Tampereen yliopisto

Joni Nikki

SAMANAIKAISOHJELMOINTI C++20-OHJELMOINTIKIELEN AVULLA

Informaatioteknologian ja viestinnän tiedekunta
Kandidaattitutkielma
Toukokuu 2021

TIIVISTELMÄ

Joni Nikki: Samanaikaisohjelmointi C++20-ohjelmointikielen avulla
Kandidaattitutkielma
Tampereen yliopisto
Tietojenkäsittelytieteiden tutkinto-ohjelma
Toukokuu 2021

Samanaikaisohjelmoinnin tarve ilmenee luonnollisesti monisäikeistetyssä ympäristössä, jossa jaettujen resurssien omistajuutta on hallittava eri ohjauksen synkronointimekanismein. Moniytimellisissä järjestelmissä fyysisten ytimien yhteisen laskennallisen tehon valjastamiseksi on tehtävä ilmaistava sen koostavilla alitehtävillä, jotka suorittavien aliohjelmien laskentojen tuloksien koostuma edustaa tehtävän tulosta. Luonnollisesti seuraa, että samanaikaisuuden ilmaiseminen on nykypäivän tyypillisissä ohjelmistoympäristöissä suoritettavien ohjelmien ilmeinen tarve.

Tässä tutkielmassa käsitellään C++20-standardin määrittelemän C++-ohjelmointikielen kykenevyyttä samanaikaisten ja rinnakkaisten ongelmien ilmaisemisessa sikäli kuin ilmaisun mahdollistaa standardin määrittelemä standardikirjasto tai C++20:ssa lisätyt vuorottaisrutiinit. Tekstissä osoitetaan, että C++20 tarjoaa keinoja niin kilpailevan samanaikaisuuden kuin yhteistyömoniajon harjoittamiseen ja että vuorottaisrutiinit ovat ilmaisuvoimainen ja suorituskyvyllisesti etevä tapa toteuttaa tyypillisiä samanaikaisohjelmoinnissa käytettyjä rakenteita ja idiomeja.

Kirjallisuuskatsauksena toteutetussa tutkielmassa painotettiin lähteitä, jotka käsittelivät vuorottaisrutiineja samanaikaisohjelmoinnin työkaluna, ja joiden avulla pystyttiin kartoittamaan vuorottaisrutiineille ominaisia sovellusalueita. Esimerkiksi huomataan, että vuorottaisrutiinit korvaavat osin FSM:n (Finite State Machine) ja CPS:n (Continuous Passing Style) sovellusalueita niiden ergonomisuuden vuoksi ja että vuorottaisrutiinien edellyttämä kekovaraus on vältettävissä tiettyjen ehtojen täyttyessä, mikä parantaa niiden käyttöönottoastetta sulautetuissa ja muuten niukkaresurssisissa järjestelmissä.

Tutkielmassa perehdytään myös siihen, miten C++:n samanaikaisuuden ja rinnakkaisuuden ilmaisukeinoja voisi vielä parantaa. Todetaan, että standardikirjaston tuki hajautetuille järjestelmille on olematonta, minkä takia hajautettua laskentaa harjoittava ohjelma on työlästä toteuttaa C++:lla ja että vuorottaisrutiinien kirjastotuki jää minimaaliseksi, missä ohjelmoijan on tehtävä itse yleisiä vuorottaisrutiinitoteutuksia kuten *generator* ja *task*.

Avainsanat: samanaikaisuus, rinnakkaisuus, C++, vuorottaisrutiinit

Sisällysluettelo

1 Johdanto

C++ on monta ohjelmointiparadigmaa yhdistävä *yleisohjelmointikieli* (general-purpose programming language), joka pohjautuu C-kieleen [14, luku 1.2]. C++ on alustariippumaton, mikäli tarkastellaan lähdekoodia – käännetty C++-ohjelma suorittuu vain tietyllä alustalla. C++ on näin ollen monikäyttöinen kieli ja käytettävissä moneen ohjelmointitehtävään. [14, luku 2.2] C++-kielen ja C++-standardikirjaston määrittelevä C++20-standardi julkaistiin joulukuussa 2020, ja se korvasi edeltävän C++17-standardin [8].

Tässä tutkielmassa tarkastellaan samanaikaisen ohjelman kirjoittamista C++20:lla. Erityisessä tarkastelussa on C++20:een lisätyt *vuorottaisrutiinit* (coroutines) (luku 4) – niihin liittyvä taustateoria ja niiden sovellusalueet. Tutkimuskysymyksenä on, millaiset eri samanaikaisohjelmoinnissa tavattavat ongelmat ovat ratkottavissa C++20:n tarjoamilla työkaluilla niin kielitasolla kuin standardikirjaston avulla ja miten C++:n antia voi tällä saralla vielä parantaa. Tekstissä ilmenee, kuinka vuorottaisrutiinien avulla on mielekästä ratkoa tiettyjä samanaikaisongelmia (luku 5) ja että ne täten ovat miellettävissä niin ikään samanaikaisohjelmoinnin työkaluiksi.

Tarve samanaikaisohjelmoinnille ilmenee luonnollisesti monisäikeistetyssä ympäristössä, jossa käytetyt resurssit joudutaan usein jakamaan säikeiden kesken, jolloin syntyy tarve haalia resurssin omistajuus sen käytön ajaksi. [7] C++ tarjoaa monisäikeistettyä suoritusta tukevan muistimallin ja abstraktion suoritussäikeille, joiden avulla ohjelmoija voi kirjoittaa monisäikeistetyn ohjelman alustariippumattomasti (luku 3.1). C++:n standardikirjasto tarjoaa ohjausmekanismeja, joiden avulla ohjelmoija hallitsee, millä suoritussäikeellä on pääsy tiettyihin jaettuihin resursseihin (luku 3.2). Vuorottaisrutiinit mahdollistavat asynkronisen koodin ilmaisemisen ilman logiikan jakamista eri funktioihin joidenkin ulkoisten tai pitkien tapahtumien tuloksista riippuvissa kohdissa (luku 5.2). Huomataan, että samanaikaisohjelmoinnin käsittelemiä ongelmia kohdataan monissa eri ympäristöissä.

Aluksi esitellään samanaikaisuus ja rinnakkaisuus, jotka ovat läheisiä käsitteitä. Tämän jälkeen seuraa vuorottaisrutiinien esittely. Vuorottaisrutiinien sovellusalueiden käsittely riippuu muiden lukujen käsittelemistä asioista, joten ne ovat tutkielmassa viimeisenä. Lopuksi on yhteenvedon muodossa pohdintaa siitä, millaista samanaikaisohjelmointi on C++20:n avulla ja millainen rooli vuorottaisrutiineilla on samanaikaisuudessa.

2 Tutkimusmenetelmä

Tutkielma on toteutettu kirjallisuuskatsauksena – sen sisältö on laajalti koontia löydetystä aiemmasta aiheeseen kuuluvasta tutkimuksesta. Ensisijaisina lähteinä on käytetty eri tieteellisten julkaisujen artikkeleja ja C++20-standardia. Ensisijaisina aineistotietokantoina toimi ACM Digital Library, SpringerLink, SFS Online ja Andor. Täydentävinä lähteinä toimi *cppreference.com*-sivusto [10] ja *The Old New Thing* -blogi [3] [4].

Aineistotietokannoissa käytettyjä hakulausekkeita oli *c++ coroutines*, *concurrency, concurrency AND parallelism* ja *(c++20 OR c++) AND concurrency*. Löydetyistä teoksista suosittiin uusimpia, ja kaikki lähteet ovatkin enintään kahdentoista vuoden takaa. C++:n vuorottaisrutiineja koskevat lähteet ovat viime vuosilta ja standardilähde on joulukuulta 2020. Vanhin artikkelilähde [11] on vuodelta 2009, koskien vuorottaisrutiinien ”renessanssia” ja asymmetrisiä vuorottaisrutiineja. Vanhin kirjalähde [6] on vuodelta 2008, koskien samanaikaisohjelmointia Windows-ympäristössä. Kaikki lähteet ovat englanninkielisiä – suomenkielisiä, käyviksi katsottuja teoksia ei löytynyt.

Kirjallisuutta etsittiin vaiheittain yhdessä luonnostelman kehityksen kanssa. Työn alussa paino oli erityisesti C++20:n vuorottaisrutiineissa, ja ne olivat yksinomainen alkuperäinen tutkimuskohde. Kuitenkin vuorottaisrutiineilla ratkotaan juuri samanaikaisohjelmoinnissa kohdattuja ongelmia, ja tämän yhteyden takia työhön tuli mittava osio samanaikaisuusohjelmoinnista, mikä heijastui myös lähteissä. Lopputuloksen esittäminen oli myös mielekkäämpää, kun leipäteksti käsitteli aihetta laajasti.

Yksittäisen lähteen tarkastelussa kiinnitin huomiota varsinkin niiden viittauksiin, avainsanoihin ja tiivistelmään. Mikäli näiden perustella katsoin lähteen olennaiseksi, sisällytin sen tarkempaan luentaan. Lähteet suodattuivat myöhemmin pois, mikäli ne eivät osoittautuneet sisällöllisesti mielekkäiksi, joko liialla päällekkäisyydellä tai työn aihetta sivuuttamalla. Lähteet luokittuivat aikakauslehtien julkaisuihin, kirjoihin, standardeihin, konferenssiartikkeleihin ja web-sivuihin.

3 Samanaikaisuus

Ohjelmointiteoriassa *samanaikaisuus* (concurrency) tarkoittaa usean laskennallisen tehtävän suorittamista samaan aikaan. Samanaikaisuutta käytetään esimerkiksi *suoritustehon* (throughput) parantamisessa valjastamalla useampi suoritin yksittäisen tehtävän suorittamiseksi tai *reagoivuuden* (responsiveness) parantamisessa jättämällä vain osa ohjelmaa odottamaan vastausta. [14, luku 5.3] [7] Samanaikaisuudessa ohjelma pyritään jäsentämään siten, että se koostuisi useasta *ohjausvuosta* (control flow), siinä missä *rinnakkaisuudessa* (parallelism) (luku 3.3) keskitytään tietyn tehtävän suorituksen nopeuttamiseen [16]. Tässä luvussa perehdytään samanaikaisohjelmoinnin teoriaan sikäli kuin se on C++-ohjelmointiin asiaankuuluvaa.

Koska nykyajan tietokoneissa parannukset suorituskyvyssä tulevat useimmiten ytimien lukumäärällisestä kasvusta eikä niinkään yksittäisen ytimen suorituskyvyn kasvusta, on samanaikaisohjelmointi tärkeä keino tietokoneen kaiken laskennallisen tehon hyödyntämiseksi [15]. Nykyisiä laitteistotason samanaikaisuuteen kykeneviä tietokoneita edelsivät yksiytimelliset järjestelmät, jotka kykenivät kerrallaan suorittamaan vain yhtä tehtävää. Tällainen järjestelmä pystyi kuitenkin limittämään tehtävien suorituksia vaikuttaakseen tekevänsä tehtäviä samanaikaisesti, harjoittaen näin *tehtävien vaihdantaa* (task switching). Vaikka näin ollen yksiytimelliset järjestelmät pystyvät käsitteellistämään samanaikaisuutta, ohjelmien käyttäytyminen saattaa erota verrattaen laitteistotason samanaikaisuuden ympäristöihin, ja esimerkiksi tietyt muistimallin huomioimattomuudesta juontuvat samanaikaisohjelmoinnin ongelmat saattavat ilmaantua vain jälkimmäisessä ympäristössä. [18, luku 1.1]

W. Morwen Gentleman [7] tunnisti kolme samanaikaisuuden paradigmaa. Vanhin ja yleisin samanaikaisuusparadigma on *kilpaileva* (competitive) samanaikaisuus, jossa säikeet riippuvat jaetuista resursseista, esimerkiksi tietorakenteista tai laitteista, ja käyttävät resurssin omistajuuden haalimiseksi erilaisia, erityisesti *poissulkevia* (exclusive) ohjausmekanismeja, esimerkiksi *opastin* (semaphore), *kriittinen alue* (critical section) ja *monitorit*, joiden avulla saavutetaan *keskinäinen poissulkeminen* (mutual exclusion) (luku 3.2.1).

Eri ohjausmekanismien soveltaminen ei ole yksiselkoista, vaan kilpailevaa samanaikaisuutta harjoitettaessa on tyypillisinä huolenaiheina muun muassa *lukkiintuminen* (deadlock), missä suoritussäikeiden suorituksen jatkuminen riippuu toistensa omistajuudessa olevista resursseista [16], *nälkiintyminen* (starvation), missä *vuorontaja* (scheduler) jättää jonkin säikeen ikuisesti suorittamatta [14, luku 42.3.1], ja *säieturvallisuus* (thread-safety), missä jokaisella säikeellä on johdonmukainen näkymä käsiteltyyn dataan [18, luku 6.1]. [7]

Kun *palvelukeskeisen arkkitehtuurin* (service-oriented architecture) eli *SOA*:n säikeet ovat vuorovaikutuksessa ennaltamäärätyllä, koordinoidulla tavalla, harjoittaa se

koordinoitua samanaikaisuutta, joka on toinen tunnistettu paradigma. SOA käyttää eri säikeitä kullekin palvelulle ja näin säikeille mielletään tiettyjä rooleja. Säikeet sitten kommunikoivat keskenään tyypillisesti järjestelmällisesti rakennettujen viestien avulla. Koska tiedetään etukäteen, miten ja milloin kommunikointi tapahtuu, avaa paradigma optimointimahdollisuuksia. [7]

Kolmas paradigma on *yhteistyöllinen* (collaborative) samanaikaisuus, jossa säikeet antavat tietyssä ajassa siihenastisen laskennan tuloksen sen epätarkkuudesta tai saatavilla olleen tiedon niukkuudesta huolimatta. Paradigman sovellusympäristöt ovat monitahoisia, laajalti tuntemattomia ja jatkuvasti muuttuvia ja vaativat täten karkeita arvioita ja ympäristön jatkuvaa luotausta. Tyypillistä paradigmalle on poikkeuksellisen tilan käsitteleminen ajoaikana ja suorituksen jatkaminen normaalisti – virheeseen ajautunut säie raportoi tällöin tuloksekseen virhettä edustavan arvon. Käytäntöä voidaan verrata IEEE 754 -liukulukujen eri virheitä edustaviin erikoisarvoihin. [7]

Samanaikaisuusohjelmointi vaatii omanlaisensa ajattelumallin. Esimerkiksi lukkomekanismien (luku 3.2.1) käytössä on ajattelumallin oltava datakeskeinen eikä koodikeskeinen – kahden samaa jaettua resurssia käyttävän funktion on käytettävä samoja eikä kaksia lukkoja. Kriittisten alueiden tarkka alku ja loppu on osattava hahmottaa. Synkronointiprimitiiveille (luku 3.2) on annettava oikea paikallisuus – globaalia muuttujaa ei ole mielekästä ja jossain tapauksissa edes mahdollista käyttää eri *ilmentymien* (instance) resurssien hallinnoimiseen. Oikean ajattelumallin opetus saattaisikin helpottaa samanaikaisuusohjelmoinnin oppimista. [15]

3.1 Monisäikeistys

C++-ohjelmassa säie on abstraktio tietokoneen laitteiston harjoittamasta laskennasta [14, luku 42.2]. C++-ohjelma sisältää globaaliin moduuliin liitetyn globaalin `main()`-funktion, joka ohjelma suoritettaessa kutsutaan sille luotavasta pääsäikeestä [8, s. 86]. C++-ohjelman funktiorungot taas tyypillisesti käsittävät *kootun lauseen* (compound statement) [8, s. 213], jonka alilauseet suoritetaan jaksotetusti [8, s. 153]. Jos tietyn suoritussäikeen suorittama evaluointi A on *jaksotettu ennen* (sequenced before) samaisen säikeen suorittamaa evaluointi B:tä, A:n suoritus edeltää B:n suoritusta, toisin sanoen A *tapahtuu ennen* (happens before) B:tä [8, s. 79–82].

Huomataan, että säikeensisäisesti *jaksotettu ennen* -suhde takaa *tapahtuu ennen* -suhteen. C++11-standardissa määriteltyä *monisäikeistettyä suoritusta* (multi-threaded execution) [16] harjoitettaessa on evaluointien tapahtumisjärjestykseen kiinnitettävä tarkemmin huomiota. C++-standardi määrittelee suoritussäikeen yksittäisenä ohjausvuona, joka käsittää säikeelle sen luomiskutsussa osoitetun alkufunktion ja rekursiivisesti tarkasteltuna kaikki sittemmin säikeen suorittamat funktiokutsut. C++-standardi määrittelee tilanteita, joissa jokin evaluointi A *synkronoituu* (synchronizes

with) jonkin evaluoinnin B kanssa. Tällöin A *tapahtuu säikeiden välisesti ennen* (inter-thread happens before) B:tä. [8, s. 81–82]

C++11:ssä tuotiin standardikirjastoon yksittäistä *suoritussäiettä* tai *säiettä* [8, s. 81] edustava `<thread>`-otsakkeen `std::thread`-luokka. Sille osoitetaan rakentajassa kutsuttava funktio ja, kuten C++-ohjelma päättyy (kutsuen `std::exit`-funktiota [8, s. 86]) palautuessaan `main()`-funktiosta, päättyy se palautuessaan osoitetusta alkufunktiostaan. [18, luku 2.1] Rakentajaan voidaan myös syöttää decay-esitykseltään *siirtorakennettavia* (move-constructible) parametreja. Luotava säie suorittaa käytännössä

```
invoke(decay-copy(std::forward<F>(f)), decay-copy(std::forward<Args>(args))...)
```

missä rakentajan ensimmäisenä parametrina annettu `f` edustaa säikeen alkufunktiota, `args`… on loput rakentajan parametrit ja `decay-copy` on [8, s. 481]

```
template<class T> constexpr decay_t<T> decay-copy(T&& v)
    noexcept (is_nothrow_convertible_v<T, decay_t<T>>)
{ return std::forward<T>(v); }
```

jonka kutsut suoritetaan rakentajaa kutsuneelta säikeeltä. [8, s. 1581–1582] Tästä seuraa, että esimerkiksi lauseke `std::thread{[](auto){},std::string{"hi"}}` aiheuttaa kolmen merkkijono-olion rakentamisen: (1) säikeen rakentajaan syötetty olio, (2) `decay-copy()`-kutsulla alustettu olio ja (3) uudelta säikeeltä kutsutun alkufunktion parametriolio.

Säieolio purkajassaan, jos olio vielä viittaa tiettyyn säikeeseen eli jos `joinable()` on `true`, kutsuu `terminate()`. Viittaaminen voidaan päättää joko odottamalla säikeen suorituksen päättymiseen asti kutsumalla `join()` tai erottamalla olion edustama säie oliosta kutsumalla `detach()`. Suoritussäiettä myöskin mallintavan `std::jthread`-luokan käyttäytyminen eroaa tästä hieman – aihetta käsitellään luvussa 3.2. [8, s. 1582–1583]

Kahden lausekkeen laskennat ovat *ristiriidassa* (conflicting), jos toinen niistä muokkaa muistipaikkaa ja toinen lukee tai muokkaa samaista muistipaikkaa. Ohjelman suoritus sisältää *data race* -tilanteen, jos se sisältää kaksi mahdollisesti samanaikaista ristiriitaavaa toimintaa eikä kumpikaan tapahdu ennen toista eivätkä molemmat ole atomisia. Tällöin ohjelman käytös on *määrittelemätöntä* (undefined behavior), eli C++-standardi ei ole asettanut sen käyttäytymiselle vaatimuksia [8, s. 8]. [8, s. 81–84] Data race on tietynlainen *kilpailutilanne* (race condition) eli tilanne, jossa laskennan tulos riippuu operaatioiden keskinäisestä järjestelystä [18, luku 3.1.1]. Ohjelmat 1 ja 2 havainnollistavat data race -tilanteen ja siltä välttymisen.

```
1 int cnt = 0;
2 auto f = [&]{cnt++;};
3 std::thread t1{f}, t2{f}, t3{f}; // undefined behavior
```

Ohjelma 1. Ohjelmassa on kolme tapahtumisjärjestykseltään määrittelemätöntä evaluointia, jotka käsittävät `cnt`-muuttujan nimeämän `int`-olion arvon muuttamisen [10].

```
1 std::atomic<int> cnt{0};
2 auto f = [&]{cnt++;};
3 std::thread t1{f}, t2{f}, t3{f}; // OK
```

Ohjelma 2. Muuttujan `f` nimeämän olion kutsuoperaattorin runko sisältää vain atomisen operaation [10].

C++-standardin määrittelemää C++-kieltä täytäntöön panevat *toteutukset* (implementations) [8, s. 1] jäljentävät standardin kuvaileman epädeterministisen *abstraktin koneen* (abstract machine) käyttäytymistä. Standardia noudattavan toteutuksen käyttäytyminen ei ole täysin täsmennetty. Joissain tilanteissa standardi kuvailee toteutuksesta riippuvaa *täsmentämätöntä käyttäytymistä* (unspecified behavior) – samalla mahdollisesti määritellen joukon sallittavia käyttäytymisiä – tai *toteutusriippuvaa käyttäytymistä* (implementation-defined behavior), jonka toteutus on dokumentoinut. Ohjelman 1 käyttäytyminen on *määrittelemätöntä* – sen suorittaminen voi tehdä mitä vain. Todellisissa ympäristöissä kohdattaneen tapauksia, joissa jokin standardin määrittelemättömäksi linjaama käyttäytyminen tuottaa sille ympäristölle ominaisen käytöksen, tai oletetaan tapahtumattomaksi – esimerkiksi etumerkillisen aritmetiikan ylivuoto on määrittelemätöntä käyttäytymistä [8, s. 74], mikä tällöin saatettaneen olettaa tapahtumattomaksi. [8, s. 5–8]

Ohjelma 2 on *lukoton* (lock-free): se on samanaikainen ohjelma, jossa ei eksplisiittisesti käytetä lukkoja (luku 3.2.1). C++-standardikirjasto tarjoaa alustariippumattomia tapoja lukottomaan ohjelmointiin, joista esimerkki on ohjelman 2 `std::atomic<int>`. [14, luku 41.3] Lukottomilla tietorakenteilla saavutetaan suurempi samanaikaisuus, ja koska ne eivät käytä lukkoja, on niiden mahdotonta lukkiintua. Kuitenkin lukoton samanaikaisohjelmointi tulee siihen harjaantumattoman ohjelmoijan rajoittaa yksinkertaisiin laskureihin, koska sen oikeelliseen harjoittamiseen vaaditaan erityistä tietämystä käytetystä tietokonearkkitehtuurista ja toteutusyksityiskohdista [14, luku 41.3]. Lisäksi edellytetyt atomiset operaatiot saattavat suoriutua vastaavia ei-atomisia operaatioita paljon hitaammin. [18, luku 7.1.4] Kun lukotonta tietorakennetta käsittelevät säikeet suoriutuvat rajallisessa määrässä vaiheita riippumatta muiden rakennetta käsittelevien säikeiden käytöksestä, on rakenne myös *odotukseton* (wait-free) [18, luku 7.1.3].

3.2 Säikeiden välinen kommunikointi

Tässä aliluvussa esitellään keinoja säikeiden väliseen kommunikointiin C++-ohjelmassa. C++-standardikirjasto tarjoaa tapoja toisella tai samalla säikeellä suoritetun funktion tuloksen – arvon tai poikkeuksen – noutamiseen [8, s. 1617]. `<future>`-otsakkeen `std::future`-luokkakaavain tarjoaa tavan vastaanottaa asynkronisen tehtävän – jota taas edustaa `std::promise`-luokkakaavain – tulos [14, luku 42.4.4]. Näin `std::future` mallintaa

asynkronista paluuoliota (asynchronous return object) – se lukee tuloksia jostain *jaetusta tilasta* (shared state), johon *asynkronista tuottajaa* (asynchronous provider) mallintava **`std::promise`** tuottaa tuloksen [8, s. 1619].

Futuuriolio tulee *valmiiksi* sen käsittämän funktiokutsun suoriuduttua joko palauttamalla arvon tai heittämällä poikkeuksen. Jäsenfunktio **`get()`** odotuttaa kutsujaa, kunnes futuuri on valmis. Promise-olion **`set_value()`**-jäsenfunktio atomisesti varastoi annetun arvon, joka voidaan noutaa **`get_future()`**-jäsenfunktion palauttaman futuuriolion avulla. Mikäli promise-olio tuhotaan asettamatta arvoa, varastoidaan poikkeus. [18, luku 4.2] Ohjelma 3 havainnollistaa toiminnallisuutta.

```
1 std::promise<int> p;
2 auto fut = p.get_future();
3 std::jthread t{[](auto p) { p.set_value(42); }, std::move(p)};
4 int res = fut.get();
```

Ohjelma 3. Lambda-funktio esittää asynkronista operaatiota, jonka tulosta edustava *int*-olio luetaan toisella säikeellä.

Ohjelmassa 3 käytetään C++20:ssa standardikirjastoon lisättyä **`std::jthread`**-luokkaa, joka toteuttaa **`std::thread`**-luokan toiminnallisuuden, sisältää pysähtymispyyntöjä välittävän **`std::stop_token`**-olion ja purkajassaan, mikäli **`joinable()`** on **`true`**, kutsuu **`request_stop()`** ja **`join()`**. Näin luokka tukee pysähtymispyyntöjen tekemistä ja automaattista liittymistä. Rakentajassa, jos lauseke on kelvollinen, uusi säie suorittaa

```
invoke(decay-copy(std::forward<F>(f)), get_stop_token(),
        decay-copy(std::forward<Args>(args))...)
```

ja muussa tapauksessa luvussa 3.1 esitellyn lausekkeen. [8, s. 1583–1585]

Edellä käsitellyt futuurioliot mallintavat *tehtäväpohjaista* (task-based) samanaikaisuutta, jossa määritellään yksittäisiä, yhden tuloksen palauttavia tehtäviä. C++ tarjoaa ohjauksen synkronointimekanismeja tilanteisiin, kun samanaikaisuutta ei ilmaista tehtäväpohjaisesti. [14, luku 42.4] **`std::counting_semaphore`** on resurssimäärää käsitteellistävä synkronointiprimitiivi, *opastin*. Sen kaavainerikoistuman alias **`std::binary_semaphore`** on kaksitilainen erityistapaus. Luokka **`std::latch`** odotuttaa siihen saapuneita säikeitä, kunnes saapujia on haluttu määrä. Vastaavanlainen luokkakaavain **`std::barrier`** antaa määritellä funktion, jota kutsutaan odotusvaiheen päättyessä ja aloittaa odotusvaiheen aina alusta. **`std::latch`** taas päästää seuraavat saapujat heti läpi. Luokka **`std::condition_variable`** tarjoaa tavan odottaa jonkin ehdon täyttymistä. Vastaavanlainen luokka **`std::condition_variable_any`** sallii odottamiseen ja täyttymisen tarkastelemiseen käytettävän muutakin lukkoa (luku 3.2.1) kuin **`std::unique_lock<std::mutex>`**. [8, s. 1606–1615]

3.2.1 Keskinäinen poissulkeminen

Suoritusagentti (execution agent) on säie tai vastaava entiteetti, joka voi suorittaa työtä rinnakkaisesti muiden agenttien kanssa [8, s. 1573]. C++-standardi määrittelee vaatimukset *lukittavalle*, ainakin `lock()`- ja `unlock()`-jäsenfunktiot omaavalle oliolle, joka voidaan syöttää standardikirjaston tarjoamalle *lukolle*, jonka omistajuuden sitten suoritusagentti voi haalia [8, s. 1612–1613]. Ohjelma 4 havainnollistaa lukittavan, binääristä opastinta edustavan `std::mutex`-olion käyttöä.

Ohjelma 4 käyttää *RAII*:tä hyödyntävää `std::lock_guard`-luokkakaavainta, joka kutsuu rakentimessaan lukittavan olion `lock()`-jäsenfunktiota ja purkajassaan sen `unlock()`-jäsenfunktiota. Ohjelmoijalle ei välttämättä riitä tämäntapainen ratkaisu, missä lukinta on voimassa koko lukon elinkaaren ajan. Standardikirjaston `std::unique_lock`-luokkakaavain mallintaa lukkoa, joka voi olla omistuksettomassa tilassa. Näin ollen omistajuuden voi siirtää ilmentymältä toiselle. Koska mahdollinen omistuksettomuuden tila edellyttää ylimääräistä kirjanpitoa, kannattaa ohjelmoijan käyttää `std::unique_lock`-kaavainta vain tarvitessaan sen tarjoamaa lisätoiminnallisuutta. [18, luku 3.2]

```
1 std::mutex mtx;
2 std::vector<int> v;
3 auto work = [&](int res){
4     std::this_thread::sleep_for(std::chrono::seconds(1));
5     std::lock_guard lk{mtx};
6     v.push_back(res);
7 };
8 { std::jthread worker1{work, 1}, worker2{work, 2}; }
9 int res = v[0] + v[1];
```

Ohjelma 4. Manipuloidakseen *v*-oliota samanaikaisesti työtä edustava lambda-olio lukitsee *mtx*-olion ja avaa sen lohkosta poistuttaessa.

Luokalla `std::mutex` harjoitettu keskinäinen poissulkeminen saattaa olla epämielekästä tilanteissa, joissa jaettua tietorakennetta päivitetään harvoin. Poissuljentaa tarvitaan vain rakennetta päivitettäessä ja muuten pääsy rakenteeseen voi tapahtua samanaikaisesti. Tarvitaan tapa ilmaista erikseen, halutaanko poissulkeva vai jaettu rakenteen omistajuus. [18, luku 3.3.2] Standardi määrittelee jaetun mutex-tyypin (*shared mutex type*), joka täyttää mutex-tyypin vaatimukset ja toteuttaa myös jäsenfunktiot `lock_shared()`, `unlock_shared()` ja `try_lock_shared()`. Jaettuja mutex-tyyppejä on standardikirjastossa `std::shared_mutex` ja `std::shared_time_mutex`. [8, s. 1592] Jaettua omistajuutta hallinnoiva `std::shared_lock`-luokkakaavain käyttää jaettua mutex-tyyppiä [8, s. 1601].

Sisäkkäisiä lukintoja eli mutex-olion jo lukinneen säikeen tekemiä lisälukintoja varten standardikirjasto tarjoaa `std::recursive_mutex`-luokan. Omistajuus vapautuu, kun kaikki säikeen tekemät lukinnat on kumottu. [18, luku 3.3.3]

3.3 Rinnakkaisuus

Rinnakkaisuudessa tyypillisesti osallistetaan useaa eri suoritinta, ydintä tai hajautettuun (luku 3.3.1) järjestelmään osallistuvaa tietokonetta saman laskennallisen tehtävän suorittamisessa suorituksen nopeuttamiseksi. Kun monisäikeistetty (luku 3.1) ohjelma suorituttaa jonkin yksittäisen tehtävän koostavat alitehtävät usealla suoritussäikeellä, toteuttaa se rinnakkaisuutta *suoritustasolla* (execution-level). *Suoritustason rinnakkaisuutta* tapahtuu niin samanaikaisuudessa kuin rinnakkaisuudessa, mutta jälkimmäisessä suoritetaan nimenomaan tiettyä tehtävää. [16]

Suoritustason ohella rinnakkaisuutta voi tapahtua:

- *bittitasolla* (bit-level), jolloin suurempi osa laskennasta arvoilla voidaan tehdä kerralla suorittimen suurentuneen sanakoon johdosta,
- *käskytasolla* (instruction-level), jolloin käskyvirran useaa eri käskyä suoritetaan yhtäaikaisesti – mikä on luontaista superskalaarisissa suorittimissa – ja
- *datatasolla* (data-level), jolloin hyödynnetään laskentaa yksityyppisten arvojen vektoreilla [16].

C++-standardi määrittelee *rinnakkaisen algoritmin* funktiokaavaimena, jolla on `ExecutionPolicy`-niminen kaavainparametri [8, s. 1046]. C++17:ssä standardikirjastoon tuodut `<execution>`-otsakkeen eri rinnakkaisuuspolitiikkoja edustavat vakiomuuttujat `std::execution::seq`, `std::execution::par` ja `std::execution::par_unseq` käsittää jaksotettua suoritusta, säietason rinnakkaisuutta ja säietason rinnakkaisuutta vektorisoinnilla (eli laskentaa *vektoreilla*, useita saman datatyypin arvoja sisältävillä rekistereillä). Nämä voidaan syöttää ensimmäisenä parametrina tiettyihin standardikirjaston algoritmeihin, jolloin algoritmi sallitaan suorittuvan kyseistä rinnakkaisuuspolitiikkaa harjoittaen. C++20-standardissa esiteltiin `std::execution::unseq`-muuttuja, joka edustaa vektorisointia ilman säietason rinnakkaisuutta. [16] [8, s. 736–737] Ohjelma 5 havainnollistaa eri rinnakkaisuuspolitiikkojen hyödyntämistä.

```
1 void calculate_sums(auto first, auto last, auto d_first) {
2     using namespace std::execution;
3     std::transform(par, first, last, d_first, [](const auto &x) {
4         return std::reduce(unseq, std::begin(x), std::end(x));
5     });
6 }
```

Ohjelma 5. Laskee annetun välin säiliöiden tai välien alkioiden arvojen summat säietasolla rinnakkaisesti siten, että summeeraukset tapahtuvat vektorisoidusti.

Rinnakkaisuuspolitiikkaa käytettäessä on huomioitava data race -tilanteen ja lukkiintumisen mahdollisuus. [16] Näiden tapahtumattomuuden takaaminen on rinnakkaisen algoritmin kutsujan vastuulla. Esimerkiksi

`std::execution::parallel_unsequenced_policy`-tyyppisellä oliolla kutsutut rinnakkaisalgoritmit saavat kutsua alkioita käsittäviä funktioita järjestäytymättömästi määrittelemättömillä suoritussäikeillä niin, että kutsut säikeensisäisesti ovat keskenään jaksottamattomia. [8, s. 1047–1048]

3.3.1 Hajautettu laskenta

C++11 toi tuen matalan tason rinnakkaisuudelle jaetun muistin alustoilla. Luvussa 3.3 esitellyt, C++17:ssä lisätyt rinnakkaisuuspolitiikat mahdollistavat jaetun muistin ympäristössä rinnakkaisuuden ilmaisemisen korkealla abstraktiotasolla pysyen. *Hajautetussa laskennassa* (distributed computing) samaa tehtävää suorittavat suorittimet eivät jaa muistia. Tällaiselle järjestelmälle ei ole vielä standardoitua tukea. Suunnitteilla oleva *Networking Technical Specification* mahdollistaisi matalan tason rinnakkaisuuden hajautetussa ympäristössä esimerkiksi *pistokkeiden* (sockets) ja kommunikaatioprotokollien muodossa. [5]

Drocco ym. [5] tutkivat hajautettujen säiliöiden, iteraattoreiden, suorituspolitiikkojen ja algoritmien suunnittelua. Tavoitteena oli tehdä standardikirjaston noudattaman rajapinnan toteuttava kirjasto, joka tällöin olisi abstraktiotasoltaan ja semantiikaltaan verrattavissa C++:n standardikirjastoon. Toteutuksen suorituskykyä *STL:ään* [1] eri algoritmeilla ja *array*- ja *unordered_set*-luokilla verrattaessa havaittiin, että toteutuksen rinnakkaisen suorituksen politiikan nopeusero STL:ään kasvoi lineaarisesti suhteessa muistiosituksien lukumäärään, kun osituksia oli kaksi jokaista klusterin kaksisuorittimista solmua kohtaan. Monella välillä operoivien algoritmien – kuten syöte- ja tulostevälillä operoiva *transform* – kohdalla luku jäi odotettavasti tätä pienemmäksi, varsinkin kun tulosteoperaatioiden käsittämä kohdeväli sijaitsi täysin eri osituksessa.

Drocco ym. osoittivat, että laskennan abstrahointi hajautetun muistin alustoilla onnistuu siinä missä jaetun muistin alustoillakin, STL:n idiomeissa pitäytyen [5].

[1] Standard Template Library. Nimitys C++:n geneerisille säiliöille ja algoritmeille.

4 Vuorottaisrutiinit

Tässä luvussa perehdytään *vuorottaisrutiinien* (coroutines) teoriaan erityisesti C++-ohjelmoinnin näkökulmasta. Ohjelmointi-idiomeihin, joiden toteutusta vuorottaisrutiinit helpottavat ja yleisesti vuorottaisrutiinien soveltamiseen käytännössä perehdytään luvussa 5.

C on esimerkillinen *proseduraalinen* ohjelmointikieli – siinä keskitytään laskennallisiin ongelmiin ja datan käypään jäsentämiseen. C:stä johtuva C++ tukee samaten proseduraalista ohjelmointia. [14, luku 1.2.1] Siinä missä proseduraalisessa ohjelmoinnissa ohjelma jaetaan *aliohjelmiin* (subroutines), joista ohjausvuo palautuu vain aliohjelman suorituksen päätyttyä, C++20:ssa esitellyt vuorottaisrutiinit ovat *keskeytettävissä* (suspend) – ohjausvuo pystyy palautumaan vuorottaisrutiinia kutsuneeseen aliohjelmaan sen suorituksen ollessa vielä keskeneräinen [16]. Toisena piirteenä vuorottaisrutiineille pidetään paikallisen datan säilymistä vuorottaisrutiinin kutsujen välissä. [11]

Yllä mainitut kaksi piirrettä jättävät tulkinnanvaraisuuksia vuorottaisrutiinitoteutuksiin. On hahmotettavissa kolme eri tavoin ratkottavaa ongelmakohtaa, joiden pohjalta luokitella vuorottaisrutiineja:

- ohjauksen välityksen mekanismi, joka on joko symmetrinen tai asymmetrinen;
- ovatko vuorottaisrutiinit ”first-class” -olioita vai rajoittuneita rakenteita;
- onko vuorottaisrutiini *pinollinen* (stackful) – voiko sen keskeyttää sisäkkäisestä funktiokutsusta. [11]

Vuorottaisrutiinit eivät suinkaan ole uusi idea – ne määriteltiin jo vuonna 1963 M. E. Conwayn kirjoittamassa tutkielmassa [2]. Seuraavan 20 vuoden ajan vuorottaisrutiinien näppäryyttä eri ohjauskäytänteiden ilmaisemisessa tutkittiin muun muassa tekoälyn, simulaation, samanaikaisohjelmoinnin ja tekstinkäsittelyn saralla. Sittemmin yleisohjelmointikielien kehittäjät laajalti lakkasivat vuorottaisrutiinien tarjoamisen. [11] Kuitenkin taas nykyään kielitason vuorottaisrutiinitukia tavataan muun muassa C#-, Lua-, JavaScript- ja Python-ohjelmointikielissä [2]. Eri kirjastototeutukset saattavat myös tarjota ”kevyitä” säikeitä (”*lightweight thread*”, ”*user-level thread*”, ”*green thread*”, ”*fiber*”) jotka harjoittavat säikeen semantiikkaa jaetussa osoiteavaruudessa. Nämä ovat usein toteutettu vuorottaisrutiinien avuin. [17]

C++-funktio on vuorottaisrutiini, jos sen runko sisältää `co_return`-lauseen, *await*-lausekkeen tai *yield*-lausekkeen. Tämä on ainoa syntaktinen ero aliohjelmiin [16]. Vuorottaisrutiini käyttäytyy kuin sen funktiorunko olisi korvattu ohjelman 6 hahmottamalla rungolla. [8, s. 216–217] Ohjelmassa käytetyllä await-lausekkeella ohjelmoija voi keskeyttää vuorottaisrutiinin suorituksen odottaakseen operandin edustaman laskennan päättymistä. Standardi listaa funktionsisäisiä tilanteita, joissa await-lauseke voi esiintyä – näitä kutsutaan *keskeytyskonteksteiksi* (suspension context). [8, s.

128] Keskeytetyn vuorottaisrutiinin suorituskonteksti, paikalliset muuttujat ja parametrit on varastoitu kääntäjän osoittamaan muistilohkoon [2].

Olkoon *e l-arvo* (lvalue), joka viittaa tiettyjen muuntosääntöjen läpi käytetyn await-lausekkeen operandin tai sillä sille käyvän **`operator co_await`**-funktiokuormituksen (luku 4.1) kutsumisen evaluoinnin tulokseen. Jos lausekkeen *e*`.await_ready()` totuusarvoesitys on **`true`**, tai kun vuorottaisrutiini *uudelleenkäynnistetään* (resume), lauseke *e*`.await_resume()` evaluoidaan ja await-lausekkeen tulos on evaluoinnin tulos. Jos esitys on **`false`**, vuorottaisrutiini keskeytyy ja lauseke *e*`.await_suspend(`*h*`)` evaluoidaan, missä *h* on await-lausekkeen sisältävän *vuorottaisrutiinin kahva* (coroutine handle). Mikäli evaluointi on tyypiltään **`std::coroutine_handle<Z>`**, missä **`Z`** on jokin tyyppi, kutsutaan evaluoinnin nimeämän olion **`resume()`**-jäsenfunktiota [2]. Muuten, jos evaluointi on tyypiltään **`bool`** ja sen tulos on **`false`**, vuorottaisrutiini uudelleenkäynnistetään. [8, s. 128]

```
{
    promise-type promise promise-constructor-arguments;
    try {
        co_await promise.initial_suspend();
        function-body
    } catch (...) {
        if (!initial-await-resume-called)
            throw;
        promise.unhandled_exception();
    }
final-suspend:
    co_await promise.final_suspend();
}
```

Ohjelma 6. Vuorottaisrutiinin semantiikkaa jäljittelevä funktiorunko [8, s. 217].

Ohjelman 6 nimen *promise* viittaama olio on *vuorottaisrutiinin promise-olio* (*promise object*). Ohjelmassa *promise-type* on **`std::coroutine_traits<R, P1, …, Pn>::promise_type`**, missä *R* on funktion paluutyyppi ja $P_1, \ldots, P_n$ käsittää funktioparametrien tyypit. [8, s. 217] **`<coroutine>`**-otsakkeen määrittelemän **`std::coroutine_traits`**-pääluokkakaavaimen **`promise_type`**-jäsen on alias *tarkennetulle tunnukselle* (qualified-id) **`R::promise_type`**, kun tämä on kelvollinen [8, s. 546].

Yield-lauseke voi esiintyä vain keskeytyskontekstissa ja se vastaa lauseketta **`co_await`** *p*`.yield_value(`*e*`)`, missä *p* on yield-lausekkeen sisältävän vuorottaisrutiinin promise-olion nimeävä l-arvo ja *e* on yield-lausekkeen operandi [8, s. 145].

---

[2] Jatkumon puuttuessa ohjelmoija voi esimerkiksi palauttaa **`std::noop_coroutine()`**-funktiokutsun palauttaman kahvan, joka ei tee mitään uudelleenkäynnistettäessä tai tuhottaessa [8, s.549].

Olkoon *p* l-arvo, joka nimeää vuorottaisrutiinin promise-olion. `co_return`-lause vastaa

`{` *S*`;` `goto` *final-suspend* `;` `}`

missä *S* on koottu lause `{` $operand_{opt}$ `;` *p*`.`**`return_void`**`();` `}` tai, jos lauseella on operandi, joka on *aaltosulkeinen alustuslista* (braced-init-list) tai ei tyyppiä `void` oleva lauseke, lauseke *p*`.`**`return_value`**`(`*operand*`)`, missä *operand* on lauseen operandi ja alaindeksillä ”$_{opt}$” merkitty symboli tarkoittaa syntaktisesti vapaaehtoista lauseketta [8, s. 11]. [8, s. 160]

Kuten ohjelman 6 rungosta huomataan, await-lausekkeissa kutsutut *p*:n jäsenfunktiot `initial_suspend()` ja `final_suspend()` tarjoavat keinon säädellä vuorottaisrutiinin keskeytyväisyyttä käyttäjän laatimaa funktiorunkoa ennen ja sen jälkeen. Vuorottaisrutiinin paluuarvoa edustava olio alustetaan ennen alkukeskeytyspistettä lausekkeella *p*`.`**`get_return_object`**`()`. [8, s. 217] Huomataan, että vuorottaisrutiinin käyttäytymisen säätelemisen keskiössä on sen promise-olio (luku 4.1).

Vuorottaisrutiinin keskeyttäminen varastoi sen jatkamiseksi tarvittavan tiedon. C++:ssa tähän ei sisälly pinoa, vaan esimerkiksi mahdolliset paikalliset muuttujat varastoidaan vuorottaisrutiinia edustavaan tietueeseen. Myöskään kutsupinoa ei tallenneta, ja näin ollen keskeytyksen on tapahduttava kutsupinon päällimmäisestä funktiosta. [17] Sitä vastoin pinollisilla vuorottaisrutiineilla on niiden kutsujasta erillinen pino, johon paikalliset muuttujat voidaan varastoida vuorottaisrutiini keskeytettäessä. [1] Kun ohjelmointikieli suunnitellaan alusta alkaen tukemaan vuorottaisrutiineja, niiden toteutus on useimmiten pinollinen. [17]

4.1 Vuorottaisrutiinien määritteleminen

Tässä aliluvussa kerrotaan omien vuorottaisrutiinien määrittelemisestä. Aiemman perusteella huomataan, että tämä saavutetaan erikoistamalla `std::coroutine_traits`-luokkakaavain funktion paluutyypin ja parametrien tyyppien suhteen (luku 4).

```
template <>
struct std::coroutine_traits<std::future<int>> {
    struct promise_type : std::promise<int> {
        auto get_return_object() { return this->get_future(); }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void return_value(int x) { this->set_value(x); }
        void unhandled_exception()
        { this->set_exception(std::current_exception()); }
    };
};
std::future<int> foo() { co_return 42; }
int main() { return foo().get(); }
```

Ohjelma 7. Vuorottaisrutiini `foo()` palauttaa heti luvun 42.

Ohjelmassa 7 käytetään standardikirjaston tarjoamaa `std::suspend_never`-tietuetta, jonka ilmentymän await-lausekkeen operandina käyttö ei koskaan aiheuta keskeytystä. Vastakohtaista käyttäytymistä edustaa tietue `std::suspend_always` (luku 5.1). [8, s. 549]

Kuten aiemmin todettua, funktion paluuarvoa edustava olio alustetaan promise-olion `get_return_object()`-jäsenfunktion kutsulla (luku 4). Ohjelman 7 vuorottaisrutiinin promise-olio on `std::promise<int>`-luokan perivän tietueen ilmentymä, jonka tulosta edustaa `std::future<int>`. Koska vuorottaisrutiini ei keskeydy alkukeskeytyspisteessään, suorittaa se käyttäjän laatiman, keskeytyksettömän funktiorungon heti, ja näin ollen promise-olion `return_value()`-jäsenfunktiota kutsutaan vuorottaisrutiinia välissä keskeyttämättä arvolla 42, jolloin palautettava futuuriolio on heti valmis (luku 3.2).

```
auto operator co_await(std::future<int> &&f) {
    struct R {
        std::future<int> f;
        bool await_ready() { return false; }
        int await_resume() { return f.get(); }
        void await_suspend(std::coroutine_handle<> h)
        { std::thread{[&,h]{ f.wait(); h.resume(); }}.detach(); }
    };
    return R{std::move(f)};
}
std::future<int> bar() { co_return co_await foo() * 2; }
int main() { return bar().get(); }
```

Ohjelma 8. `bar()` kaksinkertaistaa vuorottaisrutiinin `foo()` tuloksen.

```
template <class Rep, class Period>
auto operator co_await(std::chrono::duration<Rep, Period> d) {
    using std::this_thread::sleep_for;
    struct R {
        std::chrono::duration<Rep, Period> d;
        bool await_ready() { return false; }
        void await_resume() {}
        void await_suspend(std::coroutine_handle<> h)
        { std::thread{[&,h]{ sleep_for(d); h.resume(); }}.detach(); }
    };
    return R{d};
}
std::future<int> foo() {
    co_await std::chrono::seconds(1);
    co_return 42;
}
```

Ohjelma 9. `foo()` odottaa noin sekunnin ja palauttaa sitten luvun 42.

Ohjelmassa 8 await-lausekkeessa käytetyn tietueen ilmentymän `await_suspend()`-jäsenfunktio tekee säikeen, joka futuuriolion valmistuessa jatkaa vuorottaisrutiinia.

Vuorottaisrutiini päättyisi keskeytyksettä, jos `await_ready()` tarkastaisi futuuriolion valmiuden koska, kuten edellä todettiin, on olio heti valmis.

Ohjelmassa käytetty, vuorottaisrutiinin kahvaa (luku 4) edustava `std::coroutine_handle<>`-luokan ilmentymä tarjoaa `resume()`-jäsenfunktiollaan tavan uudelleenkäynnistää keskeytetty vuorottaisrutiini. Funktion kutsuminen eri suoritusagentilta on standardin määrittelemää käyttäytymistä, kun kutsuva ja vuorottaisrutiinin keskeyttämä suoritusagentti on `std::thread`- tai `std::jthread`-luokan tapahtuma tai `main()`-funktiota kutsunut säie, mikä pätee ohjelman tapauksessa. [8, s. 547] Ohjelmassa vuorottaisrutiini uudelleenkäynnistetään await-lausekkeen operandina annetun futuuriolion valmistuttua.

Jos ohjelman 8 `bar()` kutsuukin ohjelman 9 `foo()`-vuorottaisrutiinia, odottaa se noin sekunnin ennen palautumista. Ohjelmassa on siis noin sekunnin ajan kaksi keskeytynyttä vuorottaisrutiinia, `foo()` ja `bar()`.

4.2 Kekovarauksen välttäminen

Vuorottaisrutiinit auttavat monissa niukkaresurssisissa ympäristöissä kohdatuissa ongelmissa, ja tällaisilla laitteilla kekovarauksien välttämisen mahdollisuus on tärkeää (luku 5). Laatimassaan dokumentissa [13] Smith ja Nishanov tutkivat, missä olosuhteissa kekovaraukselta voidaan välttyä. Optimointia voidaan kutsua dokumentin nimen innoittamana *HALO:ksi* (Heap Allocation eLision Optimization).

```
1 generator<int> range(int from, int to)
2 { // user-authored coroutine body starts
3     for (int i = from; i < to; ++i)
4         co_yield i;
5 } // user authored coroutine body ends
6 int main() {
7     auto s = range(1, 10);
8     return std::accumulate(s.begin(), s.end(), 0);
9 }
```

Ohjelma 10. Sisältää alkuperäisen, käyttäjän laatiman vuorottaisrutiinin [13].

```
1 struct range$state-machine
2 { ... transformed user-authored-body-is-here ... };
3 generator<int> range(int from, int to)
4 { // coroutine ramp/thunk/trampoline
5    auto s = new range$state-machine(from, to);
6    return s->promise.get_return_object();
7 }
```

Ohjelma 11. Muutoksessa lisättiin muun muassa muistia varaava `new`-lauseke [13].

Ohjelman 10 *generator*-olion (luku 5.1) palauttava vuorottaisrutiini `range()` muutettaneen ohjelman 11 kaltaiseksi. Ohjelmasta 11 huomataan, että käyttäjän laatima funktiorunko vuorottaisrutiinille on korvattu kääntäjän generoimalla *ramppifunktiolla* (ramp function), joka alustaa vuorottaisrutiinin ja aloittaa *vuorottaisrutiinin tilakoneen* (state machine) eli käyttäjän laatimasta funktiorungosta kääntäjän tekemän muunnoksen. [13] Vuorottaisrutiinin kahva osoittaa keskeytettyyn tai suoritettavaan vuorottaisrutiiniin [8, s. 546].

Standardi nimittää *vuorottaisrutiinin tilaksi* (coroutine state) vuorottaisrutiinia varten mahdollisesti varattua lisävarastoa [8, s. 218]. Sen voi käsittää koostavan:

- *kääntäjän kirjanpito* eli tilakone ja suorituksen edistymistä seuraava tieto,
- promise-olio (luku 4) ja
- tavallista *pinokehystä* (stack frame) korvaava *vuorottaisrutiinin pinokehys* [3].

Edeltävä listaus on vain havainnollistus siitä, miten kääntäjä saattaa toteuttaa vuorottaisrutiinin tilan. Todellisuudessa esimerkiksi paikalliset muuttujat, joiden elinkaari ei sisällä keskeytyspistettä saatetaan varastoida tavalliseen pinokehykseen. [3]

Vuorottaisrutiinin keskeytyspisteiden välisesti tarvittava tieto sijaitsee joko pinossa tai keossa riippuen siitä, onko HALO voimassa ja onko käyttäjän syöttämä *varain* (allocator) käytössä. Jotta kekovaraus voidaan välttää, on varmistuttava *vuorottaisrutiinin pakenemisen* (escape) mahdottomuudesta tarkistamalla kaikkien vuorottaisrutiinin luovien ja sitä kutsuvasta funktiosta poistuvien suorituspolkujen myös tuhoavan vuorottaisrutiinin eli kutsuvan joko suorasti tai epäsuorasti (esimerkiksi purkajassa) kahvan `destroy()`-jäsenfunktiota. Jotta tämä olisi mahdollista ohjelman 7 tapauksessa, on seuraavien funktioiden oltava *inline*-optimoitavissa [3] : ramppifunktio, `get_return_object()`, `coroutine_handle<>::destroy`, sekä `generator<int>`-luokan `begin()`-jäsenfunktio, rakennin, siirtorakennin ja purkaja. Tähän ei siis lukeudu vuorottaisrutiinin runko tai `std::accumulate`, kunhan `begin()`-funktion palauttama iteraattori ei viittaa `generator<int>`-olioon. [13]

Smith ja Nishanov osoittivat, ettei kekovarausten välttäminen edellytä ei-toivottuja koodigenerointipäätöksiä, esimerkiksi rajattoman suurien funktioiden runkojen kopioimista, vaan edellytetyt generointipäätökset ovat haluttuja riippumatta HALO:n niiden edellyttämisestä [13].

---

[3] Funktiokutsun korvaaminen kutsuttavan funktion rungolla. Huomattakoon tekstissä viitattavan tämän varmaan tapahtumiseen, siinä missä `inline`-määre ei velvoita toteutuksia tekemään tätä. [8, s. 172] `inline`-määreellä itse asiassa sallitaan usean funktiomäärityksen ohjelmassa olemassaolo, koska useat saman funktion funktiomääritykset kieltävä *ODR*-säännöstys ei päde `inline`-funktioihin [8, s. 30–32].

5 Vuorottaisrutiinien sovellusalueet

Tässä luvussa käsitellään ongelma-alueita, joihin vuorottaisrutiineja on luontaista tai kätevä soveltaa. Mahdollistamalla ohjauksen palauttamisen suoritusta päättämättä ja suorituksen jatkamisen mielivaltaisesta suorituskontekstista helpottavat vuorottaisrutiinit yleisten ohjelmointi-idiomien kuten *tapahtumaohjatun* (event-driven) ohjelmoinnin, *yhteistyömoniajon* (co-operative multitasking) ja *generaattorien* toteuttamista. [16]

Vuorottaisrutiinien käyttöaste jonkin ohjelmointiongelman ratkaisemiseksi riippuu suuresti siitä, kuinka luontevasti ja tehokkaasti ongelman ratkaisun voi vuorottaisrutiinien avulla ilmaista. Eri ympäristöissä vuorottaisrutiineille käypiä sovellusalueita kohdataan eri määrin. Esimerkki vuorottaisrutiineille antoisasta ympäristöstä on *esineiden internet* tai *Internet of Things* (IoT). Internetiin yhdistettyjen sulautettujen järjestelmien on kestettävä eri tietoturvauhkia ja ohjelmavirheet on minimoitava, jotta IoT-ohjelmisto tulisi laajasti käytetyksi. IoT-laitteet ovat myös vuorovaikutuksessa anturien kanssa, mikä on pitkälti asynkroninen prosessi. Aliluvuissa huomataan, kuinka vuorottaisrutiinit sopivat tähän kuvaan. [2]

Vuorottaisrutiinien syntaktinen keveys sikäli kuin niiden avulla ratkotaan jotain niille ominaista ongelmaa tulee ilmeiseksi taulukosta 1, jossa on taulukoitu FSM- (luku 5.2) ja vuorottaisrutiinitoteutuksien lähdekoodien pituudet. Kyseessä oli yksinkertainen, sulautetulle järjestelmälle tehty asynkronisen ohjelman koodi. Jos ei oteta huomioon uudelleenkäytettävää kirjastokoodia, on vuorottaisrutiinitoteutus 75 % lyhyempi. Näin ollen vuorottaisrutiineja voi sanoa ainakin *ergonomisesti* tehokkaaksi – silti niiden käyttö maksaa suorituskyvyllisesti, ja etenkin muistin varauksen tarve vuorottaisrutiinin kehystä varten saattaa käännyttää sulautetun ympäristön ohjelmoijia. Varauksesta voi tehdä tiettyjen ehtojen (luku 4.2) pätiessä staattisen, joten kääntäjät tullevat tarjoamaan kyseisen optimoinnin tulevaisuudessa. [2]

Taulukko 1. Koodin pituutta mittaavat SLOC-lukemat asynkronisille tehtäville [2].

Platform	Library	Application	Total
Finite state machine	-	363	363
Coroutines	196	60	256

Edellä mainittu kirjasto–sovellus-jako on tärkeä havainto: siinä missä sovellukselle vartavastisen FSM-toteutuksen koodi on upotettu ohjelman logiikkaan [2], vuorottaisrutiinien käyttäytyminen määritellään sen oliolla (luku 4.1), jonka määritelmä sijaitsee muualla. C++:n vuorottaisrutiinit ovat lisäksi toiminnallisuudeltaan helposti laajennettavissa: ohjelmoija voi esimerkiksi suunnitella vuorottaisrutiinin niin, että sitä edustavaa oliota voi käyttää await-lausekkeessa, tai että sen edistys on keskeytettävissä tai tarkasteltavissa kesken suorituksen. [3]

5.1 Tarveohjattu suoritus

C#- ja JavaScript-kielissä vuorottaisrutiinin käyttäytyminen rinnastaa *dataohjattua suoritusta* (eager evaluation) – vuorottaisrutiinin suoritus alkaa heti sitä kutsuttua, palaten kutsujaan vasta ensimmäisessä keskeytyspisteessä. Pythonin vuorottaisrutiinit edustavat sen sijaan *tarveohjattua suoritusta* (lazy evaluation) eli niiden suoritus ei ala ennen kuin niiden tulosta odotetaan. C++ antaa ohjelmoijan päättää, miten vuorottaisrutiinit tällä saralla käyttäytyvät. [4]

C++:n vuorottaisrutiineista saadaan tarveohjattuja tekemällä vuorottaisrutiinin promise-olion jäsenfunktion `initial_suspend()` paluutyypiksi `std::suspend_always`, jolloin vuorottaisrutiinin runko ei ala ennen kahvan jäsenfunktion `resume()` kutsua. Vuorottaisrutiinin aloittaminen jää näin ollen sen odottajan vastuulle. [4]

Yield-lausekkeen määritelmästä (luku 4) huomataan, että sen sisältävä vuorottaisrutiini voidaan keskeyttää edellisen käytännön tavoin. Tätä hyödyntää ohjelman 12 `generator`-luokkakaavio, jota ohjelmassa käytetään päättymättömän lukujonon esittämiseen.

```
generator<int> generatorForInts(int start, int inc = 1) {
    for (int i = start;; i += inc)
        co_yield i;
}
int main() {
    auto nums = generatorForInts(-10);
    for (int i = 1; i <= 20; ++i)
        std::cout << nums << " ";
    std::cout << "\n\n";
}
```

Ohjelma 12. Vuorottaisrutiini laskee lukujonoa ilman välitilaista säiliötä [16].

Ohjelman 12 kirjoittaminen ilman vuorottaisrutiineja käyttänee väliasteista säiliötä, johon lukujono talletettaisiin, mikä vaatisi säiliön alkioiden varastointia. Vuorottaisrutiinit mahdollistavat siis usean arvon palauttamisen niin, että vuorottaisrutiinin palauttamat arvot voivat käsittää mahdollisesti päättymättömän sarjan. [16]

Synkronisesti valmistuvan operaation esitys vuorottaisrutiinina saattaa olla mielekästä myös silloin, kun vuorottaisrutiini palautuu normaalin funktion tavoin keskeytyksittä. Tarve tähän syntyy esimerkiksi silloin, kun operaation synkronisuus riippuu ympäristön tilasta, esimerkiksi siitä, onko operaation suorittamiseen tarvittava apuolio jo käynnissä vai tarvitseeko se käynnistää. [4]

5.2 Tapahtumaohjatut ohjelmat

Aiemmin mainitut IoT-laitteet ja sulautetut järjestelmät ovat arkkitehtonisesti tapahtumaohjattuja ja usein näin ollen ohjelmistototeutukseltaan asynkronisia: eri

tehtävät odottavat eri ulkoisia tapahtumia. Tapahtumaohjattu ohjelma on perinteisesti ollut vaivalloista ilmaista koodina, koska alustava logiikka erkaantuu tapahtuman käsittelevästä logiikasta. Ilmaisu vaikeutuu ennestään, kun toinenkin tapahtuma, esimerkiksi *aikakatkaisu* (timeout) pitäisi käsitellä. Lisäksi tapahtuma saattaa vaikuttaa toisia tapahtumia käsittelevien logiikkojen käyttäytymiseen. Lopputuloksena on vaikeaselkoinen, pirstaloitunut koodi, jonka huoltaminen on samoin työlästä. [1]

Työpöytäkehityksessä ratkaisu tapahtumaohjatun ohjelman kirjoittamisessa on ollut tapahtuman tulosta asynkronisesti odottavaa await-avainsanaa tukevan kielen, kuten C#:n käyttö. Kuitenkin niukkaresurssisissa sulautetuissa järjestelmissä suositaan C- ja C++-kieliä, joissa avainsanan semantiikkaa ei ole ollut saatavilla. C++20:n vuorottaisrutiinit näin ollen luovat mahdollisuuden yksinkertaistaa sulautettujen järjestelmien ja muiden resurssillisesti rajoittuneiden laitteiden ohjelmakoodia. [1]

Suuri osa IoT- tai sulautetun laitteen ohjelman kulusta on asynkronista, kun ohjelma joutuu odottamaan esimerkiksi vastausta anturilta. Mahdollisia toteutustapoja tällaiselle ohjelmalle on suorituksen *äärellisen automaatin* vaiheisiin jakava *FSM* (finite state machine) ja suorituksen jatkumon parametrina antaminen eli *CPS* (continuation passing style). Jälkimmäinen on yksinkertaisempi, mutta johtaa esimerkiksi JavaScriptissä kohdattuihin ”*callback hell*”- ja ”*pyramid of doom*” -tilanteisiin, missä usean jatkumon sarjan laadinta johtaa epämielekkääseen koodiesitykseen. Async/await-mallia on käytetty muuttamaan CPS-koodi yhdessä lohkossa oleviksi peräkkäisiksi lauseiksi. [1] C++:n kääntäjä itse asiassa tehnee vuorottaisrutiinista FSM-esityksen [2].

Ohjelma 13 havainnollistaa tilannetta, jossa käyttäjän ei tarvitse tehdä CSM- tai FSM-toteutusta asynkronisen ohjelman esittämiseksi. Käyttötilanne on tyypillinen sulautetuissa järjestelmissä. [2]

```
1 resumable someTask() {
2     co_await adc_calibrate();
3     uint8_t result = co_await adc_read();
4     process_data(result);
5 }
```

Ohjelma 13. Anturin lukeminen await-lauseketta hyödyntäen [2].

Vuorottaisrutiinitoteutuksia on jo käytetty edellä mainituissa ympäristöissä. Katsaus vuosina 2007–2018 julkaistuihin, vuorottaisrutiinien soveltamista niukkaresurssillisilla alustoilla koskeviin tutkimuksiin selvitti 19 toteutuksesta 11 olevan pinottomia. Käyttötarkoitukseksi oli 49 %:ssä tapauksista kerrottu samanaikaisuus, 43 %:ssä verkkoviestintä, 26 %:ssa anturien lukeminen ja 20 %:ssa tietovirta. Tällaiset vuorottaisrutiinien sovellusalueet eivät ole yksinomaisia resurssirajoitteisille alustoille, vaan yleisiä alustojen välisesti. Työpöytäohjelmissa anturien lukeminen olisi harvinaista ja käyttöliittymä mainittaisiin sovellusalueena. 55 % käytti 8-bittistä tai 16-bittistä

suoritinta - C++:n tuki on niille rajattua. Vähämuistisilla alustoilla vuorottaisrutiinien käyttämien dynaamisten muistivarauksien on tarpeellista olla korvattavissa kekoa käyttämättömillä versioilla. [1]

5.3 Limitetty suoritus

Nykyajan tyypillisessä suorittimessa on ydinkohtaiset L1- ja L2-tason välimuistit, missä L1-muisti on jaettu käsky- ja datavälimuistiin, ja ytimien kesken jaettu L3-tason välimuisti. Välimuisti on tyypillisesti paljon päämuistia pienempi, ja ohjelman viitatessa muistilohkoon, joka ei ole jo välimuistissa on kyseessä *välimuistihuti* (cache miss) ja ohjelman viittauksen odottaminen on *sakkausjakso* (stall). Muistioperaatioiden viivettä voidaan piilottaa hyödyntämällä *prefetch*-käskyä, joka noutaa annetun muistiosoitteen sisältämän muistilohkon, eli toisin sanoen tekee *ennaltanoudon*. Ohjelma voi tehdä käskyn esimerkiksi vuorottaisrutiinissa ja *vuoronvaihdon* [4] (context switch) toiseen, haettavasta datasta riippumattomaan käskyvirtaan, jolloin saavutetaan ajallinen ero muistihaun ja sen tuloksen käytön välille ja minimoidaan täten sakkausjaksoja. Käytäntö on esimerkki *limitetystä suorituksesta* (interleaved execution). [12] [9] Ennaltanoudon käytön mielekkyys riippuu siitä, onko sen ja lopullisen noudon välille saatavana muuta työtä [9]. Ohjelma 14 havainnollistaa käytäntöä.

```
1  template <bool suspend>
2  task<int> binary_search(vector<int> &array, int value) {
3      auto low  = 0;
4      auto size = static_cast<int>(array.size());
5      while (size > 1) {
6          auto probe = low + size / 2;
7          auto v     = co_await load<suspend>(array[probe]);
8          if (v < value) low = probe;
9          size -= size / 2;
10     }
11     if (size == 1 && array[low] < value) low++;
12     if (array[low] == value) co_return low;
13     else co_return -1;
14 }
```

Ohjelma 14. Puolitushaku joka rivillä 7 kaavainparametrista riippuen joko noutaa välittömästi viitatun arvon tai käskee sen ennaltanoudon ja keskeytyy [12].

Limitetty suoritus on hyödyllinen optimointikeino osoitepohjaisia tietorakenteita käyttäessä, missä välimuistihudit ovat todennäköisiä. Tällaisia tietorakenteita on

[4] Tekstissä viitataan ohjelmistototeutettuun vuoronvaihtoon. Käyttöjärjestelmän tekemässä vuoronvaihdossa tallennetaan CPU:n tila ja *käskyrekisteri* (instruction pointer) ja palautetaan sen tehtävän tila, jonka suoritukseen vaihdetaan. Tämä saattaa edellyttää eri käsky- ja datavälimuistinoutoja, jotka viivyttävät suoritusta edelleen. [18, luku 1.1.1]

hajautustaulut ja B+-puut ja niitä käytetään paljon esimerkiksi tietokannoissa, joiden varastoima tieto sijaitsee kokonaan päämuistissa. Taulukossa 2 on esitetty eri optimointitapojen suurin mitattu suorituskykyero suhteessa muistihauiltaan naiiviin toteutukseen. Ensimmäinen sarake kuvaa eri testattuja tietorakenteita: hajautustaulun ja binäärihaun ohella esitellään solmuja ennaltanoutava B+-puu *Masstree* ja samanaikaisuudelle optimoitu B+-puu *Bw-tree*. Optimointitapoja olivat:

- *Asynchronous Memory Access Chaining*, jossa operaatioita esitetään tilakoneina,
- *Coroutines*, jossa vuorottaisrutiini keskeytetään aina ennaltanoudon jälkeen ja
- *Group Prefetching*, jossa suoritetaan N samaa M-vaiheista operaatiota. [9]

Taulukko 2. Eri optimointitapojen paras suhteellinen suorituskyky naiiviin toteutukseen verrattuna [9].

	Compiler	Technique		
		GP	AMAC	Coro
Hash Probe	Clang	7.5	8.4	8.2
	MSVC	7.8	8.5	7.3
Binary Search	Clang	2.1	2.3	2.3
	MSVC	3.1	3.5	2.6
MassTree	Clang	-	-	3.2
Bw-tree	MSVC	1.7	1.7	1.4

Taulukosta 2 huomataan, että *Clang*-kääntäjää käytettäessä vuorottaisrutiinit suoriutuivat yhtä tehokkaasti kuin AMAC. Masstree-toteutuksen kohdalla vuorottaisrutiinit suoriutuivat naiivitoteutusta 3.2-kertaisesti tehokkaammin, edellyttäen koodipohjaan vain triviaaleja muutoksia. Näin ollen vuorottaisrutiinien avulla voidaan ilmaista limitettyä suoritusta ohjelmoijien tuottavuutta uhraamatta ja limitetystä suorituksesta saadusta suorituskyvystä menettämättä. [9]

Tietokantojen päämuisti voidaan perinteisen DRAM-varaston sijasta toteuttaa suurikapasiteettisempana *NVM*- eli Non-Volatile Memory -varastona. NVM on hitaampi, mutta limitettyä suoritusta hyödyntämällä sen muistioperaatioiden viiveet voidaan piilottaa. Uutuudellisen NVM:än käyttö tietokantojen päämuistina lisääntynee tulevaisuudessa ja limitetyn suorituksen käyttö helpottanee niiden käyttöä viivesidonnaisten työtehtävien kohdalla. [12]

5.4 Yhteistyömoniajo

Ohjelmoija valinnee tutun monisäikeistyksen moniajon ilmaisemiseksi. Säikeet kuitenkin harjoittavat *keskeyttävää moniajoa* (pre-emptive multitasking), joka vaatii monimutkaisia synkronointimekanismeja. Yhteistyömoniajossa samanaikaisten tehtävien limitys on determinististä ja kilpailutilannetta ei synny. Vuorottaisrutiini keskeytyessään jättää

käyttämänsä resurssit ennalta määrättyyn tilaan ja näin ollen tarjoaa yhteistyöllisen samanaikaisuusmallin. [11]

Ohjelmassa 15 vuorottaisrutiini keskeytyy aloitettuaan asynkronisen tiedosto-operaation. Syötetty `ctx`-olio on vastuussa vuorottaisrutiinin uudelleenkäynnistämisestä operaation valmistuttua. Käytännössä vuorottaisrutiinin voisi uudelleenkäynnistää toiselta säikeeltä – tämä riippuu `ctx`-olion toteutuksesta.

```
1 sync_task<std::size_t> file_hash(auto &ctx, auto src, auto dst) {
2     auto f          = ctx.open(src);
3     auto sz         = GetFileSize(f.native_handle, nullptr);
4     auto buf        = std::make_unique_for_overwrite<char[]>(sz);
5     auto bytes_read = co_await ctx.read(f.at(0), buf.get(), sz);
6     co_return std::hash<std::string_view>{}({buf.get(), bytes_read});
7 }
```

Ohjelma 15. Vuorottaisrutiini keskeytyy asynkronisen lukuoperaation aloitettuaan.

Windows-käyttöjärjestelmässä voi määritellä asynkronisesti valmistuvia I/O-toimituksia. Esimerkiksi funktio `WaitForMultipleObjects()` (WFMO) odottaa, kunnes jokin tai kaikki annettujen kernel-olioiden edustamista töistä on valmis. Taulukkoon voidaan kerätä asynkronisiin tiedostojen luku- tai kirjoitusoperaatioihin yhdistettyjä tapahtumaolioita, ja WFMO:lla näitä kaikkia voidaan odottaa niin, että yhden signaloiduttua WFMO-kutsu palautuu. Niin tiedosto-operaatioiden alustaminen kuin tapahtumien odottaminen ja operaatioiden tuloksista riippuvien funktioiden kutsuminen voi tapahtua yhdeltä säikeeltä ja kilpailutilanteilta (luku 3.1) vältytään. [6, luku 15] Käytäntöä havainnollistavassa ohjelmassa 15 on abstrahoitu asynkronisen I/O:n käyttämisen edellyttämä logiikka. Huomataan, ettei ohjelman synkroninen vastine eroaisi välttämättä muuten kuin `co_await`- ja `co_return`-avainsanojen puuttumisella.

Yhteistyömoniajossa saattaa ilmetä tasapuolisuusongelmia suoritusajan jaossa tehtävien kesken. Jos esimerkiksi vuorottaisrutiini käyttää synkronisesti valmistuvaa I/O-operaatiota, estyy koko ohjelman suorituksen jatkuminen. Lisäksi pitkäkestoisen operaation suorittaminen estää luonnollisesti muiden tehtävien jatkamisen – tällöin yield-lausekkeen käyttö olisi mielekästä, koska siten vuorottaisrutiini tarjoaisi muille tehtäville suoritusaikaa. [11]

6 Yhteenveto

Työn tarkoituksena oli samanaikaisten ohjelmarakenteiden ja ohjelmointi-idiomien ilmaisukeinojen kartoittaminen C++20-ohjelmointikielessä. Työssä perehdyttiin niin C++:n standardikirjaston tarjontaan kuin sen kieliominaisuuksien luomiin mahdollisuuksiin ilmaista samanaikaisia ohjelmia. Erityisesti tarkasteltiin C++20:ssa lisättyjä vuorottaisrutiineja, jotka eivät näennäisesti ole samanaikaisuuden työkalu. Kuitenkin huomattiin tilanteita, joissa vuorottaisrutiinien avulla samanaikaisen ohjelman ilmaisu oli vaihtoehtoihin verrattuna vaivattomampaa ja vähemmän koodia tarvittiin saman tehtävän esittämiseen. Huomattiin myös, että C++:n vuorottaisrutiineilla voi jopa piilottaa CPU:n sakkausjaksoja limitetyn suorituksen muodossa. Vuorottaisrutiinit osoittautuivat suorituskyvyltään lähes tai yhtä tehokkaiksi kuin vaihtoehtoiset optimointikeinot.

Kilpailevan samanaikaisuuden ohjelmoiminen on virhealtista monien eri mahdollisten kilpailutilanteiden takia. Se vaatii omanlaisensa, datakeskeisen ajattelumallin, jossa ohjelmoijan on hahmotettava jaettujen resurssien yksinomaisten ja jaettujen omistajuuksien tarkka alku ja loppu. C++ tarjoaa työkaluja kilpailevaan samanaikaisuuteen eri synkronointimekanismien muodossa ja C++20:n vuorottaisrutiinit tarjoavat yhteistyömoniajon muodossa mallin, jossa synkronointimekanismeja ei tarvita. C++:n RAII:tä hyödyntävät lukot vähentävät virheiden määrää.

C++:n tarjonta hyödyttää eri ympäristöissä kohdatuissa ongelmissa. C++20:n vuorottaisrutiinit helpottavat sulautettujen järjestelmien ohjelmoinnissa, kunhan kekovarauksen välttäviä optimointikeinoja toteutetaan kääntäjissä. C++:ssa voi suorittaa korkean abstraktion tasolla pitäytyen algoritmeja rinnakkaisesti monisäikeistämällä ja vektorisoimalla, kunhan ohjelma käsittää jaetun muistin ympäristön – hajautetun muistin ympäristöille ei ole vielä edes matalan tason tukea, mikä tosin on suunnitteilla Networking TS -luonnoksen muodossa.

Osa tutkimuskysymystä oli, miten C++:n tarjontaa voi samanaikaisuuden kohdalla vielä parantaa. Yksi selvä kehitysalue on, että standardikirjasto tarjoaisi vuorottaisrutiinitoteutuksia usein käytetyille vuorottaisrutiinityypeille, esimerkiksi useita arvoja palauttava *generator* ja kertaluontoinen *task*. Näiden itse tekeminen on työlästä ja vaatii perehtyneisyyttä liittyvään teoriaan.

Jatkotutkimusta vaatii lukottomat ja odotuksettomat samanaikaiset tietorakenteet, jotka ovat tilanteesta riippuen oleellisia suurimman mahdollisen samanaikaisuuden saavuttamisessa, sekä vuorottaisrutiinit, joiden yksityiskohtainen läpikäynti ei ollut mahdollista työn laajuus huomioon ottaen. Näihin liittyy oleellisesti C++:n atomisten operaatioiden kirjaston tarkastelu, mikä jäi tekstissä vähään.

Lähdeluettelo

[1] Belson, B., Holdsworth, J., Xiang, W., & Philippa, B. A Survey of Asynchronous Programming Using Coroutines in the Internet of Things and Embedded Systems. *ACM Transactions on Embedded Computing Systems*, *18*(3), 1–21. (2019). https://doi.org/10.1145/3319618

[2] Belson, B., Xiang, W., Holdsworth, J., & Philippa, B. C++20 Coroutines on Microcontrollers -What We Learned. *IEEE Embedded Systems Letters*, 1–1. (2020). https://doi.org/10.1109/LES.2020.2973397

[3] Chen, R. C++ coroutines: The mental model for coroutine promises. *The Old New Thing*. (2021a). *https://devblogs.microsoft.com/oldnewthing/20210329-00/?p=105015* (Haettu 23.4.2021)

[4] Chen, R. C++ coroutines: Cold-start coroutines. *The Old New Thing*. (2021b). *https://devblogs.microsoft.com/oldnewthing/20210421-00/?p=105135* (Haettu 27.4.2021)

[5] Drocco, M., Castellana, V. G., & Minutoli, M. Practical Distributed Programming in C++. *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, 35–39. (2020). https://doi.org/10.1145/3369583.3392680

[6] Duffy, J. *Concurrent programming on Windows* (1st edition). Addison Wesley. (2008).

[7] Gentleman, W. M. Concurrency Paradigms: Competitive, Coordinated, and Collaborative: Which Control Mechanisms are Appropriate? *International Journal of Parallel Programming*, *44*(2), 325–336. (2016). https://doi.org/10.1007/s10766-015-0370-9

[8] *ISO/IEC 14882:2020*. ISO; ISO/IEC JTC 1/SC 22. (2020). https://www.iso.org/standard/79358.html

[9] Jonathan, C., Minhas, U. F., Hunter, J., Levandoski, J., & Nishanov, G. Exploiting coroutines to attack the ”killer nanoseconds”. *Proceedings of the VLDB Endowment*, *11*(11), 1702–1714. (2018). https://doi.org/10.14778/3236187.3236216

[10] *Memory model—Cppreference.com*. (2021). *https://en.cppreference.com/w/cpp/language/memory_model* (Haettu 5.3.2021)

[11] Moura, A. L. D., & Ierusalimschy, R. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems*, *31*(2), 6:1-6:31. (2009). https://doi.org/10.1145/1462166.1462167

[12] Psaropoulos, G., Oukid, I., Legler, T., May, N., & Ailamaki, A. Bridging the Latency Gap between NVM and DRAM for Latency-bound Operations.

Proceedings of the 15th International Workshop on Data Management on New Hardware - DaMoN'19, 1–8. (2019). https://doi.org/10.1145/3329785.3329917

[13] Smith, R., & Nishanov, G. *Halo: Coroutine Heap Allocation eLision Optimization: The joint response*. (2018). *http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0981r0.html* (Haettu 4.4.2021)

[14] Stroustrup, B. *The C++ Programming Language, Fourth Edition*. Addison-Wesley Professional. (2013).

[15] Strömbäck, F., Mannila, L., Asplund, M., & Kamkar, M. A Student's View of Concurrency—A Study of Common Mistakes in Introductory Courses on Concurrency. *Proceedings of the 2019 ACM Conference on International Computing Education Research*, 229–237. (2019). https://doi.org/10.1145/3291279.3339415

[16] V'yukova, N. I., Galatenko, V. A., & Samborskii, S. V. Support for Parallel and Concurrent Programming in C++. *Programming and Computer Software*, *44*(1), 35–42. (2018). https://doi.org/10.1134/S0361768818010073

[17] Weber, D., & Fischer, J. Process-Based Simulation with Stackless Coroutines. *Proceedings of the 12th System Analysis and Modelling Conference*, 84–93. (2020). https://doi.org/10.1145/3419804.3421450

[18] Williams, A. *C++ Concurrency in Action: Practical Multithreading* (1st edition). Manning Publications Company. (2012).