Tampereen yliopisto

Aleksi Hietamäki

RINNAKKAISEN C++-OHJELMOINNIN MODERNIT TYÖKALUT

Kandidaatintyö
Informaatioteknologian ja viestinnän tiedekunta
Kesäkuu 2023

TIIVISTELMÄ

Aleksi Hietamäki: RINNAKKAISEN C++-OHJELMOINNIN MODERNIT TYÖKALUT
Kandidaatintyö
Tampereen yliopisto
Informaatioteknologian ja viestinnän tiedekunta
Kesäkuu 2023

Moniydinprosessoreiden yleistyessä tarvitaan tukea ohjelmointikieliltä, jotta niistä saadaan kaikki hyöty irti. Rinnakkainen ohjelmointi on perinteisesti toteutettu käyttöjärjestelmän järjestelmäkutsuilla, mutta kehitykset ohjelmointikielissä ovat tarjonneet standardoituja keinoja kirjoittaa rinnakkain suoritettavia ohjelmia käyttöjärjestelmästä riippumatta. Näihin ohjelmointikieliin lukeutuu C++. Se on tarjonnut rinnakkaisen ohjelmoinnin tukikirjaston standardiversiosta 11 alkaen. Standardi on kuitenkin jatkuvassa kehityksessä ja se on kokenut kolme päivitystä version 11 jälkeen.

Tämän työn tavoitteena on selvittää, kuinka C++-standardi on kehittynyt rinnakkaisen ohjelmoinnin näkökulmasta. Työ toteutettiin kirjallisuuskatsauksena. Työn alussa esitellään, kuinka rinnakkaisen ohjelmoinnin tukikirjasto on kehittynyt. Erityisenä tarkastelun kohteena ovat lisäykset ja muutokset sen alikirjastoissa. Mielenkiintoisimpana lisäyksenä havaitaan corutiinit, jotka ovat funktioita, joiden suoritus voidaan keskeyttää ja tarvittaessa käynnistää uudelleen myöhemmin. Työn loppupuolella selvitetään, mitä eroa on corutiineilla ja säikeillä ohjelman suorituksen ja kirjoittamisen kannalta. Tähän liittyen esitellään tutkimuksia, jotka vertailevat ohjelmien suoritusnopeutta ja itse ohjelmoinnin tehokkuutta.

Vertailevista tutkimuksista havaitaan, että corutiinit tarjoavat jopa kymmenkertaista suoritustehon kasvua, kun verrataan kontekstinvaihdon nopeutta. Corutiineja on testattu myös mahdollisuutena korvata lukot tietokantapalvelimilla, jolloin havaitaan noin neljänkertainen suoritustehon kasvu. Lopuksi corutiineja verrattiin tiedon esihaussa eri tietorakenteilla ja niiden käytön suurin tehohyöty oli noin yhdeksän prosenttia. Ohjelmakoodin kirjoittamisen kannalta corutiinit tarjoavat intuitiivisen keinon suorittaa synkronointia. Se tarjoaa varattuja avainsanoja, joita käyttäen voidaan kyseisen funktion suoritus väliaikaisesti keskeyttää. Ohjelmoijan tehtäväksi jää määrittää piste, jossa ohjelman suoritus palautuu kyseiseen funktioon. Tämä eroaa säikeistä siten, että säikeet avaavat uuden suorituksen ja pääsäie voi tarvittaessa pysähtyä odottamaan suorituksen valmistumista. Corutiinit eivät siis tarvitse odottelua, koska ohjelman suoritus siirtyy eri funktion vastuulle.

Avainsanat: C++, rinnakkaisuus, corutiinit.

SISÄLLYSLUETTELO

1. JOHDANTO

Moniydinprosessoreiden yleistyessä tarvitaan tukea myös rinnakkaiselle ohjelmoinnille. Perinteisesti C++-kieli on tarjonnut matalantason hallintaa laitteistolle ja rinnakkaisten ominaisuuksien hyödyntäminen ei ole poikkeus. Versiossa 11 standardoitu rinnakkaisuuden tukikirjasto on tuonut ohjelmoijalle ensimmäistä kertaa mahdollisuuden kirjoittaa rinnakkaista C++-ohjelmakoodia ilman käyttöjärjestelmän järjestelmäkutsuja. Se mahdollistettiin lisäämällä säiekirjasto, jossa uusi säie voidaan avata rinnakkain pääsäikeen kanssa ja liittämällä siihen suoritettava funktio. Pääsäie voi jatkaa omaa suoritustaan tai tietyssä pisteessä jäädä odottamaan tämän säikeen paluuta (ISO, 2020). Standardi on kuitenkin jatkanut kehitystään lisäten uusia ominaisuuksia kirjoittaa rinnakkaista koodia ja se tarjoaa parannuksia jo olemassa oleville keinoille. Tästä esimerkkinä uusia rajapintoja lukoille, uusi rajapinta säikeille sekä corutiinit (eng. coroutine). Lukkojen hallinta on kehittynyt tarjoamaan keinoja purkamaan ohjelman virhetilaa ajastetuilla lukoilla. Myös useiden lukkojen hallinta kerralla on lisätty mahdollisuus. Uusi säie-rajapinta tarjoaa mahdollisuuden keskeyttää säikeen toiminnan ennenaikaisesti ja palauttamaan senhetkisen tilan. Toisaalta corutiinit ovat uusi keino toteuttaa rinnakkaisuutta C++:ssa. Ne mahdollistavat funktiosta ennenaikaisen paluun, mutta funktio suorituksen tila tallennetaan ja siihen voidaan palata myöhemmin (Stroustrup, 2020).

Tämän tutkimukset tavoite on selvittää C++-ohjelmoinnin uusimpia ominaisuuksia, jotka ovat lisätty standardiversiosta 11 eteenpäin ja tarkastella, minkälaista arvoa ne tuovat ohjelmakoodin kirjoittamisen ja ohjelman suoritustehokkuuden kannalta. Työn tuloksena saadaan selville, että uudet rajapinnat säikeillä auttavat ohjelmoijaa merkittävästi. Myös corutiinit tarjoavat ohjelmoijalle tehokkaan vaihtoehdon rinnakkaisiin tehtäviin, joissa ohjelman täytyy odottaa joidenkin toisten tehtävien valmistumista

Tutkimus suoritettiin kirjallisuuskatsauksena. Siinä ensin tutustuttiin artikkeleihin ja kirjoihin, joissa esiteltiin eri versioiden uusimpia pääominaisuuksia. Tämän jälkeen etsittiin tarkempaa toteutusta C++-standardin puolelta. Työ kuitenkin painottuu corutiineihin, koska ne koettiin uusiksi ja mielenkiintoiseksi. Työn lopulla etsitään vertaisarvioituja tutkimuksia, joissa vertaillaan corutiineja sekä säikeitä. Vertailtevien tutkimusten valintakriteerinä oli se, että siinä käytettiin C++:aa ja vertailu kohdistuu säikeisiin ja corutiineihin.

Tämä tutkimus etenee seuraavasti. Aluksi luvussa 2 esitellään standardin kehitystä versiosta 11 alkaen ja sen tarkasteluväli päättyy versioon 20. Kustakin versiosta nostetaan

pääominaisuuksia ja pohditaan millaista arvoa ne tuovat ohjelmoijalle. Työ jatkuu luvussa 3 corutiineihin. Siinä esitellään yleisesti mitä ne ovat sekä tarkemmin C++-kielen toteutuksen erikoisuuksia. Luvun lopussa myös esitellään mahdollisia lisäyksiä, jotka voidaan nähdä tulevaisuudessa. Työ etenee luvussa 4 säikeiden ja corutiinien vertailulla. Vertailudataa on koottu vertaisarvioiduista artikkeleista, joissa on tehty mittauksia pääasiassa suoritustehon kannalta. Kriteerit vertailulle ovat suoritustehokkuus sekä koodin kirjoituksen tehokkuus. Työn viimeisessä luvussa 5 muodostetaan yhteenveto tutkimustuloksista ja annetaan jatkotutkimus ehdotuksia.

2. MONISÄIKEINEN OHJELMOINTI

Rinnakkainen ohjelmointi yleistyi C++:n versiosta 11 alkaen, kun standardoitiin rinnakkaisen ohjelmoinnin tukikirjasto (ISO, 2020). Se mahdollisti monisäikeisen ohjelmoinnin ilman ulkopuolisia kirjastoja tai hyödyntämällä käyttöjärjestelmien tarjoamia järjestelmäkutsuja. Kuitenkin standardi on kokenut useita päivityksiä, jotka ovat helpottaneet monisäikeistä ohjelmointia. Näihin ominaisuuksiin kuuluu uusi rajapinta säikeiden luontiin, sekä useita synkronointiin liittyviä luokkia. Tässä luvussa tutustutaan ensin C++ 11:n tarjoamiin ominaisuuksiin, minkä jälkeen esitellään uudet pääominaisuudet eri versioista ja pohditaan niiden tuomaa arvoa ohjelman kirjoittamisen kannalta. Viimeisessä kappaleessa esitetään pieni säikeinen ohjelma, jota verrataan myöhemmin corutiineilla tehtyyn toteutukseen. Luvun tavoitteena on tutustuttaa lukija säikeistenohjelmien mahdollisuuksiin, jotta niitä voidaan verrata corutiineihin jatkossa.

2.1 Säikeisten ohjelmien alku

Versio 11 toi ohjelmoijalle mahdollisuuden kirjoittaa säikeisiä ohjelmia suoraan standardikirjaston avulla (Stroustrup, 2020). Tämän tehdäkseen ohjelmoijan täytyy tuoda thread-kirjasto osaksi omaa ohjelmaansa (Andrist ym., 2020). Sen avulla voidaan kutsua uusia funktiota, jotka suoritetaan rinnakkain pääsäikeen kanssa. Säikeet voivat suorittaa vaihtoehtoisesti ohjelmaa riippumatta toisistaan. Kutsuva säie voi pysähtyä odottamaan kutsutun säikeen paluuta (ISO, 2020). Rinnakkaisuuden haasteet kuitenkin ilmenevät, kun säikeiden suoritukset riippuvat toisistaan. Yleisin ongelma on kilpailutilanne (eng. Race condition), jossa kaksi tai useampi säiettä yrittää päästä käsiksi johonkin tietorakenteeseen yhtä aikaa (Andrist ym., 2020). Esimerkiksi pankkisovelluksessa voi olla maksukäsittelijöitä, jotka hakevat jonosta aina yhden maksutapauksen. Jos kaksi säiettä ottaa saman tapauksen käsittelyyn, maksajan rahat saattavat lähteä kaksinkertaisena. Tätä ongelmaa ratkotaan standardissa lukoilla (Andrist ym., 2020). Lukkojen toiminta perustuu siihen, että ohjelmakoodiin asetetaan lukitsemispisteitä, jotka ovat oletuksena aukinaisia. Kun lukitsemispiste on auki, säie sulkee sen ja estää muita säikeitä jatkamasta, kunnes lukitsija vapauttaa lukon. Tämän jälkeen seuraava odottava säie ottaa lukon haltuunsa ja estää muita jatkamasta. Lukkojen rajapintaa kutsutaan yleisesti mutexiksi (eng. mutual exclusion).

Lukot ovat yleisin keino toteuttaa säikeiden synkronointi, kuten selviää tutkimuksesta ”An empirical study on C++ concurrency constructs” (Wu ym., 2015). Tässä tutkimuksessa

selvitettiin, mitkä rinnakkaisuuskäytännöt ovat yleisimmässä käytössä ja kuinka synkronointi toteutetaan eri säikeiden välillä. Tutkimukseen valittiin useita avoimen lähdekoodin projekteja. Heidän havaintojensa mukaan kaikista rinnakkaisen ohjelmoinnin käytänteistä monisäikeisyys ja niiden hallitseminen lukoilla on selkeästi yleisintä (62,58 %). Tästä voidaan olettaa, että merkittävimmät kehitykset ohjelmointikieleen tulevat helpottamaan ohjelmoijan arkea tarjoamalla kattavampia rajapintoja pääasiassa näiden rakenteiden osalta.

2.2 Synkronoinnin kehitys versioissa 14 ja 17

Versioiden 14 ja 17 ominaisuuksien parannukset eivät keskittyneet rinnakkaisuuteen, mutta silti tehtiin pieniä lisäyksiä lukkorajapintoihin (Stroustrup, 2020). Päivitykset suuntautuivat erityisesti suurien ohjelmien hallintaan, jossa lukkojen ja säikeiden määrä kasvaa paljon. Aiemmissa versioissa useiden mutexien hallinta on vaikeaa, sillä ohjelmoijan täytyy lukita jokainen lukko yksi kerrallaan. Ratkaisuna tähän esitettiin scoped lock -rajapinta (Stroustrup, 2020). Sen tehtävä on ottaa useita mutexeja syötteenä ja se lukitsee ne kaikki kerralla (ISO, 2020). Tämän toimintamallin edut ohjelmankirjoittamisen kannalta ovat selkeät: vähemmän toistuvia rivejä ja se myös välttää kontekstinvaihdon mahdollistaman virheen. Säie, joka on lukinnut yhden lukoista, voi kontekstinvaihdossa menettää pääsyn seuraavaan lukkoon, jos tämä uusi säie varaa sen. Tästä aiheutuisi umpilukko, joka käytännössä pysäyttäisi koko ohjelman toiminnan (Andrist ym., 2020).

Jaetun omistuksen mutexit ilmestyivät ensimmäistä kertaa versiossa 14 (Stroustrup, 2020). Niiden tarkoitus on tehdä tietorakenteita käyttävien säikeiden toiminnasta tehokkaampaa. Jaetun mutexin toiminta perustuu siihen, että sillä on kolme sisäistä tilaa. Lukitsematon, jaettu lukko ja uniikki lukko. Jaetun lukon tilassa monta säiettä pääsee tietorakenteeseen käsiksi yhtä aikaa, mikä on hyödyllistä tietoa lukiessa. Kun luettavan tietorakenteen tila täytyy päivittää, se varataan uniikilla lukolla, joka estää jaettujen säikeiden toiminnan (ISO, 2020). Version 14 jaettu mutexi oli ajastettu (Stroustrup, 2020). Se tarkoittaa sitä, että lukko voidaan varata vain tietyksi aikaa. Jos ohjelma ajautuisi umpilukkoon ajastimen lauettua lukko vapautuisi ja purkaisi virhetilan. Versio 17 lisäsi vielä ajastamattoman toteutuksen tästä ominaisuudesta (Stroustrup, 2020).

2.3 C++20 synkronointia algoritmeille

Aiemmat versiot standardista keskittyivät pääasiassa lisäämään perustoimintoja rinnakkaiseen ohjelmointiin. Kuitenkin versiossa 20 nähtiin valmiita rajapintatoteutuksia useiden säikeiden synkronoinnille. Merkittävimmät lisäykset olivat semafori, seinämä (eng. barrier) ja JThread-rajapinnat (eng. Joinable thread) (Stroustrup, 2020).

Semafori on tietotyyppi, joka pitää kirjaa saatavilla olevista resursseista ja vapauttaa odottavia säikeitä, kun resursseja on saatavilla. Rajapinnan tarjoama ominaisuus ei ideana ole uusi. Sitä on käytetty usein ratkomaan tuottaja-kuluttajaongelma, jossa tuottaja kerää dataa yhteiseen tietorakenteeseen ja kuluttaja ottaa sitä käsittelyyn. Uusi rajapinta kuitenkin vähentää mahdollisuutta ohjelmoijan itse tekemille virheille.

Seinämä-rajapinta on pääasiassa algoritmeille suunniteltu synkronointikeino. Ajatellaan listan järjestelyalgoritmia, joka toteutetaan rekursiivisesti hajota ja hallitse -menetelmällä käyttäen säikeitä. Niiden yhtäaikainen syvyyden lisäys voidaan mahdollistaa seinämällä. Sen toiminta on samankaltainen kuin mutexin, mutta erona on, että se pitää kirjaa siitä, kuinka monta säiettä odottaa kyseisellä lukolla ja avautuu, kun ennalta määrätty luku on saavutettu. (Andrist ym., 2020)

JThread-rajapinta tarjoaa ohjelmoijalle keinon keskeyttää säikeen toiminta odottamatta sen paluuta. Omistajasäie voi lähettää keskeytyspyynnön, jolloin JThread kutsuu säikeen purkajaa ja palauttaa omistajasäikeelle senhetkisen tilan (ISO, 2020).

2.4 Ohjelmaesimerkki

Tässä kappaleessa esitetään pieni säikeitä hyödyntävä ohjelma, jotta sitä voitaisiin verrata myöhemmin samankaltaiseen corutiineilla toteutettuun versioon. Se on muokattu versio Andristin ym. corutiineilla tehtyyn versioon, joka esitellään kappaleessa 3.2.

```
auto height() -> int { return 20; }
auto width() -> int { return 30; }
auto area() -> int {
    auto retHeight = async(&height);
    auto retWidth = async(&width);

    return retHeight.get() * retWidth.get();
}

int main() {
    future<int> ret = async(&area);
    int value = ret.get();
    cout << value;
}
```

Ohjelma 1 Yksinkertainen säikeinen ohjelma (Andrist ym., 2020)

Ohjelmassa alustetaan futuuriobjekteja. Futuuriobjektien tyyppimäärittely on kätketty auto-avainsanan taakse. Auto-avainsana on C++-kielen keino suorittaa implisiittinen tyyppimääritys. Pääohjelman suoritus voidaan tilapäisesti keskeyttää ja odottaa futuurin arvon valmistumista **`.get()`** metodilla. C++:n **`async()`** funktio on määritetty käynnistämään itse uusia säikeitä, joten emme tässä ohjelmassa tarvitse synkronointia. Luvussa 4 palataan tarkemmin ohjelman eroavaisuuksiin.

3. KATSAUS CORUTIINEIHIN

Corutiinit ovat Melvin Conwayn vuonna 1963 ideoima ohjelmansuoritusmalli. Se kehitettiin kilpailijaksi proseduraaliselle ohjelmoinnille, jossa funktio-ohjelman suoritus jaetaan funktioihin ja ne suoritetaan aina hierarkkisesti alusta loppuun. Corutiinit kuitenkin eroavat tästä siten, että suorituksesta poistetaan tämä hierarkkisuus ja funktiot voivat siirtää kontrollia toisilleen, kun sitä tarvitsevat (Conway, 1963). Tässä luvussa ensin esitetään kuinka corutiinit toimivat C++:ssa ja annetaan esimerkki, kuinka tiedostoja voi lukea Boost-kirjastolla rinnakkain käyttäen corutiineja. Luvun toisessa kappaleessa katsotaan, mitä lisäyksiä corutiinien standardikirjastoon on ehdotettu ja minkälainen vaikutus niillä voi olla.

3.1 Corutiinit C++:ssa

Corutiinit saapuivat osaksi C++:aa sen standardiversiossa 20 (Stroustrup, 2020). Sen mukana tulivat varatut avainsanat **`co_await`**, **`co_yield`** ja **`co_return`**. Corutiinien toiminta perustuu siihen, että corutiinifunktio voi ennenaikaisesti palauttaa arvon, jonka jälkeen sen suoritus pysäytetään ja tallennetaan myöhempää käyttöä varten. C++ toteutuksessa corutiineilla ei ole pinomuistia vaan niiden koko elinkaari suoritetaan kasamuistissa (ISO, 2020).

Corutiineja käyttääkseen kutsuja käyttää avainsanaa **`co_await`**, joka palauttaa tälle futuurin tyypin. Corutiini aloittaa futuurin toteuttamisen ja voi ennenaikaisesti palauttaa arvon **`co_yield`** operaatiolla, jolloin myös ohjelman suoritus palautuu kutsujalle. Kutsuja funktio voi tarvittaessa jatkaa corutiinin suoritusta kutsumalla sitä **`.resume()`** metodilla. Kun corutiinin suoritus palautuu **`co_return`** avainsanalla, sen elinkaari päättyy ja sen muisti vapautetaan (ISO, 2020).

Rinnakkaisuuden kannalta hyödyllisin ominaisuus on **`co_await`**. Sitä käytetään funktiokutsun yhteydessä muodossa **`co_await funktio().`** Funktion täytyy palauttaa Corutiinirajapinnan mukainen awaitable-tyyppi. Sen täytyy sisältää kolme metodia **`await_ready()`**, **`await_suspend(coroutine_handle)`** ja **`await_resume()`**. Tutustutaan seuraavaksi mitä ne ovat.

1. Kutsu **`await_ready()`** palauttaa totuusarvon, joka kertoo odottavalle funktiolle, että **`funktio()`** on valmis palauttamaan arvon.
2. Kutsu **`await_suspend(coroutine_handle)`** on viite corutiinin futuuri-olioon. Se mahdollistaa esim. viestimisen kyseiselle funktiolle siitä, että tehtävä on suoritettu.

3. metodi **`await_resume()`** on funktio, joka palauttaa lopullisen arvon corutiinilta, kun suoritus palautuu takaisin odottajalle.

Standardi ei vielä sisällä valmiita toteutuksia odottaville funktioille. Tämän työn skaalan vuoksi havainnostamme corutiinien hyödyllisyyttä käyttäen Boost.Asio kirjastoa, jossa on valmiita toteutuksia geneerisille rinnakkaisille funktioille, jotka toteuttavat yllä esitellyt rajapinnat (*Boost - 1.81.0*, 2022). Tarkastellaan GPT-4 kielimallin generoimaa esimerkkiä tiedostojen lukemisesta käyttäen Boost-kirjastoa ja corutiineja.

```
   constexpr size_t buffer_size = 4096;
   awaitable<std::string> read_file(const std::string& file_path) {
      asio::io_context io_context;
      asio::basic_file<asio::io_context::executor_type> file(io_con-
text.get_executor());
      co_await file.async_open(file_path,
asio::basic_file<>::open_mode::read, use_awaitable);

      std::string result;
      std::vector<char> buffer(buffer_size);
   std::error_code ec;
      while (!ec) {
         std::size_t bytes_read = co_await
file.async_read_some(asio::buffer(buffer), use_awaitable[ec]);
   if (bytes_read > 0) {
      result.append(buffer.begin(), buffer.begin() + bytes_read);
      }
   }
   if(ec != asio::error::eof){
      throw std::system_error(ec)
    }

   co_return result;
}

   int main() {
   std::string file_path = "example.txt";
   std::cout << "Reading file: " << file_path << '\n';
   auto file_data = asio::co_spawn(asio::make_strand(asio::io_con-
text()), read_file(file_path), asio::detached).get();
   std::cout << "File contents:\n" << file_data << '\n';
   return 0;
   }
```

Ohjelma 2 Corutiinien rinnakkaisuus Boost.Asio kirjastolla (GPT-4, 2023)

Ohjelmakoodista on poistettu direktiivit kirjastojen lisäykselle, jotta esimerkki mahtuisi yhdelle sivulle. Funktiossa `main()` luodaan rinnakkainen ajoympäristö `co_spawn()` funktiolla, jossa `read_file()` corutiini suoritetaan. Kutsu `read_file()` itsessään palauttaa yllä esitellyn awaitable-tyypin, mikä määrittää sen corutiiniksi. Ajoympäristö kutsuu implisiittisesti kyseisen awaitablen `resume` funktiota, joka jatkaa `file_read()` funktion suorittamista. Rivillä 12 parametrinä annettu tiedosto luetaan puskurin mittaisissa paloissa. Tällä rivillä `file_read()` funktion suoritus pysäytettäisiin ja ohjaus siirtyisi takaisin `main`-funktiolle, josta sen suoritusta voitaisiin jatkaa kutsumalla `file_data.resume()`. Mutta kirjaston tarjoaman ajoympäristön vuoksi funktiot suoritetaan rinnakkain ja ajoympäristön `.get()` funktio palauttaa tiedoston sisällön.

3.2 Corutiinit tulevaisuudessa

Standarditasolla corutiinit tarjoavat rajapintoja laiskasti evaluoituvien sekvenssien luontiin, kuten C++ 23:ssa saapuva generaattori. Se on keino luoda rinnakkaisesti iteroitavia funktioita. Generaattorissa luotu funktio palauttaa aina yhden arvon, jonka jälkeen ohjelman suoritus palautuu kutsujalle, joka voi vaihtoehtoisesti välittömästi palata generaattoriin jos on tarve iteroida useita arvoja (Andrist ym., 2020). Rinnakkaisuuden puolelle on kuitenkin esitetty geneeristä odotettavaa tehtäväluokkaa nimeltä `std::task<T>`. Sen tehtävä on toteuttaa awaitable-luokka, jotta funktiokutsuja voisi luoda ja pysähtyä odottamaan rinnakkaisesti niiden valmistumista. Corutiinin arvo täytyy evaluoida lopulta synkronisesti, koska pääfunktio palautuisi ennenaikaisesti ja ohjelman suoritus päättyisi. Baker (2018) esitti mahdolliseksi ratkaisuksi `sync_await(task<T>)` funktiota, joka suorittaa corutiinin loppuun. Sen käytännön toiminta vastaa säikeiden liittämistä takaisin pääsäikeeseen, jossa pääsäie odottaa toisen säikeen paluuta.

Seuraavaksi esitellään miltä näyttää ohjelma, joka käyttää edellä mainittua **`std::task<T>`** tyyppiä ja synkronista odotusta. Sen toiminta on identtinen ohjelmaan 1 verrattuna.

```
auto height() -> Task<int> { co_return 20; }
auto width() -> Task<int> { co_return 30; }
auto area() -> Task<int> {
  co_return co_await height() * co_await width();
}

int main() {
  auto a = area();
  int value = sync_await(a);
  cout << value;          // Outputs: 600
}
```

Ohjelma 3 Corutiinien Task ja sync_await (Andrist ym., 2020)

Ohjelma 3 havainnollistaa sitä, miltä rinnakkainen ohjelmointi corutiineilla voisi näyttää tulevaisuudessa. Kutsuttaessa **`area()`**-funktiota **`a`**-muuttuja asetetaan **`Task<int>`** tyyppiseksi auto-avainsanalla, joka määrittää tyypin implisiittisesti. Funktio **`sync_await(a)`** estää **`main()`**-funktiosuorituksen jatkumisen ennen kuin **`a`**:n arvo on evaluoitu.

4. CORUTIINIEN JA SÄIKEIDEN VERTAILUA

Corutiinien ollessa uusi tapa toteuttaa rinnakkaisuutta C++-ohjelmien kannalta on syytä tarkastella, mitkä ovat sen vahvuudet ja heikkoudet. Tätä on tutkittu eri käyttökonteksteissa. Tutkimukset, joita löydettiin ovat keskittyneet ratkaisemaan reaalimaailman ongelman käyttäen corutiineja ja sitä toteutusta on verrattu vastaavanlaiseen säikeiseen ratkaisuun. Tässä kappaleessa tutustutaan ensin, minkälaisissa tapauksissa tutkimustyötä on tehty. Tämän jälkeen taulukoidaan tutkimustuloksia ja tehdään johtopäätöksiä niiden perusteella. Viimeisessä kappaleessa vertaillaan ohjelmia 1 ja 3 pohditaan, mitä eroja niillä on.

4.1 Tutkimustöiden esittely

Corutiineja ovat tutkineet Benson ym. (2021) sulautettujen järjestelmien näkökulmasta. Empiirinen tutkimus on keskittynyt vertailemaan kontekstinvaihdon nopeutta verrattuna säikeisiin, lopullisen ohjelman vaatiman muistin määrää sekä koodin kirjoittamisen tehokkuutta. Tutkimuksessa kirjoitettiin ohjelma, joka vaihtoi aktiivista funktiota. Yksi funktio luki GPIO-pinnistä dataa ja toinen funktio ajoi ääretöntä silmukkaa. Corutiineilla yksi kontekstinvaihto kestää 0,209 µs. Vertailukohteena oli kaksi käyttöjärjestelmää, jotka ajastivat säikeiden kontekstinvaihdon itse. FreeRTOS käyttöjärjestelmän ajastamat säikeet vaativat vaativat 2,504 µs ja MQX Lite käyttöjärjestelmän vastaava säikeiden ajastus 2,589 µs. Muistinkäytöstä havaittiin, että corutiinien käytöllä ei ole havaittavaa eroa säikeiseen toteutukseen verrattuna. Ohjelmakoodin kirjoittamisen kannalta tutkimus keskittyi analysoimaan ohjelmoijan subjektiivista mielipidettä, sekä ohjelman koodirivien määrää. Säikeinen ohjelma kirjoitettiin 363 rivillä ilman ulkopuolisia kirjastoja ja corutiineilla tehty versio saatiin kirjoitettua 60 rivillä. Kuitenkin on hyvä huomauttaa, että tässä käytettiin kirjastoja toteuttamaan rinnakkaisuus, joka kasvattaa ohjelman yhteisrivimäärän 256 riviin. Benson ym. (2021) tutkimuksesta on hyvä huomioida, että vertailussa oli mukana myös protothread-niminen rinnakkaistamiskeino, joka suljettiin tämän tutkimuksen ulkopuolelle. Tutkimus oli myös hieman suppea sisältäen vain yhden kokeen.

Corutiineja on tutkinut Jonathan ym. (2018) tietorakenteiden näkökulmasta. Tutkimuksessa dataa esihaettiin neljästä eri tietorakenteesta: hajautustaulukosta, binäärihakupuusta, massapuusta sekä BW-puusta. Vertailussa kohteena olivat esihauton, corutiinit sekä kaksi säikeistä tekniikkaa käyttävät tiedon esihakumenetelmät GP ja AMAC. Tutkimuksessa keskityttiin tehokkuuden skaalautuvuuteen eri datamäärillä ja eri säikeiden

määrillä. Tutkimustuloksissa havaittiin, että pienillä dataseteillä ohjelman suoritus hidastuu noin 100 - 300 % verrattuna esihauttomaan toteutukseen. Tämä selitettiin sillä, että koko tietorakenne mahtui prosessorin L3-muistiin ja esihaun aiheuttamat keskeytykset vain luovat ylimääräisiä operaatioita. Pienillä dataseteillä corutiinit ovat myös suoriutuneet heikoiten verrattuna muihin esihakumenetelmiin: noin 50 % hitaampia. Suurilla datamäärillä esihaulliset menetelmät menevät merkittävästi esihauttomasta ohi 2-8 kertaisesti riippuen tietorakenteesta. Suurilla dataseteillä corutiinit yltävät lähes samalle tasolle nopeimpien säikeisten menetelmien kanssa. Ne ovat noin 0-25% riippuen datasta. Säikeiden määrää kasvattaessa skaalautuvuus corutiinien ja säikeisten menetelmien välillä on identtinen. Jonathan ym. (2018) tutkimuksessa myös huomautettiin, että corutiinien käyttöönotto tietorakenteelle oli kaikkein helpointa. Tämä tutkimus oli suoritettu ennen kuin virallinen tuki corutiineille oli tullut kääntäjille ja tuloksissa oli havaittavissa merkittäviä eroja Clang, ja MSVC-kääntäjien välillä. On siis hyvä tulkita lukuja siten, että erityisesti corutiineille suoritettava optimointi on voinut kehittyä myöhemmin.

Corutiineja on tutkinut Zhu ym. (2018) korvikkeena lukkojen hallinnalle. SQL-tietokantojen kyselyissä palvelin palauttaa asiakkaalle hakemansa tiedon, mutta usean kyselyn tullessa samaan aikaan on tärkeä varmistaa, että tieto ei muutu toisen kyselyn suorituksen aikana. Säikeisellä ohjelmamallilla tämä voidaan yksinkertaisesti ratkaista tekemällä lukko kyselyiden ajaksi. Se ei kuitenkaan palvele käyttäjää tehokkaasti, sillä säikeet jäävät testaamaan lukon aukeamista, mikä vaatii prosessointitehoa. Corutiineilla tehtävä on ratkaistu siten, että yhtäaikaisen tietovarauksen havaitessa toinen corutiini palautuu ennenaikaisesti ja sen tilan voi palauttaa, kun konflikti on ratkaistu. Kyseisessä tutkimuksessa kehitettiin kolme rinnakkaista mallia suorittamaan kyselyitä: yksi corutiineilla, säikeillä ja lukkotaulukoilla. Niiden suorituskykyä mitattiin, kun kyselyitä tekevien asiakkaiden määrää kasvatetaan. Corutiinien ja säikeisten lukkojen suorituskyvyt olivat lähes identtiset pienillä käyttäjämäärillä, koska konfliktien todennäköisyys oli pieni. Kuitenkin suurilla käyttäjämäärillä lukollisten säikeiden suorituskyky romahtaa sillä yhä enemmän suoritusaikaa käytetään testaamaan lukon tilaa. Corutiineilla datan läpivienti suppeni vakaalle tasolle käyttäjämäärän kasvaessa. Rajoittavaksi tekijäksi havaittiin prosessorin suorituskyky. Tutkimuksesta Zhu ym. (2018) on kuitenkin havaittavissa naiivi lähestymiskulma tutkimuksessa esitetyn ongelman ratkaisuun. Corutiinihallintaan kehitetään hyvin kehittynyt menetelmä ongelman ratkaisuun ja säikeiselle toteutukselle tarjotaan alkeellista menetelmää.

4.2 Tutkimusdatan koonti

Tässä kappaleessa kootaan yllä esitellyistä tutkimuksista dataa analysointia varten. Tarkoituksena on havainnollistaa käyttökonteksteja, joissa corutiinit voivat korvata säikeisen ohjelmointitekniikan. Taulukkoon on kerätty tutkimuksista nostettuja tehollisia arvoja havainnollistamista varten.

Tutkimus	käyttökonteksti	Suurin havaittu teho-hyöty (%)	Pienin havaittu teho-hyöty (%)	skaalautuvuus
1	Mikrokontrollerin kontekstin vaihto	1198	1238	-
2	Datan esihaku tietojärjestelmästä	9	-18	Skaalautuu heikosti pienillä datamäärillä, mutta suurilla tehokas
3	Lukkojen hallinta	450	0	Nopeampi suurilla asiakasmäärillä, suppenee.

Taulukko 1 Havaitut tulokset

Tutkimustuloksista voidaan tehdä seuraavanlaisia johtopäätöksiä. Tutkituista tehtävistä ongelmat ovat pääasiassa keskittyneet käyttökontekstiltaan tehtäviin, joissa täytyy odottaa tiedon saatavuutta. Säikeisillä ohjelmilla ongelmaksi tulee tässä tapauksessa se, että monta säiettä odottaa samaa lukkoa tuhlaten resursseja. Tutkimuksissa yhteiseksi tekijäksi havaittiin, että corutiineilla funktion suorituksen tallentaminen järjestelmämuistiin voidaan suorittaa nopeammin, kuin sellaisen säikeen löytäminen, jonka suoritusta voidaan jatkaa, tehtävä on silloin parempi suorittaa corutiineilla. Tuloksista kuitenkin on havaittavissa suuriakin potentiaalisia käyttökohteita, kuten kontekstinvaihto mikrokontrollereilla ja lukkojen hallinta corutiineilla. Lähes jokaisessa tutkimuksessa kuitenkin todettiin, että ohjelmakoodin kirjoittaminen on helpompaa corutiineilla. Funktiokutsu, joka jää odottamaan, kunnes sen arvo täytyy evaluoida, koettiin intuitiiviseksi. Datan esihakuun liittyvässä tutkimuksessa on havaittavissa kaikkein pienimmät erot corutiinien ja säikeiden välillä. Tuloksia voi osittain selittää, sillä että tutkimuksessa käytettiin testausversiossa olevia kääntäjiä ja kääntäjien optimointien välillä oli havaittavissa eroja; mm MSVC-kääntäjä suoriutui heikommin kuin Clang.

4.3 Koodiesimerkkien vertailu.

Tässä luvussa tutustutaan tarkemmin eroavaisuuksiin säikeiden ja corutiinien välillä ohjelmakoodin kirjoittamisen kannalta. Tavoitteena on nostaa keskusteluun yleisiä eroavaisuuksia sekä vertailla aiemmissa luvuissa 2.4 ja 3.2 esiteltyjen ohjelmaesimerkkien eroja. Säikeisessä ohjelmassa ohjelmoija saa aina käyttöönsä pääsäikeen ohjelman käynnistyessä. Tästä pääsäikeestä voidaan kutsua uusia alisäikeitä, jotka alkavat suorittaa uutta funktiota. Kun pääsäie on saanut oman suorituksensa päätökseen tai se ei kykene jatkamaan, sen on aina eksplisiittisesti jäätävä odottamaan alisäikeen suorituksen päättymistä `.join()` metodilla. Corutiineilla alisäikeen vastine käynnistetään vasta kun sille on tarvetta. Tämän takia pääsäikeessä voidaan suoraan jäädä odottamaan tarvittavaan pisteeseen ja laskea tarvittava arvo.

Ohjelma 1 on säikeillä toteutettu rinnakkainen ohjelma, jossa säikeet alustetaan `std::async()` funktiolla. Kyseinen funktio käynnistää itse tarvittavat säikeet ja palauttaa futuurin funktion paluuarvolle, jonka valmistumista voidaan pysähtyä odottamaan `.get()` metodilla. Tässä tapauksessa sen vastaa säikeen `.join()` metodia. Ainoana erona on se, että futuuri on pohjimmiltaan muuttuja ja säie on funktio. Ohjelma 3 on corutiineilla tehty ohjelmaa 1 vastaava toteutus. Esimerkeistä nähdään aiemmassa kappaleessa esitetyt odotuspisteet. Ohjelman 1 odotuspisteet löytyvät riveiltä 8 ja 12 ja ohjelma 3 odotuspisteet näkyvät riveillä 5 ja 10. Corutiiniversiosta nähdään erityisesti se, että `co_await` avainsanalla tehty funktiokutsu on paljon helppolukuisempi kuin säikeillä. Ohjelman suoritus pysähtyy siihen pisteeseen, kunnes odotettava arvo on laskettu. Säikeiden ongelmaksi muodostuu futuuriolion käsittely, jonka käyttötarkoitus ei ole nopeasti ymmärrettävissä.

On hyvä kuitenkin huomata, että ohjelma 3 ei toimi vielä tänä päivänä suoraan standardikirjaston avuin vaan ohjelmoijan tehtäväksi jää odotettavan tehtävätyypin ja synkronisen odotusfunktion toteutus. Vaihtoehtoisesti voi käyttää esimerkiksi ohjelmassa 2 esiteltyä Boost-kirjastoa, joka tarjoaa täydentäviä corutiinitoteutuksia standardikirjaston puutteille.

5. KESKUSTELU JA YHTEENVETO

Tutkimuksessa selvitetään C++-standardikirjaston kehitystä alkaen versiosta 11 ja päättyen versioon 20. Tutkimuksen tavoitteena oli löytää pääominaisuudet, jotka tulivat osaksi standardia ja tutkia millaista arvoa ne tuovat ohjelmoijalle. Tutkimuksen tuloksiksi havaittiin corutiinit. Ne lisäävät mahdollisuuden kirjoittaa helppolukuista rinnakkaista koodia ilman säikeitä. Ne myös tarjoavat kilpailukykyistä suoritustehoa tehtävissä, jossa ohjelman täytyy pysähtyä odottamaan toisen funktion valmistumista. Suoritustehon vertailussa havaitaan lähes 12-kertaista tehokasvua ohjelmankontekstinvaihdossa, mutta maltillisempaa 9 % nousua suurimmillaan datan esihaun tehtävissä. Kirjoittaminen ja lukeminen koetaan myös paremmaksi corutiineilla niiden verboosien avainsanojen takia. Niitä käyttämällä myös vältytään synkronoinnin haasteilta, koska ohjelma ei jakaudu useisiin säikeisiin.

Vertailevia tutkimuksia corutiineille ja säikeille ei ole paljoa, mikä kertoo niiden olevan vielä kehitteillä. Kuten tässäkin työssä olevat corutiini esimerkit ovat tehty ulkoisilla kirjastoilla. Standardi kuitenkin kehittyy jatkuvasti ja näemme mahdollisesti lähitulevaisuudessa puutteiden paikkaavia lisäyksiä. Henkilökohtaisesti olen hieman pettynyt tutkimusten puutteeseen, mutta optimistisesti ajateltuna suunta on vain ylöspäin kehittyvän teknologian parissa. Tutkimuskysymykseen saatiin kuitenkin vastaus, joten tutkimus on siinä mielessä onnistunut.

Jatkotutkimusideana voisi tarkastella corutiinien kehitystä muissa kielissä, kuten Golang ja Rust (Go, 2023; Rust, 2023). Näissä kielissä on ollut jo pidemmän aikaa tuki corutiineille ja ne sisältävät monia C++:sta puuttuvia ominaisuuksia, kuten odotettavan tehtävätyypin ja synkronisen odottamisen. Niiden kehityksen tarkastelu voi antaa suuntaa sille, kuinka paljon C++:lla on varaa kasvaa, sekä ominaisuuksien -että suorituskyvyn puolesta.

LÄHTEET

Andrist B, Sehr V, Garney B. C++ High Performance - Second Edition 2. p. Vsk. 2020. Packt Publishing.

Baker L. Add coroutine task type P1056R0 [Internet]. 2018 [viitattu 21. maaliskuuta 2023]. Saatavissa: http://www7.open-std.org/JTC1/SC22/WG21/docs/papers/2018/p1056r0.html

Belson B, Xiang W, Holdsworth J, Philippa B. C++20 Coroutines on Microcontrollers—What We Learned. IEEE Embedded Systems Letters. maaliskuuta 2021;13(1):9–12.

Boost.Asio - 1.81.0 [Internet]. 2022 [viitattu 22. maaliskuuta 2023]. Saatavissa: https://www.boost.org/doc/libs/1_81_0/doc/html/boost_asio.html

Conway ME. Design of a separable transition-diagram compiler. Commun ACM. 1. heinäkuuta 1963;6(7):396–408.

ISO/IEC 14882:2020, Programming languages — C++. [viitattu 2. maaliskuuta 2023]. Saatavissa: https://www.iso.org/obp/ui/#iso:std:iso-iec:14882:ed-6:v1:en

Jonathan C, Minhas UF, Hunter J, Levandoski J, Nishanov G. Exploiting coroutines to attack the ”killer nanoseconds”. Proc VLDB Endow. 1. heinäkuuta 2018;11(11):1702–14.

OpenAI. GPT-4 [Large language model]. [viitattu 22. maaliskuuta 2023]. https://chat.openai.com/chat

Stroustrup B. Thriving in a crowded and changing world: C++ 2006–2020. Proc ACM Program Lang. 12. kesäkuuta 2020;4(HOPL):70:1-70:168.

A Tour of Go [Internet]. [viitattu 23. maaliskuuta 2023]. Saatavissa: https://go.dev/tour/concurrency/1

Wu D, Chen L, Zhou Y, Xu B. An Empirical Study on C++ Concurrency Constructs. Teoksessa: 2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). 2015. s. 1–10.

Zhu T, Wang D, Hu H, Qian W, Wang X, Zhou A. Interactive Transaction Processing for In-Memory Database System. Teoksessa: Pei J, Manolopoulos Y, Sadiq S, Li J, toimittajat. Database Systems for Advanced Applications. Cham: Springer International Publishing; 2018. s. 228–46. (Lecture Notes in Computer Science).

coroutine - Rust [Internet]. [viitattu 23. maaliskuuta 2023]. Saatavissa: https://docs.rs/coroutine/latest/coroutine/