Juha Rouvinen

Rinnakkaislaskentamallien arviointi: Intel Threading Building Blocks

Tietotekniikan kandidaatintutkielma

24. kesäkuuta 2013

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Juha Rouvinen

Yhteystiedot: juha.p.rouvinen@student.jyu.fi

Ohjaaja: Anneli Heimburger

Työn nimi: Rinnakkaislaskentamallien arviointi: Intel Threading Building Blocks

Title in English: Evaluation of parallel computation models: Intel Threading Building Blocks

Työ: Kandidaatintutkielma

Suuntautumisvaihtoehto: Ohjelmistotekniikka

Sivumäärä: 31

Tiivistelmä: Moniydinprosessoreiden ollessa jo normi on rinnakkaislaskennasta tullut arkipäivää yhä useammalle ohjelmoijalle. Rinnakkaislaskenta on hankalaa hahmottaa ja toteuttaa, joten tarvitaan uusia korkeamman abstraktiotason rinnakkaislaskentamalleja tukemaan rinnakkaislaskennan yleistymistä. Tässä tutkielmassa luomme mallin rinnakkaislaskentamallien arviointiin kahdeksasta eri näkökulmasta. Sen jälkeen tutustumme Intel Threading Building Blocks (TBB) -rinnakkaistuskirjastoon, joka lupaa tehdä rinnakkaislaskennan toteuttamisesta helpompaa siirtämällä rinnakkaistuksen perusmekanismeja pois ohjelmoijan vastuulta. Lopuksi arvioimme TBB:tä aiemmin luomamme mallin pohjalta. Toteamme, että TBB täyttää hyvin siirrettävyyteen, laajaan käyttökelpoisuuteen ja käytettävyyteen liittyvät vaatimukset. Suorituskyvyn osalta tulokset ovat kaksijakoisia.

Avainsanat: Intel Threading Building Blocks, Rinnakkaislaskenta, Rinnakkaislaskentamalli, Säie, Säikeistäminen

Abstract: With multi-core processors having become the norm, parallel computing has become commonplace for more and more programmers. Parallel computing is hard to understand and implement, so there is a need for parallel computation models that operate at a higher abstraction level. In this paper we create a model for evaluating parallel computation models from eight different aspects. After that we take a look at Intel Threading Building Blocks (TBB) parallelization library which promises to make the creation of parallel

programs easier by moving the basic mechanisms of parallelization away from the programmer's responsibility. Finally, we evaluate TBB based on the model that we created earlier. We find that TBB fulfills the portability, wide applicability and usability aspects particularly well. As for performance, the results are mixed.

Keywords: Intel Threading Building Blocks, Parallel computing, Parallel computation model, Thread, Threading

Sisältö

1 Johdanto

Moniydinprosessoreiden ollessa nykyään jo normi täytyy yhä useamman ohjelmoijan pystyä kirjoittamaan rinnakkaistettua ohjelmakoodia. Tehokkaan ja virheettömän rinnakkaisen ohjelman tuottaminen on kuitenkin kaikkea paitsi helppoa. Rinnakkaislaskennan yleistyessä täytyy sen toteuttamisesta tehdä helpompaa ja tehokkaampaa. Rinnakkaislaskentaa toteutetaan rinnakkaislaskentamallien avulla. Esimerkiksi käyttöjärjestelmien matalan tason säikeet, OpenMP-kääntäjädirektiivit ja MPI-viestinvälitysrajanpinta ovat erilaisia rinnakkaislaskentamalleja. Tässä tutkielmassa luodaan malli rinnakkaislaskentamallien tarkasteluun kahdeksasta eri näkökulmasta. Nämä kriteerit ovat: *(1) Helposti ymmärrettävä*, *(2) helposti ohjelmoitava*, *(3) käyttökelpoisuudeltaan laaja*, *(4) riippumaton arkkitehtuurista (siirrettävyys)*, *(5) taattu suorituskyky*, *(6) taattu oikeellisuus*, *(7) kustannuksiltaan arvioitava* ja *(8) tuetut työkalut*.

Tämän jälkeen tutustumme lupaavaan uuteen rinnakkaislaskentamalliin, Intel Threading Building Blocks (TBB) -rinnakkaistuskirjastoon. TBB pyrkii helpottamaan ja nopeuttamaan rinnakkaisten sovellusten kehittämistä piilottamalla ja automatisoimalla rinnakkaistuksen toteutuksen perusmekanismit, kuten säikeiden luomisen, hallinnan, kommunikoinnin ja synkronoinnin. Se pyrkii myös parantamaan rinnakkaisten ohjelmien suorituskykyä, skaalautuvuutta ja siirrettävyyttä. Tutustumme lyhyesti TBB:n perusteisiin ja tehtävävuorottajaan, sekä annamme lyhyen esimerkin kuinka sen rinnakkaistusalgoritmeja käytetään.

Lopuksi tarkastelemme TBB:tä aiemmin luomamme rinnakkaislaskentamallien tarkasteluun luodun mallin kautta. Tavoitteena on hahmottaa mitkä näistä ominaisuuksista ovat TBB:n vahvuuksia ja mitkä heikkouksia.

2 Rinnakkaislaskentamallit

Skillicorn et al. [8] mukaan rinnakkaislaskentamalli on rajapinta, joka erottaa korkean tason ominaisuudet matalan tason ominaisuuksista. Se tarjoaa ylemmälle tasolle tiettyjä operaatioita, jotka taas vaativat toteutuksen kaikille tuetuille alemman tason arkkitehtuureille. Esimerkiksi ohjelmointikieliä voidaan pitää malleina tämän määritelmän mukaan, koska ne tarjoavat korkean tason näkymän ohjelman toteuttamiseen, ja kääntäjät vastaavat ohjelmakoodin kääntämisestä laitteistolla toteutettavaksi, eli alemmalle tasolle. Rinnakkaislaskentamalleiksi katsotaan siis ne ohjelmointikielet ja tekniikat, joilla rinnakkaisesti ajettavia sovelluksia toteutetaan. Esimerkkejä rinnakkaislaskentamalleista ovat mm. käyttöjärjestelmien matalan tason säikeet, OpenMP ja sen tarjoamat kääntäjädirektiivit, Java-ohjelmointikielen tarjoama toteutus säikeille, viestinvälitykseen pohjautuva MPI-ohjelmointirajapinta sekä myöhemmin tässä tutkielmassa tarkastelun kohteena oleva TBB-rinnakkaistuskirjasto.

Mallit tarjoavat sekä abstraktiota että vakautta. Abstraktiota, koska malli tarjoaa korkeamman tason operaatioita kuin sen alla oleva arkkitehtuuri. Tämä tekee mallista yksinkertaisemman ja tehokkaamman käyttää. Vakaus tulee mallin tarjoaman standardin rajapinnan kautta, joka ei ole riippuvainen alemmalla tasolla olevan arkkitehtuurin muutoksista [8].

2.1 Abstraktiotaso

Oikean abstraktiotason löytäminen on tärkeää ohjelmoinnissa [6]. Nykyään assembler-koodausta käytetään enää kaikkein suoritukseltaan kriittisimpien ohjelmakohtien viilaukseen. Erityisesti rinnakkaislaskennassa, sen haastavan luonteensa vuoksi, on oikean abstraktiotason löytäminen tärkeää. Rinnakkaislaskentaa liittyy monia ongelmia, kuten umpikujat (engl. deadlock) ja kilpailutilanteet (engl. race condition). Näitä virheitä on helppo tehdä ja vaikeaa havaita monimutkaisemmissa ohjelmissa [5]. Ne aiheuttavat usein epämääräisiä ja vaikeasti toistettavia virhetilanteita, joiden syitä ei ole helppoa löytää.

Rinnakkaistetun ohjelmakoodin hahmottaminen voi olla hankalaa ja ohjelman ajonaikainen suoritus on arvaamatonta, koska säikeiden suoritus voi vaihdella jokaisella ajokeralla. Näiden ongelmien ratkaisuun tarvitaan usein erilaisia rinnakkaista suoritusta rajoittavia meka-

nismeja, kuten lukituksia. Kääntöpuolena kuitenkin on, että nämä mekanismit rajoittavat rinnakkaisuuden hyödyntämistä ja hidastavat ohjelman suoritusta, kun rinnakkaislaskennan tarkoituksena oli juuri parantaa suorituskykyä. Tämän vuoksi niiden tehokas ja oikea käyttö on vaikeaa. Moniydinprosessoreiden ollessa jo normi täytyisi tavallisen ohjelmoijankin pystyä toteuttamaan rinnakkaistusta nopeasti ja tehokkaasti, ilman suuria lisäpanostuksia [1].

Rinnakkaislaskenta on suunnittelijan ja ohjelmoijan näkökulmasta hankalaa. Ihmisillä on luontainen kyky hahmottaa asioita rinnakkaisesti [6]. Reinders [6] käyttää esimerkkinä toimintaa kauppakeskuksen kassalla, jossa ihmisjonot jakautuvat usealle kassalle. Työt (asiakkaiden palvelu) jaetaan siis usean suorittajan (kassaneidit) kesken. Ohjelmoinnissa ihmiset kuitenkin mieltävät ohjelman suorituksen suoraviivaisena etenemisenä, kuten esimerkiksi romaanin lukemisen [8]. Monesta säikeestä koostuvan arvaamattomasti etenevän ohjelman hahmottaminen loogisesti on hankalaa [6].

Eri rinnakkaislaskentamallit ovat abstraktiotasoltaan vaihtelevia. Perinteisesti rinnakkaistuksessa ollaan oltu tekemisissä matalalla tasolla, jolloin kaiken rinnakkaisuuden toteuttaminen on jäänyt ohjelmoijan vastuulle. Reinders [6] vertaa suoraa säikeiden käsittelyä rinnakkaislaskennan assembly-kieleksi – se voi tarjota eniten joustavuutta, mutta tekee ohjelmoinnista ja ohjelmakoodista huomattavasti vaikeampaa ymmärtää, testata ja ylläpitää. Myös ohjelman siirrettävyys ja skaalautuvuus usein kärsivät [6]. Pankratius et al. [5] mukaan matalan tason ratkaisut rinnakkaisuuden toteutukseen ovat ohjelmien koon kasvaessa ja rinnakkaisen ohjelmoinnin yleistyessä riittämättömiä ja liian monimutkaisia. Monet nykyiset rinnakkaislaskentamallit ovat pyrkineet tekemään rinnakkaisesta ohjelmoinnista helpompaa ja tehokkaampaa siirtämällä rinnakkaisuuden eksplisiittistä määrittelyä pois ohjelmoijan vastuulta [8].

2.2 Abstraktiotason ja suorituskyvyn suhde

Rinnakkaisten ohjelmien suoritus ei ole determinististä. Vähänkään suuremman rinnakkaisen ohjelman suoritusta ja sen tilaa ei voida arvioida, koska erilaisia mahdollisuuksia on äärettömän paljon. Jotta rinnakkainen ohjelmakoodi olisi ihmisten hallittavissa, täytyy sen olla huomattavasti abstraktimpaa kuin sen luoma ajonaikainen entiteetti [8]. Tämä viittaa siihen, että suoritettavan ohjelman järjestämisen (säikeiden luomisen, hallinnan, kommunikoinnin

ja synkronoinnin) tulisi olla mahdollisimman implisiittistä ja sen tulisi olla johdettavissa ihmisille ymmärrettävästä ohjelmakoodista. Toisaalta mallin abstraktiotaso on liian korkea, jos sille ei voida kohtuullisilla kustannuksilla luoda suorituskyvyltään tehokkaita alemman tason toteutuksia [8]. Tämä heijastuu siten, että mallilla toteutetut sovellukset eivät ole suorituskyvyltään tarpeeksi tehokkaita.

Jotta malli olisi hyödyllinen, täytyy sen huomioida sekä abstraktiotaso että suorituskyky [8]. Koska rinnakkaislaskennalla on tarkoitus saada nopeusetua verrattuna peräkkäiseen suoritukseen, on suorituskyvyllä suuri merkitys rinnakkaislaskentamallien arvioimisessa [6]. Abstraktio ja suorituskyky ovat usein ristiriidassa keskenään. Matalalla abstraktiotasolla voidaan saavuttaa suorituskyvyltään tehokkaita ratkaisuja, mutta ohjelmointi vaikeutuu ja koodin ymmärrettävyys kärsii. Korkealla abstraktiotasolla ohjelmoiminen on helpompaa ja tehokkaampaa, mutta silloin suorituskyky usein kärsii [7], [8]. Oikean tasapainon löytäminen abstraktion ja tehokkuuden välillä onkin usein hankalaa, ja siihen vaikuttavat myös monet mallin ulkopuoliset tekijät, kuten ohjelmoijan osaamistaso sekä tuotettavan ohjelman prioriteetit, kuten suorituskyky ja ylläpidettävyys [7].

2.3 Rinnakkaislaskentamallin arviointi

Skillicorn et al. [8] esittelevät tekstissään yleiseen käyttöön suunnatulta rinnakkaislaskentamallilta toivottavia ominaisuuksia ja Singh et al. [7] listaavat templaattipohjaiselta rinnakkaislaskentamallilta toivottavia ominaisuuksia. Olen luonut näissä lähteissä esiteltyjen ominaisuuksien pohjalta mallin rinnakkaislaskentamallien arviointiin kahdeksasta eri näkökulmasta. Koska lähteet [7] ja [8] eivät käsittele pelkästään moniydinprosessoreille suunnattuja rinnakkaislaskentamalleja, on joitain niiden esittelemiä ominaisuuksia jätetty huomiotta ja joitain ominaisuuksia on muokattu moniydinprosessoreiden kontekstiin sopiviksi. Myös joitain templaattipohjaisille rinnakkaislaskentamalleille esitettyjä ominaisuuksia ei ole otettu mukaan, tai niitä on muokattu yleisempään muotoon sopiviksi. Jotkin esitellyistä ominaisuuksista voivat olla osittain tai kokonaan toisensa poissulkevia ja vaikeasti täytettäviä. Malli ei pyrikään suoraan arvottamaan olemassa olevia rinnakkaislaskentamalleja, vaan pyrkii tarjoamaan mallin niiden tarkasteluun ja vertailuun eri näkökulmista. Seuraavassa on lähteiden [7] ja [8] pohjalta kokoamani rinnakkaislaskentamallien arviointiin liittyvät ominaisuudet:

(1) Helposti ymmärrettävä

Mallin tulee olla helposti ymmärrettävä ja opetettava, jotta suuri joukko ohjelmistokehittäjiä voi omaksua sen [8]. Tämän ominaisuuden arviointi on jossain määrin subjektiivista ja riippuu ohjelmoijan osaamistasosta ja kokemuksesta. Voidaan kuitenkin vaatia, että:

- Mallin tulee olla rakennettu jonkin olemassa olevan ja yleisessä käytössä olevan ohjelmointikielen päälle. Mallin ei tulisi muuttaa kielen syntaksia ja semantiikkaa [7]
- Mallin tulee peittää sen toteutuksen monimutkaisuudet ja tarjota helppo rajapinta sen käyttöön [8]
- Peräkkäiseen koodiin ei tule joutua tekemään suuria rakenteellisia muutoksia, jotka johtuvat mallin käytöstä [7]
- Ihanteellisessa tapauksessa ohjelmakoodi ja rinnakkaisuuden määrittely olisivat toisistaan täysin erillään [7]

(2) Helposti ohjelmoitava

Mallin tulee olla mahdollisimman abstrakti ja yksinkertainen. Mallin ja sen käännösmekanismien (kääntäjä ja ajonaikainen järjestelmä) tulee hoitaa niin paljon rinnakkaisuuden toteutuksesta kuin mahdollista [8]. Tällöin ohjelmoijan tehtäväksi jää vain rinnakkaisuuden osoittaminen. Vaatimukseen liittyy:

- Mallin tulee automaattisesti jakaa ohjelmoijan osoittama rinnakkaistettava koodi ja datajoukot sopivan kokoisiksi paloiksi (tehtäviksi) [8]
- Mallin tulee automaattisesti luoda säikeet tehtävien suoritukseen ja sijoittaa ne fyysisille suorittimille [8]
- Säikeiden kommunikoinnin ja synkronoinnin tulee olla mallin vastuulla [8]
- Ohjelmakoodiin tulee joutua tekemään mahdollisimman vähän rinnakkaisuuteen liittyvää koodia, kuten aliohjelmakutsuja rinnakkaistuskirjastoon tai kääntäjädirektiivejä [7]

(3) Käyttökelpoisuudeltaan laaja

- Mallin tulee tarjota suuri kokoelma hyödyllisiä rakenteita ja menetelmiä erilaisia rin-

nakkaisia ongelmia varten, jotta se olisi hyödyllinen suurelle joukolle sovelluksia [7]

- Jos mallin abstraktiotaso on korkea, tulee käyttäjän tarvittaessa pystyä käyttämään myös matalan tason rinnakkaistusmekanismeja, kuten lukituksia, mallin tarjoamien rakenteiden ja menetelmien sijasta [7]
- Sisäkkäisen rinnakkaisuuden tulee olla mahdollista, eli rinnakkaiset rakenteet ja menetelmät voivat sisältää toisia rinnakkaisia rakenteita. Tämän tulee päteä kaikille rinnakkaisille rakenteille ja menetelmille, ei vain osalle [7]
- Käyttäjän tulee olla mahdollista lisätä uusia rakenteita ja menetelmiä rinnakkaistusmalliin, jos mallin rakenteet ja menetelmät eivät sovellu ongelman ratkaisuun [7]
- Mallin tulee olla käytettävissä mahdollisimman monessa yleisessä käytössä olevassa ohjelmointikielessä ja –ympäristössä [7]

(4) Riippumaton arkkitehtuurista (siirrettävyys)

Luomassani arviointimallissa arkkitehtuuririippumattomuudella ei ole yhtä suurta painoarvoa kuin lähteissä [7] ja [8], koska arviointimalli perustuu vain moniydinprosessoreilla suoritettavaan rinnakkaislaskentaan. Tässä kontekstissa voidaan kuitenkin vaatia:

- Mallilla toteutettujen ohjelmien tulee olla siirrettävissä rinnakkaiselta tietokoneelta rinnakkaiselle tietokoneelle ilman uudelleenkehitystä ja ei-triviaaleja muutoksia. Ohjelman suorituskyvyssä voidaan olettaa heikentymistä huonosti valitulla arkkitehtuurilla, mutta sen pitäisi silti toimia [7], [8]
- Mallin ei tule perustua minkään tietyn arkkitehtuurin ominaisuuksiin [8]
- Ohjelmistot tulee eristää alla olevan arkkitehtuurin suuriltakin muutoksilta [7], [8]
- Ohjelman tulee olla siirrettävissä usealle erilaiselle arkkitehtuurille [7]

(5) Taattu suorituskyky

Mallilla tuotettujen ohjelmien tulee antaa suorituskyvyltään tyydyttäviä tuloksia. Saavutettu nopeusetu riippuu monesta asiasta, kuten rinnakkaistuksen toteutuksesta, ratkaistavan ongelman laadusta, arkkitehtuurista, sekä itse rinnakkaislaskentamallista (korkean abstraktion mallit eivät usein ole yhtä tehokkaita [8]). Jos tyydyttävää nopeusetua ei saavuteta, ottaen huomioon ylempänä mainitut seikat, ei mallin tuottamaa ratkaisua voida pitää suorituskyvyltään riittävänä. On huomattava, että usein suorituskyvyn ei tarvitse olla paras mahdollinen

saavutettavissa oleva, erityisesti jos sen johdosta mallilla kehitettyjen ohjelmien kehitys- ja ylläpitokulut nousisivat. Nopeasti ja helposti saavutettava nopeusetu on hyväksyttävä, vaikka se ei olisikaan tehokkain mahdollinen saavutettavissa oleva [7], [8]. Seuraavat ominaisuudet tukevat tyydyttävän suorituskyvyn saavuttamista:

- Malli sijoittaa säikeet tehokkaalla tavalla käytössä oleville suorittimille. Ihanteellisessa tapauksessa kaikki ytimet ovat jatkuvasti työllistettyinä.
- Malli ei ylityöllistä tai näännytä suorittimia. Liiallinen rinnakkaistus hidastaa suoritusta ja aiheuttaa ylimääräkustannuksia (engl. overhead).
- Jos ongelma on skaalautuva, tuottaa malli automaattisesti skaalautuvan ratkaisun ytimien määrän mukaan.
- Jos ohjelmoija rinnakkaistaa huonosti rinnakkaistuvan ongelman, kääntäjä huomaa sen ja ilmoittaa siitä ohjelmoijalle. Jos kääntäjä ei tähän pysty, tulee mallille olla olemassa siihen soveltuvia tuettuja työkaluja.

(6) Taattu oikeellisuus

- Mallilla tulee olla vahva semanttinen perusta, jonka pohjalta voidaan rakentaa käännöstekniikoita, joilla ohjelmakoodi käännetään koneella ajettavaan koodiin [8]
- Mallin tulee perustua rakenteisiin, jotka tuottavat todistetusti oikeaa koodia. Perinteinen testaaminen ja virheiden etsiminen rinnakkaisista ohjelmista on mahdotonta niiden äärettömän tila-avaruuden vuoksi [8]
- Mallin tulee antaa joitain takeita ohjelman oikeellisuudesta, kuten umpikuja-vapaus, deterministinen suoritus ja virheensietokyky [7]
- Kääntäjän tulee havaita mahdolliset virhetilanteet, kuten umpikujat ja kilpailutilanteet. Jos kääntäjä ei tähän pysty, tulee mallille olla olemassa siihen soveltuvia tuettuja työkaluja.

(7) Kustannuksiltaan arvioitava

- Ohjelman kustannukset, kuten suoritusaika ja prosessoreiden käyttöaste, tulee pystyä johtamaan ohjelmakoodista, pienestä määrästä tietoa kohteena olevasta tietokoneesta (minimissään prosessoreiden / ytimien määrä) ja ohjelman syöttötietojen koosta (mutta

ei arvoista) [8]

- Mallit antavat ennustettavia ohjelmakustannuksia ja kääntäjät eivät optimoi koodia, josta ennusteet on laskettu [8]
- Ohjelman kokonaiskustannukset tulee olla johdettavissa ohjelman osien kustannuksista ja vastaavasti ohjelman kokonaiskustannuksia ei voida alentaa lisäämällä yhden osan kustannuksia [8]

(8) Tuetut työkalut

- Mallille tulee olla olemassa sitä tukevia suunnittelu-, koodaus-, testaus- ja monitorointityökaluja [7]
- Työkalujen tulee tukea saman tason abstraktiota kuin mallin [7]
- Työkalujen tulee olla helposti opittavia ja käytettäviä [7]

3 Intel Threading Building Blocks (TBB)

Tässä kappaleessa esitellään Intel Threading Building Blocks (TBB) -kirjastopohjainen rinnakkaislaskentamalli. Tutkielman laajuuden vuoksi tarkastelussa pyritään antamaan hyvä yleiskuva mallin toimintaperiaatteista menemättä liian syvälle mallin monimutkaisempiin yksityiskohtiin. Lyhyt koodiesimerkki antaa kuvan siitä miten kirjaston tarjoamia algoritmitemplaatteja hyödynnetään rinnakkaistuksen toteuttamisessa. Kappaleessa 4 tulemme tarkastelemaan TBB:tä aiemmin esitellyn rinnakkaislaskentamallien arviointiin soveltuvan mallin pohjalta.

3.1 Perusteet

TBB on C++-kielelle rakennettu avoimeen lähdekoodiin perustuva rinnakkaistuskirjasto. Sen taustalla on Intel. TBB:n ensimmäinen versio julkistettiin vuoden 2006 loppupuolella, samaan aikaan kun moniydinprosessorit alkoivat lyödä itseään läpi kuluttajamarkkinoilla. Mallin päällimmäisenä tavoitteena on helpottaa ja nopeuttaa rinnakkaisten sovellusten kehittämistä piilottamalla ja automatisoimalla rinnakkaistuksen toteutuksen perusmekanismit, kuten säikeiden luominen, hallinta, kommunikointi ja synkronointi. Näin ollen TBB on korkean abstraktiotason rinnakkaislaskentamalli. TBB pyrkii myös parantamaan rinnakkaisten ohjelmien suorituskykyä, skaalautuvuutta ja siirrettävyyttä [1], [6]. Erityisesti skaalautuvuus on tärkeää, sillä prosessoreiden ydinten määrän kasvaessa tulevaisuudessa tulisi rinnakkaisten ohjelmien hyödyntää niitä automaattisesti, ilman ohjelmakoodin muutoksia tai uudelleensuunnittelua [6].

Siirrettävyyden takaamiseksi TBB-kirjasto on rakennettu käyttäen ainoastaan C++-kielen ominaisuuksia, minkä vuoksi se ei vaadi erityistä tukea kääntäjiltä. Näin ollen TBB on periaatteessa käytettävissä millä tahansa alustalla, jolle löytyy C++ kääntäjätuki [6]. TBB ei siis ole uusi ohjelmointikieli tai laajennus ohjelmointikieleen, vaan C++-kirjasto jonka avulla voidaan rinnakkaistaa ohjelmia. TBB:lle löytyy tukea Windows, Linux ja Machintosh puolelta. Laitteistoista se tukee mm. X86 ja X64 Intel sekä AMD prosessoreita [2], [6]. Intelin omat kääntäjät eivät kuitenkaan välttämättä optimoi koodia yhtä tehokkaasti muille

kuin Intelin omille suorittimille, ja suoritinkohtaiset optimoinnit on tehty vain niitä varten [2]. Muiden kääntäjien osalta tilanne voi vaihdella.

TBB:n tarjoamassa mallissa ohjelmoija ei ole missään vaiheessa tekemisissä suoraan säikeiden kanssa. Sen sijaan ohjelmoija määrittelee ohjelmakoodista rinnakkaisia tehtäviä, eli kohtia jotka halutaan suorittaa rinnakkaisesti. Esimerkki rinnakkaisesta tehtävästä on funktio, joka suorittaa jonkin operaation kaikille taulukon alkioille. Tehtävien alustaminen ja suorittaminen on huomattavasti nopeampaa kuin säikeiden [1], [6]. Kaikki säikeisiin liittyvä logiikka on pääasiallisesti TBB-kirjaston vastuulla. TBB luo säikeet automaattisesti ja sijoittaa ohjelmoijan määrittelemät tehtävät niille. TBB myös suorittaa kuormituksen tasapainottamista (engl. load balancing) säikeiden välillä automaattisesti. Tarvittaessa ohjelmoija pystyy työskentelemään suoraan säikeiden ja muiden alemman tason ominaisuuksien parissa, jos kirjaston tarjoamat algoritmit ja rakenteet eivät riitä sovelluksen rinnakkaistuksen toteuttamiseen. Suosituksena kuitenkin on, että matalamman tason ominaisuuksiin ei turvauduta kuin pakon edessä [6].

Kuinka ohjelmakoodin rinnakkaistus sitten käytännössä toteutetaan TBB:n avulla? TBB tarjoaa algoritmitemplaatteja (engl. algorithm templates) rinnakkaistuksen toteuttamiseen. Algoritmit tarjoavat pohjan rinnakkaistukselle sopiville kohdille ohjelmakoodissa, kuten silmukoille. TBB:n algoritmit ohjaavat hyödyntämään datarinnakkaisuutta tehtävärinnakkaisuuden sijasta, koska datarinnakkaisuus on luonteeltaan skaalautuvaa [6]. TBB:n rinnakkaistusalgoritmeja käsitellään tarkemmin myöhemmin tässä kappaleessa. Ohjelmoija voi myös tarvittaessa luoda omia algoritmeja samoilla keinoilla kuin TBB:n mukana tulevat valmiit algoritmit on luotu, käskyttämällä TBB:n taustalla toimivaa tehtävävuorottajaa (engl. task scheduler) suoraan. Myös tehtävävuorottajasta kerrotaan tarkemmin myöhemmin tässä kappaleessa. TBB tarjoaa myös muita välineitä rinnakkaistuksen toteuttamiseen, esimerkiksi säie-turvallisia rinnakkaisia tietorakenteita (engl. concurrent data structures), mutekseja, lukituksia ja atomisia operaatioita [6]. Tutkielman rajatun pituuden vuoksi emme käsittele näitä tarkemmin.

TBB tukee sisäkkäistä rinnakkaisuutta (engl. nested parallelism) ja välttää monia siihen liittyviä ongelmia (mm. eksplisiittinen synkronointi), joita puhtaiden säikeiden kanssa työskentelystä seuraisi. Itse asiassa TBB suosittelee hyödyntämään sisäkkäistä rinnakkaisuutta mah-

dollisimman paljon, koska sen avulla vältetään tilanteita, joissa TBB:n luomilla säikeillä ei olisi tarpeeksi tehtäviä suoritettavana (engl. undersubscription) [6]. Kuormituksen tasapainottaminen onnistuu TBB:ltä paremmin kun rinnakkaisia tehtäviä on suoritettavana enemmän kuin säikeitä on käytössä [1]. Tämän vuoksi Contreras et al. [1] pyrkivät testeissään luomaan vähintään neljä rinnakkaista tehtävää jokaista säiettä kohden. Sisäkkäinen rinnakkaisuus toimii myös erityisen hyvin TBB:n toiminnan perustavana olevan rekursion ja tehtävien varastamisen kanssa [6]. Koska TBB pohjautuu täysin C++-kielen syntaksiin ja ominaisuuksiin, toimii se ongelmitta muiden rinnakkaislaskentamallien (mm. OpenMP ja MPI) kanssa, joten ohjelmoija ei joudu valitsemaan näiden mallien väliltä vaan voi tarpeen mukaan yhdistää ja hyödyntää niitä TBB:n kanssa [6].

3.2 Tehtävävuorottaja

TBB:n toiminnan taustalla on tehtävävuorottaja. Normaalisti ohjelmoija ei ole tekemisissä suoraan tehtävävuorottajan kanssa, vaan hyödyntää TBB:n rinnakkaistusalgoritmeja, jotka puolestaan kätkevät tehtävävuorottajan alleen. Tarvittaessa tehtävävuorottajaa voidaan käskyttää myös suoraan, jos TBB:n valmiit algoritmit eivät sovellu tilanteeseen. On kuitenkin suositeltavaa hyödyntää rinnakkaistusalgoritmeja aina kun mahdollista, koska ne kätkevät tehtävävuorottajan kompleksisuuden alleen [6].

Tehtävävuorottaja luo ja hallitsee ajonaikaista säievarantoa (engl. thread pool), joka sisältää käyttöjärjestelmän säikeitä. Tehtävävuorottaja jakaa ohjelmoijan määrittelemien tehtävien suorituksen säikeille. TBB ei tuhoa vanhoja ja luo uusia säikeitä suorituksen aikana, vaan säilyttää luomansa säikeet säievarannossa tulevia tehtäviä varten. Optimaalisen suorituksen takaamiseksi TBB luo säikeitä yleensä saman määrän kuin vapaita ytimiä on tarjolla. TBB:llä kehitetty ohjelma on skaalautuva, sillä kirjasto osaa automaattisesti sopeuttaa säikeiden määrän käytössä olevien ytimien määrän mukaan [6].

Cilk-ohjelmointikieli ja sen hyödyntämä rekursio ja tehtävien varastaminen ovat vaikuttaneet vahvasti TBB:n kehitykseen [6]. TBB:n toiminta perustuukin rekursioon. Pelkistetysti sanoen jokaisella tehtävällä on isäntätehtävä, joka luo lapsitehtäviä ratkaisemaan ongelman rekursiivisesti. Tehtävävuorottajan logiikan pohjana on puurakenne, jossa säilytetään kaikki

sillä hetkellä suorituksessa olevat ja suoritusta odottavat tehtävät. Tehtävävuorottaja pitää yllä tehtävien rekursiivista rakennetta tässä puurakenteessa. Säikeet ottavat suoritukseen puun alimman tason tehtävät joilla ei ole enää lapsitehtäviä. Tehtävien suoritus siis etenee alhaalta ylöspäin puurakenteessa. Tehtäväsuorittaja ei jaa suoritusaikaa reilulla periaatteella (engl. fair scheduling), vaan suorituksessa oleva tehtävä pyritään saattaman loppuun ennen seuraavan tehtävän suoritusta. Tämän perusteena on suorituskyvyn maksimointi [6].

Tehtävävuorottaja hyödyntää tehtävien varastamista (engl. task stealing) työnmäärän tasaisen jakautumisen saavuttamiseksi. Kun jonkin säikeen tehtävävaranto (engl. task pool) on tyhjä, etsii se satunnaisesti muiden säikeiden tehtävävarannoista suorittamattomia tehtäviä. Etsintä kohdistuu puun ylimmän tason tehtäviin [6]. Tehtävien varastaminen auttaa sovelluksia hyödyntämään rinnakkaisuutta tehokkaasti ja skaalautumaan alla olevasta arkkitehtuurista, kuten ytimien määrästä, riippumatta. Se myös parantaa kuormituksen tasapainottamista huomattavasti [1]. Tehtävävuorottajan toimintaperiaate säikeiden työnjaolle on siis tehtävien syvyyssuuntainen suorittaminen ja leveyssuuntainen varastaminen [6].

3.3 Rinnakkaistusalgoritmit

Kaikki TBB:n algoritmit pohjautuvat geneerisyyteen. Ne käyttävät C++ Standard Template Library:n (STL) määrittelyjä rinnakkaisten algoritmien tyyppimäärittelyssä. Tarkoituksena on ollut tehdä algoritmeista mahdollisimman monikäyttöisiä ja vähentää niiden rajoituksia [6]. TBB:n rinnakkaiset rakenteet eivät käytä suoraan STL:n tietorakenteita, koska ne eivät ole säie-turvallisia. Rakenteista on kuitenkin pyritty tekemään mahdollisimman samankaltaisia STL:n rakenteiden kanssa [6]. TBB:n käyttämiseksi on ohjelmoijan siis hyvä hallita geneerinen ohjelmointi, erityisesti STL-kirjasto.

TBB:n rinnakkaistusalgoritmit pohjautuvat datarinnakkaisuuteen. Datarinnakkaisuus tarkoittaa datajoukolle (esim. taulukolle) tehtävän operaation jakamista useamman säikeen kesken. Jokainen säie siis käsittelee oman osansa datajoukosta, ja lopussa (tai tarpeen tullen kesken operaation) käsitellyt osat yhdistetään takaisin yhdeksi datajoukoksi. TBB kannustaa hyödyntämään datarinnakkaisuutta tehtävärinnakkaisuuden sijasta, koska datarinnakkaisuus on luonteeltaan skaalautuva [6]. Datajoukko voidaan ytimien lisääntyessä jakaa aina yhä

pienempiin palasiin, ja tämä voidaan tehdä automaattisesti rinnakkaistuskirjaston toimesta. Tehtävärinnakkaisuus ei ole luonteeltaan samalla tavalla skaalautuva, koska se on sidottu koodissa esiintyvien loogisten tehtävien määrään. Ytimien lisääntyessä ei koodia voida enää välttämättä jakaa pienempiin tehtäviin, koska niiden välille tulisi liikaa riippuvuuksia tai niiden suorittaminen erillisissä säikeissä ei enää olisi niin tehokasta [6].

Seuraavassa Reindersin [6] esittelemässä esimerkissä näytetään, kuinka yhtä TBB:n silmukoihin perustuvaa rinnakkaistusalgoritmia (*parallel_for*) voidaan hyödyntää ohjelmakoodissa. Tämä algoritmi on tarkoitettu tilanteisiin, jossa datajoukon jokaiselle alkiolle täytyy tehdä jokin operaatio, ja operaatiot voidaan tehdä säie-turvallisesti, eli ne eivät ole toisistaan riippuvaisia. Algoritmi on kuormitukseltaan tasapainotettu, eli se jakaa tehtävien suorituksen tasaisesti vapaana olevien ytimien kesken [6].

Ennen TBB:n algoritmien käyttöä täytyy kirjaston tehtävävuorottaja aina alustaa. Tehtävävuorottaja tuhoutuu loogiselta alueelta poistuttaessa automaattisesti tai se voidaan tuhota myös aiemmin erillisellä komennolla. Tehtävävuorottajan alustaminen / tuhoaminen on kallista, joten sen alustaminen suositellaan tehtäväksi main-metodissa tai kun säikeet luodaan, eikä aina rinnakkaisten algoritmien käytön yhteydessä [6]. Seuraavassa tehtävävuorottajan alustus main-metodissa:

```
1   #include "tbb/task_scheduler_ini.h"
2   using namespace tbb;
3
4   int main() {
5       task_scheduler_init init;
6       ...
7       return 0;
8   }
```

Parallel_for algoritmilla rinnakkaistetaan for-silmukka, jossa jokaiselle datajoukon alkiolle tehdään jokin operaatio. Seuraavassa esiteltynä alkuperäinen peräkkäinen versio koodista:

```
1   void SerialApplyFoo(float a[], size_t n) {
2       for(size_t i=0, i<n; ++i)
3           Foo(a[i]);
4   }
```

Huomioitavaa on, että esimerkissä iteraatioalueen määrittelyssä käytetään muuttujatyyppiä *size_t* tavallisen kokonaisluvun sijasta. Käytännössä tämä on vain tyyliseikka [6].

Rinnakkaisen algoritmin käyttöä varten meidän täytyy muuttaa silmukkaa siten, että se operoi datajoukkoa palasittain. Tähän tarkoitukseen luodaan STL-tyylinen funktio-olio, jonka operator-metodissa määritellään alkioille suoritettavat toimenpiteet.

```
1   #include ”tbb/blocked_range.h”
2
3   class ApplyFoo {
4       float *const my_a;
5   public:
6       void operator() (const blocked_range<size_t>& r) const {
7       float *a = my_a;
8       for (size_t i=r.begin(); i!=r.end(); ++i)
9           Foo(a[i]);
10      }
11  ApplyFoo(float a[]);
12      my_a(a)
13      {}
14  };
```

Huomioitavaa on operator-metodin parametrin *r* tyyppi. Blocked_range<T> on TBB-kirjaston tarjoama templaattiluokka, jota käytetään yksiulotteisen iteraatioalueen määrittelyssä tyypille T. Kuten aiemmin todettiinkin, pohjautuu TBB vahvasti geneerisyyteen, joka on nähtävissä myös iteraatioalueiden määrittelyssä. Kirjasto tarjoaa myös luokan blocked_range2d<T> kaksiulotteisten iteraatioalueiden määrittelyyn [6].

Parallel_for algoritmi vaatii, että oliolla on kopiokonstruktori ja –destruktori, joita käytetään

luomaan ja tuhoamaan kopiot oliosta jokaista säiettä varten. Implisiittisesti luodut kopiokonstruktorit ja –desktruktorit ovat riittäviä useimmissa tapauksissa. Koska oliosta luodaan kopioita säikeitä varten, ei operator-metodin tule muokata olion muuttujia. Esimerkin tapauksessa operator-metodin ei tule muokata *my_a* muuttujaa. Se voi kuitenkin muokata sitä, mihin *my_a* osoittaa. Tämän vuoksi esimerkissä *my_a* on määritelty vakioksi ja mihin se osoittaa ei-vakioksi liukuluvuksi. Muussa tapauksessa muuttujaan tehty muutos tulisi tai ei tulisi parallel_for algoritmia kutsuneen säikeen näkyville riippuen siitä, käsitteleekö se alkuperäistä oliota vai sen kopiota. Tämän sivuvaikutuksen estämiseksi TBB vaatii, että operator-metodi on aina määritelty vakioksi [6].

Esimerkissä operator-metodi lataa *my_a* muuttujan paikalliseen muuttujaan *a*. Tämä ei ole pakollista, mutta sen perusteena voidaan käyttää tyyliä ja suorituskykyä. Määrittely luo silmukan rakenteesta enemmän alkuperäistä peräkkäistä versiota muistuttavan ja se auttaa kääntäjää silmukan optimoinnissa [6].

Funktio-olion luomisen jälkeen voimme kutsua itse parallel_for funktiota, jolle välitetään parametrina juuri luotu funktio-olio:

```
1   #include ”tbb/parallel_for.h”
2
3   void ParallelApplyFoo(float a[], size_t n) {
4       parallel_for(blocked_range<size_t>(0,n,YouPickGrainSize),
5                    ApplyFoo(a));
6   }
```

Blocked_range<T> konstruktori ottaa parametreinaan iteraatioalueen alun ja lopun, sekä kolmantena parametrina kokonaisluvun, jota kutsutaan rakeenkooksi (engl. grain size). Tämä kokonaisluku määrittelee, kuinka suuria rinnakkaisesti käsiteltävät palaset ovat. Esimerkiksi jos taulukon koko on 10 000 alkiota ja rakeenkoko on 2000, jaetaan taulukko viideksi erikseen suoritettavaksi palaksi. Rakeenkoolla on tärkeä yhteys ylimääräkustannuksiin (engl. overhead), koska jokaista rinnakkaisesti suoritettavaa palasta kohden syntyy ylimääräkustannuksia. Jos rakeenkoko on määritelty todella pieneksi, kasvaa palasten ja samalla ylimääräkustannusten määrä suhteessa rinnakkaisuudesta saatavaan nopeusetuun. Toisaalta liian suuri

rakeenkoko rajoittaa rinnakkaisuutta tilanteessa, jossa palasia luodaan vähemmän kuin vapaita ytimiä niiden suoritukseen olisi käytettävissä [6].

Rakeenkoon määrittelyssä ei tarvitse olla erityisen tarkka, sillä ainoastaan ääripäiden arvoilla (aivan liian vähän, aivan liian paljon) on suurta vaikutusta suoritukseen. On myös hyvä huomata, että rakeenkoko ei ole sidottu käytettävissä olevien suorittimien tai suoritettavien iteraatioiden määrään, vaan sen asettamisessa tulee keskittyä palasien aiheuttamien ylimääräkustannuksien ja hyödyllisen työn suhteeseen [6]. TBB:n kehittäjät tutkivat mahdollisuutta tehdä rakeenkoon määrittelyn automaattisesti kirjaston toimesta, mutta totesivat sen olevan hankalaa [6]. TBB:n versiosta 2.2 lähtien kirjasto osaa kuitenkin tehdä rakeenkoon määrittelyn automaattisesti. Käytetty heuristiikka huomioi mm. ylimääräkustannukset ja kuormituksen tasapainottamisen. Automaattisen rakeenkoonmäärittelyn käyttäminen on suositeltavaa useimmissa tapauksissa [2].

4 TBB:n rinnakkaislaskentamallin arviointi

Seuraavaksi arvioimme TBB:tä kappaleessa 2 esitellyn rinnakkaislaskentamallien arviointiin tarkoitetun mallin pohjalta. Kuten aiemmin totesimme, malli pyrkii tarkastelemaan ja arvioimaan rinnakkaislaskentamallia eri näkökulmista. Ollakseen todella onnistunut, täytyy rinnakkaislaskentamallin saavuttaa jonkinasteinen tasapaino näiden ominaisuuksien välillä. Vaikka malli olisi kuinka tehokas suorituskyvyltään, voi sillä olla vaikeuksia levitä yleiseen käyttöön jos se on vaikeasti ymmärrettävä ja ohjelmoitava.

(1) Helposti ymmärrettävä

TBB on C++-ohjelmointikielelle rakennettu kirjastopohjainen rinnakkaislaskentamalli. Se ei muuta kielen syntaksia tai semantiikkaa. TBB kirjasto on korkean abstraktiotason rinnakkaislaskentamalli, peittäen säikeiden luomisen ja hallinnan ohjelmoijalta siirtämällä sen itsensä vastuulle. Rajapintana käytetään TBB:n tarjoamia rinnakkaistusalgoritmeja ja -rakenteita.

(2) Helposti ohjelmoitava

TBB luo ja sijoittaa säikeet suorittimille automaattisesti ja skaalautuu prosessorin ytimien lukumäärän mukaan. TBB kannustaa käyttämään sen valmiita rinnakkaistusalgoritmeja ja -rakenteita, jolloin säikeiden synkronointi jää TBB:n vastuulle. Tilanteissa, joihin valmiit algoritmit ja rakenteet eivät sovellu, voi ohjelmoija joutua turvautumaan matalan tason lukituksiin, mutekseihin tai atomisiin operaatioihin. TBB:n käyttö vaatii lisäyksiä ohjelmakoodiin, kuten tehtävävuorottajan alustuksen ja rinnakkaistusalgoritmien kutsut. Näitä algoritmeja varten tulee myös luoda standardin mukaiset funktio-oliot, joihin ohjelmoija sisällyttää rinnakkaistettavan ongelman. Koska TBB pohjautuu geneeriseen ohjelmointiin ja käyttää C++ Standard Template Library:n (STL) rakenteita, vaaditaan ohjelmoijalta näiden osaamista.

Vaikka TBB:n käyttö pohjautuu suoraan C++-kielen syntaksiin, vaatii rinnakkaistuksen toteutus sillä usein enemmän lisäyksiä / muokkauksia peräkkäiseen koodiin kuin esimerkiksi OpenMP:n kääntäjädirektiivit. Esimerkiksi rinnakkaistusalgoritmeja varten täytyy luoda funktio-oliot. Marowkan [4] koodiesimerkin rinnakkaistaminen OpenMP:llä vaati 12 riviä koodia, kun sen rinnakkaistaminen TBB:llä vaati 22 riviä koodia. Contreras et al. [1] havait-

sivat testeissään tilanteen, jossa TBB:n käyttö jopa hidasti ohjelman suoritusta kasvaneiden ylimääräkustannusten vuoksi. Rinnakkaisuuden toteuttamisen rajoittaminen kirjaston aiheuttamien ylimääräkustannusten vuoksi vaikeuttaa ohjelmointia, koska se vaatii ylimääräistä profilointia ja testausta tehokkaiden rinnakkaistusstrategioiden löytämiseksi [1].

(3) Käyttökelpoisuudeltaan laaja

TBB:n tarjoamat rinnakkaistusalgoritmit perustuvat datarinnakkaisuuden hyödyntämiseen (sen paremman skaalautuvuuden takia [6]). Toisaalta tämän voisi nähdä rajoittavana tekijänä, koska algoritmit eivät sovellu tehtävärinnakkaisuuden toteuttamiseen. TBB tarjoaa myös samoja matalan tason menetelmiä kuin useimmat muut rinnakkaislaskentamallit (mm. lukitukset ja muteksit) sekä muita rinnakkaisuuden toteutukseen soveltuvia välineitä, kuten skaalautuva muistinvaraaja ja säie-turvalliset rinnakkaiset tietorakenteet. Ohjelmoija voi tarvittaessa suoraan käskyttää tehtävävuorottajaa, jota myös valmiit rinnakkaistusalgoritmit käyttävät. Näin ohjelmoija voi tarvittaessa luoda uusia rinnakkaistusalgoritmeja tehtävävuorottajan sallimien rajojen puitteissa. TBB tukee sisäkkäistä rinnakkaisuutta ja tekee sen käytöstä helppoa [6]. Koska TBB pohjautuu C++-kielen syntaksiin ja ominaisuuksiin eikä vaadi erityistä tukea kääntäjältä, on se yhteensopiva muiden rinnakkaislaskentamallien (mm. OpenMP ja MPI) kanssa. TBB:stä ei ole olemassa versiota muille kuin C++-kielelle. TBB pohjautuu avoimeen lähdekoodiin.

(4) Riippumaton arkkitehtuurista (siirrettävyys)

TBB pohjautuu standardiin C++-kieleen ja näin ollen toimii käytännössä minkä tahansa C++ kääntäjän kanssa, missä tahansa käyttöjärjestelmässä. Tuettuja kääntäjiä ovat mm. Visual C++, Intel C++, ja GNU (GCC) kääntäjät ja tuettuja käyttöjärjestelmiä mm. Microsoft Windows (XP tai uudempi), Mac OS X, Linux, Solaris ja PowerPC. Laitteistoista tuettuja ovat mm. Intelin ja AMD:n X86 ja X64 prosessorit [2], [6]. Intelin omat kääntäjät eivät välttämättä optimoi koodia yhtä tehokkaasti muille kuin Intelin omille suorittimille, ja suoritinkohtaiset optimoinnit on tehty vain niitä varten [2]. Muut kääntäjät voivat tietenkin optimoida koodin paremmin muillekin suorittimille ja arkkitehtuureille.

(5) Taattu suorituskyky

TBB luo automaattisesti yhtä monta säiettä kuin vapaita ytimiä on käytettävissä. Ohjelmoija voi halutessaan myös luoda enemmän säikeitä [1]. TBB:tä käytettäessä ei olla tekemisissä suoraan säikeiden kanssa, vaan ohjelmoija määrittelee rinnakkaisia tehtäviä, joiden suorituksen TBB jakaa luomiensa säikeiden kesken. TBB:n pohjautuminen tehtävien varastamiseen tekee sitä käyttävistä ohjelmista automaattisesti skaalautuvia ytimien määrän mukaan. Tämä on yksi TBB:n suurimmista eduista [1]. Tehtävien varastaminen myös parantaa kuormituksen tasapainottamista ytimien kesken, koska toimettomana olevat säikeet voivat varastaa tehtäviä muiden säikeiden tehtäväjonosta [1], [6].

Ainakin TBB:n versiossa 2.0 käytössä ollut satunnaisuuteen pohjautuva valinta tehtävien varastamisessa tekee heikompia valintoja mitä enemmän tehtäviä on tarjolla, koska todennäköisyys parhaan valinnan tekemiseen heikkenee. Tämä korostuu erityisesti tilanteissa, joissa kuormituksessa on suurta epätasapainoa [1]. Contreras et al. [1] kokeilivat monimutkaisempia menetelmiä varastettavan tehtävän valinnassa. Heidän menetelmänsä tutki kaikkien säikeiden (tai rajoitettujen säie-joukkojen) tehtäväjonot ja varastivat säikeeltä, jolla oli suurin työmäärä. Suuremmista ylimääräkustannuksista huolimatta nämä menetelmät parantavat suorituskykyä verrattuna satunnaiseen tehtävän varastamiseen, muutamasta prosentista aina 20% saakka, riippuen tehtävästä ja ydinten määrästä [1].

TBB pyrkii välttämään suorittimien ylityöllistämisen haittoja ”epäreilulla” suoritusajan jakamisella. Aloitetut tehtävät pyritään suorittamaan loppuun, eikä suoritusaikaa jaeta tehtävien kesken. Suoritusajan tasaisesta jakamisesta seuraisi ylimääräkustannuksia. Suorittimien nääntymistä pyritään helpottamaan tehtävien varastamisella [6].

Contreras et al. [1] testasivat TBB:n suorituskykyä seitsemän erilaisen rinnakkaistuvan tehtävän suorituksessa. Heidän käyttämänsä versio TBB:stä oli 2.0. Tämän tutkielman kirjoitushetkellä uusin versio TBB:stä on 4.1 (update 4) [2], joten testien tulokset eivät välttämättä ole enää täysin päteviä uusimman version kohdalla. Rinnakkaisuuden dynaaminen hallinta TBB-kirjaston toimesta vaatii tehtävien tallentamista, vuorottamista ja sijoittamista säikeille [1]. Testeissään Contreras et al. [1] huomasivat, että ytimien (ja sitä myötä luotujen säikeiden) määrän kasvaessa TBB-kirjaston aiheuttamat ylimääräkustannukset (kirjaston omien

funktioiden suorittaminen) kasvoivat huomattavasti. Erään tehtävän kohdalla ylimääräkustannukset 16 ytimellä olivat 3% per ydin, kun 32 ytimellä ne olivat jo 52% per ydin. Erityisesti uusien tehtävien odottaminen ja tehtävien varastaminen vaativat paljon suoritusaikaa [1]. Nämä kasvavat ylimääräkustannukset rajoittavat TBB:n soveltuvuutta hienojakoisen rinnakkaisuuden hyödyntämisessä (engl. fine-grained parallelism), jossa tehtävien koot ovat pieniä ja ne kommunikoivat usein [1], [4]. Aiempi 32 ytimen esimerkki pohjautui juuri hienojakoiseen rinnakkaisuuteen. Koska TBB:llä on kyky skaalautua automaattisesti ytimien määrän kasvaessa, ja koska nykyisten ja tulevaisuuden sovelluksien tehokas rinnakkaistaminen vaatii hienojakoisempaa rinnakkaisuutta [1], on sääli että ylimääräkustannusten voimakas kasvu rajoittaa TBB:n potentiaalia.

Contreras et al. [1] testit osoittivat, että TBB:n dynaaminen luonne (tehtävien käyttäminen säikeiden sijasta, tehtävien varastaminen) antoivat tasaisempaa suorituskykyä ytimien määrästä riippumatta. Staattisella työnjaolla kuormituksen jakautuminen ytimien kesken saattoi vaihdella suuresti [1]. Matalalla määrällä ytimiä ja suurijakoisen rinnakkaisuuden (engl. coarse-grained parallelism) kohdalla TBB antaa hyviä tuloksia. Ytimien määrän lisääntyessä jatkaa suorituskyky useimmissa tapauksissa kasvamista, mutta kasvuvauhti alkaa hidastua [1], [4]. Contreras et al. [1] testeissä TBB:n dynaaminen työnjako verrattuna staattiseen työnjakoon antoi vaihtelevia tuloksia. Joidenkin tehtävien osalta TBB:n dynaaminen työnjako antoi paremman tuloksen (3 / 7), kun taas toisten tehtävien osalta staattinen työnjako osoittautui tehokkaammaksi (3 / 7). Yhden testin kohdalla staattisen työnjaon suorituskyvyn suuren vaihtelevuuden vuoksi on vaikeampaa sanoa kumpi menetelmä oli tehokkaampi [1].

Contreras et al. [1] havaitsivat testeissään, että TBB:n hyödyntäminen ohjelmakoodissa joka luo suuria määriä tehtäviä voi jopa hidastaa ohjelman suorituskykyä ylimääräkustannuksien (tehtävävuorottaja, synkronointi, tehtävien varastaminen, odottelu) kasvaessa todella suuriksi. Heidän mukaansa ajonaikaisen kirjaston tulisi pystyä havaitsemaan tällaiset huonosti rinnakkaistuvat kohdat (esimerkiksi kun luotujen tehtävien määrä alkaa kasvaa räjähdysmäisesti) ja rajoittaa näiden kohtien suoritus pienemmälle määrälle tai jopa yhdelle ytimelle.

(6) Taattu oikeellisuus

TBB on rakennettu standardin C++-kielen päälle. Käyttämällä TBB:n valmiita rinnakkaistusalgoritmeja ja säie-turvallisia rinnakkaisia tietorakenteita voidaan tuottaa oikeaa ja turvallista rinnakkaista koodia. Algoritmit ja rakenteet hoitavat tarvittavat lukitukset ja synkronoinnit implisiittisesti. Vaikka niitä käyttämällä voidaan tuottaa turvallista koodia, voi ohjelman suorituskyky kuitenkin laskea algoritmien ja rakenteiden luomien ylimääräkustannusten vuoksi. TBB:n tarjoamat algoritmit eivät kuitenkaan sovellu kaikkiin rinnakkaisiin ongelmiin, esimerkiksi jos kaksi tehtävää käsittelevät samoja muuttujia rinnakkaisesti. Tällöin ohjelmoija voi joutua turvautumaan alemman tason ominaisuuksiin kuten suoriin lukituksiin, mutekseihin ja atomisiin operaatioihin, joiden käytön osalta koodin oikeellisuus jää ohjelmoijan vastuulle. C++-kääntäjät eivät pysty havaitsemaan rinnakkaisuuden ongelmatilanteita, kuten umpikujia tai kilpailutilanteita. Intelin julkaisemat TBB:tä tukevat työkalut kuitenkin pystyvät havaitsemaan näitä virhetilanteita (ks. kohta 8) [3].

(7) Kustannuksiltaan arvioitava

TBB:n korkea abstraktiotaso vaikeuttaa ohjelman rinnakkaisuuden kustannusten arviointia, koska tehtävien sijoittaminen säikeille, synkronointi ja muut rinnakkaistuksen matalamman tason toimenpiteet ovat dynaamisen kirjaston vastuulla [1]. Contreras et al. [1] havaitsivat yhdessä testissään, että TBB:n hyödyntäminen ohjelmakoodissa jopa hidasti ohjelman suoritusta ylimääräkustannusten kasvaessa suuriksi. Tällainen voi tietenkin vaikeuttaa kustannusten arviointia edelleen. Tehtäviin ja tehtävien varastamiseen pohjautumisen ansoista TBB pitää suorittimien käyttöasteen pääsääntöisesti parempana ja tasaisempana kuin staattinen työnjako, riippumatta ydinten tai tehtävien määrästä [1]. Koska TBB kääntyy millä tahansa standardilla C++-kääntäjällä, riippuu ohjelmakoodin optimointi valitusta kääntäjästä. Esimerkiksi GCC:llä optimoinnin tason voi valita itse, ja optimointi on mahdollista myös ottaa pois käytöstä kokonaan. Intel on julkaissut työkaluja, joiden avulla rinnakkaistuksen suorituskykyä voidaan tarkastella (ks. kohta 8) [3].

(8) Tuetut työkalut

Intel on julkaissut työkaluja, jotka tukevat TBB:n käyttöä. Nämä työkalut pystyvät TBB:n lisäksi seuraamaan myös käyttöjärjestelmien omia matalan tason säikeitä ja synkronointivälineitä (Win32 API ja POSIX-säikeet), sekä OpenMP-kääntäjädirektiivejä. Työkaluista on saatavilla graafisen käyttöliittymän versio Windowsille ja komentoriviltä toimiva versio Linuxille. Komentorivi versiolla voi vain kerätä dataa suorituksesta, tulosten tarkastelu onnistuu vain graafisen käyttöliittymän kautta [3].

Intel® Thread Checker avulla voidaan havaita kilpailutilanteita, umpikujia ja tilanteita, joissa säikeet joutuvat odottamaan pitkiä aikoja. Ohjelmalla on tarkoitus varmistaa koodin oikeellisuus [3].

Intel® Thread Profiler analysoi rinnakkaistuksen suorituskykyä ja näyttää visuaalisesti säikeiden välisen vuorovaikutuksen. Ohjelmalla on tarkoitus löytää suoritusta hidastavia kohtia ohjelmakoodista [3].

5 Yhteenveto

Moniydinprosessoreiden vallattua markkinat on rinnakkaislaskennasta tullut arkipäivää yhä useammalle ohjelmoijalle. Rinnakkaislaskenta on kuitenkin vaikeaa hahmottaa ja toteuttaa. Tässä tutkielmassa esittelimme mallin rinnakkaislaskentamallien tarkasteluun kahdeksasta eri näkökulmasta: *(1) Helposti ohjelmoitava*, *(2) helposti ymmärrettävä*, *(3) käyttökelpoisuudeltaan laaja*, *(4) riippumaton arkkitehtuurista (siirrettävyys)*, *(5) taattu suorituskyky*, *(6) taattu oikeellisuus*, *(7) kustannuksiltaan arvioitava* ja *(8) tuetut työkalut*. Levitäkseen laajaan käyttöön täytyy rinnakkaislaskentamallin huomioida kaikki nämä kohdat vähintään jollain tasolla. Koska useat näistä ominaisuuksista ovat ristiriidassa keskenään tai vaikeasti toteutettavia, täytyy onnistuneen rinnakkaislaskentamallin löytää jonkinlainen tasapaino niiden välillä.

Intel Threading Building Blocks (TBB) on vuonna 2006 julkaistu C++-kielelle rakennettu avoimeen lähdekoodiin perustuva rinnakkaistuskirjasto. Sen tavoitteena on helpottaa ja nopeuttaa rinnakkaistuksen toteuttamista siirtämällä rinnakkaistuksen perusmekanismit (mm. säikeiden luominen ja hallinta) kirjaston vastuulle. TBB pyrkii myös parantamaan rinnakkaisten ohjelmien suorituskykyä, skaalautuvuutta ja siirrettävyyttä. Ohjelmien rinnakkaistus TBB:llä hoidetaan pääasiassa sen mukana tulevilla rinnakkaistusalgoritmeilla, jotka pohjautuvat datarinnakkaisuuden hyödyntämiseen. Tarvittaessa ohjelmoija voi myös käyttää matalan tason rinnakkaistusmekanismeja, kuten lukituksia, mutekseja ja atomisia operaatioita.

Tarkastelimme TBB:tä luomamme rinnakkaislaskentamallien arviointiin tarkoitetun mallin pohjalta. Mallin kriteereistä TBB täyttää erityisen hyvin kohdat *(3) käyttökelpoisuudeltaan laaja* ja *(4) riippumaton arkkitehtuurista (siirrettävyys)*. TBB tarjoaa useita korkean tason algoritmeja ja mm. säie-turvallisia tietorakenteita rinnakkaisuuden toteuttamiseen, mutta se tarjoaa myös matalan tason välineitä, kuten lukituksia, mutekseja ja atomisia operaatioita tilanteen niitä vaatiessa. Sisäkkäinen rinnakkaisuus on mahdollista ja helppoa toteuttaa ja ohjelmoija voi tarvittaessa rakentaa omia rinnakkaistusalgoritmeja komentamalla suoraan tehtävävuorottajaa. TBB toimii ongelmitta muiden rinnakkaislaskentamallien, kuten OpenMP ja MPI kanssa. Ainoina heikkouksina laajan käyttökelpoisuuden osalta voitaneen pitää TBB:n tukea ainoastaan C++-kielelle, sekä sen algoritmien rajoittumista datarinnak-

kaisuuden hyödyntämiseen. Siirrettävyyden osalta TBB:n vahvuus on sen pohjautuminen standardiin C++-kieleen, jolloin se toimii käytännössä minkä tahansa C++-kääntäjän kanssa. Käyttöjärjestelmien ja laitteistojen osalta tuki on myös laaja, mm. Windows, Linux, Mac OS X, Solaris ja PowerPC, sekä Intelin ja AMD:n X86 ja X64 prosessorit ovat tuettuja. Pienenä heikkoutena ainakaan Intelin omat kääntäjät eivät välttämättä optimoi koodia yhtä tehokkaasti muille suorittimille, ja suoritinkohtaiset optimoinnin on tehty vain niitä varten.

TBB pärjää hyvin myös kohtien *(1) helposti ymmärrettävä*, *(2) helposti ohjelmoitava* ja *(8) tuetut työkalut* osalta. Ymmärrettävyyttä tukee TBB:n pohjautuminen täysin C++-kielen syntaksiin ja semantiikkaan. Sen käyttö ei vaadi uusien komentojen tai logiikkojen opettelua. TBB on myös todella korkean tason rinnakkaislaskentamalli, sillä se piilottaa täysin säikeiden luomisen ja hallinnan kirjaston alle. TBB:n pohjautuminen C++-kieleen ja säikeiden luomisen, hallinnan ja synkronoinnin siirtäminen kirjaston vastuulle tukevat myös helppoa ohjelmoitavuutta. TBB skaalautuu automaattisesti ytimien määrän mukaan ja suorittaa datajoukon jakamisen paloiksi automaattisesti (ohjelmoija voi kuitenkin tehdä sen myös itse). Vaikka TBB pohjautuukin puhtaasti C++-kieleen, vaatii sen käyttö enemmän lisäyksiä koodiin kuin esimerkiksi OpenMP:n kääntäjädirektiivien käyttö. TBB:n tehokas käyttö vaatii myös geneerisen ohjelmoinnin ja Standard Template Library:n (STL) hallintaa. TBB:lle löytyy hyödyllisiä ja helppokäyttöisiä työkaluja, joiden avulla voidaan analysoida rinnakkaisen ohjelman suoritusta, tutkia säikeiden välistä vuorovaikusta ja etsiä suoritusta hidastavia kohtia. Ne voivat myös tutkia ohjelmakoodia, etsien potentiaalisia virhetilanteita, kuten umpikujia ja kilpailutilanteita. Lievänä pettymyksenä näistä ohjelmista löytyy täydelliset versiot vain Windowsille.

Kohta *(6) taattu oikeellisuus* toteutuu TBB:n osalta kohtuullisen hyvin. Käyttämällä TBB:n valmiita rinnakkaistusalgoritmeja ja säie-turvallisia rinnakkaisia tietorakenteita voidaan tuottaa turvallista rinnakkaista koodia. Rinnakkaistettava ongelma ei kuitenkaan aina taivu näille rakenteille sopivaksi, jolloin voidaan joutua käyttämään matalan tason mekanismeja, kuten lukituksia, mutekseja ja atomisia operaatioita. Näiden käytön osalta vastuu koodin oikeellisuudesta jää ohjelmoijan vastuulle. Tavalliset C++-kääntäjät eivät pysty havaitsemaan rinnakkaisuuden ongelmatilanteita, kuten umpikujia tai kilpailutilanteita. TBB:lle kuitenkin löytyy työkaluja, joilla näitä ongelmia voidaan etsiä ja havaita.

Kohdan *(5) taattu suorituskyky* osalta tulokset ovat kaksijakoisia. Toisaalta TBB osaa skaalautua automaattisesti ytimien määrän mukaan ja pitää kaikki suorittimet, määrästä riippumatta, tehokkaasti ja suhteellisen tasaisesti työllistettyinä (kiitos tehtävien varastamiseen pohjautumisesta). Pienellä määrällä ytimiä ja suurijakoisen rinnakkaisuuden osalta TBB antaa hyviä tuloksia. Ytimien määrän lisääntyessä tai todella hienojakoisen rinnakkaisuuden kohdalla kirjaston ylimääräkustannukset kuitenkin kasvavat nopeasti ja suorituskyvyn kasvu hidastuu huomattavasti. Pahimmassa tapauksessa suorituskyky voi jopa laskea suuremmalla määrällä ytimiä tai liian rinnakkaisuuden vuoksi. TBB ei osaa automaattisesti puuttua tilanteisiin, joissa liika rinnakkaisuus hidastaa ohjelman suoritusta. Tämä ominaisuus olisi toivottava, jolloin ohjelmoijan ei tarvitsisi pelätä, että rinnakkaisuuden hyödyntäminen voi jopa laskea suorituskykyä. TBB asettaa yhdeksi tärkeimmäksi tavoitteekseen automaattisen skaalautuvuuden, jonka se teknisesti saavuttaakin, mutta valitettavasti nopeasti kasvavat ylimääräkustannukset heikentävät sen tehoa huomattavasti.

Kohdan *(7) kustannuksiltaan arvioitava* kanssa TBB:llä on hieman ongelmia. Koska TBB on korkean abstraktiotason rinnakkaislaskentamalli, joka peittää rinnakkaistuksen perusmekanismit alleen, vaikeutuu kustannusten arviointi. Eräässä tarkastelluista esimerkkisovelluksista TBB:n käyttö suurella määrällä ytimiä jopa laski suorituskykyä, johtuen kirjaston aiheuttamista ylimääräkustannuksista. Tällaisten ongelmien arviointi ennakkoon on ohjelmoijalta mahdotonta. Hyvänä puolena TBB kuitenkin pitää ytimet hyvin ja suhteellisen tasaisesti työllistettyinä automaattisesti.

Lähteet

[1] Gilberto Contreras, Margaret Martonosi, *Characterizing and Improving the Performance of Intel Threading Building Blocks*. Published in 2008 IEEE International Symposium on Workload Characterization (IISWC 2008), sivut 57-66. Saatavilla PDF-muodossa <URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4636091>

[2] Intel Threading Building Blocks website, <URL: http://threadingbuildingblocks.org/>. Viitattu 17.6.2013.

[3] Intel Threading Building Blocks - Threading performance and correctness analysis, <URL: http://software.intel.com/en-us/articles/intel-threading-building-blocks-threading-performance-and-correctness-analysis>. Viitattu 18.6.2013.

[4] Ami Marowka, *Towards High-Level Parallel Programming Models for Multicore Systems*. Published in 2008 Advanced Software Engineering and Its Applications (ASEA 2008), sivut 226-229. Saatavilla PDF-muodossa <URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4721348>

[5] Victor Pankratius, Christoph Schaefer, Ali Jannesari, Walter F. Tichy, *Software Engineering for Multicore Systems: An Experience Report*. Proceedings of the 1st international workshop on Multicore software engineering (IWMSE '08), sivut 53-60. Saatavilla PDF-muodossa <URL: http://dl.acm.org/ft_gateway.cfm?id=1370096&type=pdf&CFID=228032556&CFTOKEN=80227006>

[6] James Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. Publisher: O'Reilly Media, July 2007.

[7] Ajit Singh, Jonathan Schaeffer, Duane Szafron, *Views on Template-Based Parallel Programming*. Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research (CASCON '96), sivu 35. Saatavilla PDF-muodossa <URL: http://dl.acm.org/ft_gateway.cfm?id=782087&ftid=156664&

dwn=1&CFID=228032556&CFTOKEN=80227006>

[8] David B. Skillicorn, Domenico Talia, *Models and Languages for Parallel Computation*. Published in ACM Computing Surveys (CSUR) Journal, Volume 30 Issue 2, June 1998, sivut 123-169. Saatavilla PDF-muodossa <URL: http://dl.acm.org/ft_gateway.cfm?id=280278&ftid=41334&dwn=1&CFID=228032556&CFTOKEN=80227006>