Tietojenkäsittelytiede **30** Heinäkuu 2010 sivut 21–32
Toimittaja: Ari Korhonen

Supertietokoneiden tehoa janotaan: miten saada enemmän tehoa irti?

Jan Westerholm, Mats Aspnäs & Artur Signell
Åbo Akademi
Informaatioteknologian laitos
{jawester,mats,asignell}@abo.fi

1 Tausta

Supertietokoneiden ilmoitetut laskentatehot ovat vuosi vuodelta kasvaneet. Top 500 -listan [1] kärkikoneet ovat parinsadantuhannen laskentaytimen koneita, jotka teoreettisesti pystyvät tekemään yli 10^{15} liukulukulaskutoimitusta sekunnissa eli petaflopseja. Gordon Mooren havainnon mukaan transistorien lukumäärä kaupallisissa mikropiireissä kaksinkertaistuu joka toinen vuosi. Näistä transistoreista on ennen rakennettu yhä kehittyneempiä tietokonearkkitehtuureja, mutta viime vuosina niitä on käytetty varsinkin prosessoreiden laskentaytimien lukumäärän kasvattamiseen. Vaativien laskentatehtävien suorittamiselle näyttäisi siis olevan vuosi vuodelta yhä tehokkaampia koneita käytettävissä, ja laskentakapasiteetille on jatkuvasti kasvava kysyntä. Supertietokoneiden tehoa janotaan ja sitä näyttäisi olevan paljon, mutta saavatko ohjelmat kaiken tehon irti?

Tietokoneohjelmien rinnakkaistuksella käyttäjä on pyrkinyt sekä saamaan käyttöönsä suurempia muistimääriä, että lyhentämään pitkiä ajoaikoja. Lyhentyneiden ajoaikojen ansiosta ohjelmia voidaan käytännössä ajaa useammin, jolloin ohjelman toiminta, mahdolliset rajoitteet, herkkyys alkuparametreille ja ohjelman tulokset ymmärretään paremmin. Tässä mielessä voi sanoa, että tieteellisten tulosten laatu paranee rinnakkaistumisen myötä. Jos sama laskenta voidaan ajaa samoilla resursseilla nopeammin koodin optimoinnin ansiosta, säästyy energiaa ja työaikaa, ja laskentaresursseja voidaan hyödyntää paremmin palvelemalla useampia käyttäjiä.

Artikkelin kirjoittajat kuuluvat suurteholaskennan ryhmään, joka viime vuosina on osallistunut useaan supertietokoneohjelmien tehokkuutta ja skaalautuvuutta parantavaan tutkimusprojektiin [2, 3]. Yhteistyökumppanit olivat supertietokoneille ohjelmiaan tekevistä tutkimusorganisaatioista, mutta myös avointa lähdekoodia kehittävistä organisaatioista esimerkiksi bioinformatiikassa. Mukana oli muun muassa ydinfuusiofyysikoita, joille annetaan satoja tuhansia cpu-tunteja yhteiseurooppalaisissa suurteholaskentakeskuksissa.

Kysymys, johon etsimme vastausta, on seuraava: Miten supertietokoneissa ajettavat ohjelmat voivat paremmin hyödyntää käytettävissä olevat tehoresurssit? Tekemämme empiiriset havainnot suurtehokoneissa pyörivistä ohjelmista perustuvat erityisesti ohjelman rinnakkaistukseen, välimuistin käytön optimointiin, sekä vektorilaskentaan. Tarkasteltavat ohjelmat olivat osana kolmea tutkimus-

kokonaisuutta. Ensimmäisessä tutkittiin joukko kansallisen supertietokeskuksen CSC:n supertietokoneissa ajettavia eri tutkimusryhmien kirjoittamia ohjelmia, sekä yleisessä käytössä olevaa avointa lähdekoodia. Toisessa projektissa saimme EU-yhteistyön kautta käyttöömme eurooppalaisten ydinfuusioryhmien kirjoittamia lähdekoodeja, ja kolmanteen ryhmään laitamme tässä muiden yhteydenottojen kautta tulleita ohjelmia. Ohjelmointikieliä oli useampia: Fortran, C, C++, Python/C, sekä OCaml/C. Käyttöjärjestelmiä hallitsi Linux muutaman Windows-ohjelman lisäksi, ja mikroprosessoriarkkitehtuurit olivat x86 ja PowerPC. Sovellusalueita olivat pääasiassa fysiikan simulaatiot, kuten ydinfuusioplasmat, virtausmekaniikka, ydinkaskadit eli hiukkassuihkun simulointi mm. kudoksen läpi ja konfokaalimikroskopia, sekä bioinformatiikan ohjelmistot. Ohjelmat olivat kirjoittaneet fyysikot, kemistit sekä tietojenkäsittelijät ympäri Suomea, Eurooppaa ja USA:ta. Ehkä pienenä yllätyksenä tuli havainto, että joitakin koodeja oli kehitelty hyvinkin pitkään, jopa 20 vuotta, mikä näkyikin selvästi vanhimmista osista lähdekoodia. Näin vanhojen koodien jatkuva kehitys on varmaan ollut haasteellista, varsinkin kun rinnakkaiskoneet yleistyivät vasta noin 15 vuotta sitten.

2 Tavoitteet

Alusta asti oli selvää, ettemme oman resurssipulamme takia voineet lähteä rinnakkaistamaan meille lähetettyjä sekventiaalisia koodeja. Sen sijaan keskityimme jo kertaalleen rinnakkaistettujen ohjelmien parantamiseen ja tehostamiseen. Tämän lisäksi suoritimme sekä sekventiaalisten että rinnakkaistettujen koodien optimointia. Jälkimmäisessä tapauksessa tavoitteena on saada sama numeerinen tulos kuin ennen optimointia, mutta nopeammin hyödyntämällä välimuistin käyttöä sekä vektorilaskentaa. Joissakin tapauksissa koodin optimointiin saattaa sisältyä tehtävien laskutoimitusten uudelleenjärjestelyä, jolloin mikroprosessorien äärellisen liukulukulaskentatarkkuuden eiassosiatiivisuudesta seuraa, ettei 15 numeron tarkkuudella välttämättä saada samoja tuloksia kuin ennen. Joillekin koodinomistajille tämä aiheutti jonkun verran hermostuneisuutta, koska oli totuttu saamaan tietynnäköisiä numeerisia tuloksia. Meidän mielestämme ei ole syytä huoleen: jos lasketaan eri järjestyksessä saadaan usein eri tulos, mutta mikään ei viitannut siihen että alkuperäinen laskujärjestys olisi ollut numeerisesti tarkempi tai stabiilimpi.

Emme puuttuneet ongelmanasetteluun, eli emme kysyneet miksi tietyt laskut haluttiin suorittaa. Emme myöskään puuttuneet valittuun numeeriseen algoritmiin, ellei kyse ollut tarkasti modularisoidun numeerisen algoritmin korvaamisesta toisella implementaatiolla, tyypillisesti avoimen lähdekoodin kirjastorutiinilla. Tässä mielessä voi sanoa että tekemämme koodin optimointi oli lähinnä jo tehtyjen virheiden paikkaamista: laskettavat suureet ja algoritmit oli jo valittu ja pyrimme pelkästään virtaviivaistamaan laskujen suoritusta olemassa olevalle mikroprosessoriarkkitehtuurille.

3 Metodologia

Saatuamme uuden koodin sovelsimme siihen joukon perustyökaluja, joiden avulla pyrittiin hahmottamaan potentiaaliset optimointikohteet koodissa. Tämän lisäksi osoittautui hyödylliseksi tarkistaa mahdollisia muistivuotoja, joissa ohjelma ohjelmointivirheen seurauksena kirjoittaa varaamattomalle muistialueelle. Yllättävää oli, että moni käyttäjä voi elää sen kanssa, että ohjelma saattaa muistivuodon takia kaatua epäsäännöllisin väliajoin.

3.1 Koodin katselmus

Hyvin monelle tutkimukseemme osallistuneelle koodille oli ominaista, että ohjelma ajettiin omassa ympäristössä, eikä koskaan ollut ilmennyt tarvetta lähettää koodia ulkopuoliselle taholle. Tilanne oli siis uusi kun pyysimme saada lähdekoodin sekä sopivia lähtöparametreja pienehköä ajoa varten, jota käytettäisiin vertailupisteenä koodinoptimoinnin antaman nopeutuksen mittaamiseksi. Se, että ohjelmaa nyt käytettäisiin uudessa ympäristössä ilman koodinkehittäjien valvovaa silmää, aiheutti todennäköisesti pienen paineen saattaa koodi sellaiseen kuntoon, että se toimisi uudessa paikassa. Yksi merkki tästä oli usein esiintynyt ilmoitus kooditoimituksen viivästymisestä, koska oli ilmennyt ennen esiintymättömiä ongelmia.

Saatuamme koodin loimme yleensä pikaisen katseen lähdekoodiin ja sen yleiseen rakenteeseen. Tästä ilmeisen yksinkertaisesta toimenpiteestä oli usein hyötyä, sillä uudet silmäparit erottavat koodista toisia asioita kuin kehittäjien silmät. Meidän tehtävämme oli keksiä ns. tyhmiä kysymyksiä ohjelman rakenteesta ja joskus yksityiskohdistakin, ja tällä tavalla löysimme usein koodia, joka oli syntaktisesti oikein mutta toiminnallisesti väärin.

3.2 Profilointi

Ennen kuin lähdettiin optimoimaan koodeja pyrittiin paikallistamaan ne kohdat tai metodit jotka potentiaalisesti kannattaisi optimoida. Jos lähtee optimoimaan kaikkea, on aina olemassa riski, että optimointitulos on hyvä – metodin suoritusaika lyhenee – mutta sillä ei ole juurikaan vaikutusta kokonaissuoritusaikaan. Syynä tähän on yleensä se, että metodia kutsutaan vain muutama kerta suorituksen aikana ja/tai kyseisen metodin suorittamiseen menee erittäin pieni osa kokonaissuoritusajasta. Parhaiten optimoinnit purevat kun ne kohdistetaan sellaisiin metodeihin, joihin suuri osa kokonaisajasta uppoaa.

Muutamassa tapauksessa koodinomistajilla oli näppituntuma mitkä kohdat voisivat olla hyödyllisiä optimoida, kun taas toisissa ei ollut lainkaan tietoa mihin osiin kannattaisi keskittyä. Profilointityökalut, kuten *gprof*, Crayn *craypat* sekä Intelin *VTune*, eivät ota mielipiteitä tai näppituntumia huomioon, vaan antavat raakaa dataa ohjelman suoritusajasta ja muistinkäytöstä. Perusprofilointiohjelma kertoo miten paljon aikaa kuluu jokaiseen metodiin ja miten monta kertaa kyseistä metodia kutsutaan. Kehittyneemmät profilointityökalut osaavat myös kertoa miten hyvä välimuistinkäyttö oli, miten tehokkaasti laskuoperaatioita tehtiin (flops) ja rinnakkaisohjelmissa miten paljon aikaa käytetään kommunikaatioon, odotteluun sekä varsinaiseen laskentaan. Monesta ohjelmasta löytyi yksi tai muutama metodi, jossa merkittävä osa kokonaissuoritusajasta vietettiin. Yleensä kyse oli suhteellisen lyhyistä ja ytimekkäistä metodeista, joita kutsuttiin ohjelmissa erittäin monta kertaa. Jos tällaisesta metodista saa 10% tai jopa 50% suoritusajasta pois, näkyy tämän vaikutus selvästi kokonaissuoritusajassa.

Yleensä eniten kiinnostava profilointi- ja optimointikohde on suoritusaika: tuloksia halutaan nopeammin. Muistinkäyttö tulee tyypillisesti kuvioon vasta kun tämä muodostuu ongelmaksi: muisti loppuu kesken tai ohjelma kaatuu muistivuodon tai puskuriylivuodon takia. Muistivuodot ja puskuriylivuodot löytyvät yleensä muistinkäyttöanalysointityökalulla, kuten *Valgrind* [8], kunhan oppii tulkitsemaan sen tuloksia. Kokemus kertoo, että muistiongelmat kannattaa korjata saman tien kun ne ilmenevät, vaikka ne eivät haittaisikaan ohjelman suoritusta. Näin säästytään ongelmilta myöhemmin. Muistivuo-

tojen korjaaminen vähentää myös ohjelman muistinkäyttöä ja voi sallia suurempien ongelmien simulointia.

Rinnakkaistetun ohjelman muistinkäyttö voidaan kirjoittaa yksittäisen prosessorin kannalta muotoon $M = S + D$, missä M on yhden prosessorin muistinkäyttö, S prosessorin muistinkäyttö, joka ei riipu kokonaisprosessorimäärästä, ja D muistinkäyttö, joka on riippuvainen prosessorimäärästä. Jotta ohjelmaa voidaan ajaa suuremmalla prosessorimäärällä, täytyy muistinkäytön kannalta olla voimassa $D >> S$. Ideaalitilanteessa $S = 0$, jolloin muistinkäyttö prosessorissa voidaan puolittaa tuplaamalla käytettyjen prosessoreiden määrä. Profilointityökalut kertovat M:n, mutta S:n ja D:n suhde selviää vasta kun ohjelma profiloidaan eri prosessorimäärillä ja/tai parametreilla.

Tarkasteltavista ohjelmista yksi näytti profiloinnissa, että $S \gg D$. Tästä johtuen M oli lähes vakio ja melkein riippumaton prosessorimäärästä, eikä rinnakkaistetulla ohjelmalla voitu simuloida monimutkaisempaa ongelmaa kuin yhdellä prosessorilla, koska muisti loppui kesken. Syykin selvisi kooditutkiskelun jälkeen: käytettiin tiheää matriisia arvojen väliaikaistallennukseen ja matriisin koko oli valittu suoraan verrannolliseksi käytettyyn simulaatioavaruuden hilan kokoon. Vaihtamalla matriisi dynaamiseen tietorakenteeseen muistinkäyttö muuttui täysin siten, että $D \gg S$ ja hilaa oli mahdollista kasvattaa kunhan käytössä oli tarpeeksi prosessoreita.

4 Rinnakkaistaminen

Monen tieteellisen ohjelman kehitys on aloitettu yli 15 vuotta sitten, tyypillisesti Fortran77-kielellä. Nykyaikaiset rinnakkaisohjelmointikirjastot tukevat kyllä Fortran77:ä, mutta ohjelmat on alunperin suunniteltu ja implementoitu täysin sekventiaalisiksi aikakaudella, jolloin rinnakkaiskoneita ei vielä ollut yleisessä käytössä. Vuosien varrella ohjelmia on muutettu osittain rinnakkaisiksi pienin askelin rinnakkaistamalla keskeisiä silmukkarakenteita ja jakamalla simuloitava hila useammalle prosessorille koskematta oleellisesti muihin sekventiaalisiin osiin. Rinnakkaisohjelmointiparadigmaa ei siten seurattu, vaan koodia on rinnakkaistettu vain osin. Yleinen ohje rinnakkaisohjelmien kirjoittajille on suunnitella sekä algoritmit että tietorakenteet (vektorit, matriisit, struktuurit, jne.) siten, että tiedot voidaan luontevalla tavalla jakaa pienempiin yksikköihin, jotka ovat suureksi osaksi toisistaan riippumattomia. Riippumattomuus tarkoittaa, että kukin prosessori voi ainakin pääsääntöisesti tehdä laskentaa muiden prosessorien datasta riippumatta, kuitenkin siten, että välillä saatetaan lähettää suhteellisen pieniä datamääriä muutamalle toiselle prosessille.

Ohjelman osittaisenkin rinnakkaistamisen potentiaalinen hyöty on ilmeinen: suurempi laskentatyö jaetaan yhä useammalle prosessorille, joten jos rinnakkaislaskennassa tehtävä työ on verrannollinen hilapisteiden lukumäärään ja rinnakkaistaminen onnistuu hyvin (esim. hilapisteessä tehtävä laskutoimitus tarvitsee vain lähinaapuripisteiden arvoja) voimme arvioida, että rinnakkaislaskenta-aika pysyy liki vakiona mikäli ongelmakoon ja prosessorilukumäärän kaksinkertaistaminen tehdään samanaikaisesti. Tässä on oletettu hieman yksinkertaistaen, että kommunikaatio prosessorien välillä ei muodostu pullonkaulaksi. Yksi rinnakkaistamisella periaatteessa saavutettava hyöty on se, että tietty ohjelma voidaan nopeuttaa, eli tietyn kokoinen simulointi voidaan laskea nopeammin. Rinnakkaistamisen tuoma nopeutustekijä SpeedUp prosessori-

määrän funktiona lasketaan kaavasta

$$SpeedUp(N) = RunTime(1)/RunTime(N)$$

jossa RunTime(N) on ohjelman ajoaika N:llä prosessorilla.

Käytännössä harvaa ohjelmaa kuitenkaan ajetaan suuremmalla prosessorimäärällä kuin on tarpeen. Jonkinlaista minimaalista ajatusmallia edustaa havaitsemamme käytäntö, jonka mukaan simuloinnin tarvitsema muistimäärä kiinnittää käytettävien prosessorien lukumäärän.

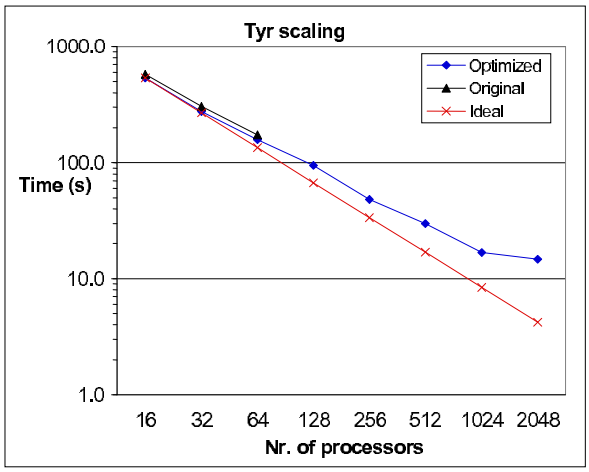
Oletetaan, että meillä on simulointiohjelma, jonka kokonaisajasta tietty osuus on sekventiaalista, kun ajetaan "pientä" ongelmaa ehkä 8 prosessorilla. Halutaan arvioida ajoaika, kun ongelman koko kasvaa ja prosessorien lukumäärä lisääntyy esimerkiksi 128:aan tai 256:een. Keskeinen kysymys ohjelman ns. skaalautuvuuden kannalta on nyt: miten käy sekventiaaliselle osalle kun simulaation koko suurenee? Jos ohjelma suunnitellaan alusta asti rinnakkaiseksi, on ainakin periaatteessa mahdollista, että sekventiaalisen osuuden ajoaika pysyy vakiona. Harva kohtaamamme ohjelma on onnistunut tässä. Sen sijaan olemme usein todenneet, että sekventiaalisen ajoajan osuus on kasvanut prosessorilukumäärän suhteessa. Tarkempi analyysi kertoo, että tiedostoihin kirjoittaminen on usein implementoitu sekventiaalisesti yhden tietyn prosessin kautta: kaikki prosessit lähettävät tuloksensa tälle prosessille, joka kokoaa tiedot oikeaan järjestykseen ja kirjoittaa välituloksia, diagnostisia arvoja, keskeytyneen ohjelmasuorituksen uudelleenkäynnistyksen tietoja tai lopputuloksia kovalevyille. Sekventiaalisen osuuden ajoaika kasvaakin suhteessa ongelman kokoon ja sitä kautta prosessien lukumäärään. Tästä seuraa, että suuremman prosessorilukumäärän tapauksessa ohjelma ei nopeudu vaan hidastuu, koska sekventiaalinen osuus kasvaa prosessorilukumäärän kasvaessa.

4.1 Triviaalisti rinnakkaistuvat

On olemassa laskutehtäviä, joiden rinnakkaistaminen on suoraviivasta ja yksinkertaista, ja niissä rinnakkaistus on onnistunut hyvin. Näihin tapauksiin kuuluvat yhteen laskentaytimeen mahtuvat ohjelmat, jotka ajetaan monta kertaa eri lähtöparametreilla suorittaen siten parametripyyhkäisyn, tai Monte Carlo -tyyppiset simuloinnit joissa sama ohjelma ajetaan monta kertaa eri satunnaislukualustuksella statistisen arvion laskemiseksi. Käytännössä yksi prosessi kerää $N-1$:n ohjelman tulokset ja suorittaa jälkiprosessoinnin. Ammattislangi kutsuu näitä ohjelmia "triviaalisti rinnakkaistuviksi" mikä saattaa kuulostaa hieman halventavalta, mutta meidän mielestämme ne edustavat hyvin tärkeää luokkaa rinnakkaisohjelmia, joille on ominaista, että ne voidaan ajaa ongelmitta erittäin suurilla prosessorimäärillä.

4.2 Prosessien välinen kommunikaatio

Rinnakkaisohjelmoinnissa pyritään aina jakamaan laskennat eri prosessorien kesken siten, ettei kaikkien prosessien tarvitse simulointiaskelten välillä vaihtaa dataa kaikkien muiden prosessien kanssa. Tällöin nimittäin N prosessia saa aikaan $N(N-1)/2$ kommunikaatiota, jolloin N:n kasvaessa yhä suurempi osuus kokonaisajasta kuluu kommunikaatioon. Kaikki tähän asti kohtaamamme ohjelmat ovat onneksi(?) olleet rakenteeltaan hilajakoon ja lähinaapurivuorovaikutukseen perustuvia. Simulointitehtävä on tällöin formuloitu tapahtuvaksi esimerkiksi kolmidimensioisella hilalla, joka on jaettu säännöllisesti pienempiin kuutiotyyppisiin osiin, ja jokainen prosessi laskee oman kuutionsa hilapisteet vaihtaen aina simulointiaskelten välillä dataa lähimpien naapuripro-



Kuva 1: Fuusiosimulaatiokoodin skaalautuvuus Cray XT4/5 koneessa

sessorien kanssa reuna-alueeseen verrannollisen määrän. Kolmessa dimensiossa lähinaapuriprosesseja on vakiomäärä, 26 kappaletta, ja kahdessa dimensiossa vain 8. Tästä seuraa että ohjelman kommunikaatiossa vaihdettava kokonaistietomäärä kasvaa suhteessa prosessorien lukumäärään eli kommunikaatio skaalautuu, jolloin on odotettavissa ettei ohjelma huku kommunikaatioon suuremmillakaan prosessorimäärillä (katso kuva 1).

Erikoisena piirteenä havaitsimme, että kommunikaation suunnittelussa on usein noudatettu ehkä hieman liioiteltuakin varovaisuutta ja suunnitelmallisuutta. Periaatteena käytettiin kuviota, jossa prosessori ensin pyyhkäisee oman hilansa yli, jonka jälkeen se lähettää tiedot laskemistaan reunapisteiden uusista arvoista ensimmäiselle naapurilleen, odottaa kuittausta vastaanottajalta eli sen lähettämää dataa ja etenee sen jälkeen seuraavalle naapurille. Varmuuden vuoksi ohjelmaan liitetään lopuksi globaali puomi (eng. *barrier)*, joka pakottaa kaikki prosessorit odottamaan kunnes jokainen prosessori on saavuttanut saman kohdan ohjelman suorituksessa. Nykyiset rinnakkaisohjelmointikirjastot tukevat asynkronista kommunikaatiota, jossa prosessori ei tarvitse naapureilta kuittausta, vaan prosessori pyyhkäisee ensin reunapisteidensä yli, lähettää nämä tiedot naapureilleen odottamatta kuittausta, pyyhkäisee tämän jälkeen kaikkien muiden pisteidensä yli ja vasta sen jälkeen tarkistaa ovatko naapuriprosessien reunapisteet saapuneet. Kaikkien uusien reuna-arvojen saavuttua prosessori voi edetä seuraavalle simulaatioaskeleelle ilman globaalia puomia.

4.3 Kuormantasaus

Tutkimissamme ohjelmissa simuloidaan usein ajassa eteenpäin siten, että kehitetään kaava, joka mallittaa miten systeemi kehittyy ajassa, jonka jälkeen edetään ajanhetkestä $t = 0$ pienin askelin Δt eteenpäin. Mikäli prosessorien simulaatio-osuus kokonaistyöstä perustuu edellä kuvattuun hilajakoon, on hilan ta-

sajaolla varmistettu, että jokainen prosessi tekee saman verran laskentaa, eikä synny tilannetta, jossa kaikki muut prosessorit odottavat eniten työtä tekevän prosessorin valmistumista. Mikäli simulaatio sisältää stokastisia piirteitä, esimerkiksi jos hilapisteiden välissä liikkuu hiukkasia, jotka voivat tilapäisesti kasaantua joihinkin hilan osiin ja toisaalta harventua muilta osin, on mahdollista että työnjako muuttuu epätasaiseksi ja useimmat prosessorit joutuvat olemaan joutilaina odottaessaan työllistettyjen prosessoreiden valmistumista. Havaintomme mukaan tällaisia työnjaon osalta epätasaisia ohjelmia esiintyy melko usein ja niiden kohdalla voi olla mielekästä toteuttaa työn uudelleenjako tietyin aikavälein. Eräässä tapauksessa nopein prosessori suoriutui työstään 10 sekunnissa hitaimman pyöriessä 30 sekuntia, jolloin kokonaisajoaika määräytyi hitaimman prosessin mukaan. Työn uudelleenjaon jälkeen, sisältäen jakoon liittyvän lisätyön, kaikkien prosessoreiden ajoaika oli 21 sekuntia. Uudelleenjaosta aiheutuva lisälaskenta oli siis pieni verrattuna kokonaisajan pienenemiseen.

4.4 Rinnakkaistiedostot

Suurten rinnakkaisajojen käsittelemät tietomäärät on helppo arvioida kertomalla prosessorien lukumäärä prosessorin muistimäärällä. Mikäli käytössä on esimerkiksi 1024 prosessoria ja jokaisella on 1 GB keskusmuistia on lopputulos 1 TB (teratavu). Simuloinnin valmistuttua halutaan mahdollisesti tallettaa tulokset, ja joskus on mielekästä esimerkiksi simuloinnin edistymisen valvomiseksi tai simulointivideota varten tallettaa myös välituloksia, esimerkiksi jokaista simuloinnin aika-askelta kohden. Sanonnalle "Teraflopsit tuottavat teratavuja" löytyy tänään katetta eritoten plasmafuusion simuloinneissa ja tilanne mutkistuu tulevaisuudessa yhä suurempien simulointien ollessa mahdollisia. Empiirisesti havaitsimme rinnakkaisohjelmien kohdalla, että tiedostosta lukeminen ja siihen kirjoittaminen usein toteutetaan keräämällä tieto pieninä osina yhteen prosessiin, joka kirjoittaa tiedot tiedostoon kovalevylle. Tässä on oiva esimerkki siitä miten rinnakkasiohjelmaa ei ole suunniteltu loppuun asti rinnakkaiseksi, vaan siihen on jäänyt sekventiaalinen osuus, joka käytännössä määrää ajoajan. Eräässä tapauksessa saimme ohjelman, jossa itse simulointilaskenta 1024 prosessorilla kesti 20 sekuntia, mutta (väli)tuloksen kirjoittaminen kovalevylle 180 sekuntia, jolloin 1023 prosessoria oli joutilaana 90 % ajoajasta. Standardirinnakkaiskirjastoissa on jo pitkään tuettu tiedostojen rinnakkaislukemista ja -kirjoittamista, ja nämä ovatkin suositeltavia mikäli vain supertietokoneeseen tai laskentaklusteriin on rakennettu rinnakkainen tiedostojärjestelmä. Lisäksi on olemassa näiden rutiinien päälle rakennettuja kirjastoja, kuten esimerkiksi HDF5 (Hierarchical Data Format [9]), jotka vielä entisestään yksinkertaistavat rinnakkaista tiedostonkäsittelyä tehden sen melkein normaaliin tiedostonkäsittelyyn verrattavaksi. Esimerkkitapauksessa tiedostoon kirjoittaminen lyheni 10 sekuntiin.

4.5 Kirjastorutiinien hyödyntäminen

Ohjelmien rinnakkaistuksen perusrakenteen lisäksi pyrimme myös selvittämään mahdollisuuksia käyttää rinnakkaisohjelmointikirjastojen tarjoamaa tukea. Rinnakkaisohjelmointistandardi Message Passing Interface MPI [10] perustuu laskentanoodien väliseen viestinvälitykseen, jossa tavuja lähetetään ja vastaanotetaan sovitun kaavan mukaisesti. MPI-kirjastojen kehittyessä niihin on rakennettu yhä enemmän toiminnallisuutta erityyppisille rinnakkaisohjelmissa usein

esiintyville algoritmeille, joita on siis turha lähteä rakentamaan itse.

Yksi esimerkki on ns. *all-reduce* operaatio, jossa jokaisesta prosessorista kootaan tietty määrä dataa yhteen prosessoriin, ja tämä prosessori suorittaa datalle jonkun operaation, esimerkiksi ynnää kaikki arvot yhteen, ja lopputulos kommunikoidaan takaisin kaikille prosesseille. Sen sijaan, että lähdemme rakentamaan rutiinia, jossa kommunikoidaan rinnakkaiskirjaston lähetys- ja vastaanottokutsuilla, voimme käyttää hyväksi yksinkertaista *MPI_All_reduce*-tyyppistä kutsua, joka hoitaa kommunikaation mitattavasti tehokkaammin kuin omat rutiinit. Toinen esimerkki liittyy tilanteeseen, jossa esimerkiksi kolmidimensioista hilajakoa halutaan kuvata prosesseille: voimme tietysti itse päätellä miten kannattaa jakaa alihilat eri prosesseille, mutta MPI:stä löytyy tukea joka helpottaa tätä työtä.

5 Koodin optimointi

Varsinaiset tekniikat joiden avulla pyrittiin nopeuttamaan ohjelmia olivat välimuistin tehokas hyödyntäminen sekä SSE-vektorointi.

5.1 Välimuistin optimointi

Välimuistin tarkoitus on luoda prosessorille mielikuva, jonka mukaan tietokoneen keskusmuisti on nopea. Jos ohjelma tarvitsee tavuja, jotka voidaan lukea välimuistista, on hakuaika noin 3 kellonjaksoa, kun taas keskusmuistista haku kestää pyöreästi 100 kellonjaksoa. Tämä mielikuva saadaan aikaiseksi siten, että uusia arvoja ladataan keskusmuistista välimuistiin sillä aikaa kun prosessori tekee laskentaa edellisellä kerralla välimuistiin luettujen arvojen kanssa. Välimuistiin kirjoitetaan ja sieltä luetaan peräkkäisiä tavuja välimuistirivin verran, tyypillisesti 64 tavua eli kahdeksan kaksoistarkkuuden liukuluvun verran. Välimuistin käytön optimointi voidaan nyt tiivistää lauseeseen "Käytä tehokkaasti hyväksesi jo välimuistiin luettuja arvoja äläkä hyppää mielivaltaisella tavalla ympäri keskusmuistia lukien yksittäisiä arvoja sieltä täältä". Jos kuitenkin tarvittavat tavut ovat kaukana toisistaan skaalalla 64 tavua kannattaa yrittää järjestää tietorakenteet uudelleen siten, että tarvittavat arvot ovat peräkkäin muistissa. Tätä periaatetta käyttäen ryhmämme onnistui nopeuttamaan yleisesti käytössä olevaa proteiinisekvensointiohjelmaa PSI-BLAST [4]. Proteiini koostuu pitkästä nauhasta aminohappoja, joita löytyy 20 eri tyyppiä. Sekvensoinnissa pyritään asettamaan kaksi proteiinia rinnakkain siten, että niillä on mahdollisimman monta osumaa eli kaksi samaa tai melkein samaa aminohappoa vierekkäin. PSI-BLAST-ohjelmassa proteiinin jokaista aminohappoa varten käytettiin 20 tavun kokoista tietuetta, jossa on ensimmäisenä jäsenenä yhden tavun pituinen aminohapon tyyppiä edustava kirjain *letter*, toisena jäsenenä kahdeksan tavua pitkä aminohapon satunnainen osumatodennäköisyys *e_value*, jne. Tietueet oli järjestetty muistiin taulukkoon.

Kuitenkin suurin osa ohjelman ajoajasta kului peräkkäisten aminohappotietueiden läpikäymiseen etsien aminohapon tyyppiä. Käytännössä siis välimuistiin luettiin jokaisella lukukerralla 64 tavua, mutta niistä käytettiin vain joka 20:s, eli noin 3 tavua, ennen kuin seuraava välimuistirivi piti lukea. Järjestimme tietorakenteet uudelleen siten, että tietuetaulukon sijaan meillä oli yksi taulukko jokaista alkuperäisen tietueen jäsentä kohden (katso kuva 2). Lukemalla nyt pelkästään tyyppitietue välimuistiin voidaan käyttää hyväksi jokaista luettua tavua, ja koko ohjelmaa voidaan nopeuttaa jopa tekijällä kolme [5].

```
typedef struct posDesc {
  signed char letter;
  unsigned char used;
  double e_value;
  int leftExtent;
  int rightExtent;
} posDesc;
```

```
typedef struct posDesc {
  signed char *letter;
  unsigned char *used;
  double *e_value;
  int *leftExtent;
  int *rightExtent;
} posDesc;
```

Kuva 2: Alkuperäinen (vasen) ja optimoitu (oikea) PSIBlast-ohjelman tietue.

5.2 Tietorakenteet

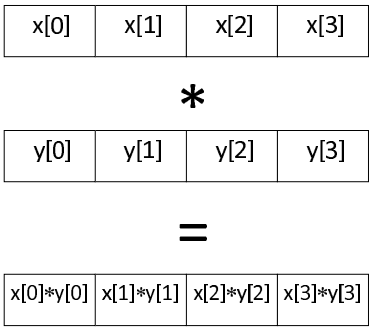
Välimuistin optimointi on käytännössä sellaisten tietorakenteiden käyttöä, jotka ovat sopusoinnussa välimuistin toiminnan kanssa. Tyypilliset ohjelmissa esiintyvät tietorakenteet ovat vektori ja matriisi, mutta joskus ohjelmissa käytetään yksinkertaisia tietueita tai linkitettyjä listoja. Suoraviivaisesta rakenteestaan huolimatta matriisit voidaan tietoteknisesti esittää eri tavoin. Esimerkkinä mainittakoon usein vastaan tullut tilanne, jossa kolmidimensioista hilaa (sivun pituus N) on esitetty oikeaoppisesti kolmidimensioisena matriisina, mutta laskennan jossakin vaiheessa muodostetaan hilapisteiden perusteella lineaarinen matriisiyhtälöryhmä, jonka matriisi on erittäin harva ja kokoa N^3. Tällöin kannattaa ilman muuta siirtyä käyttämään harvaa matriisia ja valita harvaa matriisia hyödyntävä ratkaisualgoritmi. Harvan matriisin esitystapoja on useita, joten esitystavan valinnassa täytyy suunnitella koko laskentaketju etukäteen, jotta se menee kivuitta läpi.

Toinen maininnan arvoinen tietorakenne liittyy listoihin, joista halutaan etsiä arvoja tietyin ehdoin, esimerkiksi löytyykö annettu arvo listasta tai mikä on lähinnä pienempi arvo. Mikäli lista on hieman pitempi, enemmän kuin kymmeniä elementejä, kannattaa esimerkiksi järjestää listan arvot suuruusjärjestykseen ja käyttää binäärihakua arvojen etsimiseen, tai ryhmitellä listan arvot hash-tyyppisillä avaimilla.

5.3 Vektorointi

Moderneista prosessoreista on jo pitkään löytynyt ns. vektoroituja multimediakäskyjä, Streaming SIMD Extensions SSE, (SIMD = Single Instruction Multiple Data), joissa käytetään normaalia pitempiä – tyypillisesti 128-bittisiä – rekistereitä. Näihin rekistereihin voidaan tuoda kaksi 64-bittistä kaksoistarkkuuden liukulukua tai vaihtoehtoisesti neljä 32-bittistä yksinkertaisen tarkkuuden liukulukua. Laskennallinen etu on saavutettavissa siten, että suoritetaan aritmeettisia toimintoja näiden rekistereiden kesken tätä varten olevilla SSE-käskyillä, jolloin yhden käskyn suorittaminen saa aikaan kahden (tai yksinkertaisen tarkkuuden tapauksessa neljän) samanaikaisen liukulukulaskutoimituksen suorittamisen. Teoreettisesti laskien kaksoistarkkuuden aritmetiikka voidaan nopeuttaa tekijällä 2 (katso kuva 3).

Vastaanottamiemme ohjelmien joukossa ei ollut yhtäkään, jossa x86-arkkitehtuurin SSE-multimediarekistereitä olisi käytetty tehokkaasti hyväksi! Ohjelmien tehokkuuden kannalta vektorilaskenta SSE-rekistereillä oli täysin käyttämätön resurssi. Lähdekoodeissa esiintyi monesti useampikin silmukka, joi-



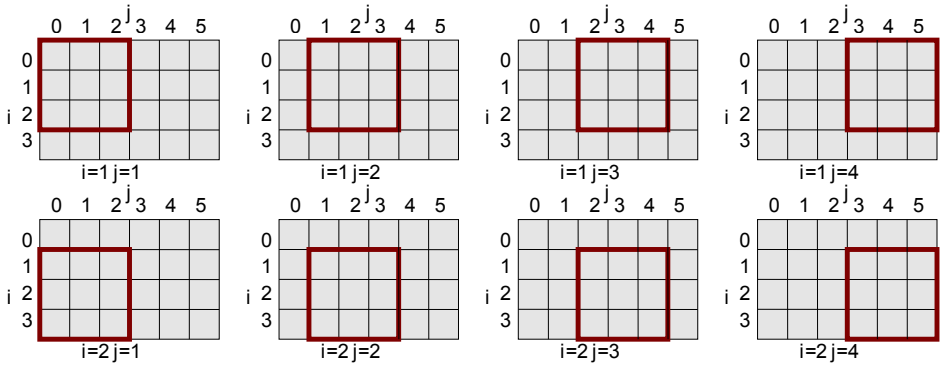
Kuva 3: Neljän yksinkertaisen tarkkuuden liukuluvun samanaikainen kertolasku SSE-vektoroinnilla.

den olisi pitänyt pystyä hyödyntämään SSE-vektorointia, mutta jostakin syystä kääntäjät eivät tuottaneet vektoroitua koodia. Myös kääntäjän tekemän periaatteessa suoraviivaisesti vektoroitavissa olevan silmukan tekemän konekielen läpikäynti osoitti, ettei kääntäjä tehnyt vektorikäskyjä ollenkaan. Tilanteen ymmärtämiseksi kirjoitimme plasmasimuloinneissa usein esiintyvälle ns. Arakawa-menetelmälle [6] konekieliversion, jossa eksplisiittisesti käytimme pelkästään vektorikäskyjä koko SSE-rekisterille. Jo 1960-luvulla huomattiin, että virtausmekaniikan simuloinneissa esiintyy numeerista epästabiilisuutta, jonka seurauksena pyörteiden kineettinen energia kasvaa rajattomasti. Keskeinen laskettava suure on kahden virtauskentän ζ ja ψ 2-dimensioinen Jakobiaani

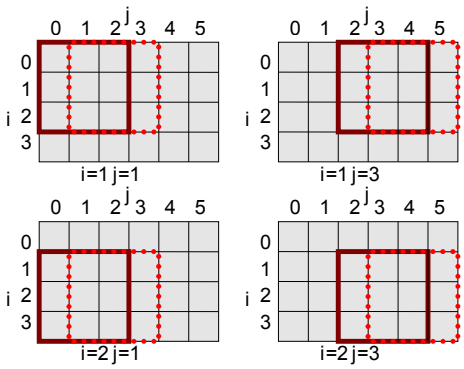
$$J(\zeta,\psi) = \frac{\partial\zeta}{\partial x}\frac{\partial\psi}{\partial y} - \frac{\partial\zeta}{\partial y}\frac{\partial\psi}{\partial x} \qquad (1)$$

Arakawa osoitti, että ensimmäisen kertaluvun osittaisderivaattojen suoraviivaisessa neljän pisteen diskretoinnissa (lähinaapurit x- ja y-suunnassa) systeemin kokonaisenergia ei säily. Sen sijaan hän johti Jakobiaanin diskretoinnille kokonaisenergian säilyttävän muodon, jossa on mukana yhdeksän pistettä kummastakin virtauskentästä muodostaen 3x3 matriisin laskettavan Jakobiaanin elementin ympärille. Kuvissa 4 ja 5 on esitetty miten paikka-avaruus on diskretoitu ja Arakawan menetelmällä pyyhitään virtauskentän 2-dimensioisen matriisin yli, kuvassa 4 ilman SSE-vektorointia ja kuvassa 5 SSE-vektoroinnilla.

Koodin optimoinnin tulos oli odotettu: ohjelma nopeutui tekijällä 1,8. Mistä syystä sitten kääntäjä ei suostu vektoroimaan Arakawa-koodia? Tälle on itse asiassa olemassa hyvä syy. Ajatellaan tapausta jossa Arakawa-rutiini toteutetaan aliohjelmakutsulla, jonka lähtöarvoina on kaksi matriisia ($\zeta_{i,j}$ ja $\psi_{i,j}$) sekä tulosmatriisina Jakobiaani $J_{i,j}(\zeta,\psi)$. Jos alirutiini kutsutaan siten, että toinen lähtömatriiseista ja tulosmatriisi ovatkin sama matriisi(!), tulokset tulevat olemaan aivan erilaiset riippuen siitä onko aliohjelma vektoroitu tai ei, jolloin kääntäjä tekee päätöksen olla vektoroimatta aliohjelmaa. Jos kuitenkin käyttäjä ilmoittaa eksplisiittisesti että erinimiset osoittimet, kuten matriisit tai vektorit, varmasti osoittavat eri muistipaikkoihin, onnistuu vektorointi hyvin. Esimerkkinä käyköön Intelin *icc*-kääntäjä [7], jolle ilmoitetaan



Kuva 4: Iterointi 6x4-kokoisen gridin yli käyttäen Arakawan 3x3 matriisia.



Kuva 5: Arakawan menetelmän iterointi 6x4-kokoisen gridin yli käyttäen SSE-vektorointia. Yhtenäinen viiva ja katkoviiva edustavat kaksoistarkkuuden lukuja, jotka sijoittuvat SSE-rekisterin vasemmalle ja oikealle puolelle.

joko komentoriviltä *icc -fno-alias* tai *restrict* avainsanalla parametreille aliohjelmakutsussa, että tarkoitamme nimenomaan eri matriiseja, jolloin kääntäjä osaa riittävän selkeissä tapauksissa generoida erittäin käyttökelpoista vektoroitua koodia. Muut C-kääntäjät, kuten GNU [11], Pathscale [12] ja Portlandin PGI [13], pystyät tuottmaaan yhtä tehokasta vektoroitua koodia vain hyvin yksinkertasille lausekkeille.

Lähitulevaisuudessa vektorirekisterien pituus kaksinkertaistuu 256 bittiin, jolloin yhteen rekisteriin mahtuu kerralla neljä kaksoistarkkuuden liukulukua avaten siten mahdollisuuden yhä suuremmille nopeutuksille.

6 Tuloksia

Mitkä olivat havaintomme rinnakkaisohjelmista ja sekventiaalisista ohjelmista joita ajetaan supertietokoneissa ympäri Eurooppaa ja myös maailmanlaajuisesti nimenomaan rinnakkaisohjelmoinnin sekä koodin optimoinnin kannalta? Yleistoteamus on positiivinen: vielä voi parantaa ohjelmia käyttämään paremmin hyväksi prosessorien tehoja. Usein sekventiaalinen ohjelma voidaan rinnakkaistaa ja tällä tavoin ohjelma voidaan nopeuttaa tekijällä 10-100. Jo rinnakkaistettujen ohjelmien uudelleensuunnittelu merkitsi usein, että ohjelmat, jotka ennen pyörivät korkeintaan 16 tai 32 prosessorilla, voidaan tarkemmin suunnitellun rinnakkaistamisen jälkeen ajaa jopa 1024 tai 2048 prosessorilla. Rinnakkainen tiedostonkäsittely on mielestämme se osa, jossa voi suhteellisen järkevällä ohjelmointipanostuksella saavuttaa merkittävimmät nopeutukset tapauksissa, joissa ohjelma tuottaa paljon dataa kovalevylle. Jos ohjelmassa on keskeisiä rutiineja, jotka voidaan vektorisoida SSE-tyyppisillä käskyillä, voidaan hyöty usein saada irti pienellä muutoksella lähdekoodiin tai kääntäjän asetuksilla olettaen, että erinimiset tietorakenteet todella ovat eri paikoissa muistissa. Koodin optimointi yleensäkin nopeuttaa ajoajat noin tekijällä 2, parhaassa tapauksessa 3, ja mikäli ajoaika alunperin on ollut tunteja tai päiviä voi nopeutus tekijällä kaksi olla hyvinkin kouriintuntuva parannus.

Viitteet

1. http://www.top500.org
2. http://www.csc.fi/csc/julkaisut/oppaat/-pdfs/finhpc.pdf/download
3. Pär Strand, Bernard Guillerminet, Frederic Imbeaux, Rui Coelho, David Coster, Lars-Göran Eriksson, Francesco Iannone, Gabriele Manduchi, Isabel Campos, Matthieu Haefele, Eric Sonnedrücker, Adrian Jackson, Jan Westerholm, Marcin Plociennik ja Michal Owsiak: *A European Infrastructure for Fusion Simulations*, toim. Marco Danelutto, Julien Bourgeois and Tom Gross: PDP 2010 — Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-Based Processing, Pisa, 17.-19. helmikuuta, 2010, s. 460–467.
4. http://blast.ncbi.nlm.nih.gov/Blast.cgi
5. Mats Aspnäs, Kimmo Mattila, Kristoffer Osowski ja Jan Westerholm, *Code Optimization of the Subroutine to Remove Near Identical Matches in the Sequence Database Homology Search Tool PSI-BLAST*, Journal of Computational Biology, vol. 17, s. 1-5, 2010.
6. A. Arakawa, *Computational design for long-term numerical integration of the equations of fluid motion: two-dimensional incompressible flow. Part I*, Journal of Computational Physics, Vol. 1, s. 119-143, 1966
7. http://software.intel.com/en-us/intel-compilers/
8. http://valgrind.org
9. http://www.hdfgroup.org
10. http://www.mpi-forum.org
11. http://gcc.gnu.org
12. http://www.pathscale.com
13. http://www.pgroup.com