hyväksymispäivä arvosana

arvostelija

Kuorman hajauttaminen hajautetussa järjestelmässä

Teemu Rinta-aho

Helsinki 27.1.2000

Tieteellisen kirjoittamisen kurssin tutkielma

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Kuorman hajauttaminen hajautetussa järjestelmässä

Teemu Rinta-aho

Tieteellisen kirjoittamisen kurssin tutkielma
Tietojenkäsittelytieteen laitos
Helsingin yliopisto
27.1.2000, 23 sivua

Kuorman hajautus on keino nostaa hajautetun järjestelmän kokonaissuorituskykyä. Tätä tarkoitusta varten on olemassa erilaisia algoritmeja, jotka poikkeavat toisistaan niin toteutettavuudessa kuin suorituskyvyssä. Tässä kirjoitelmassa luodaan katsaus tämän tutkimusalueen perusteisiin sekä eräisiin käytännön sovelluksiin.

Aiheluokat (Computing Reviews 1998): C.2.4, D.4.1

Avainsanat: kuorman hajautus, kuorman tasapainotus, kuorman jakaminen, hajautetut järjestelmät

Sisältö

1 Johdanto

Viimeisten vuosikymmenien aikana tietojenkäsittely-ympäristöt ovat vaihtuneet keskuskonemalleista hajautetuiksi järjestelmiksi, jotka muodostuvat tietoliikenneverkon välityksellä toisiinsa yhteydessä olevista henkilökohtaisista työasemista. Nämä hajautetut järjestelmät tarjoavat paremman suorituskyvyn, luotettavuuden ja laajennettavuuden kuin edeltäjänsä.

Jokaisella käyttäjällä on hallussaan vuosi vuodelta yhä suurempi tietojenkäsittelykapasiteetti. Suurin osa tästä kapasiteetista jää kuitenkin käyttämättä [ML91]. Osa koneista voi olla suuren osan ajasta käyttämättöminä, mutta toisaalta on myös käyttäjiä, jotka hyötyisivät suorituskyvyn lisääntymisestä. Tällaiset käyttäjät voivat esimerkiksi kääntää isoja ohjelmistoja, simuloida neuroverkkoja, evaluoida kryptoalgoritmien tehokkuutta tai suorittaa muuta vaativaa laskentaa. Prosessointiteho nykyaikaisissa verkoissa on siis melko tasaisesti jakautunut, mutta käyttäjien tarpeet välttämättä eivät. Koska prosessointiteho on hajautettu, täytyy myös tehtäviä hajauttaa parhaan mahdollisen suorituskyvyn saamiseksi.

Tässä kirjoitelmassa esitellään tutkimustuloksia ja ratkaisuja kuormituksen hajautukseen perustuvasta hajautettujen järjestelmien suorituskyvyn kasvattamisesta. Kirjoitelman toisessa luvussa luodaan katsaus hajautettuihin järjestelmiin yleensä, suorituskyvyn käsitteeseen sekä sen erilaisiin arviointitapoihin. Luvussa kolme käsitellään keinoja suorituskyvyn parantamiseen, eli erilaisia kuormanhajautusalgoritmejä. Luvuissa 4 – 6 esitellään esimerkkijärjestelmiä ja miten ne pyrkivät nostamaan hajautetun järjestelmän suorituskykyä, sekä niiden yhteyttä teoriaan.

2 Hajautetut järjestelmät

Nykyaikaiset tietojenkäsittelyjärjestelmät muodostuvat tietoverkon välityksellä toisiinsa yhteydessä olevista henkilökohtaisista työasemista sekä palvelimista. Tällaisten järjestelmien sekä niissä toimivien ohjelmistojen muodostamaa kokonaisuutta sanotaan hajautetuksi järjestelmäksi. Hajautettujen järjestelmien tavoiteltavia ominaisuuksia ovat resurssien jakaminen (sharing), avoimuus (openness), rinnakkaisuus (concurrency), skaalautuvuus (scalability), virheensieto (fault tolerance) sekä mahdollisimman suuri tuntumattomuus (transparency) [CDK94].

Hajautetut järjestelmät antavat tietokoneille mahdollisuuden koordinoida tehtäviään sekä jakaa resurssejaan – laitteistoaan, ohjelmistojaan sekä tietojaan muillekin kuin paikalliselle käyttäjälle. Erityisesti henkilökohtaisten tietokoneiden jatkuva suorituskyvyn nousu on johtanut painopisteen siirtymisen keskuskoneista hajautettuihin järjestelmiin.

Hajautetut järjestelmät muodostuvat autonomisista solmuista, eli henkilökohtaisista työasemista sekä erityisistä palvelimista. Nykyiset moniajokäyttöjärjestelmät, jollainen jokaisessa näistä solmuista sijaitsee, suorittavat prosesseja (process). Jokaista tietojenkäsittelytehtävää (task) suorittamaan tarvitaan prosessi, yksi tai useampia. Koska prosesseja voi olla yhtäaikaisessa suorituksessa useita, mutta prosessoreita on useimmissa tapauksissa käytettävissä vain yksi, joutuvat prosessit jakamaan prosessoriajan: jokainen prosessi saa vuorollaan tietyn aikaviipaleen käyttöönsä. Erityinen paikallinen käyttöjärjestelmän osa, skeduloija, vaihtaa prosesseja prosessorille ja takaisin työjonoon siten, että kaikki prosessit etenevät pienen ajan kerrallaan. Käyttäjälle tämä näkyy kuin prosessori suorittaisi useita tehtäviä rinnakkain. Tätä skedulointia sanotaan paikalliseksi skeduloinniksi [CDK94].

Kuorman hajautus on parhaimmillaan järjestelmänlaajuista skedulointia, missä erityiset hajautetut algoritmit huolehtivat koko järjestelmässä prosessien mahdollisimman tehokkaasta etenemisestä (katso luku 3). Miten läheistä yhteistyötä paikallinen skeduloija tekee näiden hajautettujen algoritmien kanssa riippuu toteutuksesta – onko käyttöjärjestelmää suunniteltaessa otettu huomioon kuorman hajautusta vai onko kuorman hajautus lisätty siihen jälkeenpäin erillisenä sovelluksena. Esimerkkejä kuorman hajautuksen toteuttamisesta käyttöjärjestelmään ovat mm. Amoeba [TvRvSS90] sekä Plan9 [Tho95].

2.1 Suorituskykyvyn parantaminen

Suorituskyvyn parantamisella tarkoitetaan yleensä keskimääräisen vasteajan pienentämistä [SKS92]. Keskimääräinen vasteaika kuvaa todennäköisintä aikaa joka kuluu tehtävän käynnistymisen ja sen päättymisen välillä tietyssä järjestelmässä. Tehtävän vasteaika riippuu [Gos91]:

- aikariippuvaisista tekijöistä, kuten sen käyttämistä resursseista, käytetyistä komennoista, prosessorilla suorituksessa olevien muiden prosessien määrästä, sekä

- prosessoririippuvaisista tekijöistä, kuten tietokoneen arkkitehtuurista, nopeudesta sekä käyttöjärjestelmän rakenteesta.

Vasteajan suuruus vaikuttaa suoraan kaikkiin käyttäjiin. Mitä nopeammin tietokone suorittaa tehtäviä, sitä miellyttävämmäksi sen käyttäminen koetaan. Toisaalta isojen eräajotyyppisten töiden (sääennuste, ydinsimulaatio) vasteajan pienentäminen vaikuttaa näkyvästi työn tuottavuuteen.

2.2 Suorituskyvyn arviointi

Ennakoitavissa olevan vasteajan arvioimista varten tarvitaan jokin kuormitusindeksi. Kuormitusindeksi siis kuvaa tiettyyn verkon solmuun sijoitetun prosessin odotettavissa olevaa läpimenoaikaa, eli aikaa mikä kuluu prosessin suorituksen alkamisen ja päättymisen välillä. Kuormitusindeksit ovat kuorman hajauttamisen kannalta tärkeimmät järjestelmän solmujen tiloja kuvaavat mittarit. Järjestelmän tilan tunteminen auttaa useita kuormanjakoalgoritmejä toimimaan tehokkaasti (katso luku 3.2.2).

Kuormitusindeksinä voidaan käyttää prosessorilla suorituksessa olevien prosessien muodostaman jonon, eli työjonon, pituutta, keskimääräistä työjonon pituutta, vapaan muistin määrää jne. Tutkijat ovat huomanneet huomattavia eroja eri kuormitusindeksien välillä, ja sen että yksinkertaisimmat indeksit ovat osoittautuneet tehokkaimmiksi. Monimutkaisemmilla indekseillä laskenta-aikaa kuluu indeksin arvon selvittämiseen niin paljon, että se kuluttaa kaiken tarkemman tiedon mukanaan tuoman hyödyn. Erääksi tehokkaimmista tutkituista indekseistä on havaittu prosessorin työjonon pituus [Kun91].

3 Kuorman hajautus

Kuorman hajauttamisalgoritmit (load distributing) jaetaan kahteen luokkaan: kuormanjako- (load sharing) sekä kuorman tasapainotusalgoritmeihin (load balancing). Kuormanjakoalgoritmit siirtävät tehtäviä suoritukseen muuten käyttämättömille työasemille, kun taas kuorman tasapainotusalgoritmit pyrkivät tasoittamaan järjestelmän kuorman kaikkien järjestelmässä olevien tietokoneiden kesken. Suurin ero näiden kahden algoritmiluokan välillä on, että ensimmäisessä prosesseja siirretään vain muuten käyttämättömille koneille, kun taas jälkimmäisessä prosesseja siirretään myös muutenkin käytössä oleville koneille [Gos91].

Kuormanjakoalgoritmeja käyttäviä järjestelmiä voidaan kuvata järjestelmiksi jotka "metsästävät käyttämättömiä prosessoreita" [Gos91]. Nämä järjestelmät pyrkivät kunnioittamaan käyttäjän omistussuhdetta omaan työasemaansa siten, että niitä ei käytetä kuorman jakamiseen muulloin kuin omistaja ei itse käytä konettaan. Nämä järjestelmät toimivat etäajoperiaatteella, eli prosessi lähetetään etäsuoritukseen jos järjestelmästä löytyy sopiva käyttämätön kone. Prosessia ei siirretä enää muualle paitsi jos koneella ilmenee paikallista aktiviteettia, jolloin useimmat järjestelmät joko pysäyttävät prosessin suorituksen kunnes kone jälleen vapautuu, tai lopettavat prosessin suorituksen kokonaan.

Kuorman jakaminen pyrkii lähinnä antamaan käyttäjälle mahdollisuuden lähettää prosessi suoritettavaksi muualle, ja näin parantaa yksittäisten käyttäjien mahdollisuuksia ja suorituskykyä. Sen sijaan kuorman jakaminen ei pyri näkemään koko järjestelmää kokonaisuutena, eikä se siis pyri optimoimaan koko hajautetun järjestelmän suorituskykyä.

Kuorman tasapainotusalgoritmit pyrkivät nostamaan koko järjestelmän suorituskykyä tasaamalla kuormitusta kaikkien, myös käytössä olevien tietokoneiden kesken. Päinvastoin kuin kuorman jakamista, kuorman tasapainottamista voisi luonnehtia "virtuaalisen moniprosessorikoneen" skeduloimiseksi. Kuorman tasapainotuksen näkökulma hajautettuun järjestelmään on siis laajempi – ei tehdä eroa "lokaalin" prosessorin ja tiettyjen verkosta löytyvien resurssien, "etäprosessoreiden" välillä, kuten kuorman jakamisessa. Kuorman tasapainotusalgoritmien täytyy siten tietää järjestelmästä enemmän. Esimerkiksi yksittäisen työaseman kuormitustilanne täytyy kuvata tarkemmin kuin että onko se käyttössä vai ei.

Algoritmit voivat olla joko täysin hajautettuja tai niillä voi olla keskitettyjä komponentteja. Esimerkiksi järjestelmän tilaa voidaan pitää yllä täysin keskitetysti, täysin hajautetusti tai jollain näiden yhdistelmällä. Keskitetyt komponentit tuovat algoritmeihin heikkouksia – toiminta voi keskeytyä komponentin vikaantuessa tai hidastua jos siitä tulee järjestelmän pullonkaula.

3.1 Algoritmien komponentit

Kuorman hajautusalgoritmi joutuu ratkaisemaan mm. seuraavia ongelmia: milloin prosessi siirretään, mikä prosessi siirretään, minne tai mistä prosessi siirretään sekä mitä tietoa kerätään näiden päätösten tekemiseksi. Useimmat kirjallisuudessa esitellyistä kuorman hajautusalgoritmeista koostuvat neljästä eri menettelytavasta: siirtomenettelystä, valintamenettelystä, sijoitusmenettelystä sekä informaatiomenettelystä, jotka pyrkivät vastaamaan näihin ongelmiin [SKS92]. Lisäksi jotkut algoritmit siirtävät myös suorituksensa aloittaneita, "vanhoja", prosesseja, kun taas toiset siirtävät vain uusia prosesseja (katso luku 3.1.5 sivulla 7).

3.1.1 Siirtomenettely

Siirtomenettely (transfer policy) määrää onko työasema sopivassa tilassa lähettämään prosessin suoritettavaksi jollekin toiselle työasemalle, tai vastaanottamaan prosessin suoritettavaksi. Useat ehdotetut siirtomenettelyt perustuvat jonkin kynnysarvon (treshold) käyttöön. Kynnysarvot ilmoitetaan kuormituksen asteina, joita kuvaavat erilaiset valitun kuormitusindeksin arvot. Kun prosessorille saapuu uusi prosessi suoritukseen, siirtomenettely määrää koneen lähettäväksi osapuoleksi, jos kynnysarvo T1 ylittyy. Vastaavasti jos kynnysarvo T2 alittuu, koneesta voi tulla siirtomenettelyn mukaan vastaanottaja. Algoritmista riippuen T1:llä ja T2:lla voi olla sama tai eri arvo [SKS92].

Vaihtoehtona kynnysarvon käyttämiselle ovat mm. suhteelliset siirtomenettelyt. Suhteelliset siirtomenettelyt vertaavat työaseman kuormaa muiden verkossa olevien työasemien kuormaan ja tekevät päätöksensä sen perusteella.

3.1.2 Valintamenettely

Valintamenettelyn (selection policy) tehtävänä on valita lähetettävä prosessi, jos koneesta on siirtomenettelyn perusteella tullut lähettäjä. Riippuen siirrettävän prosessin valintaperiaatteesta (katso luku 3.1.5) tämä voi olla joko jo suorituksessa oleva prosessi, tai uusi prosessi, esimerkiksi sama joka aiheutti työaseman siirtymisen lähettäjätilaan. Valintamenettelyn on otettava useita muuttujia huomioon valitessaan siirrettävää prosessia: ylimääräinen siirrosta aiheutuva työ tulisi olla mahdollisiman vähäistä, prosessin tulisi olla riittävän "pitkäikäinen", että sen siirrosta saatava hyöty riittää kattamaan aiheutuvan ylimääräisen työn kustannukset. Lisäksi esimerkiksi sijainnista riippuvaisten systeemikutsujen määrän tulisi olla mahdollisimman vähäinen [SKS92]. Sijainnista riippuvainen systeemikutsu voi olla esimerkiksi käyttäjän hiiren sijainnin lukeminen tai vaikkapa äänikortin käyttäminen, jotka pitää edelleen suorittaa sillä koneella missä prosessi on luotu.

3.1.3 Sijoitusmenttely

Sijoitusmenettelyn (location policy) tehtävänä on löytää sopiva vastaanottaja tai lähettäjä, kun kone on joutunut joko lähettäjän tai vastaanottajan rooliin. Sijoitusmenettely käyttää hyväkseen niitä tietoja mitä informaatiomenettelyn (katso luku 3.1.4) mukaisesti kerätään. Usein käytetty hajautettu menettelytapa on löytää sopiva "siirtokumppani" on yksinkertaisesti kysellä (poll) muilta järjestelmän koneilta missä tilassa ne ovat. Kysely voidaan toteuttaa joko sarjassa (lähettämällä oma viestinsä jokaiselle solmulle) tai rinnan (esimerkiksi monilähetyksellä (multicast)). Jos algoritmissa ei käytetä monilähetystä kyselyn suorittamiseen, solmujen tiloista voidaan pitää kirjaa jonkinlaisella tilavektorilla josta voidaan nopeasti tarkistaa mitkä olivat eri solmujen tilat edellisen kyselyn aikana. Näin voidaan nostaa todennäköisyyttä löytää oikeanlainen "siirtokumppani" jatkossa mahdollisimman pienellä viestimäärällä, sillä ainakin alikuormitetut koneet ovat jatkossakin mitä todennäköisimmin alikuormitettuja [ML91].

Keskitetyssä menettelytavassa lähettäjä tai vastaanottaja ottaa yhteyttä erityiseen verkon solmuun joka on nimetty koordinaattoriksi. Koordinaattori kerää tietoa järjestelmästä informaatiomenettelyn mukaisesti (katso luku 3.1.4 sivulla 6), ja siirtomenettely käyttää tätä tietoa koordinaattorissa valitakseen sopivan vastaanottajan tai lähettäjän [SKS92]. Tämä on varsin yleinen menettelytapa käytännön järjestelmissä (katso luvut 4 ja 5 sivuilla 11 ja 13).

3.1.4 Informaatiomenettely

Informaatiomenettely (information policy) määrittelee mitä järjestelmän tilaan liittyvää tietoa, koska, ja mistä on kerättävä. Informaatiomenettelyt jakautuvat kolmeen luokkaan [SKS92].

- *"Tarpeen mukaan"–menettelytapojen* (demand-driven) mukaan verkon solmu kerää tietoa muiden solmujen tiloista vain tullessaan lähettäjäksi tai vastaanottajaksi. Nämä menettelytavat voivat olla lähettäjän, vastaanottajan tai molempien tarpeen mukaan käynnistyviä.
- *Kausittaiseen tiedonkeruuseen* (periodic) perustuvat menettelytavat voivat olla joko hajautettuja tai keskitettyjä. Tiedonkeruu tapahtuu tietyin väliajoin. Nämä menettelytavat eivät yleensä sovita toimintaansa järjestelmän tilan mukaisesti, eivätkä ole kovin tehokkaita.
- *Tilanmuutoksen mukaan* (state-change-driven) tapahtuvissa menettelytavoissa verkon solmut lähettävät tilatietoaan muille siirtyessään uuteen tilaan (yli- tai alikuormitetuksi). Hajautetuissa tapauksissa tieto lähetetään kaikille kuorman hajautukseen osallistuville solmuille, keskitetyissä taas yhteiselle koordinaattorille joka pitää kirjaa solmujen tiloista. Tämä on yleinen tapa käytännön järjestelmissä (katso luku 4 sivulla 11).

Yleensä pelkkä kuormitustilanteen tunteminen ei riitä hajautuspäätöksiä tehtäessä. Joitain prosesseja ei voida siirtää koskaan, koska ne tarvitsevat jotain paikallista resurssia, toisia taas ei voida siirtää jos esimerkiksi niiden muistintarve ylittää kohdetyöasemassa olevan vapaan muistin määrän. Täysin dynaamisesti ja implisiittisesti käyttäjälle läpinäkyvästi toimivan kuorman tasapainottamisen toteuttaminen vaatii varsinkin heterogeenisessä järjestelmässä paikalliselta skeduloijalta ja mahdollisesti muiltakin käyttöjärjestelmän komponenteilta tukea kuorman hajautukselle.

3.1.5 Siirrettävän prosessin valintaperiaate

Valittaessa siirrettävää prosessia voidaan käyttää hyväkseen tietoa prosessin iästä. Prosessit, jotka eivät ole saaneet vielä suoritusaikaa, ovat "halvempia" siirrettäviä, sillä niiden mukana ei tarvitse siirtää tietoa suoritusympäristöstä (non preemptive migration), kuten avoinna olevia tiedostoja tai käytössä olevia muistialueita. Monissa esitellyissä kuorman jakojärjestelmissä "vanhoja" prosesseja siirretään (preemptive migration) pelkästään jos työasemalle ilmestyy paikallista aktiviteettia, ettei kuorman jakaminen pääse häiritsemään työasemien varsinaisia omistajia [LLM88, TvRvSS90]. Monissa järjestelmissä ei ole oikeastaan minkäänlaista kuorman jakoalgoritmia, sillä käyttäjä määrittelee mitä prosesseja siirretään, ja milloin [ZWZD92].

Aikaisemmin on esitetty useita tutkimustuloksia, joissa vanhojen prosessien siirtämisestä koituva hyöty osoitetaan näennäiseksi. Harchol-Balter ja Downey osoittavat kuitenkin että tämä on johtunut epärealistisista arvioista kuormitukselle, sekä väärästä arviosta prosessien elinkaarien jakaumista [HBD97]. He esittävät keskeyttävän valintaperiaatteen (preemptive migration policy) joka ei tarvitse manuaalisesti säädettäviä parametrejä, ja osoittavat että tämä siirtomenettely on käytännössä huomattavasti tehokkaampi kuin aiemmin suositut vain uusia prosesseja siirtävät valintaperiaatteet.

3.2 Algoritmiluokat

Kuorman hajautusalgoritmit voidaan jakaa eri luokkiin sen mukaan miten ne käyttävät hyväkseen tietoa järjestelmän tilasta.

3.2.1 Staattiset algoritmit

Staattiset algoritmit eivät ota huomioon järjestelmän tilaa hajautuspäätöksiä tehdessään. Informaatiomenettelyä ei siis ole. Valinta- ja sijoitusmenettelyssä käytetään joitain kiinteitä algebrallisia kaavoja. Staattisten algoritmien etuna on pieni ylimääräisen työn määrä. Tietoa ei tarvitse kerätä, vaan päätökset saadaan nopeasti jonkin laskutoimituksen tuloksena. Haittana on, että staattiset algoritmit eivät ota huomioon hajautettujen järjestelmien *heterogeenisuutta* – työasemat poikkeavat toisistaan niin suorituskyvyltään kuin kuormitukseltaankin, eli ympäristöä ei ole mahdollista mallintaa staattisilla kaavoilla. Myös kuormituksen kasvaessa tietyn rajan yli prosesseja siirretään turhaan koska jokainen solmu saadessaan uuden prosessin välittömästi päättää siirtää sen eteenpäin, ja järjestelmä saattaa päätyä epästabiiliin tilaan jossa prosessoreiden työjonot vain kasvavat koska kaikki prosessoriaika kuluu kuorman hajauttamiseen [SKS92]. Nämä algoritmit ovatkin mielenkiintoisia lähinnä teoreettisissa tarkasteluissa [TT85] sekä luonnollisesti järjestelmissä joissa tunnetaan yläraja mahdolliselle yhtäaikaiselle prosessien määrälle jo järjestelmää suunniteltaessa.

3.2.2 Dynaamiset algoritmit

Dynaamiset algoritmit käyttävät joitain järjestelmän tilaa kuvaavia muuttujia ja pystyvät tekemään tämän avulla parempia ratkaisuja kuin staattiset algoritmit. Nämä algoritmit siis lisäävät vähintään informaatiomenettelyn staattisiin algoritmeihin verrattuna. Myös valinta- ja sijoitusmenettelyissä käytetään kerättyä tietoa hyväkseen hajautuspäätöksiä tehtäessä. Esimerkkinä algoritmi joka ei vie järjestelmää ikinä epästabiiliin tilaan voisi toimia siten että se siirtää prosesseja vain koneille jotka ovat käyttämättöminä. Tämä on hyvin yksinkertainen dynaaminen algoritmi, mutta kuitenkin paljon käyttökelpoisempi kuin yksikään staattinen. Itse asiassa yksinkertaisimmat kuorman jakojärjestelmät perustuvat tähän periaatteeseen. Dynaamiset algoritmit tuovat kuitenkin lisäkuormaa järjestelmään. Tietoa järjestelmän tilasta täytyy kerätä, lähettää verkossa sekä tallettaa. Suurin osa viimeaikaisesta kuorman hajauttamiseen liittyvästä tutkimustyöstä on keskittynyt dynaamisiin algoritmeihin [WT98, HBD97].

3.2.3 Adaptiiviset algoritmit

Adaptiiviset algoritmit ovat dynaamisten algoritmien erikoistapaus. Ne osaavat muuttaa parametrejään ja jopa menettelytapojaan järjestelmän tilan muuttuessa. Tiedonkeruun määrä on tämän luokan algoritmeissa yleensä suurempi kuin muissa, riippuen siitä miten tarkasti ne osaavat ottaa huomioon järjestelmän tilan. Esimerkiksi jos jotkin menettelytavat toimivat paremmin järjestelmän ollessa joissain tietyissä tiloissa, adaptiivinen algoritmi voisi valita kulloinkin menettelytavan järjestelmän tilan mukaan. Esimerkiksi jos adaptiivinen algoritmi huomaa että järjestelmä on niin ylikuormitettu että sen omasta toiminnasta on enemmän haittaa kuin hyötyä, se voi yksinkertaisesti keskeyttää toimintansa kunnes kuormitus taas laskee tasolle joka mahdollistaa sen toiminnan jatkumisen [SKS92].

3.3 Järjestelmän vaikutus algoritmien suorituskykyyn

Eri algoritmit reagoivat eri tavoin järjestelmän muuttujiin, kuten siihen miten tasaisesti töitä eli prosesseja tulee suoritettavaksi, ja miten tasaisesti prosesseja syntyy eri solmuissa. Riippuen valituista menettelytavoista ja parametreistä, toisilla algoritmeillä voi olla huomattavasti korkeampi suorituskyky tiettyjen olosuhteiden vallitessa kuin toisilla. Naiivisti voisi ajatella, että voitaisiin muodostaa adaptiivinen algoritmi joka osaisi kullakin hetkellä valita kaikista tunnetuista algoritmeistä parhaiten tilanteessa toimivan ja joka näinollen olisi tietysti kaikkiin järjestelmiin sopiva. Tällainen algoritmi joutuisi kuitenkin keräämään niin paljon tietoa järjestelmän tilasta, että ylimääräisen työn määrä ylittäisi saavutetun hyödyn. Laajoissa järjestelmissä ajan-tasaisen tiedon hankkiminen onkin yleisesti ottaen mahdotonta, varsinkin jos sekä koneet että verkko ovat kuormitettuja.

Algoritmi voi toimia joko lähettäjän tai vastaanottajan aloitteesta – *lähettäjä-* ja *vastaanottaja-algoritmit*. Lähettäjä on verkon solmu joka tullessaan ylikuormitetuksi haluaa lähettää prosessin muualle suoritettavaksi. Vastaavasti vastaanottaja on solmu jolla ei ole tehtäviä, ja se on valmis suorittamaan jonkin muun lähettämää prosessia.

Lähettäjäalgoritmit toimivat tehokkaimmin silloin kun järjestelmä on vähän kuormitettu, koska tällöin todennäköisyys löytää nopeasti kone joka haluaa vastaanottaa lähetettävän prosessin on suuri. Kuormitetussa järjestelmässä vastaanottavia koneita ei löydy ja lähettäjäalgoritmi tekee paljon turhaa työtä etsiessään vastaanottajaa. Vastaavasti vastaanottaja-algoritmi toimii huonosti vähän kuormitetussa järjestelmässä koska todennäköisyys että se löytää siirrettävän prosessin niin nopeasti että sen siirtämisestä on vielä jotain hyötyä on pienempi kuin kuormitetummassa järjestelmässä.

Vastaanottaja-algoritmit ovat herkempiä varianssille töiden saapumisaikajakaumissa, koska todennäköisyys että algoritmi osaa kysyä juuri ylikuomitetulta koneelta prosessia siirrettäväksi ajoissa, on sitä pienempi mitä enemmän saapumisajat vaihtelevat – kerättyä tilatietoa ei voida käyttää jos saapumisaikojen varianssi on suuri. Vastaavasti lähettäjä-algoritmi on herkempi varianssille prosessien aikavaatimusjakaumissa, sillä varianssin kasvamisen on osoitettu lisäävän turhaa vastaanottajan etsimisaktiviteettia, mikä lisää ylimääräisen työn määrää [Dan98].

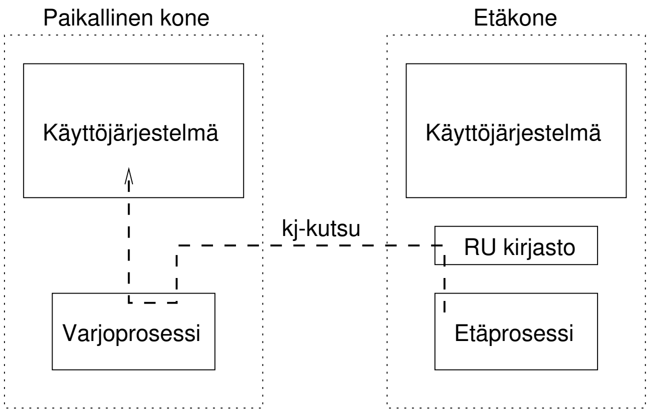
Nämä algoritmit voidaan yhdistää siten, että käytetään lähettäjäalgoritmia kun järjestelmän kokonaiskuormitus on pieni, ja vastaanottaja-algoritmia kun kuormitus on suuri. Tällainen algoritmi on *symmetrinen* [SKS92], eli solmu toimii aktiivisesti kuorman hajauttamiseksi ollessaan joko lähettäjän tai vastaanottajan roolissa. Symmetrinen, adaptiivinen algoritmi joka tallettaa tilatietoa on osoittautunut suorituskyvyltään parhaimmaksi niin kuorman määrän kuin heterogeenisuudenkin kasvaessa muihin yksinkertaisempiin algoritmeihin verrattuna [SKS92]. Tässä kuorman heterogeenisuudella tarkoitetaan vapaiden koneiden määrän osuutta kaikkien koneiden määrästä, eli heterogeenisuus on suurin mahdollinen kun yksi kone tuottaa koko järjestelmän kuorman.

4 Condor

Condor on yksi ensimmäisiä kuorman hajauttamiseen tehtyjä järjestelmiä jotka toimivat olemassaolevan käyttöjärjestelmän päällä ja olemassaolevilla sovelluksilla [LLM88]. Järjestelmä käyttää Up-Down –algoritmia taatakseen prosessoriajan tasaisen jakautumisen käyttäjien kesken [ML87]. Condor käyttää prosessien etäsuorittamiseen Remote Unix (RU) ohjelmistoa [Lit87]. RU on suunniteltu pitkien, eräajotyyppisten töiden suorittamiseen etäajona. Se osaa myös tallettaa prosessin tilan (checkpointing) siten, että talletetun prosessin suoritusta voidaan myöhemmin jatkaa samasta kohdasta, samalla tai eri työasemalla. Condor–projektissa yhdistettiin havainnot työasemien käyttöprofiileista [ML91], Up-Down –algoritmi [ML87] sekä RU [Lit87].

Condorissa käyttäjä joutuu päättämään mikä prosessi etäsuoritetaan, mutta ei sitä millä koneella se suoritetaan. Kyseessä on siis eksplisiittinen kuorman jakaminen, kuten useimmissa uudemmissakin käytännön järjestelmissä. Siirtomenettely on siis toisaalta käyttäjän laukaisema, mutta toisaalta myös järjestelmän laukaisema siinä tapauksessa, että etäsuorituksessa oleva prosessi siirretään toiselle koneelle jos etäkoneella ilmenee paikallista aktiviteettia.

Condorissa käytetään keskitettyä koordinaattoria, joka tekee kuormanhajautuspäätökset. Jokaisella työasemalla on oma paikallinen skeduloijansa, sekä taustajono (background job queue). Käyttäjän käynnistämät prosessit menevät taustajonoon. Koordinaattoriprosessi sijaitsee yhdellä työasemista sen oman paikallisen skeduloijan ja taustajonon lisäksi. Koordinaattori kysyy työasemilta kahden minuutin välein niiden tilatiedon (onko työasemalla ylimääräistä kapasiteettia jaettavaksi). Informaatiomenettelynä on siis kausittainen tiedonkeruu, joka aiheuttaa ylimääräistä työtä aina kun verkon tila ei muutu. Paikalliset skeduloijat tarkistavat puolen minuutin välein paikallisen aktiviteetin, ja etäsuorituksessa oleva prosessi pysäytetään jos paikallinen käyttäjä on palannut koneelleen. Koordinaattori tietää työasemien kuormitustilanteen, ja jakaa kapasiteettia työasemien paikallisille skeduloijille, eli prosessi voi siirtyä suoraan työasemalta toiselle. Sijoitusmenettelyssä tarvitsee löytää vain vastaanottaja, joka löytyy koordinaattorin määräämänä, kuormitustilanteen mukaisesti. Paikallinen skeduloija päättää mikä prosessi taustajonosta siirretään. Prosessin valintaperiaatteena on siis valita vain prosesseja jotka eivät vielä ole siirtyneet suoritukseen käyttöjärjestelmään asti. Valintamenettely on varsin yksinkertainen, prosesseja otetaan taustajonosta siinä järjestyksessä suoritukseen tai lähetetään etäsuoritukseen



Kuva 1: Remote Unixin toiminta

missä ne ovat jonoon saapuneet. Condorin hajautusalgoritmi on dynaaminen, sillä se käyttää hyväkseen järjestelmän tilatietoa, mutta ei adaptiivinen, sillä se ei esimerkiksi osaa lopettaa toimintaansa, vaikka järjestelmästä tulisikin niin kuormitettu että kuormanhajautuksen ylläpitäminen aiheuttaisi pelkästään ylimääräistä työtä.

Toimintaperiaatteen taustalla on yleinen käsitys siitä, että työasemat ovat itsenäisiä tietojenkäsittelyresursseja ja omistajansa tai käyttäjänsä henkilökohtaisia työkaluja. Tästä syystä vain muuten tyhjillään olevia koneita käytetään kuorman jakamiseen. Koordinaattorista on haluttu tehdä yksinkertainen, eikä se talleta paljoakaan tietoa työasemien tilasta. Tämä mahdollistaa järjestelmän skaalautuvuuden, ja tekee koordinaattorin toipumisen helpommaksi mahdollisissa virhetilanteissa. Koordinaattorin vikaantuminen ei vaikuta jo hajautettuihin töihin millään tavalla, se vain keskeyttää uusien hajautuspäätösten tekemisen kunnes koordinaatori käynnistetään uudelleen. Koordinaattori voidaan käynnistää uudestaan mille tahansa työasemalle, mutta tämä ei tapahdu automaattisesti.

Remote Unixin käyttäminen vaatii hajautettavien ohjelmien kääntämisen uudelleen. Etäsuoritettavaa ohjelmaa käännettäessä siihen linkitetään Remote Unix –kirjasto, joka mm. luo varjoprosessin sekä hoitaa käyttöjärjestelmäkutsujen lähettämisen varjoprosessille. Kaikki käyttöjärjestelmäkutsut mitä etäprosessi tekee, välitetään paikallisella koneella sijaitsevalle varjoprosessille käyttäen verkkoyhteyttä (katso kuva 1), ja vastaavasti paluuarvot samaa reittiä takaisin etäprosessille [Lit87]. Paljon käyttöjärjestelmäkutsuja tekevät prosessit kannattaa jättää hajauttamatta, sillä ne vievät paikallisen koneen tehoa, sekä kuluttavat verkkokapasiteettia.

Kun prosessi halutaan siirtää muualle, käytetään RU:n tilantallennusmekanismia (checkpointing), joka tallettaa prosessin teksti– ja datasegmentit, vakiot, pinon, keon, prosessorin rekisterit, sekä tiedot avoimista tiedostoista. Tietoja lähetetyistä viesteistä ei tarvitse tallentaa, sillä Condorissa odotetaan kunnes kaikkiin lähetettyihin viestihin on saatu vastaus varjoprosessilta. Näiden tietojen perusteella prosessin suoritusta voidaan jatkaa millä tahansa klusterin työasemalla milloin tahansa.

Up-Down –algoritmia käytetään resurssien tasaiseen jakamiseen. Algoritmi estää suurkäyttäjiä käyttämästä kaikkia resursseja ja antaa myös satunnaisille käyttäjille mahdollisuuden käyttää järjestelmää. Algoritmi pitää skedulointi-indeksiä jokaisella solmulla. Alussa kaikki indeksit alustetaan nolliksi. Kun solmu saa käytettäväkseen resursseja, indeksiä kasvatetaan. Kun solmu haluaa resursseja, mutta ei saa, indeksiä pienennetään. Koordinaattori päivittää indeksejä kahden minuutin välein saadessaan uusia tietoja paikallisilta skeduloijilta. Pienempi indeksi tarkoittaa korkeampaa prioriteettia. Jos korkeamman prioriteetin omaava työasema haluaa prosessoriaikaa, eikä koneita ole vapaana, koordinaattori keskeyttää matalamman prioriteetin työn ja korkeamman prioriteetin työ voi siirtyä suoritukseen.

Sekä paikallinen skeduloija että koordinaattoriprosessi vievät vähemmän kuin yhden prosentin työaseman kapasiteetista. Vuoden 1988 järjestelmällä prosessin sijoittaminen ja tilan tallentaminen veivät keskimäärin 2.5 sekuntia, joten kovin pieniä töitä Condorilla ei kannata hajauttaa. Prosessit jotka tekevät paljon käyttöjärjestelmäkutsuja, kuten lukuja ja kirjoituksia, suhteessa muiden käskyjen määrään, on parempi jättää suoritettavaksi paikallisesti, sillä kutsujen suorittaminen RU:n välityksellä on huomattavasti hitaampaa kuin paikallisten järjestelmäkutsujen. Prosessit tarvitsevat prosessorin lisäksi myös muistia sekä usein myös levytilaa. Jos näitä resursseja ei ole saatavilla, jää tarjolla oleva prosessoriaika käyttämättä.

Condor on toimiva kuorman jakojärjestelmä pitkäkestoisten taustaprosessien eksplisiittiseen suorittamiseen muuten käyttämättömillä prosessoreilla. Tunnin etäsuoritukseen tarvitaan vain alle kuusi sekuntia paikallista prosessoriaikaa.

5 NOW

NOW (Networks of Workstations) on Berkeleyn yliopiston tutkimusprojekti jossa hajautettu järjestelmä nähdään yksittäisten sovellusten resurssina. Projektin tarkoituksena on osoittaa, että työasemaverkolla voidaan hoitaa kaikki tietojenkäsittelyn tehtävät, myös ne mitkä ennen vaativat supertietokoneen hankkimisen.

Loppukäyttäjät hyötyvät kolmesta uudesta mahdollisuudesta verrattuna perinteiseen lähiverkkoon: verkon työasemien muistin käyttäminen suurena levyvälimuistina, levyjen käyttämisen eräänlaisen hajautetun RAID:in (Redundant Array of Inexpensive Disks) muodostamiseen sekä prosessoreiden käyttämisen rinnakkaiseen laskemiseen [ACP94]. Näiden toteuttamiseksi projektissa on tunnistettu useita haasteita: miten mahdollistaa tehokas prosessien välinen kommunikaatio niin laitteiston kuin ohjelmistonkin tasolla, verkon autonomisten käyttöjärjestelmien koordinointi sekä verkonlaajuiset tiedostojärjestelmät.

Uudet nopeat verkkoteknologiat mahdollistavat hajautetun muistin käyttämisen isona levyvälimuistina. Tämä ei ole ollut käytännössä mahdollista perinteisessä Ethernetissä, koska se veisi liikaa jaetun tiedonsiirtotien kapasiteettia, eikä tiedonsiirto maksiminopeudellakaan ole ollut nopeampaa kuin haku työaseman paikalliselta levyltä. Uudet kytkentäiset verkkotekniikat, kuten ATM (Asynchronous Transfer Mode), ovat kuitenkin niin paljon nopeampia, että simulaatioiden mukaan hajautetun muistin käyttäminen virtuaalimuistina on 5–10 kertaa nopeampaa kuin paikallisen levyn. Kytkentäisissä verkkotekniikoissa verkon maksimaalinen tiedonsiirtonopeus voidaan saada verkon jokaiselle solmulle yhtä aikaa, sillä verkon solmut eivät joudu kilpailemaan samasta (fyysisestä) tiedonsiirtokanavasta, vaan datakehykset menevät erityisen reitittimen kautta solmulta toiselle.

Verkon levyistä on mahdollista muodostaa RAID:in tyyppinen levyjärjestelmä ohjelmiston tasolla. RAID (Redundant Array of Inexpensive Disks) –teknologia on nykyisissä palvelimissa useimmiten käytössä oleva levyjärjestelmä, missä useista fyysisistä levyistä on muodostettu yksi looginen levy. Tälla tavalla levyjä voidaan käyttää toistensa varmistamiseen; useissa RAID–järjestelmissä levykön yksi levy voikin vaikka tuhoutua kokonaan: sen voi vaihtaa uuteen toiminnan keskeytymättä, sillä rikkoutuneen levyn tiedot voidaan automaattisesti palauttaa levykön muilta levyiltä. Lisäksi tiedonsiirtonopeus RAID–järjestelmissä on suurempi kuin yksittäisillä levyillä, koska tiedostoja voidaan hajauttaa useille fyysisille levyille. Perinteiset RAID–levyjärjestelmät vaativat kalliin laitteistotuen, mutta hajauttamalla tietoa verkon levyille NOW:ssa saadaan käyttöön turvallinen tietovarasto valtavalla kapasiteetilla jonka maksimaalista tiedosiirtonopeutta rajoittaa ainoastaan verkkoyhteyden nopeus. Tällainen levyjärjestelmä toimisi aivan kuten RAID–järjestelmäkin, mutta laitteisto-RAID–ohjain olisi korvattu ohjelmistolla, ja sisäinen tiedonsiirtoväylä nopealla lähiverkolla.

Kuorman hajauttaminen voi tarkoittaa ison laskennan rinnakkaista suorittamista, kuten nykyisissä rinnakkaissupertietokoneissa (MPP, Massively Parallel multiProcessor), tai interaktiivisten käyttäjien töiden hajauttamista. Tiettyjä supertietokoneita vaativia tehtäviä voidaan suorittaa yhtä tehokkaasti myös NOW:lla hinnan jäädessä noin kuudesosaan. Esimerkkinä 32 verkon solmua käyttävä rinnakkainen laskenta hidastaa interaktiivisten käyttäjien koneita 64 koneen verkossa vain 10 prosenttia [ACP94].

Eräs projektin tavoitteista on saada viestin lähettämisen vaatima ylimääräinen työ niin pieneksi, että pieni viesti verkon solmulta toiselle vaatii vain saman verran työtä prosessorilta kuin nykyaikaisen supertietokoneen sisäisessä väylässä, eli n. 10 mikrosekuntia. Tämä vaatii työtä sekä verkon laitteiston, että käyttöjärjestelmän parissa.

Toinen projektin päätavoitteista on hajautettujen resurssien hallinta NOW:ssa. Tätä varten on kehitetty Global Layer UNIX (GLUnix), joka on ohjelmallinen kerros käyttöjärjestelmän ja sovelluksen välissä. Tämä kerros kaappaa ja kääntää sovelluksen systeemikutsuja luodakseen sovellukselle illuusion globaalista käyttöjärjestelmästä. Kaikkia palveluja ei ole kuitenkaan mahdollista luoda ilman muutoksia käyttöjärjestelmän ytimeen. Projektin yksi tavoitteista on myös tutkia mitkä ovat pienimmät muutokset mitä tarvitaan olemassaolevien kaupallisten käyttöjärjestelmien saamikseksi “NOW-valmiuteen”.

5.1 GLUnix

GLUnix on käyttöjärjestelmän kerros verkon työasemaklusterille. GLUnix mahdollistaa NOW:in tarjoamat palvelut olemassaoleviin käyttöjärjestelmiin. Nimestään huolimatta GLUnix voidaan toteuttaa muihinkin käyttöjärjestelmiin kuin Unixeihin [GPR97]. GLUnix on suunniteltu tarjoamaan tuntumattoman etäsuorituksen, tuen sekä interaktiivisille rinnakkaisille että sarjallisille tehtäville, kuorman tasapainotuksen, sekä tuen olemassaoleville ohjelmille. GLUnix on ollut käytössä Berkeleyssä yli kaksi vuotta ja parhaimmillaan 100–solmuisessa Sun UltraSparc klusterissa. Mittausten mukaan 100 solmulle hajautettavasta rinnakkaisesta laskennasta voidaan suoriutua reilussa sekunnissa, eikä keskitetty GLUnix koordinaattorisolmu ole järjestelmän pullonkaula.

Projektin alkuperäisenä tavoitteena oli yhdistää aikaisempien järjestelmien (Condor, LSF, Utopia, PVM) hyvät ominaisuudet. Tavoitteena oli tukea sekä interaktiivista että eräajotyyppistä etäsuoritusta sekä rinnakkaistuvalle (parallel) että perättäissuoritettavalle (sequential) laskennalle, tukea perinteistä UNIXin I/O–semantiikkaa, täysin läpinäkyvää suorituksessa olevien prosessien siirtämistä sekä toipua solmujen kaatumisista.

GLUnix ei sisällä kaikkia alkuperäisen suunnitelman mukaisia ominaisuuksia, sillä niiden toteuttaminen olisi vaatinut muutoksia käyttöjärjestelmän ytimeen sekä rajoittanut sen siirrettävyyttä. GLUnix muodostaa käyttöjärjestelmään uuden GLUnix–domainin prosesseille entisen UNIX–domainin rinnalle.

GLUnixin nykyinen toteutus tarjoaa etäsuorituksen, mutta ei täysin tuntumattomasti. GLUnix tarjoaa kuorman tasapainotuksen prosessin käynnistysvaiheessa, käyttäen senhetkistä tietoa järjestelmän tilasta, mutta ei tue käynnistettyjen prosessien uudelleensiirtämistä. GLUnix on siis selvästi pudonnut suunnitellusta kuorman tasapainotusjärjestelmästä kuorman jakojärjestelmäksi. GLUnix tukee rinnakkaisesti suoritettavien töiden skedulointia ja suorittaa olemassaolevia sovelluksia. GLUnix osaa toipua minkä tahansa solmujen paitsi koordinaattorisolmun vikaantumisesta.

GLUnix olettaa, että klusterin työasemat käyttävät samaa yhteistä tiedostojärjestelmää, sekä samoja käyttöjärjestelmiä. Lisäksi GLUnix käyttää jaettua hakemistoa tallettamaan koordinaattoriprosessin osoitteen GLUnix–prosessien käynnistämistä varten.

GLUnix koostuu kolmesta komponentista. Jokaisessa klusterissa on yksi koordinaattoriprosessi ja jokaisessa klusterin solmussa on sekä GLUnix–prosessi että sovelluskohtainen GLUnix–kirjasto. Koordinaattoriprosessi pitää kirjaa klusterin tilasta, eli työasemien kuormituksesta sekä prosesseista. Koordinaattori myös päättää resurssien käytöstä, eli käynnistettävien prosessien sijoittamisesta. Hajautusalgoritmissa on keskitetty komponentti, koordinaattorisolmu, aivan kuten Condorissakin.

Paikallinen GLUnix–prosessi kerää tietoa paikallisista prosesseista sekä kuormituksesta ja lähettää tiedon tilan muutoksista koordinaattorille, aivan kuten Condorissakin. Se myös käynnistää ja signaloi paikallisia prosesseja sekä huomaa työn päättymisen. Informaatiomenettelynä on siis kausittaiseen tiedonkeruuseen perustuva menettelytapa. Siirtomenettely on vielä yksinkertaisempi kuin Condorissa, sillä pelkästään käyttäjän eksplisiittinen pyyntö aiheuttaa prosessin hajauttamisen. Kerran siirrettyä prosessia ei tämän jälkeen enää siirretä. Tästä seuraa, että myös valintamenettely määräytyy samalla tavalla: siirrettävä prosessi määräytyy käyttäjän pyynnöstä. Sijoitusmenettely sitävastoin on kehittyneempi, eli se käyttää hyödykseen tietoa järjestelmän tilasta, ja hajauttaa prosesseja kuormitustilanteen mukaan: keskitetty koordinaattori määrää prosessin sijoittamisesta. Myöskin siirrettävän prosessin valintaperiaatteena on siirtää vain uusia prosesseja. Hajautusalgoritmi muistuttaa siis läheisesti Condorin vastaavaa. Vaikka tämäkin algoritmi on dynaaminen, eikä adaptiivinen, ei se kuitenkaan vie järjestelmää epästabiiliin tilaan, sillä prosesseja ei hajauteta kuin kerran, eikä siis voida päästä tilaan missä prosessit pelkästään kiertäisivät verkon solmulta toiselle pääsemättä ikinä suoritukseen. Tämä pätee myös Condorissa, jossa ongelma ratkaistaan keskeyttämällä prosessin suoritus kunnes vapaa kone löytyy. Epästabiili tila voisi olla mahdollinen dynaamisessa kuorman tasapainotusjärjestelmässä. Pelkkä kuorman jakamiseen tyytyminen ei siis vaadi kaikkein hienoimpien algoritmien käyttämistä – tämän ovat huomanneet myös monet esimerkkijärjestelmien suunnittelijatkin.

GLUnix–kirjasto toteuttaa GLUnix–rajapinnan mahdollistaen sovelluksille eksplisiittisen GLUnixin palvelujen käyttämisen. Olemassaolevien sovellusten etäajo on mahdollista erillisen tynkäprosessin avulla joka siirtää sovelluksen binäärikoodin suoritettavaksi jollekin muulle klusterin solmulle. Rinnakkaisia ohjelmia voidaan toteuttaa käyttämällä GLUnix–kirjastoa hajautettujen prosessien synkronointiin. Käyttöjärjestelmän standardit I/O –kanavat ohjataan uudelleen käyttäen TCP:n tarjoamia tietoliikenneyhteyksiä, joiden määrä on useimmissa käyttöjärjestelmissä rajoitettu, ja siten myös rinnakkaisten prosessien määrä NOW:ssa.

GLUnixin avulla Berkeleyn 100–solmun UltraSPARC-I verkko saavutti 10.1 GFLOPS (Billions of Floating Point instructions Per Second) vuonna 1997, ja oli maailman 345. tehokkain supertietokone. GLUnix on saatavilla lähdekoodeineen ainakin Solaris–käyttöjärjestelmälle.

6 POPCORN

POPCORN on Israelissa, Jerusalemissa sijaitsevan Hebrew-yliopiston tietojenkäsittelytieteen osaston tutkimusprojekti, jonka tarkoituksena on mahdollistaa koko Internetin ylimääräisen prosessointikapasiteetin käyttäminen rinnakkaisen laskennan nopeuttamiseen. Projektissa on kehitetty Java-pohjainen järjestelmä joka mahdollistaa prosessoriajan markkinat myyjineen ja ostajineen [NLRC98].

Internet tuo kuorman hajauttamiseen uusia ongelmia verrattuna "perinteiseen" lähiverkossa tapahtuvaan kuorman hajauttamiseen. Ongelmat ovat lähinnä teknisiä ja liittyvät ohjelmakoodin siirtämiseen, turvallisuuteen, verkon heterogeenisyyteen ja rinnakkaisen laskennan koordinaatioon. Laajalle levinnyt Java-arkkitehtuuri ratkaisee nämä ongelmat suurimmalta osin, tarjoamalla yhtenäisen ja turvallisen suoritusalustan.

Teknisten ongelmien ratkaisemisen jälkeenkin jäljellä on huomattavia ongelmia jotka liittyvät globaaliin hajauttamiseen. Ainakin kaksi perustavanlaatuista ongelmaa syntyy siirryttäessä lähiverkon tasolla tapahtuvasta hajauttamisesta Internetin laajuiseen hajauttamiseen. Näistä ensimmäinen on luonnollisestikin järjestelmän koko. Internet on paljon hajautetumpi kuin yksikään lähiverkko. Tiedonsiirtokapasiteetti on pienempi, viiveet isommat ja luotettavuus paljon pienempi. Lisäksi Internetissä olevat koneet ovat tavoitettavissa muista riippumattomasti, eikä niiden kontrolloimiseksi ole keinoja. Internetin etuja on kuitenkin se, että potentiaalinen prosessoreiden määrä on valtava. Vaikka Internetiä ei tällä hetkellä voikaan toivoa käyttävänsä täysin yleiskäyttöisenä tehokkaana rinnakkaisprosessorikoneena, sitä voi kuitenkin käyttää useiden mielenkiintoisten laskennallisten ongelmien ratkaisemiseen. Internetin luonteen vuoksi POPCORN järjestelmänä eroaa sekä Condorista että NOW:sta huomattavasti.

Toinen suuri ero lähiverkkojen ja Internetin välillä on, että Internetissä myös prosessoreiden omistus on hajautuneempaa. Tietokoneiden omistajat ovat yksityishenkilöitä tai organisaatioita, joten rinnakkaisen ongelman ratkaisemiseen ei ole valmiiksi olemassaolevaa motivaatiota. Tämän vuoksi globaalin hajautuksen täytyy sisältää motivaatiotekijä, kuten maksu prosessoriajasta. Lisäksi Internetissä sijaitsevat koneet voivat olla viallisia tai vääristävät tahallaan laskennan tuloksia. Tämän vuoksi koneiden täytyy tarkistaa toistensa tuloksia ja ne täytyy olla suojassa toisiltaan.

POPCORNin pääasiallinen tehtävä on tarjota kenelle tahansa ohjelmoijalle Internetissä virtuaalinen rinnakkaistietokone. Tämä virtuaalikone on toteutettu käyttämällä kaikkia tarjolla olevia prosessoreita. Prosessoreiden saamiseksi käytettäviksi sisältyy järjestelmään oleellisena osana vapaisiin markkinoihin perustuva laskutusjärjestelmä. Järjestelmä on toteutettu Javalla, ja turvallinen etäsuoritus perustuu Java-sovelmiin joita voidaan ajaa kaikissa nykyaikaisissa www-selaimissa.

POPCORN koostuu kolmesta erillisestä komponentista:

- rinnakkaisesta Java-ohjelmasta joka käyttää POPCORN-ohjelmointiparadigmaa sekä ohjelmointirajapintaa (API),
- prosessoriajan myyjästä, sekä
- prosessoriajan markkinoista.

POPCORN-ohjelmointiparadigma jota ostaja-ohjelma käyttää, toteuttaa rinnakkaisuuden pilkkomalla laskennan useisiin rinnakkaisiin alilaskentoihin, eli etäsovelmiin (computelet). POPCORN-sovellus lähettää näitä etäsovelmia käyttäjän valitsemille prosessoriaikamarkkinoille joka lähettää ne edelleen prosessoriajan myyjille ja palauttaa tulokset. Sijoitusmenettely on monimutkaisempi kuin "perinteisissä" lähiverkon laajuudella toimivissa hajautusalgoritmeissa. Markkinoiden valitessa suorittajia jonossa oleville etäsovelmille ei käytetä hyväkseen tietoa myyjien kuormitustilanteesta, vaan lähinnä talouden lakeja, myyjien ja ostajien määrätessä hintoja prosessoriajasta. Lisäksi sijoitusmenettelyssä voidaan ottaa huomioon esimerkiksi koneiden etäisyys verkossa. Informaatiomenettely on tilanmuutoksen mukaan tapahtuvaa – sekä myyjät että ostajat ilmoittavat markkinoille, koordinaattorille, lähinnä tarjoamiensa tai pyytämiensä hintojen muutoksista, sekä myyjät tullessaan käytettäviksi. Myyjien ja ostajien kohtaaminen on dynaamista ja toimii siis talouden periaatteiden mukaisesti, ja tuloksena on maksu ostajalta myyjälle. Järjestelmä on suunniteltu hyvin karkealle rinnakkaisuudelle – pieniä ongelmia ei kannata ratkaista hajautetusti sillä viestien lähettämisestä koituu ylimääräistä työtä.

POPCORN-sovellus toimii siten, että pääsäie, joka toimii paikallisella prosessorilla, luo alilaskentoja suoritettavaksi etäprosessoreilla. Etäsovelma kapsuloi laskennan Java-olioksi kun se lähetetään etäprosessorille. Etäsovelma sisältää sekä ohjelmakoodin että datan mikä tarvitaan laskelman suorittamiseen. Vastauksena paikallinen kone voi saada minkä tahansa Java-objektin. Etäsovelma-mekanismi poikkeaa Javan RMI:stä (RPC:n olioversio, Remote Procedure Call, etäproseduurikutsu) siinä, että suoritettava ohjelmakoodi lähetetään suorituspyynnön mukana, vaikka ne syntaktisesti vastaavatkin toisiaan. Etäsovelmaa voidaan pitää erittäin rajoitettuna agenttina – se ei voi matkustaa verkossa pidemmälle kuin yhteen kohteeseen, sillä on rajoitettu elinaika, se ei tiedä sijaintiaan, eikä se voi kommunikoida vapaasti.

Koko POPCORN-sovelluksen suorituksen ajan etäsovelmia lähetetään ja niiden tuloksia vastaanotetaan. Tulosten saapumisjärjestystä, eikä etäsovelmien vaatimaa suoritusaikaa ei voi ennustaa. Pääohjelma täytyy tehdä asynkroniseksi, eli se voi edetä huolimatta tulosten saapumisesta epäjärjestyksessä. Tuloksia ei välttämättä saada ollenkaan. POPCORN-järjestelmä huomaa tämän ja ilmoittaa siitä pääohjelmalle, jolloin pääohjelma voi suorittaa kyseisen alilaskutoimituksen uudelleen. Tulokset voivat olla myös vääriä, joko etäkoneen Java-toteutuksen virheiden takia, kommunikaatiovirheiden takia tai etäkoneen tahallisen huijaamisyrityksen takia. Tulosten oikeellisuuden tarkastaminen on pääohjelman vastuulla.

Prosessoriajan myynti onnistuu helposti käymällä tietyllä www-sivulla Javaa tukevalla selaimella. Tällä sivulla on Java-sovelma joka aloittaa suorituksen myyjän koneella vastaanottaen ja suorittaen etäsovelmia. Markkinapalvelin huolehtii etäsovelmien sekä tulosten siirtämisestä ja pitää yllä myyjien sekä ostajien tilejä. Markkinapalvelimen kaatumisesta tai vikaantumisesta toipumiseen ei tosin oteta kantaa millään tavalla.

POPCORN-järjestelmällä on luontevaa toteuttaa rinnakkaisia sovelluksia jotka ovat isoja ongelmia joita on helppo jakaa useisiin toisistaan riippumattomiin aliongelmiin, jotka yksistäänkin vievät paljon prosessoriaikaa. Tällaisia sovelluksia voivat olla esimerkiksi ongelmat joihin haetaan ratkaisua kokeilemalla kunnes oikea vastaus löytyy, koska parempaa ratkaisualgoritmia ei kyseiselle ongelmalle tunneta. Tällaisia ongelmia ovat esimerkiksi useat koodinmurtamisongelmat. Myös geneettiset algoritmit sekä päätöspuu-algoritmit ovat sopivia sovelluskohteita.

7 Yhteenveto

Hajautettujen järjestelmien yleistymisen myötä tärkeimmät tietojenkäsittelylliset resurssit – vapaa prosessoriaika sekä vapaa muisti ovat hajautuneet verkkoon. Yksittäisen prosessorin skedulointi prosessien kesken, sekä moniprosessorikoneen skedulointi yhteisen muistin koneessa käyttöjärjestelmän sisällä hallitaan nykyisin melko hyvin. Hajautetut järjestelmät ovat käyttäjän näkökulmasta oma työasema sekä verkosta löytyvät resurssit.

Käyttöjärjestelmän tai sovelluksen suunnittelijan näkökulmasta hajautettu järjestelmä tarjoaa aivan uusia mahdollisuuksia: suurin osa työasemista on suurimman osan ajasta käyttämättöminä, joten niiden prosessoriaikaa, muistitilaa sekä levytilaa käyttämällä voidaan toteuttaa ohjelmistoja jotka aiemmin vaativat supertietokoneita. Toisaalta myös käyttäjien olemassaolevia sovelluksia voidaan nopeuttaa ottamalla käyttöön läpinäkyvä kuorman hajauttaminen käyttöjärjestelmän tasolla. Tämän mahdollistamiseksi täytyy ratkaista, ei pelkästään ongelmaa hajautetun muistin moniprosessorikoneen skeduloinnista, vaan myös prosessien tarpeesta työasemien paikallisiin resursseihin sekä prosessien siirtoon liittyvistä tietoliikenne– sekä käyttöjärjestelmäteknologian asettamista rajoituksista.

Tiettyjen rinnakkaista laskentaa suorittavien sovellusten nopeuttaminen kuorman jakamista käyttäen on suorastaan helppoa verrattuna käyttäjälle läpinäkyvään kuorman tasapainottamiseen jossa hajautettavat prosessit valitaan implisiittisesti tiettyjen periaatteiden mukaisesti. Useita kaupallisia tuotteita sekä tutkimusprojektien toteuttamia kokeellisia järjestelmiä on saatavilla erilaisiin tarpeisiin. Useimmissa käytännön järjestelmissä on tähän mennessä rajoituttu kuorman jakamiseen käyttäen jonkinlaista keskitettyä koordinaattoria. Alueen teoreettinen tutkimus tarjoaa ongelmaan kuitenkin huomattavasti kehittyneempiä ratkaisutapoja, joiden soveltaminen tosin vaatisi muutoksia olemassaoleviin käyttöjärjestelmiin, tai kokonaan uuden kirjoittamista. Toisaalta uusi avoimen lähdekoodin trendi, sekä käyttöjärjestelmien tuki ladattaville moduleille laskevat kynnystä siirtyä toteuttamaan ja soveltamaan teoriatietoa käyttöjärjestelmän tasolle. Yleiskäyttöisen ja käyttäjälle läpinäkyvän kuorman hajauttamisen toteuttaminen esimerkiksi Linux–käyttöjärjestelmään voisi tarjota aiheen oivalliselle tutkimusprojektille, jonka tuloksille monilla olisi paljon käyttöä.

Lähteet

ACP94 T. Anderson, D. Culler, and D. Patterson. A case for now (networks of workstations). Technical report, CS Division, University of California at Berkeley, 1994.

CDK94 G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems - Concepts and Design*, chapter 1, pages 1–25. Addison-Wesley, 1994.

Dan98 S. Dandamudi. Sensitivy evaluation of dynamic load sharing in distributed systems. *IEEE Concurrency*, pages 62–72, July-Sept. 1998.

Gos91 A. Goscinski. *Distributed Operating Systems - The Logical Design*, chapter 9.5–9.13, pages 465–514. Addison-Wesley, 1991.

GPR97 D. Ghormley, D. Petrou, and S. Rodrigues. Glunix: a global layer unix for a network of workstations. Technical report, CS Division, University of California at Berkeley, 1997.

HBD97 M. Harchol-Balter and A.B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems*, 15(3):253–285, 1997.

Kun91 T. Kunz. The influence of different workload descriptions on a heuristic load balancing scheme. *IEEE Transactions on Software Engineering*, 17(7):725–730, 1991.

Lit87 M. Litzkow. Remote unix. In *1987 Summer Usenix Conference, Phoenix, Arizona*, June 1987.

LLM88 M. Litzkow, M. Livny, and M.W. Mutka. Condor - a hunter of idle workstations. *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, June 1988.

ML87 M.W. Mutka and M. Livny. Scheduling remote processing capacity in a workstation-processor bank computing system. In *7th International Conference of Distributed Computing Systems, Berlin, West Germany*, pages 2–9, Sept. 1987.

ML91 M.W. Mutka and M. Livny. The available capacity of a privately owned workstation environment. *Performance Evaluation*, (12):269–284, 1991.

NLRC98 N. Nisan, S. London, O. Regev, and N. Camiel. Globally distributed computation over the internet - the popcorn project. In *18th IEEE Distributed Computing Conference*, pages 592–601, 1998.

SKS92 N. Shivarathi, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *IEEE Computer*, 25(12):33–44, 1992.

Tho95 K. Thompson. Plan 9 from bell labs. Technical report, Bell Labs, http://plan9.bell-labs.com/plan9/doc/9.html, 1995.

TT85 A. Tantawi and D. Towsley. Optimal static load balancing in distributed computer systems. *Journal of the ACM*, 32(2):445–465, April 1985.

TvRvSS90 A. Tanenbaum, R. van Renesse, H. van Staveren, and G. Sharp. Expriences with the amoeba distributed operating system. *Communications of the ACM*, 33:336–346, 1990.

WT98 J. Watts and S. Taylor. A pratical approach to dynamic load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 9(3), March 1998.

ZWZD92 S. Zhou, J. Wang, X. Zheng, and P. Delisle. Utopia: A load sharing facility for large, heterogeneous distributed computer systems. Technical report, Computer Systems Research Institute, University of Toronto, April 1992.