SDMAPeG

– att utöka processorkraft via nätverk

Niclas Alexandersson
Oskar Nyberg
Victor Nilsson
William Dahlberg

Abstract

Följande projekt utforskar möjligheten att koppla samman ett nätverk av datorer för att utföra krävande beräkningar, med målet att kunna utnyttja deras gemensamma beräkningskraft för att få mer arbete utfört på kortare tid. Det har resulterat i en fullt fungerande prototyp, bestående av tre separata applikationer, som samverkar för att kunna skicka ut ett flertal krävande uppgifter som sedan distribueras till och utförs parallelt av olika datorer.

Innehåll

Inledning

Parallellism – konceptet att låta ett program köras på flera processorer eller processorkärnor samtidigt – är en term som på senare tid har blivit allt mer populär inom datorindustrin. Allteftersom den fysiska begränsningen för processorers frekvens börjar nås så har utvecklingen i stället börjat fokusera på att öka antalet kärnor i ett processorchip. Fler kärnor gör i sig själva inte en processor snabbare; vad de däremot gör är att låta processorn utföra flera beräkningar samtidigt, och på så sätt få mer arbete utfört under samma tid. Genom att dela upp ett program i flera delar som är oberoende av varandra, och sedan låta dessa olika delar köras parallellt på olika processorkärnor kan programmet utföra sitt arbete snabbare än om det bara körts på en enda kärna.

En dator är dock fortfarande begränsad av sin hårdvara, vilket innebär att en uppgift som utnyttjar alla processorkärnor fullt ut inte längre kan göras snabbare genom att delas upp i fler delar. En uppenbar lösning på detta är förstås att öka antalet processorer och kärnor i datorn – något som också görs i bland annat servrar och massivt parallella superdatorer. Dock är detta också begränsat av hårdvaran, närmare bestämt hur den kan kopplas ihop och uppgraderas. En vanlig dator kan inte uppgraderas hur mycket som helst, och superdatorer är inte alltid lättillgängliga.

I detta projekt utforskas en annan lösning; möjligheten att distribuera många krävande arbetsuppgifter över ett nätverk av datorer. I stället för att låta en enda kraftfull dator utföra alla beräkningar kopplas flera datorer ihop i ett nätverk för att dela på arbetsbördan. Uppgifter skickas ut till datorerna i nätverket för att utföras och sedan skickas tillbaka, och fördelas på ett sådant sätt att alla datorers processorer alltid utnyttjas i så hög grad som möjligt. Fler datorer kan sedan kopplas upp mot nätverket för att öka beräkningskapaciteten vid behov.

Syfte

Projektets syfte är att utveckla ett system för att distribuera ett stort antal prestandakrävande beräkningar över ett nätverk för att sedan utföras parallellt av ett antal datorer. Målet är att visa hur ett nätverk av datorer kan användas för att utföra många fler uppgifter på samma tid, och därmed kunna slutföra dessa på kortare tid än om alla uppgifter skulle ha utförts av bara en enda dator.

Det är även ett mål att visa att systemet enkelt kan skalas upp genom att lägga till fler datorer i nätverket, och att detta ger en faktisk ökning av systemets beräkningskapacitet. Detta bör i sin tur visa att systemet åtminstone i teorin kan användas för att utföra uppgifter som annars hade varit för tidskrävande för en ensam dator att utföra – givet att dessa uppgifter kan delas upp i tillräckligt många oberoende delar.

Teori

Ett program var traditionellt sett en sekvens av instruktioner som utfördes av en enda processor (Göetz et al., 2006). Instruktionerna utfördes en efter en, fram tills dess att programmet hade kört färdigt. Operativsystemens intåg gjorde det sedan möjligt för flera program att köras samtidigt genom att växla mellan olika *processer* – program som oberoende kunde köras sida vid sida av varandra. På samma sätt som program kunde köras parallellt i olika processer, så kunde individuella program senare även delas upp i *trådar* – delar av program som också kunde köras oberoende och samtidigt som varandra.

Det blir allt vanligare med datorsystem med fler och fler *CPU:er* – det vill säga processorer eller processorkärnor. På grund av fysiska begränsningar för hur snabba processorer kan göras med nuvarande tekniker, arbetar man numera på att kunna ha så många kärnor i ett processorchip som möjligt (Göetz et al., 2006). Som ett exempel kan nämnas att i en undersökning gjord av Steam i April 2013 om vilken hård- och mjukvara som deras användare använder, så hade över 95 % av användarna en dator med mer än en CPU (Steam, 2013). Detta behöver inte nödvändigtvis vara representativt för alla datoranvändare, då Steam är en plattform för datorspel, men det visar ändå att övergången till fler CPU:er är en förändring som redan är här.

För att kunna utnyttja mer än en CPU åt gången så krävs det att ett program består av mer än en tråd. En tråd kan bara köras av en CPU åt gången, och därför krävs det att ett program delas upp i flera trådar för att kunna utnyttja flera av datorns kärnor samtidigt.

Det kan vara olika svårt att dela in ett program i trådar beroende på hur mycket programmets olika delar är beroende av varandra. Ett program där de olika delarna i stort sett inte är beroende av varandra över huvud taget kallas för “pinsamt parallelt”. I ett sådant program kan trådarna hållas isolerade från varandra, och hade till största del kunnat köras på helt olika datorer. Detta kan utnyttjas för att dela upp tyngre uppgifter i oberoende delar, som kan utföras parallellt av flera olika CPU:er och på så sätt skapa mer kraftfulla datorsystem – så kallade *parallella datorsystem*.

Parallella datorsystem kan vara uppbyggda på flera olika sätt. De kan bestå av en enda stor dator med flera CPU:er kopplade till samma minne. De kan bestå av ett *datorkluster* – ett flertal separata men nära sammankopplade datorer som arbetar som om de vore en och samma dator. De kan också bestå av en *gridd* – ett nätverk av flera löst sammankopplade datorer som samarbetar via internet. Det sistnämnda är den sortens system som detta projekt fokuserar på.

Ett konkret exempel på en gridd är plattformen *BOINC*. Boinc används för att utföra tunga vetenskapliga beräkningar med hjälp av volontärer som kopplar upp sina datorer mot systemet och låter dem användas för att utöra beräkningar när de inte används till något annat (BOINC, 2013). BONC används i vetenskapliga projekt såsom SETI@home och MilkyWay@home, och har stora likheter med det system som detta projekt har som mål att utveckla.

Metod

Arkitektur

Då projektets syfte var att utveckla ett system bestående av flera samarbetande datorer var arkitektur hela tiden en central del i processen. Mycket tid lades därför på att planera arkitekturen för systemet och kommunikation mellan dess olika delar, då detta skulle vara avgörande för hur alla delar av resten av systemet skulle komma att se ut. Valet som gjordes var att bygga ett centraliserat system med en klient-serverarkitektur, där all kommunikation skedde via en central server, och ingen av de andra datorerna var direkt kopplade till varandra.

Tre nödvändiga delar identifierades; *klient*, *server*, och *arbetare*. Klienten var delen av systemet som skulle bestämma vilka uppgifter som skulle utföras. Flera kilenter skulle kunna koppla upp sig mot en server och skicka iväg uppgifter till servern för att sedan få tillbaka resultaten när uppgifterna slutförts. Serverns uppgift skulle vara att ta emot uppgifter från klienterna, för att distribuera dem bland arbetarna, och sedan skicka tillbaka resultaten från arbetarna till kilenterna. Arbetarna skulle också koppla upp sig mot en server och fungera som en sorts klienter i systemet, men med en mycket annorlunda uppgift än vad som här kallas för klienter. Arbetarnas uppgift skulle vara att hålla sig uppkopplade mot servern, och vänta på att bli tilldelade arbete. Därefter skulle de utföra uppgifterna, skicka tillbaka resultaten till servern, och återgå till att vänta på nya uppgifter.

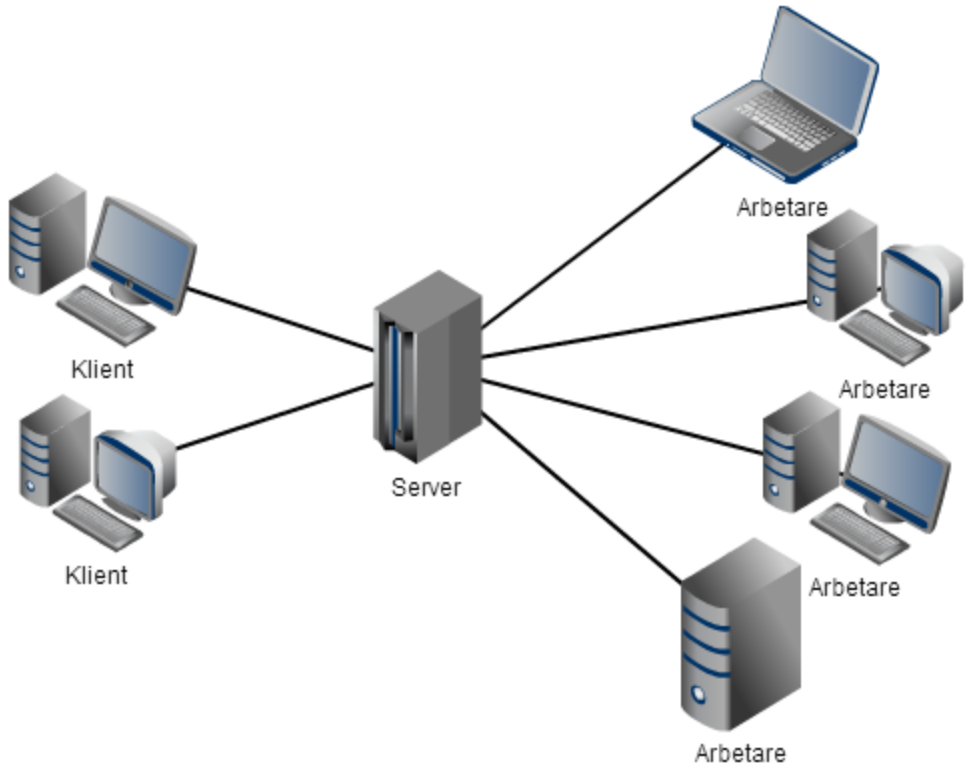

Fig 1: Planerad nätverksstruktur för projektet.

Implementation

Implementationen av systemet skedde i Java. För att kunna bygga och paketera projektet, hantera beroenden, och se till att det höll en enhetlig struktur så användes verktyget Apache Maven. Detta hade två fördelar; dels så kunde hela gruppen arbeta tillsammans trots skillnader i operativsystem och utvecklingsmiljöer, och dels så kunde projektet enkelt delas upp i separata moduler. Allt som allt bestod systemet i sex moduler; *client*, *server* och *worker* som var körbara applikationer, *server-client*, och *server-worker*, som innehöll klasser gemensamma för server- och client-modulerna respektive server- och worker-modulerna, samt *common*, som innehöll klasser gemensamma för samtliga moduler.

Det första som gjordes var att skapa alla systemets viktigare gränssnitt enligt de arkitekturella beslut som tidigare fattats. Först efter att samtliga gränssnitt skapats och dokumenterats påbörjades implementationen av de konkreta klasserna i systemet.

Något som tidigt implementerades var nätverksdelarna. Nätverkskommunikationen implementerades genom att använda javas standard-API för *sockets*, kombinerat med javas serialiserings-API. Detta gjorde att lågnivåkonstruktioner såsom användandet av byte-strömmar kunde abstraheras bort och gjorde det möjligt att skicka objekt direkt mellan de olika applikationerna utan att behöva programmera nere på byte-nivå.

Något som också implementerades ganska tidigt var möjligheten att köra Pythonskript, vilka möjliggjorde för godtyckliga uppgifter att definieras i klientapplikationen och sedan köras av arbetarapplikationen. Detta var från början inte planerat, men visade sig vara väldigt enkelt att bygga in stöd för tack vare Maven, då det räckte att lägga till Pythonbiblioteket i en konfigurationsfil för projektet.

En central del i implementationen var hanterandet av trådar. Samtliga applikationer behövde lyssna både efter användarinteraktion och nätverkskommunikation samtidigt – i serverns fall för flera uppkopplingar samtidigt. Dessutom behövde arbetarapplikationen kunna utföra flera uppgifter parallellt för att kunna utnyttja alla datorns CPU:er till så stor grad som möjligt. För att hindra systemet från att bli för komplext och hålla det trådsäkert användes ett antal olika tekniker. Icke-muterbara objekt användes i så stor utsträckning som möjligt, kommunikation mellan trådar minskades till så få platser och tillfällen som möjligt, och den kommunikation som skedde använde sig av javas trådsäkra concurrency-API.

Det grafiska gränssnittet var den del som implementerades sist. Användargränssnittet kopplades bara löst mot den underliggande arkitekturen, och lades utanpå resten av applikationerna som ett skal. De grafiska delarna implementerades med hjälp av Swing, och gjordes bara väldigt enkla, då fokus för det här projektet låg på det underliggande systemet.

Resultat

Funktionalitet

Projektet har resulterat i ett fullt fungerande system, byggt enligt den arkitektur vi ursprungligen kom fram till. Systemet stödjer att flera klienter och arbetare kopplas upp mot en server, och uppgifter kan sedan skickas från klienterna för att utföras av arbetarna. De uppgifter som stöds är primtalsfaktorisering, hittande av nästa primtal givet ett tal, samt exekvering av användardefinierade Pythonskript.

Systemet har funktionalitet på plats för att fördela uppgifter mellan arbetare på ett sådant sätt att samtliga arbetares kapacitet alltid utnyttjas till så stor grad som möjligt. Systemet kan även hantera om en ny arbetare ansluter, eller om en arbetare skulle få slut på arbete medan andra arbetare har hög arbetsbelastning på sig; och kan då omfördela arbetet så att belastningen blir jämnare. På samma sätt kan det hantera om en arbetare skulle koppla ifrån sig, genom att dela ut den frånkopplade arbetarens återstående uppgifter till återstående arbetare. Funktionalitet finns även för att avbryta uppgifter, både från klientsidan som följd av användarinteraktion, och från serversidan i fallet då uppkopplingen till en klient stängs.

Både klienten och arbetaren har grafiska gränssnitt för att se information och historik. Det grafiska gränssnittet i arbetarapplikationen fyller ingen särskilt viktig funktion, utan används snarare för att visualisera statistik. Klientens användargränssnitt har ett viktigare syfte, då en uppgift måste kunna definieras på ett överskådligt sätt. Klienten har funktionalitet för att skapa, skicka iväg och avbryta uppgifter, samt visa resultat för slutförda uppgifter. För Pythonskript finns en textredigerare med syntaxmarkering (*syntax highlighting*), som även stödjer inladdning av filer. Servern har ett väldigt enkelt kommandoradsgränssnitt för att kunna köras av en administratör via terminalen, och exempelvis kunna startas och köra via SSH.

Begränsningar

För att systemet ska kunna fungera krävs det att datorerna i fråga kan nå varandra via nätverket. Detta kräver att inga brandväggar är i vägen, eller att de portar som används av systemet är öppna. Om servern dessutom ligger bakom en router i ett hemnätverk krävs det att routern konfigureras för att vidarebefordra anslutningar till dessa portar till serverdatorn.

Det finns inte heller något sätt att snabbt skapa stora mängder av uppgifter på en och samma gång i klientapplikationen. För närvanrande måste alla uppgifter skapas för hand, vilket begränsar klientens nuvarande när det kommer till att användas i annat än demonstrationssyfte.

Avslut

Diskussion

Den färdiga produkten är fullt fungerande på så sätt att den uppfyller de krav som ställdes i början av projektet. Dock finns det ett antal frågeställningar att behandla:

Valet av Python som skriptspråk gjordes enbart på grund av hur lätt det kunde läggas till i systemet, utan några större reflektioner över vilka alternativ som fanns tillgängliga. Det gav systemet stöd för att utföra godtyckliga uppgifter; men kunde detta ha gjorts på ett bättre sätt? Python är ett språk som ändå är förhållandevis trevligt att skriva saker såsom matematiska beräkningar i, så ur användarvänlighetssynpunkt är valet inte helt fel.

Vad man däremot kan ifrågasätta är om Python är snabbt nog för tyngre beräkningar. Det skulle med stor sannolikhet gå snabbare att köra kompilerad javakod i stället. Jämförelsevis är dock javakod betydligt mycket svårare att föra över mellan datorer, då det krävs att en kompilerad klassfil laddas in med en klassladdare innan koden kan köras. Dessutom är javakod betydligt mycket tyngre för användaren att skriva än Python, men beroende på hastighetsskillnaden skulle det fortfarande kunna vara värt det. Om hastighetsskillnaden inte är särskilt stor kan det tänkas att man tjänar mer på att använda Python och koppla upp en extra arbetare mot systemet, än man hade gjort på att implementera java.

Något annat som är värt att reflektera över är det fokus som valdes vad gällde typen av uppgifter som systemet skulle kunna utföra. Planen var från början att stödja ett fåtal specifika typer av uppgifter, vilket var något som också följdes. Men då funktionalitet för att köra godtyckliga skript byggdes in kunde fokus kanske ha lagts på att enbart stödja skript, och förbättra stödet för dem i stället.

Hade systemet designats med Pythonskript i fokus från början, så hade systemet kanske sett annorlunda ut, och kanske varit mer användbart nu. Det skulle ju ha gått att fokusera på att låta användaren skriva en funktion som bara skickas till servern en gång, och sedan bara skicka indata till funktionen, som även den kan genereras av användarskript. Detta skulle antagligen ha gjort systemet mer praktiskt användbart, och är definitivt något värt att tänka på i framtiden.

Vidare kan man alltid diskutera systemet ur säkerhetssynpunkt. Systemet bygger för närvarande på antagandet att samtliga parter är pålitliga, något som säkerligen inte kan förväntas vara sant om systemet skulle användas i större sammanhang. Ett förhållandevis enkelt skydd vore att kräva autentisering vid anslutning av klienter och arbetare för att åtminstone försvåra för obehöriga parter att ansluta sig till systemet. Det är långt ifrån vattentätt, men det är fortfarande bättre än inget skydd alls.

Samtidigt finns det faktiskt vissa begränsningar för vad Pythonskript kan göra när de körs av arbetarapplikationen. De körs i en lite mer skyddad miljö, som är gjord för att begränsa deras

privilegier i så stor mån som möjligt. Inte heller detta är vattentätt, men det försvårar för illvilliga kilenter att ställa till alltför stor skada. Dessutom finns det en begränsning för hur länge ett skript kan köra innan det avbryts. I detta fall kan man i stället vända på resonemangen och fråga sig om denna extra säkerhet verkligen är nödvändig i ett system som bygger på antagandet att alla parter är pålitliga. Om man ändå litar på klienterna kanske det är onödigt begränsande att sätta in spärrar mot vad de får göra och hur länge deras skript får köra. Om de märker att något blivit fel så kan de trots allt avbryta uppgiften.

Källförteckning

Göetz, Peierls, Bloch, Bowbeer, Holmes, Lea: *Java Concurrency in Practice*
Steam http://store.steampowered.com/hwsurvey/ 2013-05-15