Universitetet i Oslo
Institutt for informatikk

Forbedring av kjøresystemet i PRP

Jørn Christian Syversrud

Masteroppgave

1. August 2007

Forord

For noen år siden møtte jeg opp på Universitet i Oslo som en uerfaren og spent student. 5 år senere står jeg igjen med en oppgave som avslutter mitt masterstudie ved instutt for informatikk. Det har vært en lang og innholdsrik opplevelse, og det flere personer jeg vil takke gjennom denne tiden.

Først og fremst vil jeg takke min veileder Arne Maus. Han har bidratt med gode råd og ideer, og oppmuntret på tider det har stagnert litt.

Jeg vil også takke Cuong Van Truong(15), som har vært min samarbeidspartner gjennom denne masteroppgaven.

Jeg vil også rette en stor takk til min samboer Ingrid som har holdt ut med meg, og oppmuntret meg når problemer har dukket opp.

Takk til min familie, spesielt mamma og pappa for god oppmuntring underveis.

Videre vil jeg takke Philip, Nils og Kenneth for tiden vi bodde sammen i Blindernveien.

Takk til Ole Sverre, Øyvind, Arnstein, Håvard og Håkon for artige tider utenfor studiet.

Jørn Christian Syversrud

Innhold

Figurer

Tabeller

Kapittel 1

Introduksjon

Parallellisering har blitt et veldig aktuelt tema de siste årene innenfor software-utvikling. En vesentlig grunn til dette er at hardwareprodusenter velger å gi ut datamaskiner og spillekonsoller med flere prosessorer. Vi kan f.eks nevne Sony som i disse dager gir ut Playstation 3 med 7 prosessorer.

I denne masteroppgaven vil jeg beskrive hvordan vi kan utnytte flere maskiners totale regnekraft ved å parallellisere rekursive metoder. Nedenfor gir jeg en kort beskrive av hva de forskjellige kapitlene vil inneholde.

Kapittel 2: Datasystemer og parallellitet I det første kapittelet går jeg igjennom kjente problemer rundt parallellisering og synkronisering. Videre vi jeg nevne kjente modeller for parallellisering. Deretter beskriver jeg rekursive metoder og hvordan man parallelliserer disse i PRP-systemet.

Kapittel 3: Videreutvikling av JavaPRP I dette kapittelet beskriver jeg kort hva min masteroppgave går ut på. Videre beskriver jeg viktige klasser som PRP-systemet i dag består av.

Kapittel 4: Utnyttelse av flerkjernemaskiner Her beskriver jeg hva som skal til for å kunne utnytte maskiner med flere kjerner, hyperthreadmaskiner eller maskiner med flere single cpuer. Videre viser jeg til kjøreresultat-

er og måle disse resultatene opp mot andre kilder som har utført lignende tester.

Kapittel 5: Oppsplitting av vanskelige parametersett I dette kapittelet går jeg igjennom de forskjellige problemene som dukker opp når vi skal splitte vanskelige parametersett. Jeg vil vise til hvordan parametersett håndteres i det eksisterende systemet, og argumentere videre hva som skal til for å løse oppgaven.

Kapittel 6: Design av problemstilling og uttesting for Travelling Salesman Her drøfter jeg og velger en passende problemstilling som videre brukes for å teste oppsplitting av vanskelige parametersett. Her viser jeg til resultater av 3 forskjellige kjøringer og begrunner de forskjellige utfallene.

Kapittel 7: Oppsummering og videre arbeid I dette kapittelet går jeg igjennom hva jeg har oppnådd i oppgaven. Jeg beskriver videre hva som gikk bra, og hva jeg ville gjort annerledes om det hadde vært mer tid. Avslutningsvis vil jeg nevne tanker og ideer for videre arbeid.

Kapittel 2

Datasystemer og parallellitet

Vi kan i dag merke oss at nye maskiner som kommer på markedet ikke har så store forbedringer i forhold til det forrige slippet, og da spesielt med tanke på CPU-hastighet. Listen har nå lenge stått på ca. 3 GHz og det ser nesten ut til at det må skje noe revolusjonerende for at framgangen skal ta fatt igjen.

Intel og AMD har valgt å komme ut med doble prosessorkjerner og vi ser at fokuset har gått fra å gjøre en oppgave raskest mulig til å gjøre mest mulig på en gang. En prosessor utstyrt med to kjerner gjør at man kan kjøre to programtråder samtidig, uten at de må dele på prosessorkraften.

Grovt sett kan man dele parallellisering inn i tre nivåer. Øverst ligger programmet brukeren benytter seg av. Laget under ligger selve programmet. Her finner vi eksempelvis MPI, HPF og PRP som vi kommer tilbake til senere i denne teksten. Under program ligger maskinen programmet kjører på. Emner rundt sistnevte kommer ikke til å være fokuset i denne teksten.

2.1 Løs/Fast kobling

Ved parallell prosessering trenger prosessene i systemet å kommunisere med hverandre, og man skiller her mellom to typer koblinger: løst og fast.

I ett fast koblet system jobber prosesseringsenhetene via en og samme buss(mange prosessorer i en maskin). Kommunikasjonen mellom proses-

sene har lav feilrate og har høy ytelse. Supercomputere er ett eksempel på et fast koblet system. I ett løst koblet system er maskinene koblet via f.eks ett LAN, og kommunikasjonen mellom dem er mer upålitelig og langsommere en i ett fast koblet system.

Fra beskrivelsen kan det virke som om det er fast koblet system som er det beste, men vi kan i dag se at det er sistnevnte som virkelig har slått ann, og det er mange grunner til dette. Vi kan her nevne noen:

- Maskiner med høy ytelse er i dag relativt billig.
- Hastigheten over internett har blitt markant bedre(opp mot 25 Mbit/sek for en vanlig bruker, og et Gigabit LAN gir effektivt 300-500 Mbit/sek).
- Om det skulle komme en ny prosessor er det relativt enkelt å sette inn denne i maskinen.
- Eksisterende software kan bli brukt eller modifisert.

Det som til gjengjeld gjør det vanskelig ved slike systemer, enten det er løst eller fast, er designet av algoritmene som tas i bruk. Da man fra før er vant med å bruke algoritmer for sekvensiell kjøring, må man nå tenke parallelt og dette er ofte utfordrende og krever mye tid.

2.1.1 Cluster

Clustere(klynger) er et eksempel på ett løst koblet system. I klynger er flere maskiner koblet sammen via ett nettverk, som oftest ett LAN, og fungerer som en stor maskin. Måten det skjer på er at det er en koordinator som deler opp jobber som skal utføres i mindre biter, og deler deretter ut disse til maskinene på nettet. Maskinen som mottar en oppgave, vil løse denne og returnere svaret så fort den er ferdig.

2.1.2 Grid

Et annet eksempel på ett løst koblet system er det man kaller Grid computing(gitter eller samkjøringsnett hos Gyldendal). Tanken bak Grid

computing(først beskrevet av Foster og Kesselman (9) i 1999) er å utnytte den totale regnekraft i ett distribuert system for å løse ett enkelt problem, samtidig. Dette kan være vitenskaplige eller tekniske problemer som til felles krever tilgang til store maskinressurser eller store datamengder. I forhold til klynger skiller Grid seg ut med at det strekker seg over ett helt land eller hele verden, og ikke bare til ett lokalt nettverk. Man kan derfor se på Grid computing som en mengde av klynger koblet sammen.

Man kan koble tanken bak Grid til strømnettverket som i dag finnes over hele verden. I ethvert hjem har man ett strømuttak som man vet vil gi den strømmen man trenger. Denne ideen vil man overføre til dataverden. Ved å enkelt koble en maskin til nettverket så vil den til enhver tid har de ressursene den trenger, enten det er dataressurser eller maskinkraft til å løse ett komplisert matematisk problem.

Ett stort prosjekt innenfor grid computing foregår i disse dager hos CERN i Europa. CERN driver med partikkelforskning og har utviklet en partikkelakselleratør(Large Hadron Collider) som vil starte opp i 2007. 12-14 PetaByte(1 PB = 1 mill GB) med data vil bli generert hvert år og man trenger prosessorkraft tilsvarende 70.000 maskiner for å analysere dette. CERN skjønner at de selv ikke vil klare å skape denne ressursen lokalt og søker derfor Europa og andre verdensdeler om hjelp for å klare å skape ett grid som vil gjøre denne dataanalysen mulig(1).

2.2 Synkroniseringsmekanismer

Når flere prosesser kjøres samtidig, medbringer dette ganske mye problemer og faren for deadlock er stor. Deadlock vil si at prosess A venter på prosess B og at prosess B venter på A. Dette gjør at programmet aldri vil terminere, noe som ikke er ønskelig. Ett annet problem som kan oppstå er kampen om delte ressurser(variable) og at utfallet av ett program ikke alltid vil være den samme(ikke deterministisk). Vi kan illustrere med et lite eksempel:

```
int x = 0;  //felles variable

prosess 1(P1) skal kjøre: x = x + 1;
prosess 2(P2) skal kjøre: x = x - 1;
```

```
Det er 4 atomære x-operasjoner: P1 leser (R1) verdien i x, P1
skriver (W1) en verdi til x, P2 leser (R2) verdien i x, og P2
skriver (W2) en verdi til x. R1 må skje før W1 og R2 før W2, så
disse operasjonene kan sekvenseres på 6 måter:

R1 R1 R1 R2 R2 R2
W1 R2 R2 R1 R1 W2
R2 W1 W2 W1 W2 R1
W2 W2 W1 W2 W1 W1
-----------------
 0 -1  1 -1  1  0
```

Utifra et så lite problem, er det 3 forskjellige utfall med 6 ulike sekvenseringer av operasjonene. Når store parallelle programmer skal synkroniseres så er dette en komplisert oppgave for programmereren. Siden man har vist om disse problemene lenge har det blitt utviklet forskjellige mekaniser for å kontrollere flere prosesser, og man deler dem grovt inn i to forskjellige former:

- Ekslusiv tilgang
  Her passer man på at kun en prosess har tilgang til en kritisk region. Med kritisk region mener vi tilgangen til felles ressurser.

- Betingelse synkronisering
  Her venter en gitt prosess på at en betingelse skal bli sann.

2.3 Ulike arkitekturer for parallell programmering

Flynn presenterer en modell(2) for å klassifisere forskjellige arkitekturer innenfor parallellprogrammering:

SISD - Single Instruction, Single Data

Dette beskriver en maskin med en prosessor. Her kan det utføres en instruksjon på ett dataelement. Det er derfor ikke mulig med parallell prosessering.

SIMD - Single Instruction, Multiple Data

Dette beskriver en maskin som utfører en instruksjon på mange dataelementer samtidig. Eksempel på dette er vektor prosessorer.

MISD - Multiple Instruction, Single Data

Her utfører mange instruksjoner på ett og samme data. Denne metoden ses på som lite effektiv og er nesten ikke brukt.

MIMD - Multiple Instruction, Multiple Data

Dette metoden er den som ses på som den mest normale. Her jobber uavhengige prosessorer på forskjellige data. Flere prosessorer i ett nettverk er ett eksempel på dette.

SPMD - Single Program, Multiple Data

Her vil ett enkelt program kjøre på flere uavhengige prosessorer, hvor hver prosess kjører ulik data gjennom samme program.

2.4 Systemer for parallell programmering

Før vi nå retter fokuset på PRP og dens virkemåte, så kan det være av interesse å nevne andre former for parallell programmering:

2.4.1 PVM

Står for Parallell Virtual Machine og er ett meldingsbasert system hvor brukeren selv kan konfigurere flere arbeidsstasjoner som en distribuert parallell maskin over ett nettverk. Systemet består av to deler: en prosess(eller flere) som kjører på alle maskiner som tilsammen tilsvarer den virtuelle maskinen og en som tar for seg koordineringen mellom maskinene.

Det er i dag mulig å kjøre PVM på nesten alle arbeidsstasjoner. Når en pakke blir sendt fra en prosess til en annen så sørger PVM(PVM deamon) selv for å levere riktig pakke til riktig prosess og på korrekt format. En annen ting som lar seg ordne i PVM er forholdet mellom prosesser og prosessorer og kjøringen av disse. Er det flere kjørende prosesser en tilgjengelige prosessorer, så tilordner PVM de forskjellige prosessene til prosessorene så lenge brukeren selv ikke spesifiserer noe annet i koden.

Vi kategoriserer PVM som et MIMD system. Dette fordi den støtter flere typer kommunikasjon mellom prosessene og kan kjøre forskjellige deler av applikasjonen i parallell.

2.4.2 HPF

Står for High Performance Fortran og er en utvidelse av FOTRAN 90 for parallell programmering. Utvidelsen er ment for å løse numeriske simuleringer parallelt, som f.eks array multiplikasjon.

HPF ses på som et SPMD system da den benytter seg av dataparallellisme, og utfører en instruksjon på forskjellige data samtidig. Som vi skal se senere så har HPF mye likt med PRP systemet med det at brukeren setter forskjellige direktiver som kompilatoren senere parallelliserer.

2.4.3 MPI

Står for Message Passing Interface og er ett sett bibliotekrutiner for meldingsutveksling. Protokollen har nå blitt en de facto standard for kommunikasjon mellom noder i ett parallellt program. Fokuset til MPI er

å få en effektiv kommunikasjon over ett heterogent nettverk av prosesser og prosessgrupper.

Siden MPI er ett sett bibliotekrutiner så blir det vanskelig å klassifisere det som noen av modellene under pkt. 2.2. I steden kan vi se på det som ett verktøy for utvikling av distribuerte parallelle systemer.

2.4.4 Barrier synchronization

Barrier synchronization er en form for sykroniseringsmekanisme for parallelle prosesser. Det er slik at når flere prosesser kjører en og samme metode så blir de som oftest ikke ferdig samtidig(om dette er ønskelig). Det som da gjøres er å ha ett punkt i programmet(en barriere) hvor man da kan be prosessene å vente til alle prosesser har kommet ditt. Dette kan enkelt gjøres ved å ha en teller som vet hvor mange prosesser det er. Når telleren er like stor som antall prosesser så tar siste prosess å sier ifra til alle andre prosesser om at nå fortsetter vi eksekveringen, og samtidig nuller ut telleren.

2.5 PRP

Det å bygge opp et parallelt program fra grunnen kan være en vanskelig og ikke minst en tidkrevende jobb. Viss man isteden hadde ett program som kunne hjelpe brukeren i å parallellisere sine programmer, så hadde dette vært det ideelle, og det var denne tanken Arne Maus presenterte i 1978. Ideen var først ment for ett fast koblet system, hvor hver prosessor hadde sitt eget lokale minne, men i årene som har gått, har maskiner i nettverk blitt veldig vanlig. Ideen har derfor blitt utvidet til å fungere på maskiner tilknyttet internett og/eller LAN.

2.5.1 Administrator/arbeider-modell

PRP baserer seg på en administrator/arbeider-modell. Det vil si at vi har en maskin som administrerer kall ut til en eller flere arbeidermaskiner. Arbeidermaskinene vil arbeide parallelt på hvert sitt delproblem og returnerer svaret til administrator. På denne måten blir PRP en parallell til

klient/tjener-modell, hvor klienten er administrator, og arbeidermaskinene er tjenere.

2.5.2 Kommunikasjon

I de siste versjonene av PRP har man valgt Java som programmeringsspråk. For å få til kommunikasjon mellom arbeider og administrator har valget derfor falt på javas egne implementasjon av RPC, nemlig RMI(Remote Method invocation). Java RMI benytter seg av Javas plattformuavhengighet til å tilby robust mellomvare som fungerer på tvers av forskjellig maskinvare og operativsystemer.

Som i RPC, skal man også i RMI oppleve kall på en metode som ett lokalt kall, selv om koden eksekveres på en annen virtuell maskin.

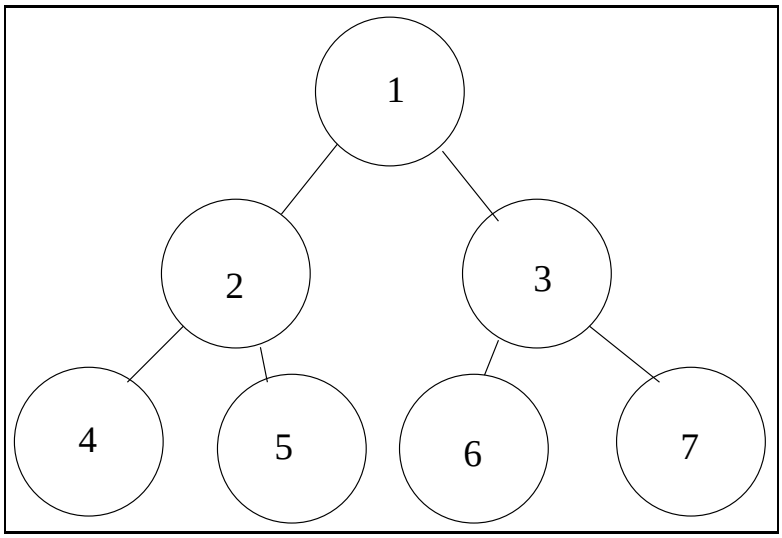
2.5.3 Rekursjon

Som tittelen antyder så kan ikke PRP parallellisere ett hvilke som helst program, men begrenser seg til de som løser sine problem rekursivt. En metode er direkte rekursiv om den kaller seg selv minst en gang, og indirekte rekursiv om det er slik at metode 1 kaller metode 2, som igjen kaller metode 1. Om en rekursiv metode også skal parallelliseres så er det viktig at den har en fanout(treets grad) større eller lik 2. Om en rekursiv metode har fanout lik 1(halerekursjon), vil ikke parallellisering være mulig For ett sekvensielt program ville det også vært ineffektivt da ett kall på en metode skaper mer overhead, enn om man brukte en vanlig for_/while løkke.

For en rekursiv metode vil det være normalt å traverse treet bredde først, eller dybde først. Fra figur 2.1 på neste side skal vi se på hva de to forskjellige måtene vil si:

1. **Bredde først:** Her vil nodene bli traversert i følgende rekkefølge: 1,2,3,4,5,6,7.

2. **Dybde først:** Her vil nodene bli traversert i følgende Rekkefølge: 1,2,4,5,3,6,7.



Figur 2.1: Rekursjon

Vi skal senere i avsnittet om administratoren, se at PRP benytter seg av en blanding av disse to måtene å traversere ett tre på.

2.5.4 Språkprimitiver

For en normal bruker med minimal kunnskap om parallellisering, skal det å gjøre om sine sekvensielle program være enkelt å intuitivt. Man har derfor i PRP valgt å innføre to(eller tre) nøkkelord som brukeren skriver inn i sine sekvensielle programmer.

```
/* PRP_FF */
class PrpEksempel {

    .....

    /* PRP PROC */
    int rekursivMetode(...) {
        for(int i = 0; i <= 2; i++){

            /* PRP CALL */
            int x = rekursivMetode(...);
```

```
            .....

        }
    }
}
```

Som ett eksempel viser vi i programmet over hvordan disse nøkkelordene kan brukes:

- **PRP_PROC**: Dette er et nøkkelord som må puttes inn rett før den rekursive metoden.
- **PRP_CALL**: Dette nøkkelordet puttes inn rett før det rekursive kallet blir gjort inne i metoden.
- **PRP_FF**: Dette nøkkelordet er ikke nødvendig, men kan brukes om det er ønskelig å bruke full fanout. Med full fanout mener vi at alle noder(utenom rotnoden) er direkte barn av rotnoden.

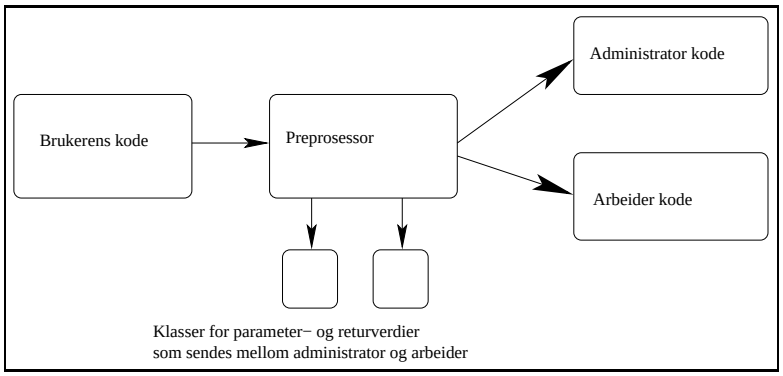
Det disse språkprimitivene har til felles er at de alle forteller hvor preprosessoren skal legge til kode for å få programmet parallellisert.

2.5.5 En dypere forståelse av PRPs virkemåte

For en bruker skal det å bruke PRP være en ganske enkel oppgave, og man slipper å tenke på alle de vanskelige mekanismene som ligger rundt parallellisering. Vi skal derfor i dette avsnittet gå bak kulissene og se på de ulike komponentene og deres virkemåte.

De forskjellige komponentene

I GUI-PRP har man 3 komponenter å forholde seg til: preprosessoren, arbeidermaskin og administrator



Figur 2.2: Bruk av PRPs preprosessor

Preprosessoren

Det første som må gjøres i PRP er at preprosessoren leser inn brukerens kode. Den vil deretter lese seg fram til de stedene hvor brukeren har skrevet inn de forskjellige nøkkelordene. Utifra disse nøkkelordene vil den muliggjøre for distribuert og parallell kjøring av programmet.

Den vil videre generere kode som arbeidermaskinenen kan kjøre, kode som tjenermaskinene kan kjøre og klasser for parametre og returverdier som skal brukes i kommunikasjonen mellom disse.

Når preprosessoren er ferdig med å generere filer, må disse kompileres. Ved den siste implementasjonen av Kristiansen skjer dette automatisk. En fullstendig illustrasjon av hva preprosessoren gjør er illustrert i figur 2.2

Arbeidermaskinen

For å starte opp en arbeider holder det at den er tilknyttet internett. Ved oppstart starter arbeidermaskinen en tjener som hele tiden vil stå å lytte på om det kommer noe forespørsel fra administrator. Når den mottar en oppgave vil den ta hånd om dette ved å kjøre normal dybde først rekursjon, for deretter å returnere svar objektet. I dette objektet vil det også være informasjon om hvor i stakken administratoren skal legge svaret.

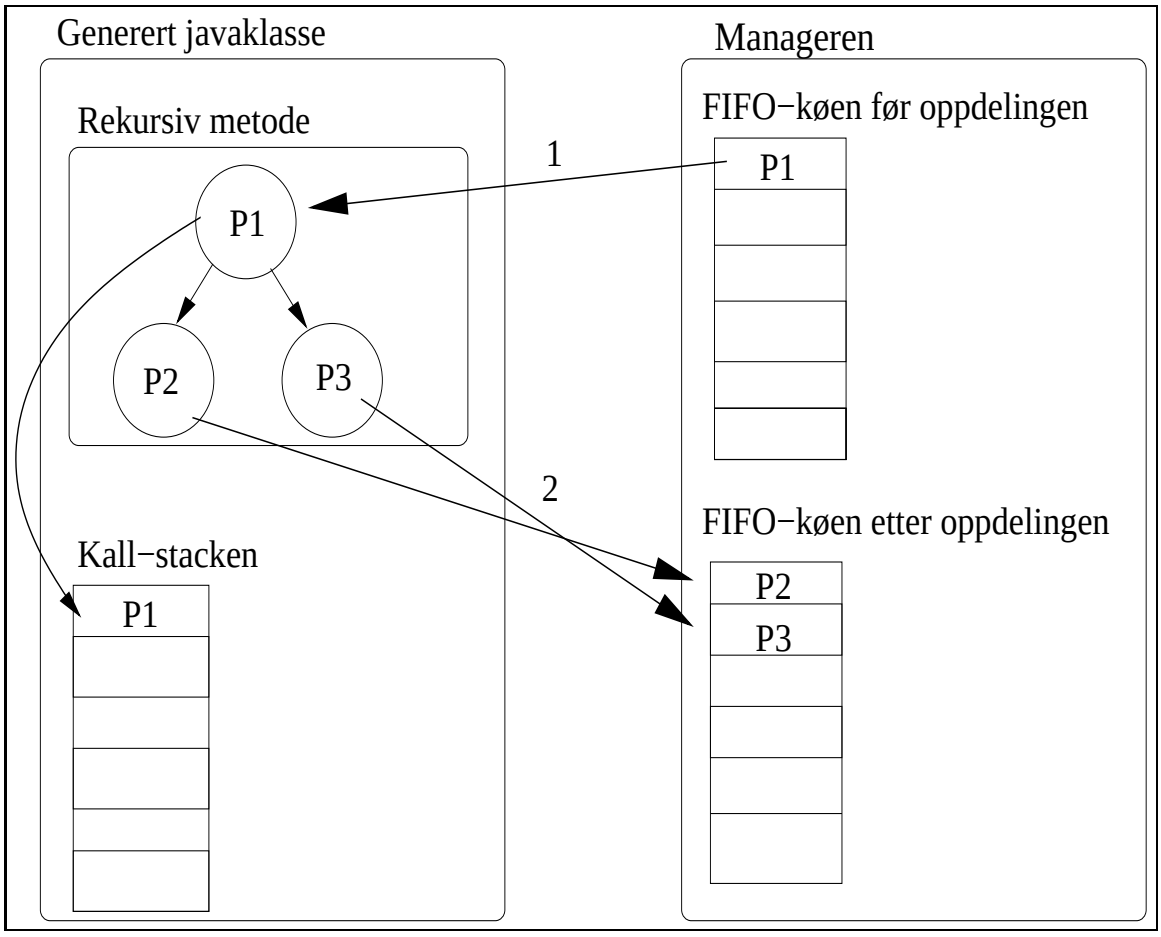
Administrator

Ved oppstart av administratoren så vil den forsøke å få kontakt med arbeidermaskinene. Siden java støtter bruk av tråder("threads") blir det laget en tråd for hver arbeidermaskin. Hver og en av disse trådene vil kontakte en prosess(RMIregistry) på den maskinen den er tildelt. Denne prosessen fungerer som en navnetjener og vil returnere en peker til en arbeiderfabrikk. Videre vil Tråden kalle denne arbeiderfabrikken med koden den ønsker den skal kjøre. Arbeiderfabrikken vil da lage ett objekt instansiert med denne koden og returnere en peker tilbake til tråden. Når tråden mottar denne pekeren vil den sjekke med en gang om administrator er ferdig med parametergenereringen. Er den ikke det vil tråden sove imens.

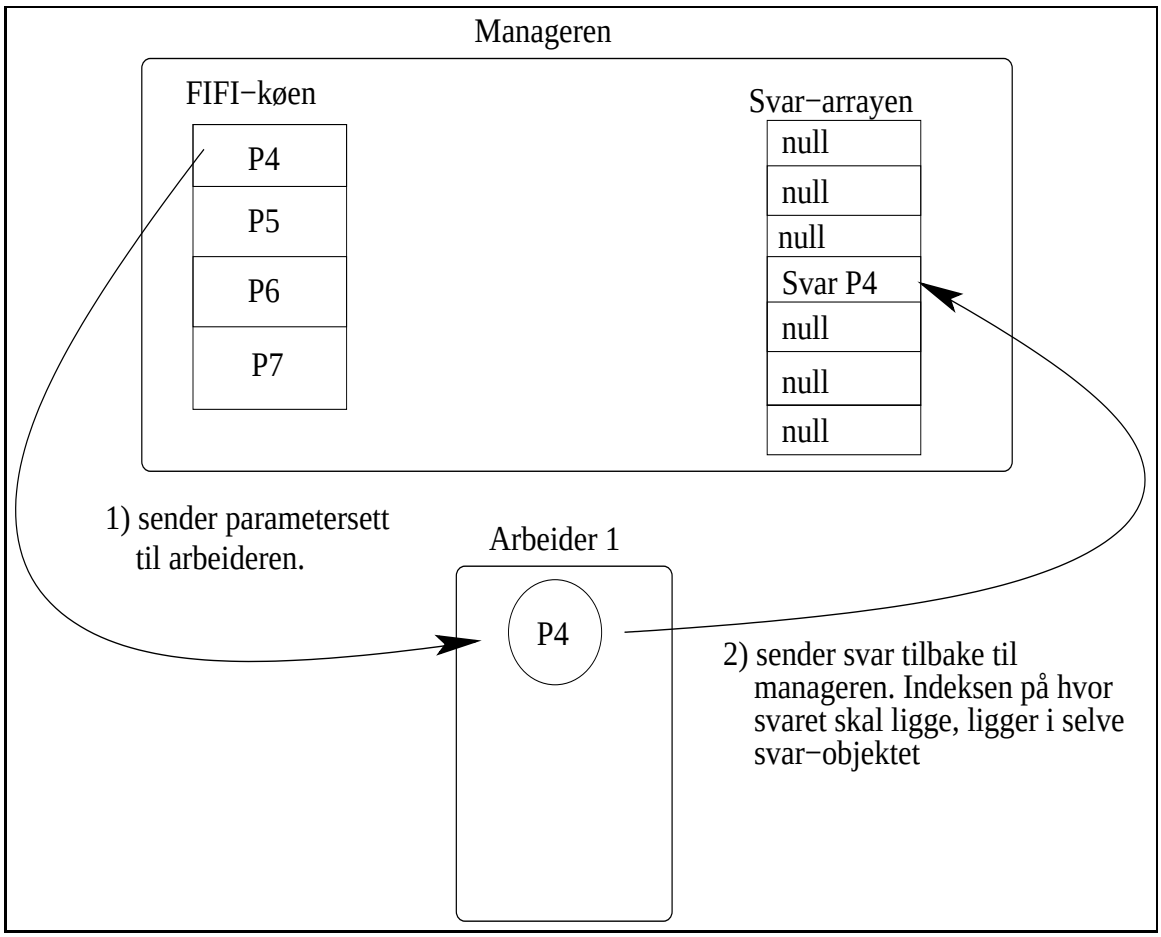
I det administratoren startet tråder for å kontakte alle arbeidermaskinene, startet den også en tråd for å generere parametersett. Det å generere parametersett vil si å dele opp startparameteret til den rekursive metoden i mindre biter. På denne måten kan administrator gi forskjellige deler av problemet til arbeiderene senere. Vi skal her gå stegvis å se hvordan parametergenereringen foregår(se figur 2.3 på neste side).

- Det initsielle parametersettet P_1 sendes til den rekursive metoden. P_1 blir først lagt på kall-stacken. Denne blir brukt senere under svargenereringen. Videre blir koden kjørt fram til kallstedet, og parametersett P_2 og P_3 blir generert og puttet på en FIFO-kø.
- Administrator henter ut P_2 fra FIFO-køen og sender det til den rekursive metoden. Koden blir kjørt fram til kallstedet, og parameterett P_4 og P_5 blir generert og puttet på FIFO-køen.
- Administrator henter ut P_3 fra FIFO-køen. Videre skjer det samme som i punktet over. Dette vil fortsette til FIFO-køen er fylt opp med parametersett(som regel 20 ganger antall maskiner). Fordi vi bruker en FIFO-kø ser vi at kall-treet blir traversert bredde først.

Administratoren er nå klar for å løse oppgaven. Ved å sende parametersett fra FIFO-køen til arbeidermaskinene, vil disse jobbe på hvert sitt delproblem, parallelt. Hver arbeider vil nå løse sin delproblemer ved hjelp av dybde først rekursjon. Fra figur 2.4 på side 28 kan vi se at manageren sender ut parametersett P4 til arbeider 1.



Figur 2.3: Parametergenerering.

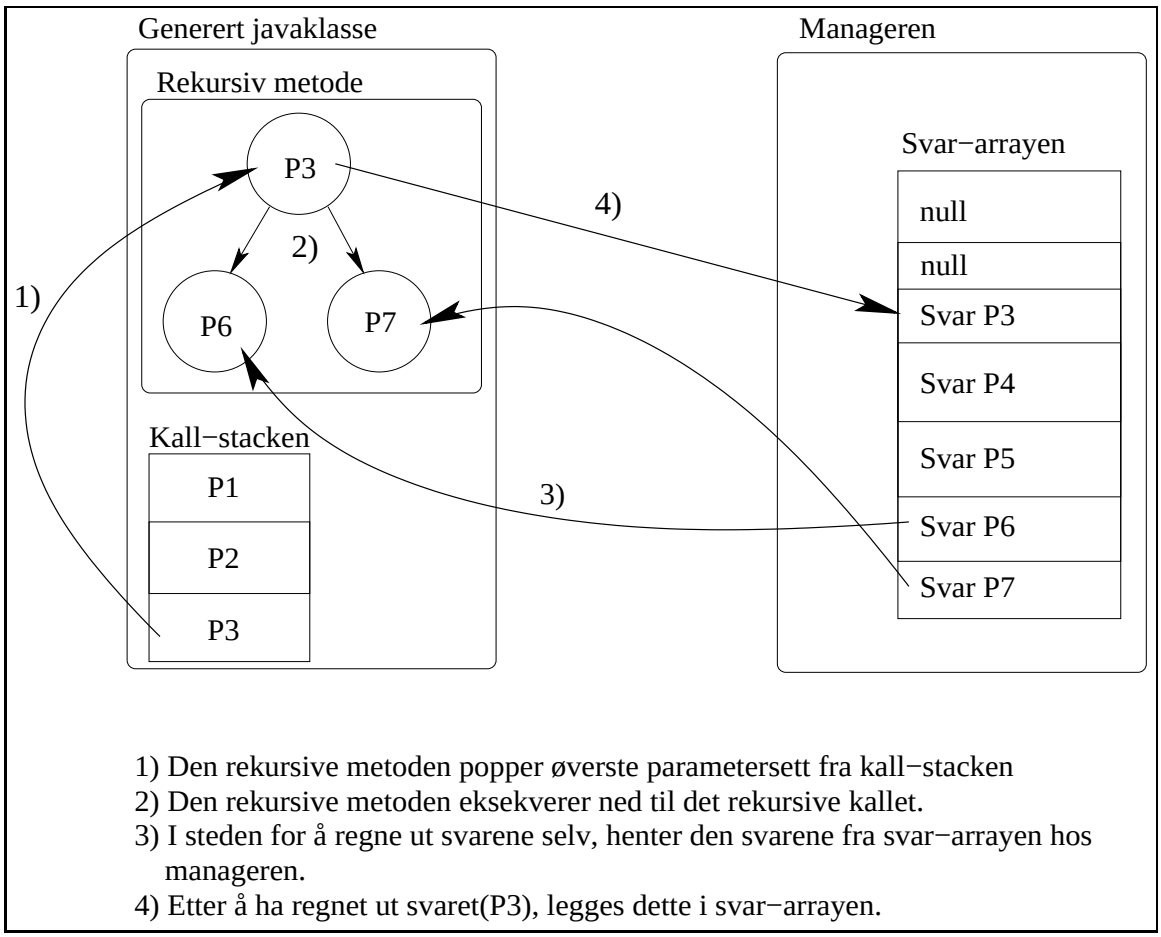


Figur 2.4: utsending og mottak av parametersett under runtime

Ettersom arbeiderne blir ferdig vil administrator motta svar, og legger disse i en svararray(hvilke posisjon svaret skal ligge i arrayen, finner man i svarobjektet). Fra samme figur nevnt over kan vi se at arbeider 1 er ferdig med sitt parametersett og sender svaret tilbake til manageren.

Når samtlige parametersett er løst vil administrator starte svargenereringen(se figur 2.5 på neste side). Svargenereringen skjer i følge steg:

- Administratoren henter øverste parametersett P_n, fra kallstakken(punkt 1 fra figuren).
- Administrator vil deretter kjøre den rekursive metoden med dette parametersettet fram til kallstedet(punkt 2 fra figuren).



Figur 2.5: svargenerering

- Nå er det ikke noe poeng i å gjøre det rekursive kallet siden dette er blitt løst av arbeiderne. Administratoren henter derfor ut disse svarene fra svararrayen(punkt 3 fra figuren).

- Svarene blir videre brukt til den delen av koden som kommer etter det rekursive kallet, og et nytt svar vil bli generert og lagt på svararrayen(punkt 4 fra figuren).

- Administrator henter ut parametersettet P_{n-1}, og samme prosedyre gjentar seg. Til slutt vil man komme til det initielle parametersettet P_1, og det endelige svaret kan kalkuleres og vises til bruker.

2.5.6 Historikken til PRP-systemet

Etter Maus(12) sitt ide-notat på slutten av 70-tallet, har det blitt gitt flere hovedfagsoppaver på PRP systemet. Vi skal her ta en titt på hva som er blitt gjort fram til i dag.

Thorfinn Aas I 1994 ble den første hovedfagsoppgaven på PRP systemet ferdig. Oppgaven til Aas(3) ble å implementere ett løst koblet PRP system, og han valgte å gjøre dette i programmeringsspråket C. Aas valgte i sin implementasjon å benytte seg av ett sett språkprimitiver som brukeren skriver i sitt program. Programmer går deretter igjennom en preprosessor som så legger til kode for parallellisering.

For kommunikasjon mellom prosessene velger Aas å bruke RPC(Remote Procedure Calls). RPC er en teknikk som gjør det mulig å kalle en annen prosedyre utenfor samme adresserom. Dette vil si at prosedyrenene kan være på samme maskin eller koblet sammen via ett nettverk(LAN).

Yan Xu Xu(16) leverte sin oppgave i 1997 og hennes mål var å videreutvikle systemet ved å gjøre det mer brukervennlig og portabelt. I steden for Aas valg av RPC(prosedyre kall) velger Xu å benytte seg av meldingsutveksling og faller derfor på MPI(beskrevet i pkt 2.4). Siden disse to er så ulike i oppbygningen velger hun å bygge opp systemet fra grunnen av.

Noe av forbedringene Xu gjorde var:

- Systemet kan bruke vilkårlig mange parametre og returverdier.
- Man slipper å tenke på prosedyrens fanout da Xu's implementasjon beregner dette selv.
- Man kan finne ut hvilke rekursjonsgren et svar kommer ifra.
- Innfører en boolsk matrise som hele tiden holder oversikt over hvilke maskiner som er ledig til enhver tid.

Videre velger Xu å kun benytte seg av kun en versjon av den rekursive prosedyren da hun syntes disse(parallelle og ikke parallelle) fungerer ganske likt ved å legge til noen tester. Etter testing viser dette seg å ikke være lønnsomt.

Arne Høstmark Høstmarks(10) hovedfagsoppgave var ferdig i 1997, og hans oppgave bygger videre på Aas. Oppgaven nå ble å skalere systemet til å fungere på mange maskiner. En annen oppgave Høstmark sto ovenfor var å gjøre systemet mer tolerant for feil. Han valte derfor å innføre ett backup system. Systemet funket slik at ved oppstart så ble det satt av noen maskiner som ikke skulle brukes enda. Hvis det nå skulle skje noe kommunikasjonssvikt mellom to noder senere i eksekveringen så tar backsystemet av seg det ved å eksludere den maskinen som feilet og tar dens oppgave og sender til en av de ventende maskinene i backup systemet. En annen ting Høstmark implementerte var kalkuleringen av prosedyrens fanout, slik Xu gjorde i sin implementasjon.

Viktor Eide Eide(7) leverte sin oppgave i 1998 og også hans oppgave byggere videre på Aas. I denne oppgaven blir PRP systemet implementert ved hjelp av PVM(beskrevet i pkt 2.4). Han velger å innføre en løsning ved fordeling av kall som brukes til dags dato. Måten denne fordelingen skjer på er at administratoren deler opp problemet i mange flere deler en antall maskiner. Når administrator får ett svar fra en tjener sender den ut ett nytt kall til samme maskin. På denne måten vil alle kall tilslutt være løst og man vil også her få mer utnytte av raske maskiner. Problemet rundt feilhåndtering løses også ved bruk av denne metoden siden uløste problemer sendes ut til ledige maskiner, og om det da er en maskin som låser seg så vil likevel problemet bli løst.

Tore Andre Rønningen Rønningen(13) leverte sin oppgave i 2003 og det store målet nå var å gjøre PRP systemet objektorientert. Fra før har systemet vært implementert i C, og for å gjøre det objektorientert velger han, etter vurdering mellom C++ og Java, å skrive systemet i Java. Fra før har også PRP systemet vært begrenset ved at det kun kan kjøre over ett lokalt nett, men Rønningen tar i bruk Java RMI og får derfor systemet til å fungere over internett, siden RMI støtter dette.

Rønningen velger også å innføre bufring av parametersett. Fra Eides versjon fikk en ledig maskin utdelt ett nytt parametersett når den var ferdig med ett. I Rønningens løsning får hver tjener utdelt to parametersett. Så fort den har løst ett sett, sender den svar til administratoren og begynner på det andre. I mellomtiden vil adminitrator registrere dette og sende ut ett nytt. På denne måten effektiviserer man systemet og unngår at overføringen blir en flaskehals for systemet.

Mats Bue Bue(6) leverte våren 2005 sin hovedfagsoppgave(kort oppgave), og oppgaven gikk ut på å oversette PRP(som sist ble implementert i java av Rønningen) til C# og .Net plattformen. Etter å ha beskrevet den generelle virkemåten til javaPRP går Bue inn på C# virkemåte og beskriver likheter og ulikheter mellom C# og java. Videre går han dypt inn i oppbygningen av JavaPRP og ser på hvordan hele systemet er konstruert, noe som er veldig viktig ved konvertering fra ett språk til ett annet.

I konverteringsprosessen går Bue inn på hva som kan beholdes fra JavaPRP(Java og C# er ganske like når det gjelder klasser, variable og metodekall) og hva som må forandres. JavaPRP bruker RMI mens man i .NET har valget mellom web services og Remoting. Valget faller på sisnevnte da dette er ganske likt RMI pluss at det viser seg å være raskere ved bruk av TCP/IP. Siden JavaPRP baserer seg på bruk av kommandolinje og lange kommandoer tar Bue å forenkler dette ved å lage ett enkelt brukergrensesnitt.

Etter å ha implementert PRP i C# viser Bue til positive resultater, da det viser seg at C# versjonen er markant raskere(opp mot 40 prosent).

Christian O. Søhoel Søhoels(14) oppgave var ferdig høsten 2005 og målet med oppgaven var å implementere en sjakk computer og se på muligheten for å benytte det eksisterende PRP systemet for parallellisering. Før han går inn på selve parallelliseringen av sjakk computeren går han i detalj og ser på ulike komponenter som trengs i ett sjakkspill. Dette er ting som evalueringsalgoritme, hvordan representere brettet og hvordan man i det hele tatt kan finne ett lovlig trekk.

Ved parallellisering finner Søhoel ut at PRP kun støtter ett rekursiv kall av programmet som skal parallelliseres. Han måtte derfor forenkle sin algoritme som skulle traversere ett posisjonstre. En annen ting som dukket opp er PRPs krav til statiske variable og metoder. Fordi systemet aldri oppretter objekter av brukerens klasse må samtlige metodenavn og klassevariable være static. Søhoel måtte derfor forandre sin kode for å tilpasse PRPs krav.

Etter å ha tilpasset den parallelle versjonen av sjakk spillet viser Bue til både positive og negative resultater. Det viser seg at den sekvensielle versjonen er raskere i noen posisjoner, mens den parallelle er raskere i andre posisjoner.

Bjørn Arild Kristiansen Kristiansens(11) oppgave var ferdig våren 2006 og er det siste som er blitt gjort på PRP systemet. Målet for oppgaven var å få det fra kommandolinjebasert styring til GUI(graphical Unit Interface). Kristiansen velger her å bruke swing biblioteket etter å ha sammenlignet dette med AWT(Abstract Window ToolKit).

Ved oppstart av programmet slik Kristiansen har implementert det så kan man enkelt ved noen tasteklikk starte nye prosjekter. Ved å fylle inn noe informasjon, som hvor java filer ligger o.l. så er systemet klar for kjøring. Under kjøring har man mulighet til å se f.eks følgende informasjon:

- Hvem er den raskeste/tregeste arbeideren.
- Hvor mye av oppgaven er løst.
- Hvor mange parametersett har en gitt tjener løst.
- Hvor lang tid en tjener har brukt på ett sett den har løst.

Det ble i denne oppgaven også startet på en styringsmekanisme. Ved å sende kontrollsignaler til en hvilke som helst tjener så har man mulighet for å styre litt mer under kjøring. Kristiansen implementerte styrings kommandoen "kill" som dreper en prosess, men vi ser her at det er mulighet for å utvide dette med flere andre kommandoer som "hold" for å sette en prosess på vent o.l.

Kapittel 3

Videreutvikling av JavaPRP

Vi har til nå sett på hvordan prp-systemet fungerer. Vi skal i dette kapitellet og utover se på hva min oppgave går ut på, og hva som skal til for å løse de forskjellige problemstillingene.

3.1 Problemstilling 1: Maskiner med flere kjerner

Første problem jeg skal se på, er utnyttelsen av maskiner med flere kjerner/cpu'er. Dagens fokus på hastighet i en datamaskin har gått fra å få en prosossor raskere til å få flere prosessorer/kjerner inn i en datamaskin. På denne måten klarer man å kjøre ting i parallell, noe som igjen fører til at man får utført flere arbeidsoppgaver samtidig. En oppgave vil bli å se på muligheten for å utnytte dette i Prp-systemet. Alt om dette i Kapittel 4.

3.2 Problemstilling 2: Forbedre lastbalansen mellom arbeiderne under runtime

Min andre problemstilling, som også er mitt hovedfokus, vil bli å se på hvordan vi kan forbedre lastbalansen mellom de forskjellige arbeiderne under runtime. Vi vet fra rekursive trær, at eksekveringstiden for de

forskjellige nodene kan varierere betraktlig. Som eksempelet i Kapittel 6, så er Travelling Salesman med cutoff et godt eksempel på dette. Hvordan jeg har valgt å løse denne problemstillingen er beskrevet i kapittel 5 med gjennomgang av resultater fra testkjøringer i kapittel 6.

3.3 En oversikt over klassene i PRP-Systemet

I de neste kapitlene vil vi gjennomgå disse to problemstillingene nevnt over, men før vi gjør det skal vi beskrive viktige klasser som prp-systemet i dag består av.

3.3.1 Oversikt over oppstartssystemet

Før runtime-systemet starter opp, er det oppstartssystemet som tar av seg preprosesseringen av brukerens kode. Nedenfor vises inndelingen av de forskjellige lagene. Inndelingen baserer seg på tre-lags arkitekturen, hvor man har presentasjonslaget, kontrollaget og et datalag. Presentasjonslaget er det brukeren forholder seg til.

UI - presentasjonslaget

AdminProjectGUI Dette vinduet er for modifisering, rekompilering av prosjektfiler og oppstart av runtime-systemet.

PrpGui Hovedvinduet ved oppstart av systemet. Man har her mulighet-en mellom å lage et nyttprosjekt, eller å kjøre et tidligere opprettet prosjekt.

NewProjectGUI Denne klassen tar seg av oppsettet av et nytt prosjekt. Dataen som brukeren fyller inn blir sendt videre til NewProjectExecutor.

Control - kontroll laget

NewProjectExecutor Denne klassen tar seg av all informasjon som brukeren plottet inn. Den vil lagre dette i dataklassen. Den sender også oppgaver til preprosessoren og CommandLineExecutor.

AdminProjectExecutor

CommandLineExecutor Denne klassen tar seg av systemkommandoer. Dvs kompilering av java filer og tilgangsrettigheter for de distribuerte filene.

PrpPP Som beskrevet i Kapittel 2, så er dette preprosessoren. Den tar inn brukerens kode og lager nye klasser som muligjør parallellisering for den rekursive metoden.

FileCopyFunctions Denne klassen kopierer filer fra et område til et annet. Den har også en metode som kopierer de distribuerte filene til område spesifisiert av bruker.

ConfigFileHandler Klasse som tar seg av lesing og skriving fra/til konfigurasjonsfilen.

MouseEvents Klasse som oppdaterer gui når musepeker er over bestemte komponenter.

ListFilter Den grafiske filvelgeren brukeren denne klassen for å vise bestemte typer filer.

Data - Modell laget

ConfigurationData Denne klassen lagrer informasjon skrevet inn av bruker.

3.3.2 Oversikt over runtime-systemet

Etter at oppstartssystemet har tatt i mot brukerens kode og registrert nødvendig info, så er det runtime-systemet som starter opp. Laginndelingen her følger også tre-lags arkitekturen.

UI - presentasjonslaget

PrpRuntimeGui Gui til runtimesystemet.

SpringUtilities Klasse brukt for fin layout ved bruk av swing.

Control - kontroll laget

PrpManager Denne ble beskrevet i kapitell 2 og er kjernen i systemet. denne klassen holder oversikt over alle arbeidere, og har ansvaret for all datalagring av parametersettene.

PrpMonitoring Denne klassen henter monitoreringinformasjon fra arbeiderne. Denne dataen blir hentet ut av klassen workerstatistics

StatsCalculator Denne klassen regner ut forskjellige tider ut i fra data som monitoren har hentet inn.

RuntimeController Denne klassen har ansvaret for kontrollsignaler sendt av bruker. Om et signal skal sendes så er det denne klassen som sender beskjeden videre til en arbeider.

PrpArbeiderImp: Controller Denne indre klasse mottar kontrollsignal fra runtimecontroller på managersiden.

PrpArbeiderImp: InternalMonitor Denne indre klasse samler inn forskjellige data på arbeidersiden, og lagrer det i et workerstatistics objekt.

Data - Modell laget

WorkerStatistics Denne klassen lagrer informasjon hentet fra en bruker. Dette omfatter følgende:

- int parameters - Hvor mange parametre den har løst.
- long calls - antall prosesserte kall.
- long paramSetStart - tidspunkt da eksekveringen av gjeldende parametersett startet.
- long currentTime - Nåværende tidspunkt hos arbeideren.
- boolean finished - angir om arbeideren er ferdig med sine parametersett.
- boolean init - angir om arbeideren er initalisert
- String hostname - navn og ip til arbeideren.

Kapittel 4

Utnyttelse av flerkjernemaskiner

Før man går inn på selve implementasjonendelen, vil vi se nærmere på hvordan koden allerede er bygget opp. Ut i fra dette kan man utforske forskjellige muligheter man har for å utvide systemet.

4.1 Strukturen til arbeiderprosessen

Koden til en arbeider er grovt delt inn i to deler:

- Klassen PrpArbeiderFabImp.
- Klassen PrpArbeiderImp

Klassen PrpArbeiderFabImp inneholder main metoden til en arbeider. Den setter opp RMI registre, securitymanager og hvilke port all aktivitet skal sendes/mottas på. Videre har den en metode, nyArbeider, som prp-manageren kaller på når den vil initiere en ny arbeider med kode den skal kjøre. Metoden oppretter da et objekt av klassen PrpArbeiderImp og returnerer denne pekeren til manageren. Fra nå av vil manageren kun kommunisere med PrpArbeiderImp-objektet.

Klassen PrpArbeiderImp er selve motoren i arbeideren. Det er denne klassen som mottar nye parametersett, levering av svar og eksekvering av brukerens kode. En skisse av klassen ser slik ut:

```
class PrpArbeiderImp {

  class PrpBufferTraad extends Thread {}

  class InternalMonitor extends Thread {}

  class PrpProcess extends Thread {}

  class Controller extends Thread {}

}
```

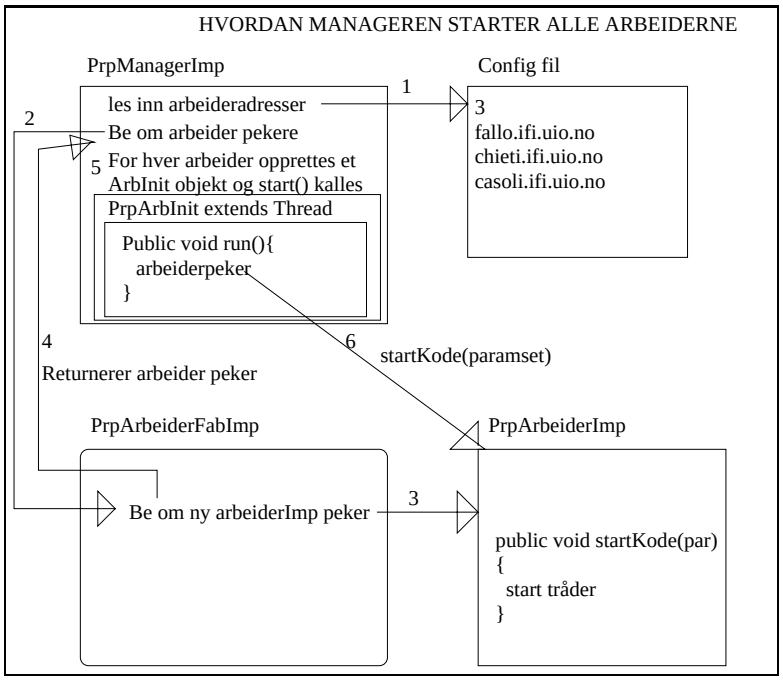
Som vi ser er hele arbeideren tråd-basert. Det vil si at flere prosesser kjøres samtidig, som alle trenger å synkroniseres. De forskjellige trådene har følgende funksjon:

- PrpBufferTraad : Denne tråden henter ned parametre fra manageren til bufferen er av ønsket størrelse.
- InternalMonitor : Denne tråden samler lokale data, slik som løste sett, tid den bruker og annen viktig info.
- PrpProcess : Denne tråden ber om parametre fra buffer-tråden og kjøres brukerens kode med dette parameteret.
- Controller : Denne tråden tar imot signaler fra manageren. Eksempel på dette er kill kommandoen som dreper arbeideren.

4.2 Strukturen til manageren

Hovedkomponenten i manageren er klassen PrpManagerImp. Det er denne klassen som kontakter arbeiderne, overleverer parametersett og tar imot svar. Ved opprettelse av denne klassen skjer følgende:

- Leser inn fra konfigurasjonsfil hvor navnenene til arbeiderne ligger



Figur 4.1: Strukturen til manageren

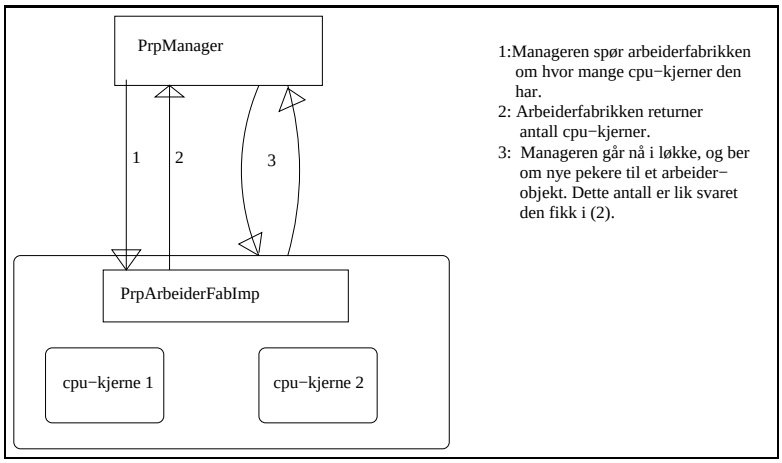
- For hver arbeider opprettes det et PrpArbInit objekt. Dette er en indre tråd-klasse i manageren.
- Når run metoden til prpArbInit starter for hver arbeider, kontaktes PrpArbeiderFab. En peker til et nytt arbeiderobjekt blir returnert, og tråden starter arbeiderobjektet med et parametersett.

Fra figur 1 ser vi en illustrasjon av manageren og hvordan den jobber.

4.3 Drøfting av mulige Implementasjoner

Etter å ha sett på hvordan manageren og arbeideren er strukturert, kan vi nå se på hvordan det er mulig å utnytte det at maskinen har flere cpu-kjerner. Slik jeg ser det, har man to muligheter:

1. Man kan la manageren ta seg av jobben. Ved å første finne ut hvor mange cpu-kjerner maskinen den kontakter har, for deretter å be om



Figur 4.2: Kommunikasjonen mellom manager og arbeider

like mange pekere til et arbeiderobjekt.

2. Man kan la arbeidermaskinen ta seg av det å utnytte flere cpu-kjerner. Ved å opprette en lokal manager på hver arbeidermaskin, som skal ha ansvaret for å starte opp like mange arbeidere som antall cpu-kjerner. Denne manageren vil da også få ansvaret med å kommunisere med den globale manageren som holder oversikt over alle arbeidermaskiner.

Hvis vi ser på alternativ 1, så krever ikke dette mye forandringer i koden. Ved å spørre arbeiderfabrikken om hvor mange cpu-kjerner den har, for deretter å opprette like mange arbeiderobjekter på arbeidermaskinen, blir dette som å legge til en ny arbeider. Arbeiderprossesene vil da være uvitende om hverandre. Det eneste som vil skape mer tidsbruk er i det manageren spør arbeiderfabrikken om hvor mange cpu-kjerner den har(se figur 4.2), men tidsbruken vil uansett være minimal.

Alternativ 2 er en annen mulighet for å utnytte samtlige cpu-kjerner. Det som er viktig her, er at de ikke skriver til felles variable. Skulle de gjøre dette, krever dette mer synkronisering og ekseveringen av koden vil ta lenger tid. Siden vi ønsker å få et raskere system, er dette derfor ikke ønskelig. Alternativ 2 krever også forandring i manageren. For det første så må den vite hvilke av trådene som returnerer svaret. En annen ting som må forandres er monitoreringen av de lokale dataene på en prosess. Siden manageren kun har en peker til en arbeider som kanskje har to cpu-kjerner,

krever dette omstruktureringer. Med store omstruktureringer menes det at det vil foregå mer kommunikasjon mellom arbeider og manager en det systemet gjør i dag. Mer kommunikasjon vil igjen føre til større tidsbruk og systemet vil kanskje ikke løse oppgaven så fort som ønskelig.

4.4 Hvordan vite at alle kjerner blir utnyttet?

Spørsmålet som etterhvert dukker opp, er om alle kjerner blir utnyttet ved en slik ordning. Kan man be en prosess kjøre på en bestemt cpu-kjerne? Dessverre er ikke dette mulig i java 1.5.0_04. Dette er noe en bruker ikke har mulighet til å påvirke, men må overlate til implementasjonen av javas virtual machine(JVM) og operativsystemet.

4.5 Mulige maskinene vi kan utnytte

Før vi starter uttesting, skal vi se nærmere på hvilke maskiner vi kan dra nytte av.

Hyper-threading(HT) maskiner

Man har i HT teknologien optimalisert vekslingen mellom prosessene som kjører, noe som medfører at operativsystemet ser og behandler prosessene som ett system med to "vanlige" prosessorer.

Ved å kjøre denne java kommandoen:

```
Runtime.getRuntime().availableProcessors()
```

får vi ut 2 som svar, selv om maskinen har en cpu kjerne.

Intel selv mener en ytelseforskjell rundt 30% mer, med en maskin med HT teknologi i forhold til en uten HT teknologi. En annen test utført av hardware.no viser følgende:

- Ved å starte to SETI@home klienter viste resultatet at maskinen med HT teknologi var 39% raskere.

- En annen test hvor man startet et spill og samtidig startet et program for å konvertere et filmklipp til divX, så viste testen at maskinen med HT var 34% tregere.

Av disse opplysningene kan vi forvente oss at en arbeider i PRP-systemet maksimalt utgjøre rundt 1,3 arbeider.

dual-core maskiner

Med dual-core maskiner så har prosessoren to cpu-kjerner. Med denne arkitekturen klarer maskinen å utføre 2 instruksjoner samtidig. Selv om en maskin har 2 cpu-kjerner kan man likevel ikke forvente at maskinen jobber dobbelt så fort. Siden kjernene deler samme system bus og minne vil dette trekke litt ned på ytelsen. Intel mener selv at man kan merke et forbedring på rundt 40-50% ved bruk av dual-core. Dette vil si at i vårt system kan vi maskimalt forvente at en arbeider kan utgjøre rundt 1,5 arbeider.

4.6 Testkjøring

4.6.1 Første testkjøring for utnyttelse for hyperthread-maskiner

Etter å ha tenkt litt på de to forskjellige valgmulighetene valgte jeg å gå for alternativ 1, hvor manageren har like mange pekere til en arbeidermaskin som antall cpuer den har. Ved å bruke denne framgangsmåten blir hver arbeiderprosess på en arbeidermaskin utvitende om hverandre.

Med denne løsningen kan man beholde det meste av strukturen i det eksisterende systemet, og for det andre så legger man her ansvaret på manageren, noe som virket veldig naturlig. I tillegg til dette virket det ikke fornuft å gi mer arbeid til en arbeider. En arbeider har som hovedoppgave å gjøre en beregning og returnere svaret så fort som mulig. Hvis man i tillegg skal legge til mer arbeid i et arbeiderobjekt ville ikke dette gjøre systemet så raskt som ønskelig.

Ut ifra tabell 4.1 på neste side og 4.2 på neste side ser vi resultatet av første testkjøring. Resultatet viser at den nye versjonen løser oppgaven rundt

Antall arbeidere	Kjøretid	hva ble kjørt?
3 (hyper-thread) maskiner	158 s	Goldbach 1 000 000
3 (hyper-thread) maskiner	3645 s	Goldbach 5 000 000

Tabell 4.1: Testkjøring uten utnyttelse av hyperthreading

Antall arbeidere	Kjøretid	hva ble kjørt?
3 (hyper-thread) maskiner	128 s	Goldbach 1 000 000
3 (hyper-thread) maskiner	2834 s	Goldbach 5 000 000

Tabell 4.2: Testkjøring av ny versjon med utnyttelse av hyperthreading

23%raskere. Det viste seg derimot at det var noe ved denne testkjøringen som ikke gikk som normalt, noe jeg vil gå igjennom i neste avsnitt.

4.6.2 Et problem med eksisterende PRP-Systemet

Selv om den nye versjonen viste seg å være betraktlig raskere så var det noe ved PRP-systemet som ikke virket som det skulle. Kjøringen av goldbach gikk bra, og vi fikk det riktige resultatet, men gui'en som viser informasjon om hver arbeider, stoppet opp etter noen få sekunder. Etter å ha lagt inn litt testutskrifter viste det seg også at det bare var en maskin som klarte å returnere svar i passende tempo, og de andre kom sakte men sikkert etter at 17% av oppgaven var løst. Jeg prøvde derfor å kjøre et enklere java program(sum.java) for å se at min nye versjon fungerte som den skulle, noe den gjorde.

Dermed begynte debuggingen for å finne hvor feilen lå, slik at Goldbach kunne kjøre som normalt. Første jeg tenkte på var om det kunne være et problem rundt preprosessoren og koden den lagde utifra brukerens kode. Dette viste seg ikke å være tilfelle.

Neste mulighet kunne ligge rundt operativsystemet og bruken av tråder. Siden statistikken kjørte en liten stund, for deretter å stoppe helt opp, kunne det kanskje ligge et problem rundt operativsystemet og dens håndtering av interupts. Siden Goldbach har enn mye større jobb å gjøre en sum, kunne det hende at arbeidermaskinen hadde problemer med å kontakte den kjørende filen. Men etter å ha skrevet ut masse testresultater

av forskjellige variable, så viste deg seg at feilen ikke lå her heller. Arbeidermaskinen klarte å hente ut data fra den kjørende javafilen, og administrator fikk de dataene den trengte for å regne ut de forskjellige statistikkene.

Jeg måtte derfor gå enda dypere for å finne feilen. Etter en god stund fant jeg der det stoppet helt opp. Det viste seg at PRP hadde problemer med å oppdatere grafen i gui'en. Denne grafen viser arbeidshastighet. Hastigheten beregnes ved å se på antall løste kall for hver arbeider i forhold til den akkumulerte mengden løste kall. Fra programkoden kan vi her se hva som gjorde at monitoreringstråden låste seg:

Listing 4.1: FeilIGui.java

```
private void updateGraph() {
  int multiplier = 2;
  int[] val = new int[noOfWorkers];



  long callsTotal = statsCalc.getAccumulatedCalls();
  long[] workerCalls = statsCalc.getWorkerCalls();


  for (int i = 0; i < noOfWorkers; i++) {
    if (callsTotal == 0) {
      val[i] = 0;
    }
    else {
      //her ligger feilen
      val[i] = (int)(workerCalls[i] /(callsTotal / 100));

    }
    if (val[i] > 50) {
      multiplier = 1;
    }
  }

  for (int i = 0; i < noOfWorkers; i++) {
    pb[i].setValue(val[i] * multiplier);
    pb[i].setString(workerNames[i] + "   " + val[i] + "%");
  }



}
```

Ser vi på else grenen i for-løkken gjør den en beregning og lagrer hele resultatet i val[i]. Problemet er derimot:

- Hvis callsTotal er liten vil (CallsTotal / 100) bli et desimaltall. Siden den prøver å caste hele beregningen til int vil dette bli 0. Når man videre tar workerCalls[i] / (CallsTotal / 100) ville dette bli å dele på 0, noe som ikke er lov.

Grunnen til at dette problemet ikke har blitt oppdaget før, er at verdiene man har jobbet på er såpass store, slik at en deling på 100 ikke har skapt desimaltall mindre en 1. Et av grunnene til at jeg selv brukte en del tid på å finne feilen var at det ikke kom noe exception feil i konsoll vinduet. Når det ikke kommer en slik feil, tror man ikke med det første at det er en caste feil et sted i programmet. Det java derimot har gjort er å er å melde fra om en java.lang.ArithmeticException: / by zero, men siden GUIPRP er et todelt program, hvor du har preprosesseringen på den ene siden og runtime systemet på den andre siden, så vet ikke java hvilke terminalvindu den skal sende meldingen til.

4.6.3 Andre testkjøring for utnyttelse av hyperthread-maskiner

Etter å ha forandret koden i oppdateringen av grafen til en arbeider, vil jeg nå teste systemet på nytt. Vi kan nå stole mer på resultatet, da alle tråder vil være i aktivitet og ingen prosesser stopper opp underveis.

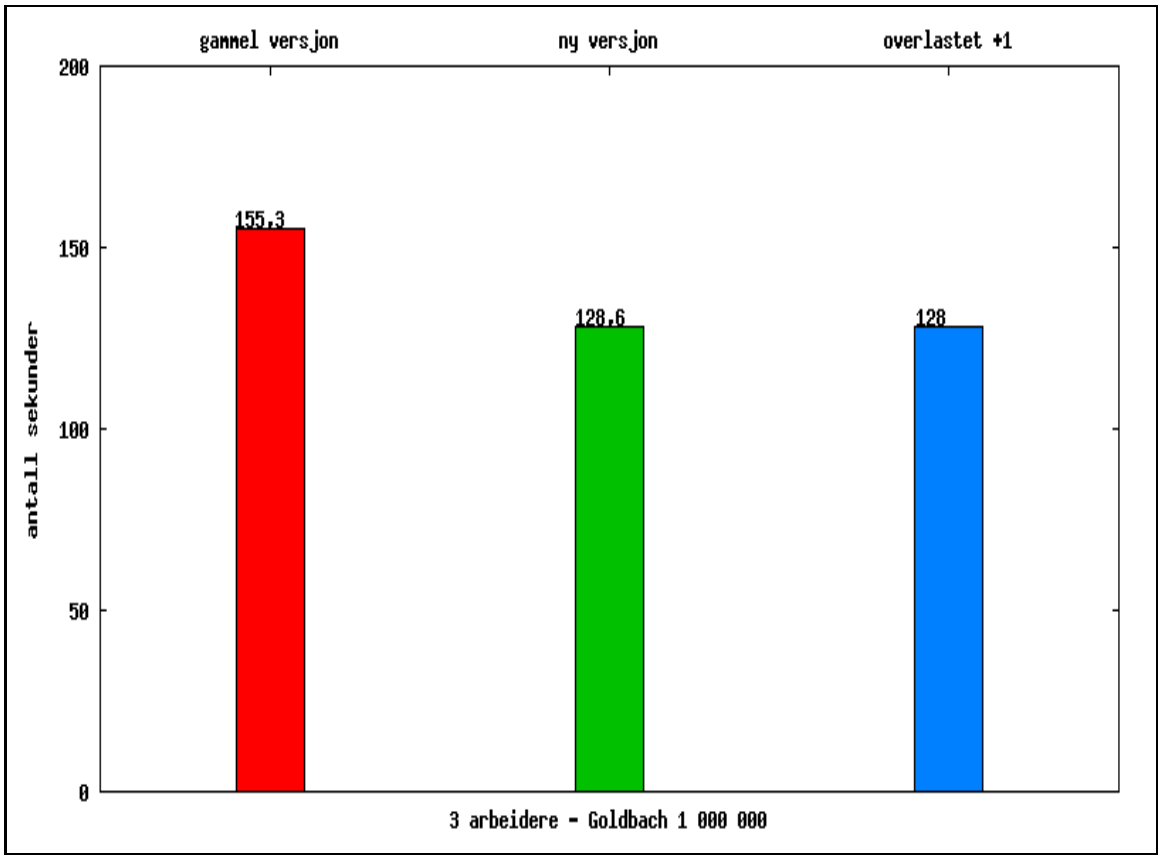
Bakgrunnsinformasjon

Fra figur 4.3 på neste side ser vi testkjøringen med to forskjellige input. alle søyler er et resultat av 3 kjøringer hvor snittet ble regnet ut. Som i første testkjøring bruker også her maskiner med HT-teknologi. Søylene med tittel overlastet+n er der hvor jeg har kjørt n arbeidere for mye på hver maskin. Det vil si:

1. ny versjon: Administrator finner x cpu'er hos arbeidermaskinen, og starter x arbeiderprosesser på denne maskinen.

2. overlastet+1: Administrator finner x cpu'er hos arbeidermaskinen, og starter x+1 arbeiderprosesser på denne maskinen.

Testen med å overlaste en arbeidermaskin gjorde jeg for å se hvor oppgaveløsningen begynte å gå tregere igjen.



Figur 4.3: Testresultat av andre kjøring ved utnyttelse av hyperthread-maskiner

Resultat

Fra figur 4.3 på forrige side ser vi resultatet av goldbach med input på 1 million. Utifra søylene ser vi at den gamle versjonen bruker ca 21% prosent mer tid en den nye versjon. Dette er et veldig positivt resultat. Når vi overbelaster arbeideren, ser vi at tiden den bruker ikke blir noe raskere, men holder seg på samme tid. Man kan derfor forholde seg til antall cpu'er javakommandoen Runtime.getRuntime().availableProcessors() returnerer.

Sammenligner vi denne testen med harware.no og SETI klientene, så har vi ikke her så stor forbedring. Noe av grunnen kan være at kommunikasjonen til og fra manager fordobles. Det at hver arbeiderprosess kaller samme metode i manageren ved levering av svar skaper noe delay.

Kapittel 5

Oppsplitting av vanskelige parametersett

Hva skal i denne delen se på problemet rundt vanskelige parametersett. Det vil si de parametersettene som skaper et større kall-tre en de andre settene. Ved å splitte opp slike vanskelige parametersett, får vi en bedre lastbalanse mellom arbeiderne, og utregningen vil bli gjort hurtigere.

5.1 Problemer rundt det å oppdage en treig arbeider

Det å oppdage en treig arbeider er ikke så vanskelig å implementere i det eksisterende PRP-systemet. Bjørn Arild la inn mekanisme for å ta tiden på hvor lang tid en arbeider bruker på et parametersett. Vi kan bruke dette til å finne gjennomsnittstider, og utifra dette avgjøre under runtime om en arbeider er treg eller ikke.

Oppdager vi at en arbeider er treg, kan det være tre grunner til dette:

- Maskinen er treig.
- Maskinen er belastet med andre oppgaver.
- Parametersettet genererer et større kall-tre en normalt.

Hvordan skal man avgjøre om en maskin er treig/overbelastet, eller om den har et vanskelig parametersett? Den eneste tiden vi har å måle opp mot, er hvor lang tid hver arbeider bruker på hvert parametersett. Om denne tiden blir stor i forhold til gjennomsnittet kan være av begge grunner nevnt ovenfor.

Et annet problem rundt det å finne en treig arbeider, er å avgjøre når den faktisk er treig. Det å sette en formel på hvor mye langsommere en maskin kan være før dens parametersett splittes opp, kan være ganske avgjørende for tidsbruken. En situasjon som vil bruke mer tid, er når arbeideren nesten er ferdig med parametersettet, men manageren bestemmer seg for å splitte opp, siden den har brukt x sekunder for mye tid på et parametersett.

5.2 Håndtering av parametersett i det eksisterende systemet

Før vi går inn på en løsning av hvordan man kan splitte opp et parametersett, så skal vi se på hvordan det fungerer i det eksisterende PRP-systemet.

Under oppstart av runtime-systemet startes den preprosesserte filen Prp<navn>.java. I denne klassen ligger metoden for å generere nye parametersett. Utifra antall arbeidere og splittfaktoren som er satt(default 20), så genereres det antall arbeidere * splittfaktor paramersett. Vi skal her se på hvordan filen er strukturert.

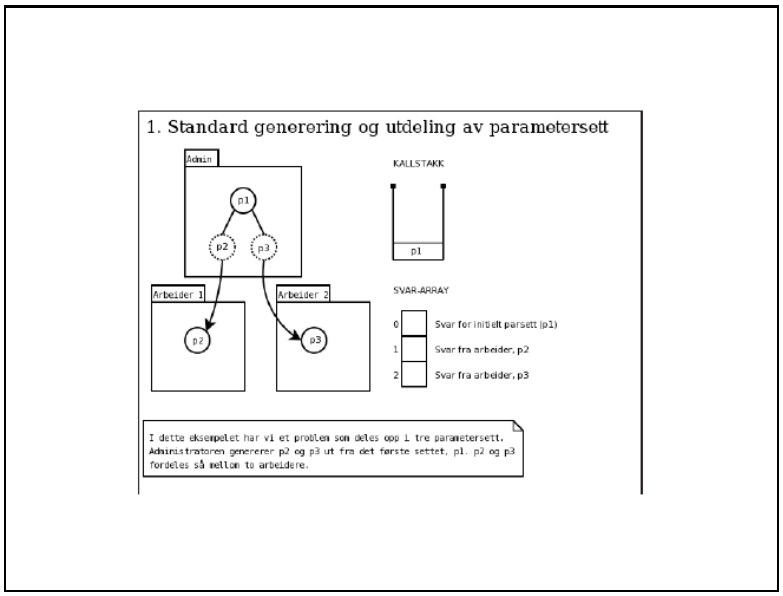
```
public class Prp<navn> {
  public static void main(String [] args) {
     < brukerens kode i main blir uendret >
     <type> <variabelnavn> = rekursivmetode(<parametre>);
  }

  < Egendefinerte hjelpemetoder >

  public static <type> rekursivmetode(<initialparametre>){
    < Oppretter PRPmanageren >
    < Legger initialparametre i FIFO–køen >
```

```
    //////PARAMETERGENERERING/////////

    while(<FIFO kø ikke er full>){
       <hent paramsett fra FIFO kø>

       PRP_proc(<paramsett>);
    }


    //////START PARALLELL FASE //////
    < Her blir metoden i manageren kalt og den begynner å sende ut >
    < parametersett til arbeiderne. Svarerene lagres i en svar-array >
    < lokalt hos prpmanageren >



    //////SVARGENERERING //////

    while(<kallstakk ikke er tom>){
       PRP_proc(<hent paramsett fra kall stakken>)

    }


    < returnerer svaret til main-metoden >

  }


  public static void PRP_proc(<paramsett>){

    < henter ut verdiene fra parametersettet >
    < legger kopi av settet på kall stakken >

    < Brukerkode for å bestemme parameterverdier til de nye kallene >


    for(int i = 0; i < N; i++){

      if( <parametergenerering> ){
         < parametersettet blir her delt opp og puttet på FIFO stakken >
         < i steden for å kalle den rekursive metoden med dette settet. >
      }
      else if ( <svargenerering> ){

        < hent ut utregnet svar på parametersett fra svar arrayen >

      }


      if(<svargenerering>){
  <Legg svaret som ble hentet ut inn i et returobjekt>
        <Legg svarobjektet på returstakken >

        return;
      }

    }


  }
}
```

Ved oppdeling[1] av paramersett skjer følgende:

- Første paramerersett,P1, blir puttet i en FIFO kø hos manageren. Dette settet blir lagret som et Param Objekt. Hvert param objekt får

[1]Parametergenerering i koden beskrevet over



Figur 5.1: Generering av parametersett

variable som brukeren skrev i sitt eget program. I tillegg lagres det en indeks som brukes under svargenereringen.

- P1 blir hentet ut fra FIFO-køen og kaller den rekursive metoden med dette settet. FIFO-køen hos manageren er nå tom.
- Rekursive metoden lagrer en kopi av parametersett P1 i kallstacken.
- P1 blir delt opp i to sett, P2 og P3. Disse blir puttet i FIFO køen hos manageren. Hver av disse settene får en indeks som forteller hvor de skal legges i svar-arrayen.
- P2 og P3 kan nå sendes ut til arbeiderne

En illustrasjon hentet fra Bjørn Arilds(11) oppgave er vist i figur 5.1.

Når alle svar er blitt returnert starter svargenereringen. Fra koden over ser vi at man nå behandler de parameterne som er blitt lagt på kall-stacken. Dette skjer i følgende steg:

- Man popper øverste parameters,P_n, som er lagt på stakken.
- Videre kalles PRP_proc med dette parametersettet.

- Og istedenfor å gjøre de rekursive kallene, henter den ut svar fra svar-arrayen til manageren. Disse svarene har blitt regnet ut av arbeiderne under runtime.
- P_n blir regnet ut og lagt i svar-arrayen.
- Dette gjenntas helt til P_1 henter svarene fra P_2 og P_3.

5.3 Prp-manageren

Ved oppsplitting av parametre så er det på managersiden de største forandringene ligger. Vi skal i dette avsnittet gå igjennom steg for steg, og se på hva som skal til for å splitte opp et parameter. Jeg vil først beskrive hvordan vi skal ta i bruk refleksjon og hvordan man overvåker tidene til de forskjellige arbeiderne. Videre vil jeg beskrive backup-mekanismen i system og hvilke utfordringer dette gir oss når vi skal splitte opp et parametersett. Til slutt beskriver jeg hvordan splittmetoden vil se ut og stegvis gå igjennom fra det tidspunktet da manageren oppdager en treig arbeider til parameteret er oppsplittet.

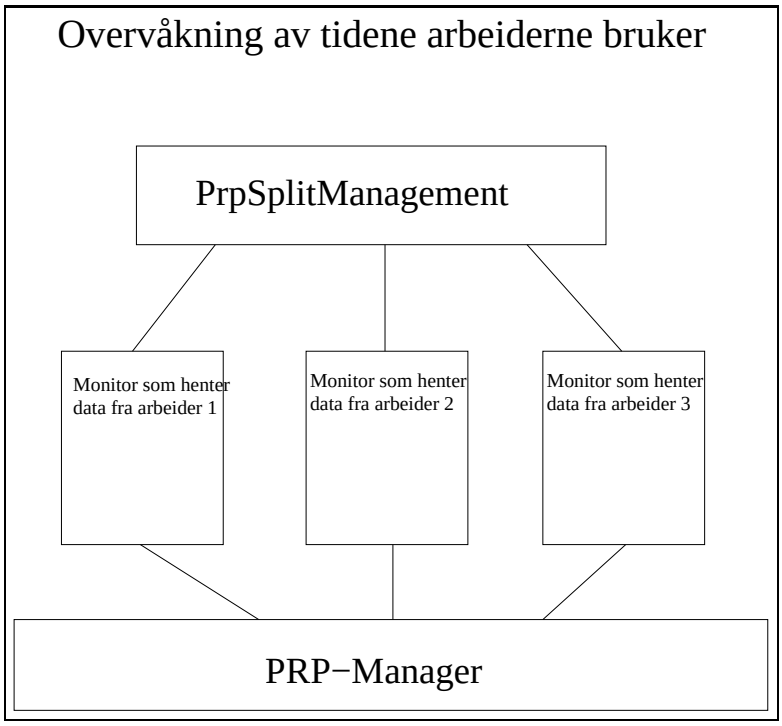
5.3.1 Overvåkning av alle tider

Som en viktig del av oppsplittingen, trenger vi en mekanisme som holder oversikt over tidsbruken på de forskjellige parametersettene. Det er her to tider vi er interessert i:

- Hvor lang tid hittil en arbeider har brukt på et parametersett under runtime.
- Hvor lang tid det tok totalt for en arbeider å løse et parametersett.

Fra den eksisterende koden starter manageren en monitortråd for hver arbeider. Denne tråden har som oppgave å hente forskjellige tids-data fra arbeidermaskinen. Hver av disse trådene har data som vi trenger for å sammenligne tidene under runtime.

Strukturen jeg har valgt for å overvåke alle tider er illustrert i 5.2 på neste side. Hver av monitortrådene har en peker til PrpSplitManagement(PSM).



Figur 5.2: Splitmanagement

PSMs oppgave er å samle alle tider som monitorene sender. Sender monitoren tiden et parametersett tok, lagres dette i en liste hos PSM. tidene fra denne listen blir brukt senere for å oppdage treige arbeidere. Sender monitoren en mellomtid på hvor lang tid arbeideren har brukt hittil på et sett, bruker vi tiden fra listen over de ferdige parametersette å måle opp mot. På denne måten kan vi bestemme om arbeideren er treig i forhold til de andre utifra en gitt formel. Denne formelen kommer vi tilbake til i Kapittel 6.

5.3.2 Bruk av Reflection i PRP

Ved oppsplitting av parametre trenger manageren å kontakte klassen som er blitt generert av preprosessoren. I denne klassen ligger det en metode[2] som tar imot et parameter, splitter det opp og legger de nye

[2]Denne metoden tar å splitter opp et parametersett i mindre subsett. Beskrivelsen av denne kommer vi tilbake til senere i dette kapittelet

settene på FIFO-stakken. Det som derimot skaper problemer, er at denne klassen blir generert under kjøring og kan ha forskjellig navn fra kjøring til kjøring. Dette fører til at vi ikke kan legge en peker til denne klassen i programkoden, da det ikke vil kompilere. For å komme rundt problemet kan vi ta i bruk javas refleksjon. Ved bruk av refleksjon kan vi f.eks få tak i følgende informasjon:

- Avgjøre klassen til et objekt.
- Finne metoder,superklasser,deklarasjonen og annen informasjon til en klasse.
- Kalle på en metode i et objekt, hvor metoden ikke er kjent før runtime.

Mulige varianter for å kontakte en ukjent klasse

Det er den genererte klassen som oppretter manageren, og den kan derfor sende med noe informasjon om seg selv til konstruktøren i manageren, noe som letter arbeidet litt. Hvis den sender navnet på klassen vi skal kontakte(som nevnt over), kan vi kontakte metoden med følgende kode:

```
String klasseNavn = ‘‘PrpSumsddfre5Sdf’’;
String metodeName = ‘‘PrpSplitParam’’;

//ForName returnerer klasseobjektet utifra stringen.
Class classPointer = Class.forName(klasseNavn);

//Hvis metoden aksepterer parametre, må vi først definere dem i en
//klasse array
Class[] classParams = new Class [] { PrpParameter. class};

//Deretter oppretter vi en metode utifra metodenavn og parametre
Method someMethod = classPointer.getMethod( metodeName, classParams );

//legger parameterne som skal sendes med til metoden i en
//object array. par peker få et parameterobjekt.
Object[] arguments = new Object [] { par };
```

```
//kaller på metoden via refleksjon
someMethod.invoke( dynaclass.newInstance(), arguments );
```

På denne måten vil metoden PrpSplitParam i den generte klassen bli kalt. Med metoden sender vi også en parameter som vi vet metoden aksepterer. Vi kunne her gjort koden ennå mer generell, ved å sjekke at den må kalles på med forskjellige parametre. Men i vårt tilfelle er ikke noe slik nødvendig da vi vet hva metoden heter og hvilke parametre den trenger.

Når vi skal kalle på metoden for å splitte opp parameteret, kan det være greit å få tilbakemelding på om dette gikk bra. Siste linje i koden over blir da i steden slik:

```
Object retObj = someMethod.invoke( dynaclass.newInstance(), arguments);
boolean splitOK = (Boolean) retObj;
```

En annen mulighet for å kontakte metoden er å la den genererte klassen opprette et objekt av seg selv, og sende med denne pekeren til manageren. Koden hadde nesten blitt den samme. Eneste forandringen hadde vært at vi lagret pekeren som ble tilsendt i en objektvariabel. Videre måtte følgede linje

```
Class classPointer = Class.forName(klasseNavn);
```

byttes ut med

```
Class classPointer = objektvariabel.getClass();
```

Som en liten test ville jeg teste ut hvilke av disse framgangsmåtene som var raskest. Og det viste seg at de så og si var like raske. Det er derfor ingen grunn til at koden i preprosessoren må forandres for å få den generte klassen til å opprette et objekt av seg selv, og sende den til manageren. I stedet kan den enkelt bare sende navnet på klassen som en string.

5.3.3 Hvordan beholde backupmekanismen i JavaPRP

Prp-systemet har en backupmekanisme som gjør at maskiner som blir fort ferdig, tar å kopierer over parametersett fra uferdige arbeider for å se om den klarer å bli raskere ferdig. Dette skaper et problem når vi snakker om oppsplitting av parametersett. Et enkelt eksempel kan beskrive problematikken:

```
 Gitt at vi har 4 arbeidere som jobber på hvert sitt parametersett.
Vi kaller disse arbeiderne for A,B,C og D. Dette er de 4 siste parameterne
igjen. Etterhvert blir A,B og C ferdige med sine parametersett, og
starter derfor med samme parametersett som arbeider D. I det A,B
og C har fått utdelt parametsettet til D, så bestemmer manageren
at D har brukt for lang tid å velger å splitte om dens parametersett.
```

Allerede her ser vi problematikken vi står ovenfor. I dette tilfellet vil 3 arbeidere stå å jobbe på et stort parametersett, og en arbeider vil jobbe på delproblemet av det store problemet.

Måter å løse dette på kan være flere:

1. Første mulighet er å ikke dele opp flere parametre så fort en av arbeiderne er ferdige. Dette er den enkleste løsningen.

2. Vi kan hindre arbeider D i å spitte opp parametersettet. Dette kan vi enkelt gjøre ved å sjekke om noen har det gitte parametersettet i sin utsendt liste.

3. Når et parametersett blir splittet opp, må vi først stoppe alle arbeidere som jobber på samme sett, for deretter sette de igang med de nye settene igjen.

uansett hvilke av løsningene vi velger så kreves det synkronisering mellom metoden i manageren som splitter opp et parametersett, og metoden som arbeideren kaller på for å få andres parametersett. Dette kan vi enkelt gjøre i java på følgende måte:

```
class PrpManagerImp {
```

```
    Object lock = new Object();

    public void metode1(){
      synchronized(lock){

<kritisk kode>

      }
    }

    public void metode2(){
      synchronized(lock){

<kritisk kode>

      }
    }


  }
```

Spørsmålet som blir vanskelig, er om man skal gå for punkt 1,2 eller 3. Første punkt er klart enklest å gjøre, og krever lite arbeid av systemet. Man kan her sette en boolsk variabel så fort en arbeider ber om parametersett fra andre arbeidere. Siden man kan sykronisere denne metoden med splittmetoden ville dette fungere uten at systemet låser seg på noe som helst måte. Det som derimot er dumt med denne metoden er hvis det siste parametre skaper et veldig stort subtree. Om vi ikke velger å splitte opp, mister oppsplitting av vanskelige parametersett noe av sin funksjon.

Punkt to er en utvidelse av punkt en, som gjør at det fortsatt er mulighet for oppsplitting selv om noen arbeidere er blitt ferdige med sine parametersett.

Når vi ser på siste punktet så er dette den mest krevende. Her vil det foregå mye kommunikasjon mellom arbeiderne. Et eksempel vil være at man stopper og starter alle arbeiderne om og om igjen ettersom parametersettet trenger oppsplitting. Det vil her bli brukt mye unødvendig tid av alle

arbeiderne. Det som derimot er positivt er at man beholder oppsplitting av parametersett under alle omstendigheter.

Etter å ha tenkt gjennom de forskjellige alternativene, så valgte jeg å implementere siste punkt. Siden min oppgave var å utvide systemet slik at det kunne splitte opp vanskelige parametersett, så ville jeg la manageren splitte opp under alle omstendigheter.

5.3.4 Splitt-metoden

Vi har nå en mekanisme for å overvåke alle tidene til arbeiderne. Videre har vi en framgangsmåte for å kontakte en ukjent klasse. Vi skal nå se på en metode som kan splitte opp et parametersett i flere subsett. En slik metode har vi sett tidligere ved gjennomgangen av preprosessoren. Preprosessoren genererte en ny klasse, som hadde en metode(PRP_PROC) som tok imot en parameter og splittet det opp i to eller flere deler. Videre la den de nye parameterne på FIFO-køen, som senere ble beregnet av arbeiderne. Dette er akkurat hva vi trenger ved oppsplitting av parametre under runtime. Vi trenger derimot en liten forandring fra den genererte metoden under oppstart av prp systemet.Etter diskusjon men Cuong van Truong kom vi fram til følgende løsning på en metode for å splitte opp parameteret:

```
<metode som legges i den genererte klassen Prp<Navn><generert streng>


public boolean splitParam(Parameter par, PrpStackImp stack){

  if(sjekk på variabel om vi kan splitte){

    <push par på kall-stack>

    //Resten av koden i denne if testen blir nesten den samme
    //som PRP_PROC metoden under parametergenereringen.
    //Forskjellen ligger i indekseringen. I steden for en teller som
    //brukes i oppstarten av prp-systemet, må vi her i steden spørre
    //manageren hvor stor svar-arrayen er, da det er denne størrelsen
    //som kan si oss noe om hvor mange parametersett det finnes.
```

```
    }
    else {
      //Man kan ikke splitte opp parameteret noe mer.
      //Vi returnerer false for å si ifra til manageren.
      //Dette vil gjøre at manageren ikke kontakter arbeideren,
      //men lar den utføre den store oppgaven.
      return false;
    }

    return true;

  }
```

Det som derimot skaper et problem, er at navnet på den genererte klassen kan skifte fra kjøring til kjøring. Når vi under runtime skal kontakte denne klassen via refleksjon, trenger vi også navnet. Dette kan løses på en enkel måte. Siden det er den genererte klassen som oppretter manageren kan det enkelt sende med sitt eget navn til kontruktøren som en streng. Denne kan vi bruke senere ved refleksjon.

Et annet problem som følger er en løsning rundt kall-stakken. Denne er definert i den genererte klassen og brukes under parametergenereringen og senere i svargenereringen. Når vi under runtime splitter et parameter, trenger vi å kontakte splitParam metoden beskrevet over. Det første metoden gjør(akkurat som PRP_PROC) er å pushe parametersettet på kall-stakken. Problemet vårt er derimot at klassen vi kontakter via refleksjon ikke er den samme som under oppstart. man vil derfor komme til å pushe på en stakk som ikke er den samme som ble brukt under parametergenereringen. Dette vil igjen føre til feil under svargenereringen. Løsningen på dette er å la den genererte klassen sende med pekeren til kall-stakken på samme måte som den sendte med navnet på seg selv. Vi kan dermed sende stakk-pekeren med til den nye metoden beskrevet ovenfor.

Vi har nå en metode vi kan kalle på for å splitte opp et parameter, kall-stakken og returlisten behandles på riktig måte. Vi skal videre se på hva

som skjer fra det tidspunktet hvor klassen PrpSplitManagement oppdager en treig arbeider, til oppsplittingen er fullført.

Steg 1: PrpSplitManagement får en mellomtid brukt på et parameter for en gitt arbeider. Etter beregning finner metoden ut at den har brukt for lang tid. Den velger derfor å kontakte metoden SplitParam(int id) i manageren.

Steg 2: Metoden Splitparam mottar hvilke arbeider som jobber for langsomt. Det første den må gjøre er å stoppe den aktuelle arbeideren sin monitor. Hvis vi ikke gjør dette vil monitoren igjen hente en ny mellomtid på samme settet hos den aktuelle arbeideren. Det vil igjen føre til at man prøver å splitte opp samme parametersett. Videre må vi undersøke om det er noen andre arbeidere som jobber på samme parametersett(jfr. 5.3.3), og i såfall stoppe de aktuelle monitorene. Når dette er utført stopper vi alle arbeidere som jobbet på samme sett.

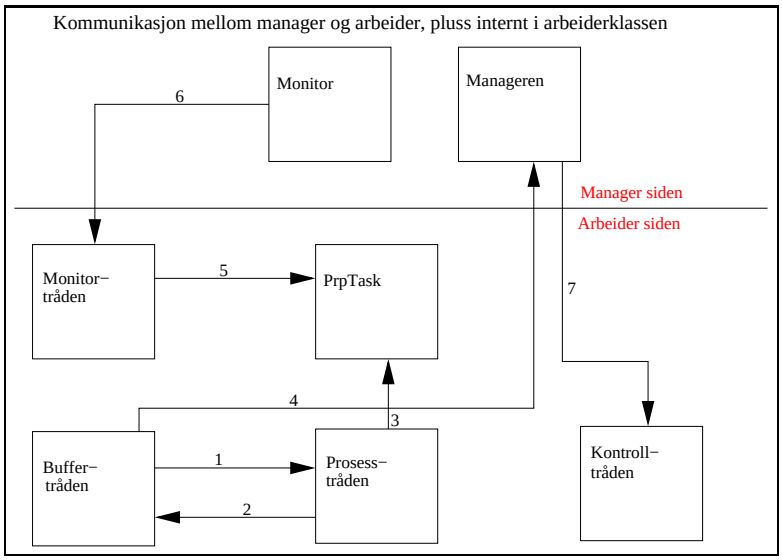
Når monitoren(e) er slått av(det vil si at den stopper å polle statistikk fra arbeideren), er vi klare får å splitte opp parameterett. Settet som skal splittes opp kan vi finne ut ved å se på listen over tilsendte parametersett til arbeideren. Vi vet at settet den hadde problemer med, er det første i listen. Vi kaller på metoden beskrevet over via refleksjon(jfr. 5.3.2) med dette settet og får beskjed tilbake at oppsplittingen var vellykket.

Steg 3: Parameteret er nå splittet opp. Neste steg blir å fylle opp bufferen(e) til de aktuell(e) arbeiderne med uløste parametersett.

Steg 4: Vi kan nå starte monitoren(e) og arbeidern(e) igjen med de nye settene.

5.4 Arbeidermaskinen

Vi skal i denne delen se på hva som skjer på arbeidersiden når man skal splitte opp et parametersett. Vi skal se hvordan vi stopper alt arbeid den driver med, for deretter å la den starte opp med et nytt parametersett. Men



Figur 5.3: Kommunikasjonen mellom de forskjellige trådene hos arbeideren og manageren

før vi gjør det, skal vi gi en kort repetisjon over alle trådene som kjører i arbeideren og hvilke kommunikasjon som skjer internt mellom disse og manageren.

De forskjellige trådene

Buffertråden Buffertråden er den som skaffer parametre til arbeideren. Fra figur 5.3 kan vi se at den hele tiden passer på at bufferen er av ønsket størrelse. Dette gjør den ved å kalle på getParam i manageren. Videre leverer den løste parametersett til manageren.

Prosesstråden Prosesstråden er den som utfører selve arbeidet. Den spør gjevnlig buffertråden om et nytt sett den skal løse, kaller PrpTask med dette settet og legger det deretter i svarkøen. Svarkøen tar buffertråden seg av som beskrevet over.

Monitortråden Denne tråden spør PrpTask om antall ganger den har kalt seg selv. Videre lagrer den antall parametersett arbeideren har løst

og hvor lang tid den har brukt. Det er disse dataene monitoren på managersiden spør om for å finne en treig arbeider.

Kontrolltråden Denne tråden gjør egentlig ikke så mye. Hvis brukeren velger å drepe en arbeider, så er det denne tråden som tar imot dette signalet.

5.4.1 Hvordan stoppe alle tråder og starte de opp igjen

For å stoppe tråder kunne man før i java bruke Thread.suspend() og Thread.stop(). Men i ettertid har disse blitt forkastet grunnet fare for deadlock. I steden anbefaler java å bruke en boolsk variabel som run metoden for de forskjellige trådene sjekker ofte. Vi skal her se på to mulige måter å gjøre dette på:

```
  class trådEksempel extends Thread {
    private boolean stop = false;

    public void run(){
      while(!stop){
<kode>
      }
    }

    public void stopThread(){
      stop = true;
    }
  }
```

Den andre måten vi kan terminere en tråd på er ganske lik. Forskjellen ligger i at sjekken på den boolske variabelen legges inn i while-løkken. Koden ville derfor sett slik ut:

```
  public void run(){
    while(true){
```

```
      <kode>

      if(stop){
return;
      }
      <kode>
    }
  }
```

Som vi ser er begge måtene like gode. Fra den tidliger koden var det lagt inn en en variabel "stop" som i første eksempelet i hver tråd i arbeideren. Det eneste vi trenger å gjøre er å sette denne til true. På denne måten vil hver tråd returnere fra sine run metoder. Dette er akkurat hva vi ønsker.

Men det er en ting vi må gjøre før vi setter den boolske variabelen til true, og det er å stoppe oppgaven som løses i PrpTask. Prosess tråden går hele tiden i while løkke og gir nye oppgaver til PrpTask. Det vil si at den ikke tar en nye iterasjon før oppgaven i PrpTask er løst. Dette fører igjen til at når vi setter den boolske variabelen til true, så ville ikke prosesstråden sjekket dette før oppgaven i PrpTask er løst.

Vi trenger derfor en metode som får den rekursive metoden i PrpTask til å stoppe så fort som mulig. Dette skal vi se på i neste avsnitt.

Når dette er utført er alt dødt i arbeideren. Manageren kan nå fylle opp bufferen må ny, og starte trådene på den vanlige måten.

5.4.2 Hvordan stoppe en rekursiv metode på raskest mulig måte

Vi skal i dette avsnittet se på hvordan man raskest mulig kan stoppe en rekursiv metode. I forrige avsnitt så vi på hvordan man raskest mulig kunne stoppe alle trådene som kjørte på arbeidersiden, men vi fant fort ut at det ikke bare holdt å sette en boolsk variabel i whileløkken, men at vi også må inn i PrpTask og stoppe den manuelt. Vi skal nå vise i detalj hvordan vi kan løse dette.

Fra før har vi denne koden å jobbe med:

```
  class PrpArbeiderImp {

    <variabledeklarasjoner og indre klasser>

    /* indre klasse som kaller på PrpTask med et parameter */
    class PrpProsess {

      public void run() {

while( <betingelser> ){

          Object par = < be buffertråd om nytt parameter >

          /* selve kallet til prpTask */
          prpTask.execute(par);

          < returner svaret til buffertråden >

}
      }
    }
  }


  class PrpTask {

    < Main metoden >

    < andre hjelpemetoder >

    /* den rekursive metoden */
    public < type > rekursivMetode(<parametre>) {

      < variabel deklarasjoner >

      if(<splittbetingelse>) {

        for/while( <x antall ganger> ) {

          < rekursivMetode(<parameter>) >
        }
```

```
    } else {

      < beregn svar >
    }
  }

  public Object execute(Object p){
    < kaster om p til et PrpParam object >
    < kaller rekursiv metode med p >
    < returnerer svaret til arbeideren >
  }

}
```

Som vi ser så kaller arbeideren på execute metoden i PrpTask objektet. Det er denne metoden som videre kaller på den rekursive metoden som brukeren selv har laget.

Første forslag: returnere null

Første ide som slo meg var å ha en metode i PrpTask som arbeideren kalte på. La oss kalle denne metoden terminate. Eneste oppgaven til terminate var å sette en boolsk variabel til true. Videre kunne den rekursive metoden sjekke hver gang den kalte seg selv om den skulle terminere eller fortsette jobben videre. Hvis den skulle terminere så måtte den returnere noe, som f.eks null. Videre ville null returneres videre oppover i treet hvor til slutt metoden execute mottok null.

Problemet som derimot dukker opp her, er at den rekursive metoden ikke alltid er av typen objekt. Veldig vanlig vil være å skrive en rekursiv metode som har typen int. Om metoden er int vil denne ikke kunne returnere null, da dette vil skape kompileringsfeil. Alternativ løsning til dette ville være at preprosessoren f.eks døper om int til Integer og double til Double i metodedeklarasjonen. Det vil da fungere, men skape mye arbeid.

Andre forslag: if-statements

Et annet alternativ som dukket opp var å legge inn flere if-statements i den rekursive metoden. hver if-statement kunne sjekke på hvilke type metoden var, for deretter å returnere et svar av riktig type. Dette svaret ville blitt returnert oppover i treet, hvor tilslutt arbeideren mottok et svar. dette svaret ville man bare forkastet.

Ved hjelp av denne løsningen ville man sluppet noe ekstra arbeid fra preprosessoren, men koden ville blitt mindre elegant.

Et bedre alternativ: returnere exception

Selv om de to første alternativene ville fungere, var jeg ikke helt fornøyd med hvordan koden ville blitt. Jeg ville derfor se på et bedre alternativ for å stoppe en rekursiv metode på best mulig måte.

I Java har man mulighet til å bruke try/catch setninger. Dette kan man utnytte for å stoppe PrpTasken som kjører. I steden for å kaste uriktige svar eller null videre opp i treet kan man kaste exceptions.

Ved å lage en egen subklasse av exception kan man kaste dette objektet oppover i treet når variabelen terminate blir satt til true. Å definere en egen subklasse av exception gjøres på denne måten:

```
public class RecursionException extends Exception {
  public RecursionException() {
    super();
  }

  RecursionException(String msg){
    super(msg);
  }
}
```

Hadde vi trengt mer tilleggsinformasjon i exception objektet kunne vi latt den extende RunTimeException, men i vårt tilfelle holder det å extende

Exception. Den endelige versjonen av eksempelet over vil derfor se slik ut.

```
  class PrpArbeiderImp {

    <variabledeklarasjoner og indre klasser>

    /* indre klasse som kaller på PrpTask med et parameter */
    class PrpProsess {

      public void run() {
while( <betingelser> ){

          Object par = < be buffertråd om nytt parameter >

          /* selve kallet til prpTask */
          try {
            prpTask.execute(par);
          } catch(Exception e){
            return;
          }

          < returner svaret til buffertråden >
}
      }
    }
  }

  class PrpTask throws RecursiveException {

    static boolean terminate = false;

    < Main metoden >

    < andre hjelpemetoder >

    /* den rekursive metoden */
    public < type > rekursivMetode(<parametre>) {
      < variabel deklarasjoner >
```

```
      if(terminate){
        throw new RecursionException(‘‘terminate’’);
      }

      if(<splittbetingelse>) {

        for/while( <x antall ganger> ) {
          try{
            < rekursivMetode(<parameter>) >
          }catch(RecursionException r){
            throw new RecursionException(‘‘terminate’’);
          }
        }
      } else {

        < beregn svar >
      }
    }

    public Object execute(Object p){
      < kaster om p til et PrpParam object >
      try {
< kaller rekursiv metode med p >
      }catch(RecursionException r){
throw new RecursionException(‘‘terminate’’);
      }
      < returnerer svaret til arbeideren >

    }

    public void terminate(){
      terminate = true;
    }
  }
```

Som vi ser er alle kall på den rekursive metoden satt inn i try/catch. I tillegg er det lagt inn en test øverst i metoden, som sjekker terminate variabelen. På denne måten vil en exception bli kastet oppover i treet, hvordan tilslutt når prosesstråden. Vi ser ser også at kallet fra prosesstråden har fått

en try/catch blokk rundt seg. På denne måten vet vi enkelt når det returnerer et feilaktig svar. Skjer dette så kan prosesstråden være sikker på at vi ønsker å stoppe den og den velger derfor å returnere fra sin run metode.

5.4.3 Fjerne bruken av static i PrpTask

En av de absolutte kravene til programmet brukeren skriver, er at alle metoder og klassevariable skal være static. Grunnen til dette er at klassen(PrpNavn.java) som starter hele runtime-systemet ikke oppretter noe objekt av seg selv. I dette tilfellet blir også alt statisk i PrpTasken som preprossessoren oppretter. Det er dette som skaper problemer når vi nå skal kjøre flere tasker på en og samme maskin.

Fra forrige avsnitt fortalte vi hvordan man skulle stoppe en PrpTask, ved å sette en boolsk variable til true, og deretter kaste en exception oppover i treet. Dette skaper et problem når vi oppretter to arbeiderobjekter på samme maskin. Når den ene arbeideren setter den statiske variabelen til true vil dette påvirke den andre. Dette er ikke ønskelig. Derfor måtte jeg inn og se på en løsning for å hindre bruken av statiske variable.

Når jeg så på koden som lå i PrpTask, så var den nesten helt identisk med koden brukeren skrev selv. Eneste forskjellen var at det var lagt inn noe tellervariable som Bjørn Arild brukte i sin oppgave. Videre viste jeg at main metoden som lå i klassen aldri ble kjørt. Grunnen til dette er at den allerede er blitt kjørt i PrpNavn under oppstart av runtime-systemet. Videre opprettes det et objekt av klassen fra PrpNavn som arbeidermaskinen får utdelt. Av dette sa det meg at det ikke var noen grunn til å ha statiske variable i PrpTask klassen. Det er heller ikke noe poeng i å ha mainmetoden da koden som står der aldri blir kjørt.

Dermed bestemte jeg meg for å legge til kode i preprosessoren som fjernet statisk bruk og main-metoden i PrpTask. Ved å gjøre dette fjernet vi problemet rundt det å terminere ved å sette en klassevariabel, men vi setter nå en objektvariabel i steden.

5.5 Oppdatering av gui

5.5.1 Grafisk framstilling av treet etter runtime

Ved oppsplitting av parametre så er det ikke lett å holde oversikten over hva som skjer bak kulissene. Det kunne derfor vært interessant å representere hvordan oppdelingen foregikk etter at oppgaven er løst.

Man kan her tenke stort og se for seg fin gui med noder(representert som sirkler) som er linket til hverandre. Men tenker man seg at det er 100-talls oppsplittinger blir det mye sirkler å representere på en skjerm, og brukeren vil kanskje ikke ha så mye glede av det likevel. Etter en oppfordring fra min veileder ville jeg gå for en enklere løsning, nemlig å representere treet ved hjelp av stringer. Måten dette kan gjøres på, kan illustreres på følgende måte:

Gitt at det er 5 startparametere. Under kjøring blir parametersett 3 og 4 oppsplittet i 2 deler hver. Videre blir det ene nye parametersettet fra oppdeling av 4 igjen oppsplittet i to nye delere. Dette kan vi enkelt vise ved følgende notasjon:

```
1

2

3
|____6
|____7

4
|____9
     |____11
     |____12
|____10

5
```

Man får her en enkel og klar framstilling av hvordan parameterne har blitt splittet opp under runtime. Framstillingen vil heller ikke kreve så mye

plass på skjermen som en mer avansert løsning. Et skjermbilde tatt under kjøringen av Prp er vist i figur 5.4 på neste side.

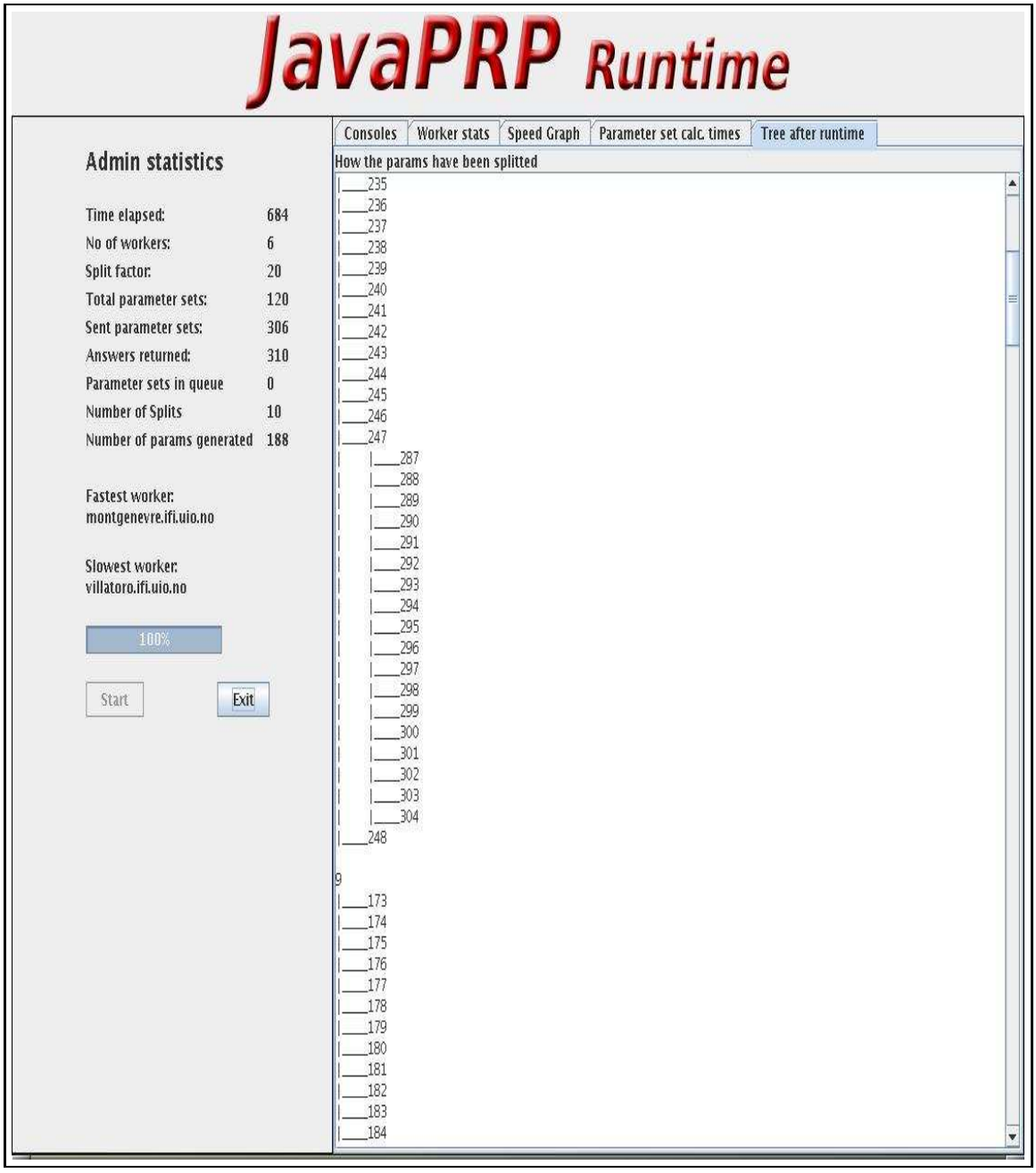
Videre har jeg oppdatert informasjonen i Admin stats og Worker stats. Fra figur 5.5 på side 78 ser vi et skjermbilde av den nye oppdateringen.

I Admin stats er det to nye felter med informasjon:

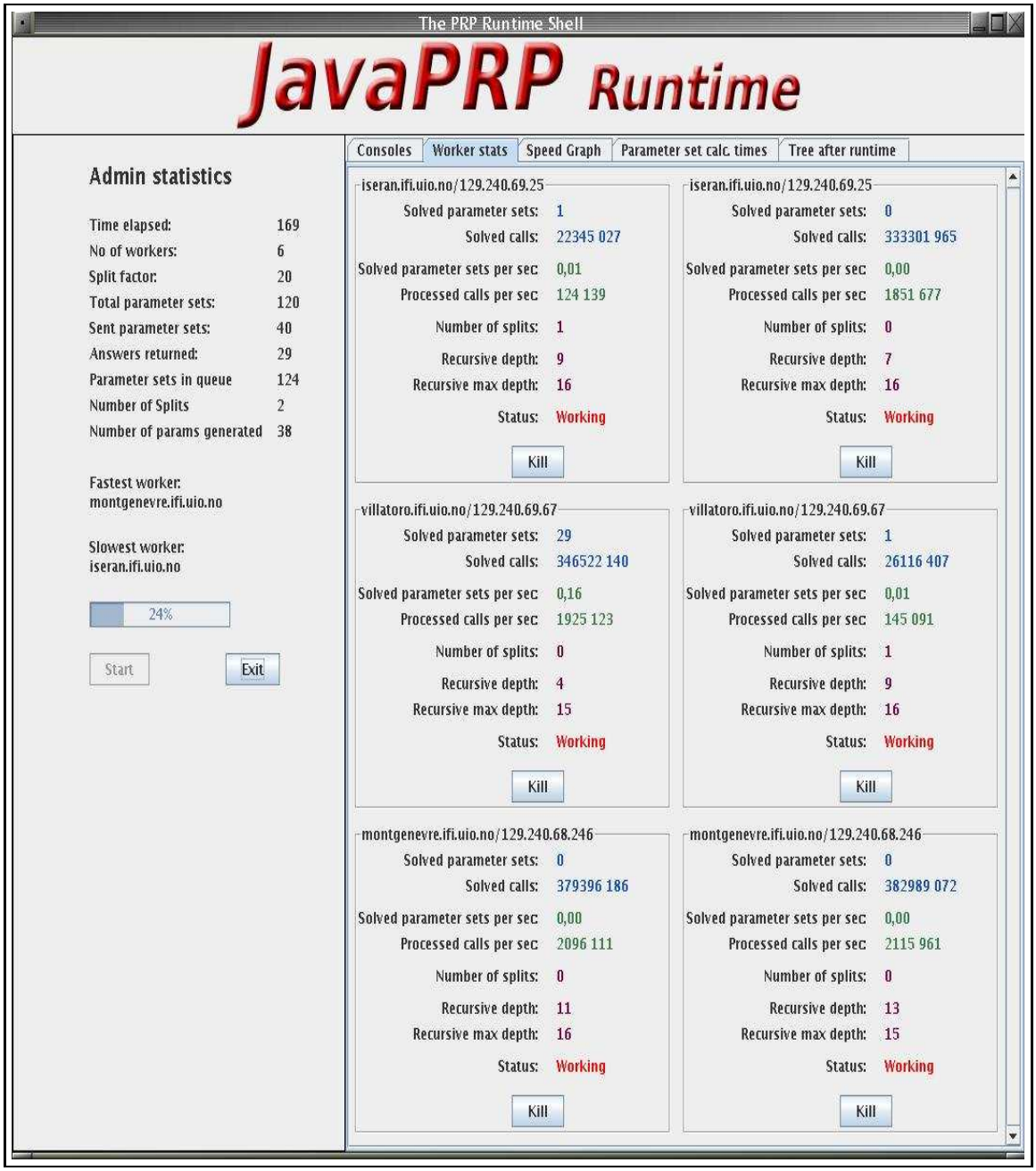
- Number of Splits: Dette felter viser totalt antall oppsplitinger.
- Number of params generated: Dette feltet viser hvor mange nye parametersett som er blitt generert.

I Worker stats er det 3 nye felter med informasjon:

- Number of splits: Dette felter viser antall oppsplittinger som er blitt gjort for den aktuelle arbeideren.
- Recursive depth: Dette felter viser hvile nivå arbeideren er i den rekursive metoden.
- Recursive max depth: Dette feltet viser den maksimale dybden en arbeider har nådd på et gitt tidspunkt.



Figur 5.4: oppdatert gui - treet av de nye genererte parametersettene



Figur 5.5: Oppdatert gui - arbeiderstatistikk

Kapittel 6

Design av problemstilling og uttesting for Travelling Salesman

Vi skal i dette kapittelet se på uttestingen for oppsplitting av vanskelige parametersett. Til nå har vi drøftet hva som skal til for å splitte opp et parametersett, og det er derfor på tide å se effekten av dette.

6.1 En passende problemstilling

Før vi går inn på resultatet, vil jeg drøfte problemet jeg vil teste systemet på. For at oppsplitting av sett skal ha noen effekt, så trenger vi en rekursiv metode som lager et ujevnt rekursivt tre. Skulle vi testet ut oppsplittingen på en jevnt rekursivt tre, så ville være node ta like lang tid, og det ville mest sannsynlig ikke skje noe oppsplitting i det hele tatt. Bare for å bevise dette startet jeg opp travelling salesman uten cufoff. Dette var resultatet vi fikk:

arbeidere	**Kjøretid**	**oppslitting**	**oppsplittinger**	**paramersett generert av oppsplittingen**
6	1460 s	uten	0	0
6	1410 s	med	3	36

Tabell 6.1: Testkjøring av Travelling salesman uten cutoff: 14 byer i Burma.

Som vi ser så er ikke tidsdifferansen veldig stor, og man kan ikke si at oppsplittingen ga så mye effekt. Travelling salesman i seg selv lager ikke et ujevnt rekursivt tre, men ved å legge til cutoff i treet så vil ikke hver node ta like lang tid.

Etter samtale med min veileder gikk jeg i gang med å finne en passende initsiering av byer for travelling salesman. Det vi vil ha, er parametersett, hvor de ulike nodene tar forskjellige tid å løse. Og etter litt testing kom jeg fram til følgende eksempel for N byer(her N=18), som er initsiert fra den uniforme fordeling av 1..N tallene.

```
public static int[][] cities =

{{0, 16, 16, 10, 17, 16, 15, 17, 13, 14, 7, 3, 15, 5, 5, 9, 15, 10 },
 {16, 0, 4, 17, 12, 17, 13, 12, 17, 17, 1, 7, 16, 2, 2, 12, 4, 1 },
 {16, 4, 0, 11, 16, 5, 2, 1, 1, 14, 14, 14, 2, 11, 18, 3, 7, 2, },
 {10, 17, 11, 0, 18, 18, 11, 1, 8, 14, 5, 16, 15, 9, 16, 1, 3, 10 },
 {17, 12, 16, 18, 0, 5, 3, 7, 14, 15, 5, 5, 5, 14, 5, 11, 9, 8 },
 {16, 17, 5, 18, 5, 0, 14, 8, 17, 11, 2, 8, 1, 1, 4, 10, 14, 16 },
 {15, 13, 2, 11, 3, 14, 0, 6, 2, 14, 7, 4, 2, 6, 15, 2, 4, 15 },
 {17, 12, 1, 1, 7, 8, 6, 0, 15, 13, 15, 4, 4, 9, 7, 9, 1, 8 },
 {13, 17, 1, 8, 14, 17, 2, 15, 0, 6, 8, 17, 1, 12, 2, 6, 6, 16 },
 {14, 17, 14, 14, 15, 11, 14, 13, 6, 0, 3, 5, 15, 13, 2, 2, 11, 12 },
 {7, 1, 14, 5, 5, 2, 7, 15, 8, 3, 0, 17, 7, 16, 5, 12, 8, 17 },
 {3, 7, 14, 16, 5, 8, 4, 4, 17, 5, 17, 0, 1, 12, 9, 18, 13, 3 },
 {15, 16, 2, 15, 5, 1, 2, 4, 1, 15, 7, 1, 0, 10, 17, 13, 13, 7 },
 {5, 2, 11, 9, 14, 1, 6, 9, 12, 13, 16, 12, 10, 0, 9, 15, 16, 16 },
 {5, 2, 18, 16, 5, 4, 15, 7, 2, 2, 5, 9, 17, 9, 0, 12, 7, 6 },
 {9, 12, 3, 1, 11, 10, 2, 9, 6, 2, 12, 18, 13, 15, 12, 0, 6, 3 },
 {15, 4, 7, 3, 9, 14, 4, 1, 6, 11, 8, 13, 13, 16, 7, 6, 0, 12 },
 {10, 1, 2, 10, 8, 16, 15, 8, 16, 12, 17, 3, 7, 16, 6, 3, 12, 0 }
};
```

Som vi ser, så er det stor variasjon av avstandene som igjen skaper stor cutoff i det rekursive treet.

6.1.1 Hva kan vi forvente oss?

Et interessant spørsmål som dukker opp, er hva vi burde forvente oss av oppsplittingen. Selv om vi får et positivt resultat ut av testkjøringen, så burde vi ha et begrep om dette er bra nok eller ikke. Å begynne å regne på denne problemstillingen kan være litt problematisk, siden tiden det tar å løse parametersettene under en kjøring kan variere fra 1 til noen tusen sekunder.

En mulighet for å se om kjøringen var optimal, er å sammenligne tiden det tar å kjøre den rekursive problemstillingen på en maskin(sekvensiell kjøring), mot tiden prp-systemet bruker ved å kjøre på X antall maskiner. Blir oppgaven løst X ganger raskere så kan vi med trygghet si at det ikke går ann å løse oppgaven raskere.

6.2 Uttestingen

Vi har nå en problemstilling, og det er på tide å få ut noen tider som viser hva systemet er god for. Men før vi gjør det skal vi se på hvilken formel vi kommer til å bruke for å avgjøre hva som gjør en arbeider treig. Videre skal vi se på 2 forskjellige måter å håndtere parametersettene, og et alternativ på valg av tidspunkt for oppsplitting. Disse tidene vil vi tilslutt sammenligne og ta noen avsluttende konklusjoner.

Valg av formel Noe som er avgjørende ved oppsplittingen er formelen som forteller manageren at nå må vi stoppe arbeideren og splitte opp parametersettet. Denne avgjørelsen foregår i klassen SplittManagement. Her finner man to metoder:

- En metode som lagrer tider for ferdige parametersett i en liste, paramTimes.
- En metode, checkTime, som går utifra listen paramTimes nevnt over og sjekker mellomtider til parametersett som en arbeider driver å regner på.

En tanke jeg kom på fort var det at man kunne overse parametersett som tok veldig kort tid. Slike sett ville trekke ned snittet betraktlig, og gjøre

Antall arbeidere	Kjøretid
6	5054 s

Tabell 6.2: Testkjøring av Travelling salesman med 18 byer med bruk av cutoff, men uten bruk av oppsplitting

at vi kanskje splittet et parametersett alt for tidlig. En annen grunn er at prp-systemet er best egnet for rekursive problemstillinger som bruker en del tid. Jeg valgte derfor å legge inn et absolutt tall som splittmanageren sjekket på når den skulle lagre tider en arbeider hadde brukt på et parametersett.

Videre var det formelen som skulle brukes i metoden checkTime. Jeg bestemte å gå for følgende formel som avgjorde om vi skulle splitte eller ikke:

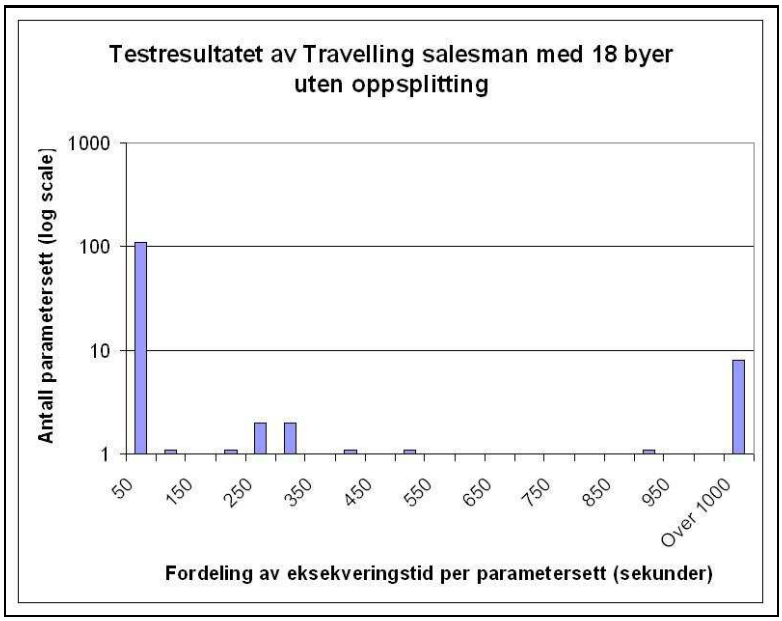
- Mellomtiden som kom inn måtte være større en 10 sekunder.
- Mellomtiden som kom inn må være dobbelt så stor som snittet av ferdige parametersettene.
- Det må finnes minst 1 ferdig parametersett. Dette for å ha noe å måle opp mot.

6.2.1 Ingen oppsplitting

For å ha en tid å måle opp mot, ville jeg kjøre en test uten oppsplitting på travelling salesman. Utifra denne tiden kan jeg videre avgjøre hvilke av de neste alternativene som fungerer best.

Etter testkjøring kom jeg fram til følgende resultat som vist i tabell 6.2. Observasjoner jeg la merke til var at noen parametersett gikk veldig fort, mens andre kunne bruke opp mot 1000 sekunder. Eksekveringstidene for de forskjellige parameterne vises i figur 6.1 på neste side [1].

[1] I tabellen har jeg brukt logaritmisk skala langs y-aksen. Dette har jeg gjort for å framheve de få parametersettene som bruker veldig lang tid, da det er disse vi er mest interessert i. Videre i kapittelet vil jeg bruke samme fremgangsmåte.



Figur 6.1: Kjøretider for de forskjellige parametersettene - ingen oppsplitting

6.2.2 Alternativ 1: Legge nye sett sist i FIFO-køen

Første testkjøring jeg ville prøve meg på var som følger:

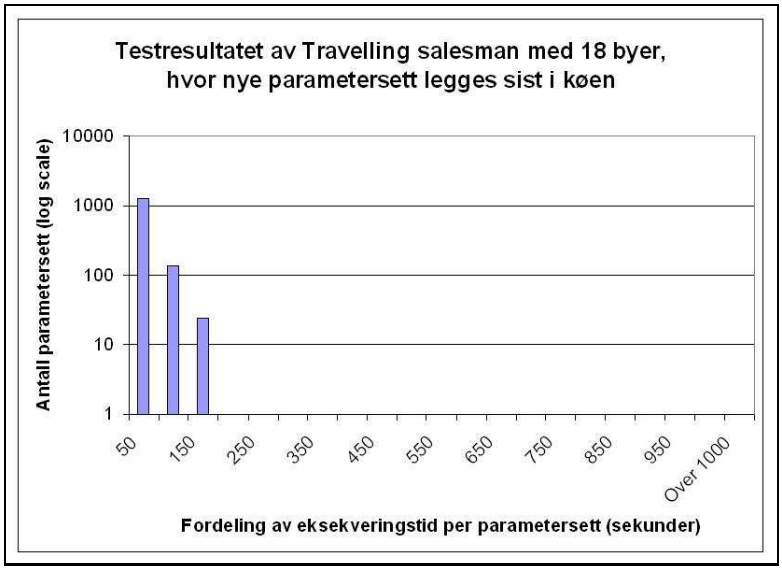
- Når SplitManagement fant ut at det var en treig arbeider, så ville den splitte momentant.
- Nye genererte parametersett skulle legges sist i FIFO-køen, som arbeiderne senere ville hente parametersett fra.

Av dette forventet jeg at tiden ville gå ned betraktlig. Gjennomsnittstiden mellom de forskjellige arbeiderne ville ligge rundt det samme, og det ville mest sannsynlig skje mange oppsplittinger.

Fra figur 6.3 på neste side kan vi se resultatet av kjøringen. Observasjoner jeg her la merke til var at noen parametersett ble stoppet veldig fort, og snittet mellom arbeiderne lå på ca på 20 sekunder. Fordelingen mellom hvor lang tid hvert parametersett tok vises i figur 6.2 på neste side. Vi kan her se at de fleste settene holder seg 100 sekunder, noe som igjen viser en bedre lastbalanse mellom settene.

Antall arbeidere	Kjøretid	oppsplittinger	parametersett generert av oppsplittingen
6	3616 s	87	1291

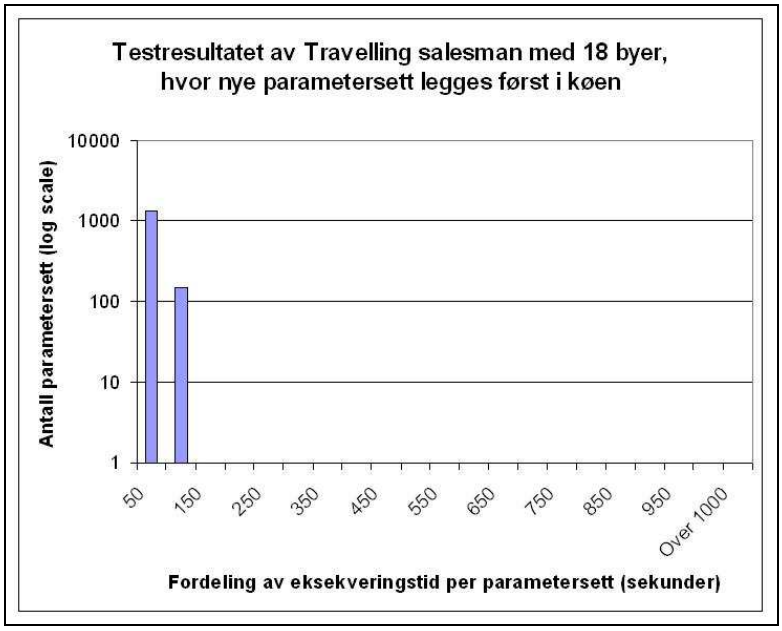
Tabell 6.3: Testkjøring av Travelling salesman med 18 byer, hvor nye parametersett legges sist i FIFO-køen.



Figur 6.2: Kjøretider for de forskjellige parametersettene - legge nye sett sist i køen

Antall arbeidere	Kjøretid	oppsplittinger	parametersett generert av oppsplittingen
6	3521 s	98	1447

Tabell 6.4: Testkjøring av Travelling salesman med 18 byer, hvor nye parametersett legges først i køen.



Figur 6.3: Kjøretider for de forskjellige parametersettene- legge nye sett først i køen

6.2.3 Alternativ 2: Legge nye sett først i køen

Neste alternativ jeg ville teste ut var ganske lik første alternativ. Eneste forskjellen jeg ville gjøre, var å legge nye genererte parametersett først i køen.

Fra tabell **??** på side ?? ser vi resultatet av kjøringen. vi kan her se at det ble flere oppsplittinger, og tiden den brukte var litt raskere. Andre observasjoner jeg la merke til var at snittiden på de forskjellige parametersettene gikk ned litt. Fra 20 i alternativ 1, lå snittet nå på ca 15 sekunder. Grunnen til dette er at mindre parametersett som blir lagt først i køen, er med på holde tiden nede. Dette vil igjen påvirke splittmanageren

til å splitte raskere en om vi legger de nye parametersettene sist i køen. Hvor lang tid de forskjellige settene brukte er vist i figur 6.3 på forrige side. Her ser vi at det raskeste alternativet er å splitte med engang, og samtidig legge de nye parametersettene først i køen.

6.2.4 Alternativ 3: Ikke splitte før det er ledige arbeidere

Tredje og siste alternativ jeg ville teste ut, var å ikke avbryte arbeidere så fort som mulig, men vente til det var ledige arbeidere. I vårt tilfelle vil det si når det finnes andre arbeidere som jobber på samme parametersett(jfr backupmekanismen).

Hva vi her kunne vente var jeg veldig usikker på. Det jeg først om fremst ville tro var at snittet til arbeiderne seg imellom ville gå betraktlig opp, men hva sluttiden ville bli i forhold til de andre alternativene nevnt over var jeg veldig usikker på.

Det skulle vise seg å bli litt problemer å kjøre akkurat denne testkjøringen. Det var flere grunner til dette:

1. Når en arbeider kalte på ferdigmetoden, så gjorde den dette fordi FIFO-køen var tom for parametersett. Den tok ikke hensyn til om det var blitt splittet opp i mellomtiden. Fra tidligere logikk ville den derfor kopiere parametersett fra en annen arbeider, selv om det lå nye sett på FIFO-køen.

2. Arbeiderne skjønte ikke når alle parametersett var løst, men fortsatte å kopiere sett fra hverandre i det uendelige.

3. Buffertråden på arbeidersiden hoppet ut av sin løkke før alle parametersett var løst.

4. Hvis det skjedde en oppsplitting etter at første arbeider hadde kalt ferdigmetoden, så skapte dette krøll på utsendt listene i det vi fjernet fra listen til arbeideren som trengte en oppsplitting.

Alle disse punktene inntraff ikke når vi kjørte de to første kjøreeksemplene, så det var bra at vi kom til et eksempel som virkelig testet ut de forskjellige funksjonene.

Antall arbeidere	Kjøretid	oppsplittinger	parametersett generert av oppsplittingen
6	4737 s	6	93

Tabell 6.5: testkjøring av Travelling salesman med 18 byer hvor manageren venter med oppsplitting til det er en ledig arbeider.

Når det gjelder punkt 1, så la jeg inn nye kode som sjekket akkurat dette. Hvis FIFO-køen ikke var tom så skulle den bare kopiere sett fradenne listen og ikke fra en uferdig arbeider.

Punkt 2 løste jeg ved å la ferdigmetoden kalle på en ny metode som sjekket følgende:

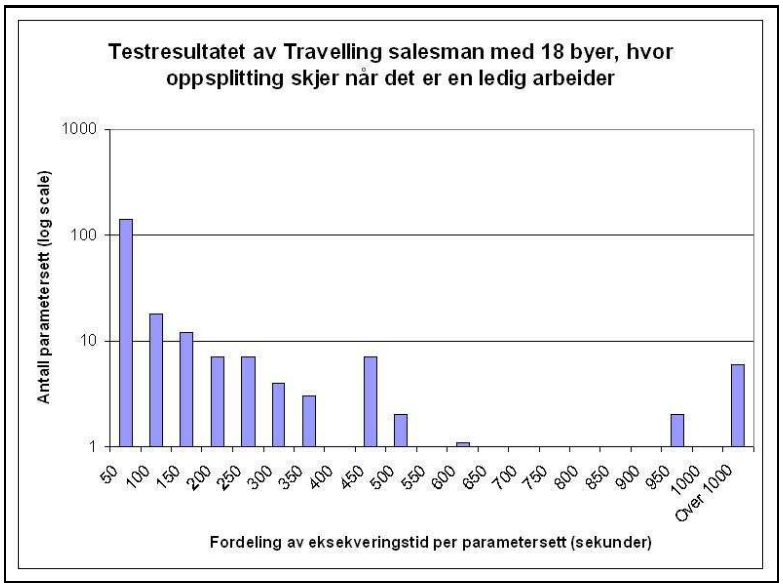
- Hvis returstakken innholdt nullobjekter på indekser som ikke var splittet, så betydde det at det var en oppgave der ute som ikke hadde returnert enda.

Det tredje punktet var en triviell sak, da jeg la til en sjekk i løkken som sjekket om variabelen "finish" var satt til true. Hvis den ikke var dette så skulle den fortsette å hente nye parametersett, eller kopiere sett fra uferdige arbeidere.

Det fjerde punktet var også lett å løse, men kanskje ikke så lett å oppdage. Grunnen til at det ble slettet parametersett fra flere utsendtlister når du egentlig ville bare fjerne et, var at ferdigmetoden kopierte ikke sett fra en utsendt liste til en annen. Den satt bare utsendt listen til en arbeider til å peke på en annen arbeider sin utsendt liste. Dermed vil fjerning av parametersett påvirke flere arbeidere. Ved å kopiere settene i steden så fjernet man denne problematikken.

Etter å ha løst disse problemene var vi klare for testkjøring. Fra tabell 6.5 kan vi se at tiden den brukte ble raskere en om den ikke hadde splittet opp i det hele tatt. Men den klarte ikke å måle seg opp mot de to første alternativene.

Observasjoner jeg la merke til etter kjøring var som forventet, at snittet mellom de forskjellige arbeiderene varierte fra 50 til 400 sekunder. Dette igjen førte til at antall oppgaver som ble løst mellom arbeiderne også varierte fra 15 til over 100 sett. Fra figur 6.4 på neste side ser vi eksekveringstidene mellom de forskjellige parametersettene.



Figur 6.4: Kjøretider for de forskjellige parametersettene- vente til det er ledige arbeidere

Som vi ser, så skaper ikke denne løsningen en god lastbalanse mellom arbeiderne og vil derfor ikke være et alternativ man ønsker å benytte seg av.

6.3 Konklusjon

Vi har nå sett på 3 alternativer for å løse oppsplitting av parametersett, og av de forskjellige framgangsmåtene så vi også en viss variasjon. På figur 6.5 på side 90 ser vi en sammenligning av de forskjellige alternativene og det er to som skiller seg klart ut, med en forbedring på 30 prosent.

Det som nå gjensto var å se hvor rask prp-systemet hadde vært i forhold til samme kjøring på en maskin. Her viste det seg at den sekvensielle kjøringen var mye raskere. Dette var ikke som jeg håpet. Etter å ha tittet på travelling salesman sammen med min veileder, så viste det seg at koden(Vedlegg A) ikke var helt korrekt. Feilen lå i utregningen av cutoff. Den ble ble ødelagt etterhvert som koden kjørte. Derfor viste Maus meg en bedre løsning som jeg igjen skulle teste ut på(se vedlegg C).

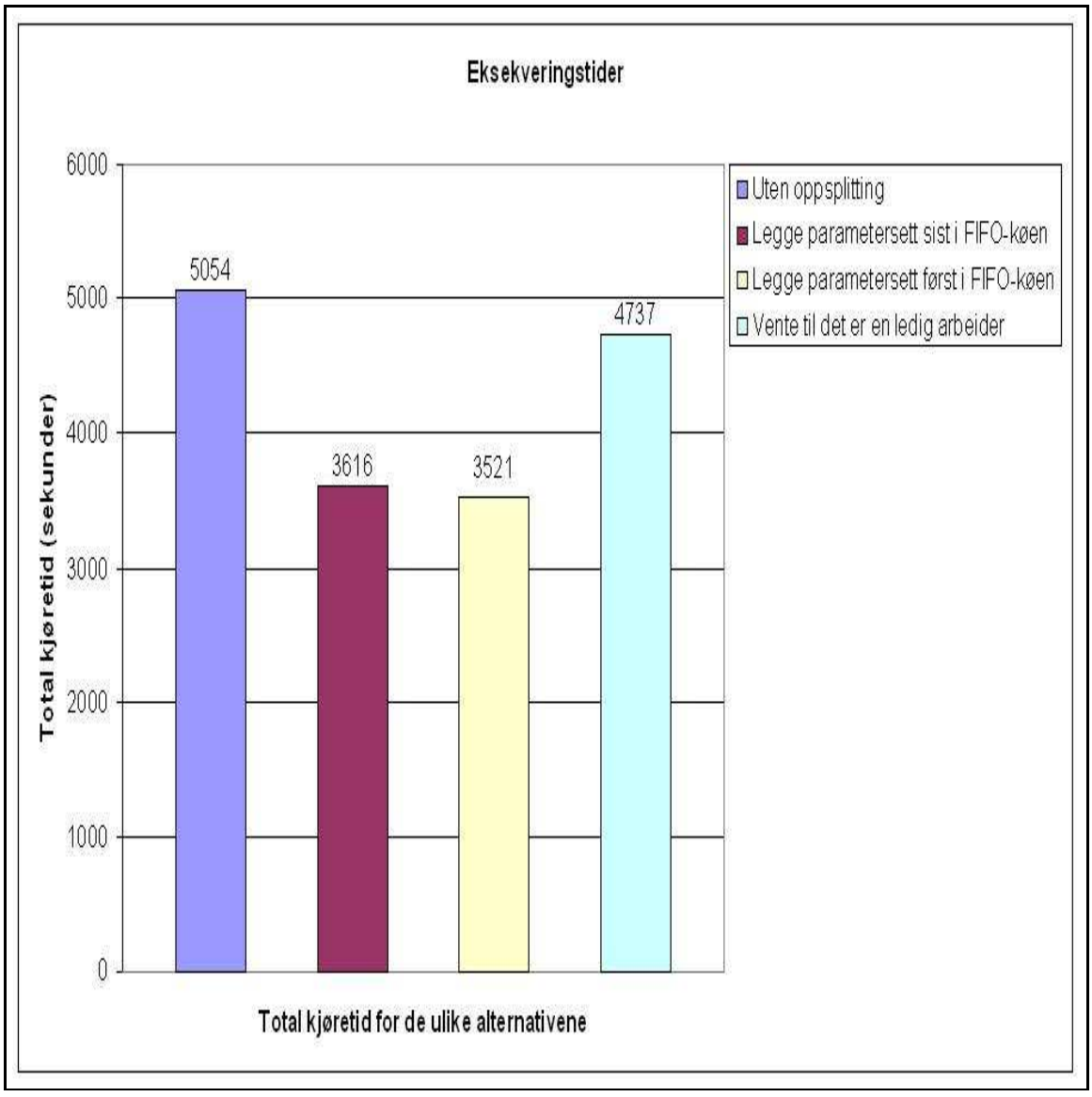
Hva ble kjørt	Kjøretid
Sekvensiell kjøring	ca 1000 s
3 arbeider i prp-systemet(uten utnyttelse av flere kjerner)	ca 500 s
3 maskiner starter opp to arbeidere hver	ca 1000 s
6 maskiner uten utnyttelse av flere kjerner	ca 4-500s

Tabell 6.6: Testresultatene for ny versjon av Travelling salesman med 21 byer

Kjøreeksempelet jeg kjørte var med 21 byer. Resultatet viser i tabell 6.6

Fra disse resultatene kunne vi ta noen konklusjoner. Ved å sammenligne tiden 3 maskiner bruker(uten utnyttelse av flere kjerner) med tiden 3 maskiner, hvor hver maskin starter 2 arbeidere, så er førstnevnte raskest. Grunnen til at dette skjer, er pga oppsplittingen før oppstarten av runtime-systemet. Oppsplittingen før runtime lager 20 ganger antall arbeidere parametersett. I vårttilfelle vil det her si 60 og 120 parametersett. Grunnen til at kjøringen hvor det genereres 120 parametersett går tregere er fordi det fortsatt er et stygt parametersett som bruker lang tid. Når det da er dobbelt så mange sett som den andre kjøringen nevnt over, vil dette ta lenger tid.

For å se om jeg klarte å komme rundt denne problemstillingen, ville jeg kjøre samme testeksempel, men denne gangen ha en splittfaktor på 100 i steden for 20. Av resultatene denne teste viste, så nærmet de to kjøreeksemplene hverandre, men fortsatt var 3 maskiner raskere i forhold til 3 maskiner hvor hver maskin starter 2 arbeidere. Pga mye tidsbruk på dette måtte jeg derfor legge saken død, og slutte meg til resultatene vist i figur 6.5 på neste side.



Figur 6.5: Sammenligning av de forskjellige alternativene

Kapittel 7

Oppsummering og videre arbeid

Gjennom Bjørn Arilds(11) oppgave gikk fokuset på PRP-Systemet fra å få systemet til å løse en problem raskest mulig, til å gi et bedre og mer intuitivt brukergrensesnitt.

Gjennom min masteroppgave har igjen fokuset kommet inn på ytelse, og hvor mye raskere vi kan få systemet til å løse en rekursiv problemstilling. I dette kapittelet vil jeg gi en liten oppsummering på hva som har blitt oppnådd og hva jeg eventuelt kunne gjort annerledes. Til slutt vil jeg belyse noen ideer på hva man kan jobbe med videre for å få systemet ennå bedre.

7.1 Oppnåde mål

Når jeg startet på denne masteroppgaven, så var det to store hovedmål jeg sto ovenfor(som nevnt i kapittel 3).

1. Utnyttelse av flere kjerner/cpuer:
2. iterativ fordypning under runtime:

Utnyttelse av flere kjerner/cpuer var oppgaven jeg skulle løse først. På forhånd var jeg veldig usikker på hvor store forandringer som måtte til for å få dette til å fungere. Men etter å ha tenkt godt igjennom

problemstillingen, og drøftet flere sider av saken, så viste det seg at oppgaven var overkommelig. Fra resultatene kunne vi se at min implementasjon oppnåde målet ved å utnytte de ressurser en maskin hadde tilgjengelig. Ved å sammenligne mine resultater med testresultater fra andre informasjonskilder, kunne vi se at systemet utnyttet maskinen fullt ut.

Videre gikk jeg over på punkt 2, og det viste seg at dette var en mer krevende problemstilling. For å oppnå iterativ fordypning under runtime, så måtte jeg omstrukturere store deler av koden. Dette innebar forandringer i preprosessoren, managersiden og arbeidersiden. Siden det innebar så store forandringer, så prøvde jeg å dele opp problemstillingene i mindre biter. Det ble derfor gjennomført mye testeksempler utenfor prp-systemet som prøvde å løse problemer som refleksjon, stoppe rekursive kjøringer samt stoppe og starte tråder på en effektiv måte uten tap av informasjon. Videre brukte jeg disse kodeeksemplene som mal når jeg senere skulle putte det inn i Prp-systemet.

Fra testresultatene presentert i kapittel 6, så viste min implementasjon at jeg på et vis fikk systemet raskere iforhold til det å ikke splitte opp. Med dette tenker jeg på at den sekvensielle kjøringen gikk raskere en den parallelle(mer om dette i neste seksjon).

Avsluttningsvis kan jeg si at begge oppgaver har blitt løst, og jeg er fornøyd med resultatene jeg har oppnådd. Min implementasjon viser at jeg klarte å få systemet raskere, noe som var en overordnet målsetning for denne oppgaven.

7.2 Selvkritikk

Selv om jeg er fornøyd med hva som er blitt oppnådd i min masteroppgave, kan det likvel være viktig å se tilbake på hva som kunne blitt gjort annerledes.

7.2.1 metodedesign for oppsplitting av vanskelige parametersett

Metoden jeg lagde som ble lagt i den preprosesserte filen, hadde som ansvar å ta i mot en parametersett og splitte det opp i mindre biter. Løsningen jeg valgte når det gjaldt å splitte settet opp i mindre biter, var å senke det rekursive nivået med 1. Det vil si at ved kjøring av Travelling Salesman med 18 byer, så vil et vanskelig parametersett bli splittet opp i 17 mindre biter.

En annen mulighet for å løse denne oppgaven, er å heller tenke på hvor mange deler man ønsker det vanskelige parametersettet skal splittes opp i. For å løse dette kunne man hatt en variabel som programmereren lett kunne forandre for å sjekke effekten av å splitte opp i få eller flere biter. Man kunne også valgt å legge dette inn i brukergrensesnittet for en bruker.

7.2.2 Uttesting på flere typer maskiner

Når jeg startet på denne masteroppgaven, var det å løse iterativ fordypning først og fremst tenkt på maskiner med flere kjerner/cpuer. Men på grunn av liten tilgjengelighet av dette på universitetet, ble det til at jeg kjørte mine tester på hyperthread maskiner i steden.

Selv om det ikke var så mye å gjøre med, skulle jeg likevel ønske at jeg hadde hatt tid til mer uttesting på maskiner med flere kjerner eller flere single cpu'er. På denne måten kunne jeg med sikkerhet ha sagt at det å starte opp x arbeiderprosesser på en maskin forholdt seg likt om det var hyperthreading eller flere kjerner.

7.3 Videre arbeid

Gjennom en rekke hovedfagsoppgaver er ideen om å spre en rekursiv beregning(Parallelle Rekursive Prosedyrer) på en større antall maskiner blitt utviklet. Enda gjenstår mye interessant arbeid, og det er også slik at jo mer man jobber med denne problemstillingen, desto flere nye problemstillinger har kommet opp.

I dette avsnittet vil jeg snakket litt om mulige forbedringer og utvidelser av PRP-Systemet som jeg har kommet på underveis. Deler er også ideer fra tidligere masterstudenter.

7.3.1 Innsyn i parametersett og svar under runtime

En interessant utvidelse av prp-systemet ville være å lage en mekansime for å se hvilke verdier som blir sendt ut til de forskjellige arbeiderne. I dag ser man kun hvilke parametersettnummer en arbeider får, men ikke konkret hvilke verdier den jobber på. Dette kunne også vært en fordel ved debugging for senere utvidelser av systemet.

For å få til dette må man se på måter for å aksessere javaklassene PrpParam og PrpRetur. I dag er det kun brukerens preprosesserte program som har tilgang til disse. Måter å løse dette på, er ved bruk av refleksjon. Hvordan man bruker refleksjon har jeg beskrev grundig i Kapittel 5.

7.3.2 Lettvekts webtjener

Om Prp-systemet en dag blir tilgjengelig for folk utenfor UiO, så er det et problem en bruker vil møte. Dette problemet gjelder det å ha et eget webområde. Mats Bue(6) lagde i sin masteroppgave en egen lettvekts webtjener. Det kan derfor være interessant å se om noe lignende er mulig i java.

7.3.3 Interaktiv og enkel pålogging

Et irritasjonsmoment ved PRP-GUI i dag er måten man må sette opp en arbeider. Man må selv gå inn på en maskin, starte arbeiderprosessen og etterpå legge denne maskinen til når man starter administrator. Dette tar tid, og for en ny bruker er dette både tidkrevende og kjedelig.

I Bjørn Arilds(11) oppgave blir det diskutert flere muligheter for å forenkle dette.

1. Automatisk oppstart:

Programmet selv logger seg inn på andre maskiner. Dette kan derimot være vanskelig med tanke på brannmur og lignende

2. Dedikerte maskiner:
   En annen mulighet er å alltid ha programmet kjørende på noen dedikerte maskiner på UIO. På denne måten vil man alltid ha noen arbeidere som står klare.

Alternativ 1 kan man kanskje komme seg rundt, men for en bruker som ikke har noen maskiner den kan logge seg på blir dette vanskelig. Alternativ 2 er en mulighet, men i et større perspektiv med flere nye brukere blir også dette en litt smal løsning. Om man setter opp 5 dedikerte maskiner, og det er 20 brukere som vil teste det ut samtidig, så ser man fort at ting ikke vil gå så fort.

Etter å ha lest om SETI(Search for Extra-Terrestrial Intelligence) prosjektet, som jobber med å finne intelligent liv utenfor vår jord, likte jeg deres løsning på bruken av andres maskiner. De har en egen hjemmeside hvor du selv kan laste ned et program. Ved å installere og kjøre dette er du med på å analysere data fra verdensrommet. Denne tankegangen kan føres over på GUI-PRP. Jeg ser for meg følgende:

- Opprette en hjemmeside for GUI-PRP hvor man kan laste ned arbeiderprogrammet.
- Man oppretter en server på en dedikert maskin på UIO, som lytter etter meldinger fra en arbeiderprosess.
- Ved kjøring av en arbeider vil den sende sin ip-adresse til serveren, samt annet info(cpu antall, ledig/ikke ledig).
- Ved kjøring av administrator, vil den be serveren om disse opplysningene. Videre vil den kontakte alle ledige arbeiderprosesser som er registrert.
- Når arbeideren får en oppgave, vil den registrere til serveren at den er opptatt med en beregning.

7.3.4 Open source

Hittil har det vært ganske begrenset på hvem som har brukt GUI-PRP. Gjennom flere oppgaver er det kun masterstudenter som har tatt i bruk programmet. En ide er å gjøre det åpent for andre. Vi har allerede pratet om å lage en hjemmeside for registrering. Det kunne også på denne siden vært aktuelt å legge ut kildekode og gjort GUI-PRP til et open source prosjekt. Fordelen med dette er at man får mange meninger om hva som er bra, og hva som kan forandres. Ved en slik ordning vil man også få en raskere utviklig.

7.3.5 Overvåkning av CPU-bruk

Når jeg startet uttestingen for oppsplitting av vanskelige parametersett, så var det en informasjon jeg gjerne skulle hatt tilgang til, nemlig cpu-bruk. Det kunne vært interessant å se hvor mye ressurser prp-systemet brukte på de forskjellige prosessorene. Java har i dag ikke noe bibliotek for å hente ut slik informasjon. For å løse denne oppgaven må man se nærmere på andre muligheter for å kontakte operativsystemet.

Referanser

[1] Et nytt forskningsnett for grid og lambda [online; aksessert våren 2006] http://www.uninett.no/uninytt/2003-3/gri.html.

[2] The main architectural classes. [online; aksessert høsten 2006] http://www.top500.org/orsc/2004/architecture.html.

[3] Torfinn Aas. Prp - en implementasjon av system for parallelle rekursive prosedyrer(prp). Master's thesis, Universitetet i Oslo, Institutt for informatikk, 1994.

[4] Gregory R. Andrews. *Mutlithreaded, Parallel, and Distributed Programming*. 2000.

[5] Micheal Allen Barry Wilkinson. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. 1999.

[6] Mats Bue. Prp-systemet implementert med c#/.net. Master's thesis, Universitetet i Oslo, Institutt for informatikk, 2005.

[7] Viktor Eide. Parallel recursive procedures, a manager/worker approach. Master's thesis, Universitetet i Oslo, Institutt for informatikk, 1998.

[8] Ian Foster. *Designing and bulding parallel programs*. 1995.

[9] Kesselman Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufman Publishers, 1999.

[10] Arne M. Hørstmark. Parallel recursive procedures ver. 2.0 extension of a distributed parallel system. Master's thesis, Universitetet i Oslo, Institutt for informatikk, 1997.

[11] Bjørn Arild Kristiansen. Gui-prp: et brukervenlig, høynivåsystem for parallellprogrammering. Master's thesis, Universitetet i Oslo, Institutt for informatikk, 2006.

[12] Arne Maus og Torfinn Aas. Prp parallel recursive procedures. 1995.

[13] Tore Andre Rønningen. Javaprp. Master's thesis, Universitetet i Oslo, Institutt for informatikk, 2003.

[14] Christian O. Søhoel. Parallel recursive procedures in a chess engine. Master's thesis, Universitetet i Oslo, Institutt for informatikk.

[15] Cuong Van Truong. Parallelle rekursive prosedyrer: Preprosessor. Master's thesis, Universitetet i Oslo, Institutt for informatikk, 2007.

[16] Yan Xu. Prp-systemet implementert på mpi. Master's thesis, Universitetet i Oslo, Institutt for informatikk, 1997.

Tillegg A

Goldbach.java

```
/**
 *
 * @author Jørn Christian Syversrud
 *
 * Program checks that a even number is always the sum of two prime numbers
 * (still not proven in mathematics)
 *
 *  USAGE: java Goldbach
 *
 */

public class Goldbach2 {
    static boolean [] prime = null;

    public static void main(String [] args) {

   /**
    * start checking even numbers
    */
   int sumGB = 0;
   try {
       sumGB = checkEvenNumbers(1,1000000);
   }catch(Exception e){

   }


   System.out.println("sumGB = " + sumGB);

    }

    /*
     *  determine primes < N using Sieve of Eratosthenes
     */
    public static void getPrimeNumbers(boolean isPrime[]){

   for (int i = 2; i * i < isPrime.length; i++) {
       if (isPrime[i]) {
      for (int j = i; i * j < isPrime.length; j++)
          isPrime[i*j] = false;
       }
   }

    }

    /*PRP_PROC*/
    public static int checkEvenNumbers(int from, int to){

   int newFrom = 0; //used when splitting interval
   int newTo = 0;   //used when splitting interval
   int sumGB = 0;   //number of GB sums
```

```
    //used when child returns from recursive call
    int[] childCheck = new int[2];

    //calculate the prime numbers
    if(prime == null){

        prime = new boolean[1000000];
        for (int i = 0; i < prime.length; i++)
       prime[i] = true;

        getPrimeNumbers(prime);

        prime[0] = false;
        prime[1] = false;


    }


    /*PRP_SPLIT*/
    if((to-from) > 5001 ){//Split the interval

        newFrom = from;
        newTo = (to-(to-from)/2);


        /* fanout = 2 */
        for(int i = 0; i < 2; i++){

       if(i > 0){
           newFrom = newTo+1;
           newTo = to;
       }

       /*PRP_CALL*/
       childCheck[i] = checkEvenNumbers(newFrom,newTo);

        }

        return childCheck[0]+childCheck[1];

    }
    else{

        //the sum is small enough. check every even number in the interval
        if((from % 2) != 0)
       from = from + 1;


        for(int n = from; n < to; n+=2){
       for(int i = 3; i <= n/2; i+=2){

           if(prime[i] && prime[n-i]){
          sumGB++;
           }
       }
        }
        return sumGB;
    }
     }
}
```

Tillegg B

TravelingSalesman.java

```
import java.util.*;

/**
 * Travelling Salesman for testing JavaPRP
 * by Bjørn Arild Kristiansen
 * Corrected - expanded by Arne Maus
 * ---
 * USAGE: javac Salesman <city data file>
 *
 * inludes:
 * -greedy cutoff
 *
 */

public class Salesman2 {


    public static int[][] cities =

     {{0, 16, 16, 10, 17, 16, 15, 17, 13, 14, 7, 3, 15, 5, 5, 9, 15, 10 },
      {16,  0,  4, 17, 12, 17, 13, 12, 17, 17, 1, 7, 16, 2, 2, 12, 4, 1 },
      {16,  4,  0,  11, 16, 5, 2, 1, 1, 14, 14, 14, 2, 11, 18, 3, 7, 2, },
      {10, 17, 11,   0, 18, 18, 11, 1, 8, 14, 5, 16, 15, 9, 16, 1, 3, 10 },
      {17, 12, 16,   18, 0, 5, 3, 7, 14, 15, 5, 5, 5, 14, 5, 11, 9, 8 },
      {16, 17,  5, 18, 5, 0, 14, 8, 17, 11, 2, 8, 1, 1, 4, 10, 14, 16 },
      {15, 13,  2, 11, 3, 14, 0, 6, 2, 14, 7, 4, 2, 6, 15, 2, 4, 15 },
      {17, 12,  1, 1, 7, 8, 6, 0, 15, 13, 15, 4, 4, 9, 7, 9, 1, 8 },
      {13, 17,  1, 8, 14, 17, 2, 15, 0, 6, 8, 17, 1, 12, 2, 6, 6, 16 },
      {14, 17, 14, 14, 15, 11, 14, 13, 6, 0, 3, 5, 15, 13, 2, 2, 11, 12 },
      {7,   1, 14, 5, 5, 2, 7, 15, 8, 3, 0, 17, 7, 16, 5, 12, 8, 17 },
      {3,   7, 14, 16, 5, 8, 4, 4, 17, 5, 17, 0, 1, 12, 9, 18, 13, 3 },
      {15, 16,  2, 15, 5, 1, 2, 4, 1, 15, 7, 1, 0, 10, 17, 13, 13, 7 },
      {5,   2, 11, 9, 14, 1, 6, 9, 12, 13, 16, 12, 10, 0, 9, 15, 16, 16 },
      {5,   2, 18, 16, 5, 4, 15, 7, 2, 2, 5, 9, 17, 9, 0, 12, 7, 6 },
      {9,  12,  3, 1, 11, 10, 2, 9, 6, 2, 12, 18, 13, 15, 12, 0, 6, 3 },
      {15,  4,  7, 3, 9, 14, 4, 1, 6, 11, 8, 13, 13, 16, 7, 6, 0, 12 },
      {10,  1,  2, 10, 8, 16, 15, 8, 16, 12, 17, 3, 7, 16, 6, 3, 12, 0 }};

    public static int noOfCities = 18;

    /*
    public static int[][] cities =
    {{0, 33, 101, 130, 176, 110,  82,  15,  25,  63,  25, 101, 62, 78,  88, 44, 113},
     {33, 0, 81, 120, 183, 115, 95, 39, 58, 88, 58, 103, 79, 72,         201,99, 100},
     {101, 81, 0, 48, 139, 79, 89, 94, 122, 164, 120, 67, 92, 40,        3,156,92    },
     {130, 120, 48, 0, 96, 54, 82, 119, 145, 191, 142, 47, 95, 51,       67,133,169  },
     {176, 183, 139, 96, 0, 68, 95, 161, 178, 224, 174, 81, 116, 116,    39,101,200  },
     {110, 115, 79, 54, 68, 0, 36, 96, 117, 164, 112, 13, 56, 48,        91,14,173   },
     {82, 95, 89, 82, 95, 36, 0, 67, 83, 130, 79, 35, 21, 49,            108,35,93   },
     {15, 39, 94, 119, 161, 96, 67, 0, 28, 72, 25, 87, 47, 67,           241,123,10  },
     {25, 58, 122, 145, 178, 117, 83, 28, 0, 47, 4, 110, 62, 94,         50,70,120   },
```

```
 {63, 88, 164, 191, 224, 164, 130, 72, 47, 0, 51, 157, 109, 140,    89,153,209 },
 {25, 58, 120, 142, 174, 112, 79, 25, 4, 51, 0, 106, 57, 90,        117,166,66},
 {101, 103, 67, 47, 81, 13, 35, 87, 110, 157, 106, 0, 52, 35,       199,213,56},
 {62, 79, 92, 95, 116, 56, 21, 47, 62, 109, 57, 52, 0, 53,          100,200,75},
 {78, 72, 40, 51, 116, 48, 49, 67, 94, 140, 90, 35, 53, 0,          45,75,93},
 {88,201,3,67,39,91,108,241,50,89,117,199,100,45,     0,67,104},
 {44,99,156,133,101,14,35,123,70,153,166,213,200,75,  67,0,82},
 {113,100,92,169,200,173,93,10,120,209,66,56,75,93,   104,82,0}};
public static int noOfCities = 17;
*/


/*
 * Greedy algorithm that returns a cutoff value for main method
 * by always taking the shortest path to next city.
 */
public static int getCutoff() {
    int cutoff = 0;

    boolean[] beenHere = new boolean[noOfCities];

    /* Starting from city 0 */
    int currentCity = 0;
    int step = 0;


    /* Traverse cities */
    while(step < noOfCities) {

        /* Marking current city as "visited" */
        beenHere[currentCity] = true;

        int shortestYet = Integer.MAX_VALUE; //2^31-1
        int nextCity = -1;

        /* Finding next destination (w/ shortest distance from current) */
        for (int i = 1; i < noOfCities; i++) {
            if (!beenHere[i] && cities[currentCity][i] < shortestYet) {
                shortestYet = cities[currentCity][i];
                nextCity = i;
            }
        }

        /* Been everywhere, returning to city 0 */
        if (nextCity < 0) {
            shortestYet = cities[currentCity][0];
            nextCity = 0;
        }

        cutoff += shortestYet;


        /* Travelling... */
        step++;
        currentCity = nextCity;
    }

    System.out.println("Cutoff set to " + cutoff);

    return cutoff;
}



/*
 * Additional cutoff. Principle find shortest arcs, sum thenm
 *  to create may be not connected: shortest two-arks, three arcs,four arcs  ..
 */
public static int[] getFakeShortest() {
   int cutoffs[] = new int[noOfCities];
   // Copy cities
   int [] ccopy = new int [(noOfCities)*(noOfCities-1)/2];
   int shortestYet;

   int rowlen = noOfCities-1, sum=0;
   for (int i =0 ; i < noOfCities; i++) {
      rowlen --;
      for (int j =i+1; j<noOfCities;j++)
         ccopy[sum + j -1] = cities[i][j];
      sum += rowlen;
   }
```

```
    Arrays.sort(ccopy);

    for (int i = 1; i< noOfCities; i++)
       cutoffs[i]= ccopy[i-1] + cutoffs[i-1];

    return cutoffs;
} // end getFakeShortest



/*
 * Recursive algorithm for finding shortest path
 * between cities
 */

 /*PRP_PROC*/
 public static int travel(int city, boolean[] beenHere,int currentDistance, int cutoff,int[]
       fakeShortest, int citiesLeft) {
     //Initial call: travel(0, visited, 0, cutoff, fakeShortest, noOfCities - 1);
     int shortestPath = Integer.MAX_VALUE; //2^31-1
     boolean leafNode = true; /* Remains true if all cities are  visited */

     /* Marks current city as visited */
     beenHere[city] = true;

     int [] lastPath = new int [noOfCities];


     /* Visit all cities that has not yet been visited */
     for (int i = 0; i < noOfCities; i++) {
         if (!beenHere[i]) {
             leafNode = false;

             /*PRP_SPLIT*/
             if (currentDistance + cities[city][i] /*+ fakeShortest[citiesLeft]*/ <= cutoff) {

                 if (city == 0) {
                     System.out.println("Launch in direction " + i);
                 }


                 /*PRP_CALL*/
       lastPath[i] = travel(i, beenHere, currentDistance + cities[city][i], cutoff, fakeShortest,
             citiesLeft-1);

       //-----------------
                 lastPath[i] += cities[city][i];


                 if (lastPath[i] < shortestPath)
                     shortestPath = lastPath[i];

             }
             else {
                  shortestPath = cutoff + 1;
             }

          // ---------------------------

             if (city == 0) {
                 // arrived at city 0 after full roundtrip
                 if (shortestPath < cutoff){
      cutoff = shortestPath;
       }

                 System.out.println("Result for " + i + ": " + lastPath[i]);
                 System.out.println();
             }

             beenHere[i] = false;
        } // end ! been here
     } // end for i


     if (!leafNode) {
         return shortestPath;
     }
```

```
        /* Leaf node will return to starting city */
        return cities[city][0];

    }


    public static void main(String[] args) {
        //return summen av kortestevei i hver by
        int cutoff = getCutoff();
        boolean[] visited = new boolean[noOfCities];


        /*
         * return [] kortestevei i hver by.
         * ex
         * [0]=12;
         * [1]=[0](12)+46;
         * osv
         */
        int[] fakeShortest = getFakeShortest();


   long time = System.currentTimeMillis();

        /* Find shortest path */
        int totalLength = travel(0, visited, 0, cutoff, fakeShortest, noOfCities - 1);
   time = System.currentTimeMillis() - time;

        System.out.println("Total length: " + totalLength+ ", time used: "+ time + " millisec.");


    }
}
```

Tillegg C

Forbedret Travelling Salesman

```
import java.util.*;

/**
 * Travelling Salesman for testing JavaPRP
 * by Bjørn Arild Kristiansen
 * Corrected - expanded by Arne Maus - new ver 2007 with syntetic data
 * ---
 * USAGE: javac Salesman  noOfcities <city data file>
 *
 * inludes:
 * -greedy cutoff
 *
 */

public class Salesman6{
  final static boolean disableCut = false;   // use cutoff (false)
  static long numCall =0;
  final static int reportFreq = 1000000; // 1 M

  static int depthCall=0;
  static int cutoff =Integer.MAX_VALUE ;
  static long [] depthStat= new long[100];
  static long [] cutLevel = new long[100];
  static long numCut;
  public static int [] solution = new int[100];
  public static int [] tried    = new int[100];
  public static int [][] fakeShortestNSteps;
  //Total length: 37, time used: 1984 millisec.
  // Path:0, 11, 12, 8, 6, 4, 14, 9, 15, 3, 16, 7, 2, 17, 1, 10, 5, 13, 0,
  public static int[][] cities =

  /*
    {{  0, 16, 16, 10, 17, 16, 15, 17, 13, 14,  7,  3, 15,  5,  5,  9, 15, 10,},
    { 16,  0,  4, 17, 12, 17, 13, 12, 17, 17,  1,  7, 16,  2,  2, 12,  4,  1,},
    { 16,  4,  0, 11, 16,  5,  2,  1,  1, 14, 14, 14,  2, 11, 18,  3,  7,  2,},
    { 10, 17, 11,  0, 18, 18, 11,  1,  8, 14,  5, 16, 15,  9, 16,  1,  3, 10,},
    { 17, 12, 16, 18,  0,  5,  3,  7, 14, 15,  5,  5,  5, 14,  5, 11,  9,  8,},
    { 16, 17,  5, 18,  5,  0, 14,  8, 17, 11,  2,  8,  1,  1,  4, 10, 14, 16,},
    { 15, 13,  2, 11,  3, 14,  0,  6,  2, 14,  7,  4,  2,  6, 15,  2,  4, 15,},
    { 17, 12,  1,  1,  7,  8,  6,  0, 15, 13, 15,  4,  4,  9,  7,  9,  1,  8,},
    { 13, 17,  1,  8, 14, 17,  2, 15,  0,  6,  8, 17,  1, 12,  2,  6,  6, 16,},
    { 14, 17, 14, 14, 15, 11, 14, 13,  6,  0,  3,  5, 15, 13,  2,  2, 11, 12,},
    {  7,  1, 14,  5,  5,  2,  7, 15,  8,  3,  0, 17,  7, 16,  5, 12,  8, 17,},
    {  3,  7, 14, 16,  5,  8,  4,  4, 17,  5, 17,  0,  1, 12,  9, 18, 13,  3,},
    { 15, 16,  2, 15,  5,  1,  2,  4,  1, 15,  7,  1,  0, 10, 17, 13, 13,  7,},
    {  5,  2, 11,  9, 14,  1,  6,  9, 12, 13, 16, 12, 10,  0,  9, 15, 16, 16,},
    {  5,  2, 18, 16,  5,  4, 15,  7,  2,  2,  5,  9, 17,  9,  0, 12,  7,  6,},
    {  9, 12,  3,  1, 11, 10,  2,  9,  6,  2, 12, 18, 13, 15, 12,  0,  6,  3,},
    { 15,  4,  7,  3,  9, 14,  4,  1,  6, 11,  8, 13, 13, 16,  7,  6,  0, 12,},
    { 10,  1,  2, 10,  8, 16, 15,  8, 16, 12, 17,  3,  7, 16,  6,  3, 12,  0,}};
```

```
  public static int noOfCities = 18;
*/

// Total length: 67, time used: 1407079 millisec.
//  Path:0, 3, 6, 18, 11, 4, 2, 12, 19, 7, 15, 20, 17, 1, 14, 16, 10, 9, 5, 8, 13, 0

  {{  0, 14,  2,  1,  3, 21, 15, 18,  1, 15, 10,  1, 11,  4, 17, 15,  3,  9,  9, 13, 16,},
   { 14,  0, 21, 17, 21, 17, 19, 10, 13, 12, 18, 20, 13, 17,  1, 15, 17,  9, 12, 14, 11,},
   {  2, 21,  0,  8,  3, 12, 20,  7, 18, 12, 17, 18,  2, 20,  5, 21,  1,  9,  2,  2, 16,},
   {  1, 17,  8,  0,  9, 13,  2, 21, 16, 12, 13,  2, 21, 14, 13,  7,  7,  4, 17,  8, 14,},
   {  3, 21,  3,  9,  0, 18, 17, 10, 18, 10, 17,  2,  8, 10, 14, 12,  8, 14,  9, 10, 19,},
   { 21, 17, 12, 13, 18,  0, 10, 10,  5,  5, 15, 14,  6, 17,  5, 13, 10, 19,  9, 10, 16,},
   { 15, 19, 20,  2, 17, 10,  0, 18, 10, 15, 20, 15, 17, 13, 17, 11, 12, 10,  4, 17,  4,},
   { 18, 10,  7, 21, 10, 10, 18,  0, 20, 21, 12, 12, 18, 10,  8,  4, 12,  4, 11,  4,  6,},
   {  1, 13, 18, 16, 18,  5, 10, 20,  0, 18,  3, 12, 18,  4, 19, 12, 11, 10, 14,  7, 10,},
   { 15, 12, 12, 12, 10,  5, 15, 21, 18,  0,  3, 16,  8, 13,  6, 10, 13, 18,  9,  8, 18,},
   { 10, 18, 17, 13, 17, 15, 20, 12,  3,  3,  0, 20, 13, 14, 17, 14,  2, 10, 16, 20, 18,},
   {  1, 20, 18,  2,  2, 14, 15, 12, 12, 16, 20,  0,  4,  8, 16,  1, 14,  2,  2, 12, 10,},
   { 11, 13,  2, 21,  8,  6, 17, 18, 18,  8, 13,  4,  0, 18,  7,  9, 19,  9,  7,  5, 17,},
   {  4, 17, 20, 14, 10, 17, 13, 10,  4, 13, 14,  8, 18,  0,  8, 18, 18, 11, 15, 10, 16,},
   { 17,  1,  5, 13, 14,  5, 17,  8, 19,  6, 17, 16,  7,  8,  0,  1,  1, 11,  5,  8,  4,},
   { 15, 15, 21,  7, 12, 13, 11,  4, 12, 10, 14,  1,  9, 18,  1,  0, 19,  1, 16,  6,  1,},
   {  3, 17,  1,  7,  8, 10, 12, 12, 11, 13,  2, 14, 19, 18,  1, 19,  0, 14,  5, 17, 16,},
   {  9,  9,  9,  4, 14, 19, 10,  4, 10, 18, 10,  2,  9, 11, 11,  1, 14,  0,  8, 18,  3,},
   {  9, 12,  2, 17,  9,  9,  4, 11, 14,  9, 16,  2,  7, 15,  5, 16,  5,  8,  0, 18, 19,},
   { 13, 14,  2,  8, 10, 10, 17,  4,  7,  8, 20, 12,  5, 10,  8,  6, 17, 18, 18,  0,  9,},
   { 16, 11, 16, 14, 19, 16,  4,  6, 10, 18, 18, 10, 17, 16,  4,  1, 16,  3, 19,  9,  0,}};

public static int noOfCities = 21;


/*
 * Greedy algorithm that returns a cutoff value for main method
 * by always taking the shortest path to next city.
 */
public static int getCutoff() {
  int cutoff = 0;

  boolean[] beenHere = new boolean[noOfCities];

  /* Starting from city 0 */
  int currentCity = 0;
  int step = 0;
  //println(" ...(start greedy)...");

  /* Traverse cities */
  while(step < noOfCities) {

    /* Marking current city as "visited" */
    beenHere[currentCity] = true;

    int shortestYet = Integer.MAX_VALUE; //2^31-1
    int nextCity = -1;

    /* Finding next destination (w/ shortest distance from current) */
    for (int i = 1; i < noOfCities; i++) {
 if (!beenHere[i] && cities[currentCity][i] < shortestYet) {
   shortestYet = cities[currentCity][i];
   nextCity = i;
 }
    }

    solution[step] = currentCity;
    /* Been everywhere, returning to city 0 */
    if (nextCity < 0) {
 shortestYet = cities[currentCity][0];
 nextCity = 0;
    }


    cutoff += shortestYet;

    /* Travelling... */
    step++;
    currentCity = nextCity;
  }

  return cutoff;
}
```

```
/*
 * Additional cutoff. Principle find shortest arcs, sum thenm
 *  to create may be not connected: shortest two-arks, three arcs,four arcs  ..
 */
public static int[] getFakeShortest() {
  int cutoffs[] = new int[noOfCities];
  // Copy cities
  int [] ccopy = new int [(noOfCities)*(noOfCities-1)/2];
  int shortestYet;

  int rowlen = noOfCities-1, sum=0;
  for (int i =0 ; i < noOfCities; i++) {
    rowlen --;
    for (int j =i+1; j<noOfCities;j++){
 ccopy[sum + j -1] = cities[i][j];
    }
    sum += rowlen;
  }

  Arrays.sort(ccopy);

  for (int i = 1; i< noOfCities; i++)
    cutoffs[i]= ccopy[i-1] + cutoffs[i-1];

  return cutoffs;
} // end getFakeShortest

/*
 * Additional cutoff. Principle find shortest arcs, sum thenm
 *  to create may be not connected: shortest two-arks, three arcs,four arcs
 * Also checkagains fake1 (not shorter than that).
 */
public static int[] [] getFakeShortest2(int [] fake) {
  int [][]fakeShortN = new int[noOfCities][noOfCities];

  for(int i= 1;i<noOfCities;i++){
    fakeShortN[i][1]= cities[i][0]; //lengt of path of length 1 to city 0
  }
  for(int i= 2;i<noOfCities;i++){
    // for every pathlength i to city 0 in fakeShortN[city][i]
    for(int j= 1;j<noOfCities;j++) {
 // for all cities j with pathlegth i from city 0
 fakeShortN[j][i] = Integer.MAX_VALUE;
 for(int k= 1;k<noOfCities;k++){
   // test city 2,3,. for shorter pathlength
   if(k!=j && fakeShortN[j][i] > fakeShortN[k][i-1] + cities[k][j] ){
     fakeShortN[j][i] = fakeShortN[k][i-1] + cities[k][j];
   }
 }
 // but not shorter that fake[i]
 if (fake[i] >  fakeShortN[j][i]){
  fakeShortN[j][i] = fake[i] ;
 }
    } // end j
  }// end i

  return fakeShortN;
} // end getFakeShortest

/*
 * Recursive algorithm for finding shortest path
 * between cities
 */

/*PRP_PROC*/
public static int travel(int city, boolean[] beenHere, int currentDistance, int[][]
      fakeShortestNSteps, int citiesLeft, int startCutoff) {

  if(city == 0){
    System.out.println(city);
  }


  //Initial call: travel(0, visited, 0, cutoff, fakeShortest, noOfCities-1 );
  numCall++;  // statistics
  depthCall ++;
  depthStat[depthCall]++;
  if(startCutoff < cutoff) {
    cutoff = startCutoff;
  }
```

```
    int shortestPath = Integer.MAX_VALUE; //2^31-1
    int lastPath = 0;

    /* Marks current city as visited */
    beenHere[city] = true;

    /* Visit all cities that has not yet been visited */
    for (int i = 1; i < noOfCities; i++) {
      if (!beenHere[i]) {

   /*PRP_SPLIT*/
   if (disableCut | currentDistance + cities[city][i] + fakeShortestNSteps[i][citiesLeft-1] < cutoff) {
     if (city == 0) {
       System.out.println("Launch in direction " + i);
     }


     tried[depthCall] = i;

     /*PRP_CALL*/
     lastPath = travel(i, beenHere, currentDistance + cities[city][i],fakeShortestNSteps, citiesLeft-1,
           cutoff);
     if (lastPath < shortestPath){
       shortestPath = lastPath;
     }
   }
   else {
     cutLevel[depthCall]++;
     numCut++;
   }

      } // end not been here

    } // end i

    beenHere[city] = false;

    if(citiesLeft == 1) {
      // bottom, return to city 0
      shortestPath = currentDistance + cities[city][0];

      if (shortestPath < cutoff) {
   //System.out.println("\nFound new better:"+ shortestPath);
   cutoff = shortestPath;
   for (int j=0; j< noOfCities; j++){
     solution[j] = tried[j];
   }
      }
    }


    //if (numCall%reportFreq==0)
    //  System.out.print("\rnumCall:" + numCall +", depth:" + depthCall);
    depthCall--; // register dept of call;

    return shortestPath;

  } // end **travel **



  static long fak(int i) {
    if (i == 1) return 1; else return i*fak(i-1);
  }

  static void print(String s) {
    System.out.print(s);
  }

  static void println(String s) {
    System.out.println(s);
  }

  public static void main (String [] args)
  {
    cutoff = getCutoff();
    boolean[] visited = new boolean[noOfCities];

    /*
     * return [],  korteste vei av lengde 1,2,3,..
     */
```

```
    int[] fakeShortest = getFakeShortest();
    fakeShortestNSteps = getFakeShortest2(fakeShortest);

    long time = System.currentTimeMillis();

    /* Find shortest path */
    int totalLength = travel(0, visited, 0,  fakeShortestNSteps, noOfCities, cutoff);
    time = System.currentTimeMillis() - time;

    System.out.println("Total length = " + totalLength + " Time total = " + time/1000 + "sec");

  } // end main
} // end class
```