UNIVERSITETET I OSLO
Institutt for informatikk

PRP-2014: Parallelle, faseoppdelte og rekursive algoritmer

Masteroppgave

Peter Ludvik Eidsvik

27. april 2014

Sammendrag

Parallell programmering blir mer og mer viktig ettersom vi får flere prosesseringsenheter i en datamaskin. Derimot kan slik programmering være vanskelig, fordi det kreves ekstra administrativ kode for å sette i gang parallellitet. Programflyten er også vanskeligere å se for seg, når forskjellige oppgaver i programmet kjører selvstendig i parallell. Derfor er Java PRP(*Parallel recursive procedures*) laget for å være et program mellom brukerens sekvensielle program, og kompilatoren. Java PRP lager et parallelt program, basert på det sekvensielle programmet, uten at det har noe påvirkning på resultatet. Dette gjør at man kan muligens oppnå bedre kjøretider, uten at man må ta for seg den administrative koden selv.

Denne versjonen av Java PRP har sett nærmere på påvirkningen av delte variabler i parallelle programmer, og hvordan dette kan håndteres. Løsningen er at Java PRP tar for seg faseoppdelte programmer, der hver fase har en parallell del og en sekvensiell del, der vi kan håndtere delte variabler i den sekvensielle delen, og dermed få full parallellitet i den parallelle delen.

Denne masteroppgaven tar for seg tre eksempler, hvor vi sammenligner en sekvensiell versjon og Java PRP sin parallelle versjon. De to første, nemlig finn største tall i en mengde og Quicksort, er ikke kjørt i flere faser, men baserer seg på én metode. Det siste og største eksempelet er Delaunay triangulering, og er et større program i tre faser. I alle tre eksemplene får vi en speedup. Selve Java PRP programmet kan man finne på *http://folk.uio.no/peterlei*. Her finner man filen under navnet *JavaPRP.java*.

Forord

Etter flere læringsrike år på Instituttet for Informatikk ved Universitet i Oslo, avsluttes mitt studium med denne masteroppgaven. Det er flere som har bidratt til at jeg har klart å skrive denne oppgaven.

Jeg vil først og fremst takke min veileder Arne Maus, som tilbød meg en veldig interessant oppgave. Igjennom hele masterperioden har han bidratt med mange gode råd og ideer. Det har også vært flere interessante diskusjoner rundt temaet til oppgaven, som jeg har funnet nyttige og inspirerende.

Jeg vil også takke mine foreldre og min bror, for deres støtte gjennom hele min studieperiode, samtidig som de har også vist interesse for studiet.

Til slutt vil jeg takke mine venner for deres støtte, og at de har bidratt til å gjøre de siste årene morsomme og spennende.

Institutt for Informatikk
Universitetet i Oslo
Blindern, April 2014
Peter Ludvik Eidsvik

Innholdsfortegnelse

1. Innledning

1.1.1 Introduksjon

I de siste årene har en multikjernet arkitektur blitt standard i de fleste datamaskiner. Dette har ført til at parallell eksekvering av programmer har blitt mer og mer relevant ettersom vi har fått flere prosesseringsenheter tilgjengelig i en maskin. Å programmere parallelle programmer krever mer administrativ kode for å opprette nødvendige datastrukturer, og det er vanskeligere å se for seg flyten av et slikt program, da hver oppgave, som løses i parallell, kjører for seg selv.

1.1.2 Java PRP

Til tross for at parallelle programmer introduserer nye problemstillinger, har de potensialet til å være raskere enn de sekvensielle variantene. Dette gjør at programmer, som automatisk genererer parallelle programmer, basert på en sekvensiell versjon, kan forenkle prosessen. Gjennom flere masteroppgaver, av min veileder Arne Maus, har det blitt laget flere versjoner programmer basert på PRP(*Parallel recursive procedures*), som genererer parallelle programmer basert på en rekursiv metode. Denne versjonen av PRP tar for seg Java programmer på multikjernede datamaskiner, som er blitt kalt Java PRP tidligere. Jeg skulle i denne masteroppgaven se spesielt på innvirkning og håndtering av problemer, som delte variabler introduserer. Dette gjør at oppgaven handler mye om delte variabler og hvordan disse påvirker parallelle programmer, og hvordan disse kan håndteres på et generelt nivå, som i Java PRP.

Denne oppgavens løsning baserer seg på at Java PRP kan parallellisere faseoppdelte programmer, slik at vi parallelliserer i flere faser. Hver fase er todelt. Den første delen vil inneholde en rekursjonsmetode, som parallelliseres. Dette etterfølges av en sekvensiell del, der brukeren har mulighet til å kjøre en del av koden sekvensielt. Her kan man, for eksempel, håndtere delte variabler.

1.1.3 Innholdet i oppgaven

Her vil vi se kort på innholdet i hvert av kapitlene i denne masteroppgaven. I kapittel 2 ser vi på generelt om parallellitet og bakgrunnen til at vi har begynt med parallell programmering på dagens datamaskiner, som baserer seg på en multikjernet arkitektur. Vi ser på utviklingen av antall transistorer på en brikke, og hva som har skjedd med maskinvaren for at vi har begynt med flere kjerner. Til slutt ser vi på hvordan minnehierarkiet fungerer i en multikjernet maskin.

Kapittel 3 går dypere i hvordan vi kan bruke Java til parallell programmering. Her gjennomgås det hvordan vi lager, starter og håndterer tråder i Java. Vi ser også på hvordan vi kan gjøre tidsmålinger i Java, som skaper grunnlaget for kjøretidstester senere i oppgaven.

Vi begynner i kapittel 4 med å se på bakgrunnen til Java PRP. Her nevnes det hvorfor Java PRP baserer seg på rekursjon, spesielt dens fordeler når det gjelder parallellitet. Dette fører oss videre til implementasjonen av denne oppgavens Java PRP for programmer med én rekursjonsmetode, og hvordan strukturen er i forhold til klasser og arbeidsoppgaver.

Kapittel 5 omhandler delte variabler. Her er det generelt om variabler i Java og hvordan de delte variablene passer inn i dette. Det vil bli diskutert rundt alternativer rundt håndteringen av dette i forhold til et generelt nivå, som Java PRP skal klare. Det er her et eksempel, som viser at en god og en dårlig måte å synkronisere rundt delte variabler.

På grunn av at Java PRP baserer seg på rekursive metoder, vil vi i kapittel 6 se nærmere på parametre og parameteroverføring i Java. Parametre kan komme i flere former, basert på hva type variabler de er. Spesielt er det forskjell om parametrene er primitive typer eller objektspekere. Siden parametre er kopier av de originale verdiene, kan vi bruke dette til å håndtere de delte variablene som kopier via parametrene.

Kapittel 7 begynner med å se på hva type rekursjonsmetoder, som Java PRP skal kunne takle i forhold til delte variabler. Fra ideen om at tråder arbeider lokalt i parallellitet og vi samler lokale resultater til ett samlet resultat etterpå, går vi over til faseoppdelte programmer. Hver fase vil bestå av en rekursjonsmetode, som kjører i parallellitet, etterfulgt av en sekvensiell del hvor vi kan eventuelt ordne delte variabler. Et program kan bestå av flere faser, og fasene vil bli satt i gang av en administratormetode, som bestemmer flyten av programmet. I kapittel 8 viser utvidelsen av Java PRP til å håndtere faseoppdelte programmer. Det viktigste for brukere av Java PRP er at de må lage en administratormetode og legge til noen nye nøkkelord.

Det første eksempelet med Java PRP er i kapittel 9 og tar for seg oppgaven med å finne største tall i en mengde. Dette er et eksempel på et program med én rekursjonsmetode. Vi ser her at vi får en speedup ved å bruke det parallelle programmet generert av Java PRP.

Neste eksempel, i kapittel 10, tar for seg Quicksort. Dette er også et eksempel med bare én rekursjonsmetode. Her får vi en speedup, men vi ser også på effekten med flere tråder i det parallelle programmet, nemlig lastbalanse, og merker at dette gir en enda bedre speedup.

Det største eksempelet kommer i kapittel 11 og er Delaunay triangulering. Dette er et eksempel på et program med tre faser. Den første fasen er at vi finner den konvekse innhyllingen, den andre er at vi finner de k nærmeste nabopunktene per punkt og til slutt har vi standard søk for å finne resten av trianguleringen. I dette eksempelet får vi også en speedup.

I kapittel 12 ser vi på de delte variablene i Delaunay trianguleringsprogrammet, og hvordan de forskjellige fasene håndterer dem. Det viktigste er at vi unngår synkroniseringsmetoder.

Den andre fasen i Delaunay triangulering, nemlig finne de k nærmeste naboene, kan utnyttes på flere måter. I kapittel 13 ser vi nærmere på hvordan den sekvensielle versjonen kan være bedre ved at vi finner en god måte å finne nabopunktene i fase to. Vi ser også på hvorfor fase to søker etter de 10 nærmeste naboene, og prosentmessig hvor mye av den totale trianguleringen dette gir oss.

Til slutt, i kapittel 14, har vi en konklusjon og forslag til videre arbeid, både andre retninger for Java PRP, og utvidelser av denne versjonen. Deretter følger det to vedlegg, nemlig brukermanualen til Java PRP, og kodeeksemplene for å finne største tall av en mengde og Quicksort, både de sekvensielle versjonene, annotert for Java PRP, og de parallelle versjonene, generert av Java PRP.

2. Bakgrunn for parallellitet i datamaskiner

2.1 Parallellitet

2.1.1 Hva er parallellitet

Parallellitet vil si at vi kan gjøre flere operasjoner i samme tidsrom. I en datamaskin kan dette bety at vi kan gjøre flere kalkulasjoner samtidig. Ofte er en løsning på et større problem å gjøre det til mindre problemer, som egner seg til å eksekveres i parallell. Når de små problemene er løst akkumuleres de til et nytt svar som gir oss en løsning på det større problemet.

2.1.2 Speedup

Det viktigste med å utnytte parallellitet i en datamaskin er at man kan bedre kjøretider ved å gjøre mange av beregningene samtidig, enn å gjøre alt sekvensielt. Man vil altså ha en *speedup* i kjøringen. Speedup *S* kan man definere som:

$$S = \frac{tiden\ det\ tar\ å\ kjøre\ en\ algoritme\ sekvensielt}{tiden\ det\ tar\ å\ kjøre\ algoritmen\ parallelt}$$

Et eksempel på speedup er som følgende. Dersom vi har et program, som tar 12 sekunder på én kjerne og 3 sekunder på en multikjernet maskin med fire kjerner, da er speedup: 12/3 = 4. Samme som antall prosessorer vi har på maskinen.

Ifølge Gregory R. Andrews [4] har man tre typer for speedup:

- lineær speedup, som er tilfellet hvor speedup er lik antall prosessorer. Beregningen ovenfor er et eksempel på dette.
- Sublineær speedup, som er slik at speedup er mindre enn antall prosessorer. Dette kan være tilfelle hvor en del av koden må kjøres sekvensielt.
- Superlineær speedup er slik at speedup er mer enn antall prosessorer tilgjengelig.

Fra et praktisk perspektiv vil det alltid være en del sekvensielt som må kjøres i et program. I begynnelsen av et program må det alltids være en sekvensiell oppstart av tråder. Først etter dette kan vi gjøre oppgaver i parallell.

2.1.3 Overhead med å starte tråder

For å bruke parallellitet i Java, må vi sette i gang tråder for å utføre de forskjellige oppgavene. Det medfører en ekstra tidskostnad ved å starte tråder, slik at vi kan si at det følger en overhead ved å utnytte parallellitet i et program.

I en annen masteroppgave[7] av Petter A. Busterud ble det gjort tidsmålinger på hvor lang tid forskjellige parallellitetsmekanismer tok i Java. Tidsmålingene baserte seg på en Intel Pentium M 760 med 2GHz single core, med 2 GB DDR2 RAM med en klokkefrekvens på 533 MHz der det ble gjort tester på en Windows 7 32-bit og Linux Mint 13 32-bit. På Windows, fikk han i overkant av 1500 mikrosekunder (µs) for å starte én tråd samt å sette denne tråden i gang. De påfølgende trådene, samt å sette dem i gang, tok rundt 200 µs. På linux fikk han bedre tider, der det tok rundt 700 µs for å starte og sette i gang én tråd og i overkant av 100 µs for de neste trådene.

Dette viser at de programmene vi ønsker å parallellisere må være store nok, slik at den faktiske, parallelle utførelsen veier opp for denne overheaden. Dersom det sekvensielle programmet klarer å gjøre oppgaven før det parallelle har startet opp trådene sine, vil vi ikke få en speedup. Tidene viser også at det er tidsmessige forskjeller på overheaden basert på hvilket operativsystem man bruker. De vil begge håndtere tråder og trådgenerering på forskjellige måter. I denne oppgaven vil det bli brukt Windows 7 64-bit, da dette var mest tilgjengelig.

2.1.4 Amdahls og Gustafsons lover

Amdahls lov[4] er en enkel, men viktig, observasjon når det gjelder maksimal speedup til en kode. Denne loven tar for seg at ikke all kode kan kjøres i parallell. I en maskin med flere kjerner kan denne koden ikke optimaliseres mer enn den faktiske tiden det tar å kjøre den sekvensielle kodedelen. For eksempel, hvis en sekvensiell gjennomkjøring av en kode tar 10 sekunder, og 2 av de sekundene må kjøres sekvensielt, vil en parallell kjøring ikke kunne ta mindre tid enn 2 sekunder. Dette er et eksempel hvor vi kan få sublineær speedup. Dette kommer av at uansett hvor mange kjerner vi har tilgjengelig, vil det være deler av programmene hvor alle disse ikke kan bli utnyttet. For Amdahls lov vil en speedup *S2* se slik ut:

$$S2 = \frac{1}{sekvensiell\ del + \frac{parallell\ del}{speedup\ for\ parallell\ del}}$$

Her vil den sekvensielle delen og den parallelle delen utgjøre det totale programmet, altså 1. Så hvis vi kan parallellisere 80 % av programmet vil den sekvensielle delen være 0,2 mens den parallelle vil være 0,8.

John L. Gustafson[13] mente at det var visse mangler med Amdals lov, nemlig at den ikke tok for seg varierende størrelser på programmenes problem og hvor mange *n* prosessorer vi kjører den parallelle delen av programmet. Med størrelsen av problemet menes det, for eksempel, om vi vil sortere 100 eller 10 000 tall. For Gustafson måtte man inkludere disse faktorene dersom man skulle riktig beregne speedup. For større

problemer samt flere, kraftigere prosessorer kan man oppnå bedre speedup fra et praktisk perspektiv, enn hva Amdahls lov tilsier. Den parallelle delen av programmet vil bli en større andel av programmet, ettersom vi øker problemet. For Gustafsons lov vil speedup *S3*, der P antall kjerner vi har tilgjengelig, mens α er andel sekvensiell kode, være slik:

$$S3(P) = P - \alpha(P - 1)$$

2.2 Prosessorens utvikling

2.2.1 Moores lov

I 1965 så Gordon Moore[5], en av grunnleggerne av Intel Corporation, en utvikling i prosesseringskraften til datamaskiner i forhold til antallet transistorer som er på en brikke. Dette la grunnlaget til det vi i dag kjenner som *Moores lov*. På den tiden antok man at det ville være en fordobling i antall transistorer per brikke hvert år i de 10 neste årene. Denne ideen viste seg å vare lengre enn 10 år, og har blitt justert i etterkant. Det var heller ikke en fordobling hvert år, men det viser seg å være nærmere hver 24. måned, ifølge Tabell 3.

I utviklingen av prosessoren og ytelsen, fant man ut at man måtte gjøre andre grep enn nødvendigvis bare øke antallet av transistorer vi kan ha på en brikke. Hyperthreading er et av disse grepene og dette ga oss muligheten til å dele opp en prosessor i flere logiske enheter. Dette førte til at selv om vi har en fysisk prosessor, vil operativsystemet registrere det som flere. Dette har flere fordeler, spesielt i tiden det tar å bytte mellom tråder. En hyperthreaded prosessor kan bytte tråd fra en klokkesykel til en annen, mens en normal prosessor kan trenger opp til 20 000 sykler på å skifte mellom tråder[6]. Siden det faktisk bare er én fysisk kjerne vil vi ikke få 100 % parallellitet, men vi kan bytte fort mellom tråder. Det som derimot kunne brukes for 100 % parallellitet var å legge inn flere fysiske kjerner, slik som i en multikjernet arkitektur.

Det som kan være av interesse er å se hva slags faktisk utvikling vi har hatt i prosessorkraft, spesielt i forhold til hvordan transistorantallet per brikke har vokst. Vi tar utgangspunkt i Intel og AMD for å se denne utviklingen.

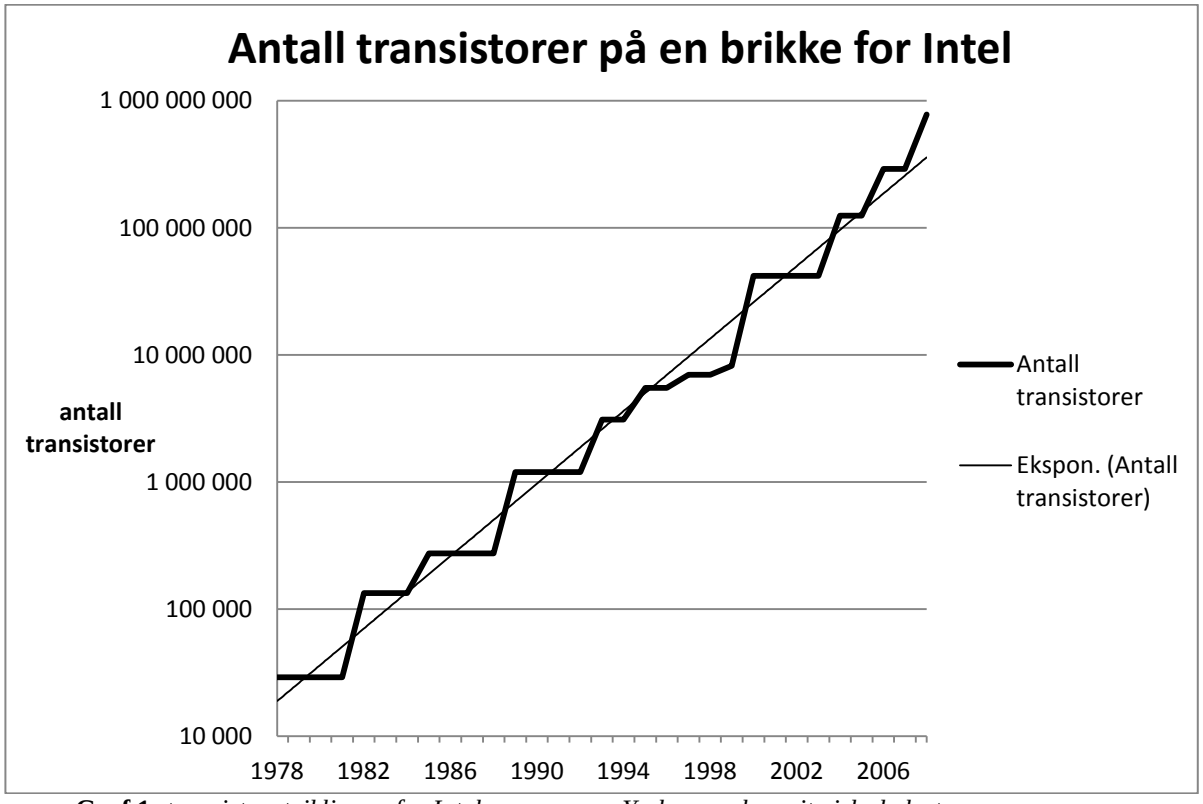
2.2.2 transistorutviklingen for Intel

For Intel starter vi fra 1978[5] med 16-bitsprosessoren Intel 8086 helt til Intel Core i7 i 2008, som inkluderte hyperthreading og en flerkjernet arkitektur.

År	Navn	Transistorer(i millioner)
1978	8086	0,03
1982	80286	0,1
1985	i386	0,3
1989	i486	1
1993	Pentium	3
1995	PentiumPro	6
1997	Pentium II	7
1999	Pentium III	8
2000	Pentium 4	42
2004	Pentium 4E	125
2006	Core 2	291
2008	Core i7	781

Tabell 1: *transistorutvikling for Intel prosessorer*

Vi kan bedre illustrere denne utviklingen ved å sette verdiene i en graf. I den neste grafen ser vi hvordan Intel sine prosessorer har økt dramatisk. Den tykke linjen indikerer de faktiske økningene i antall transistorer på en brikke, mens den smallere gir oss en trendlinje av dette.



Graf 1: *transistorutviklingen for Intel prosessorer. Y-aksen er logaritmisk skalert.*

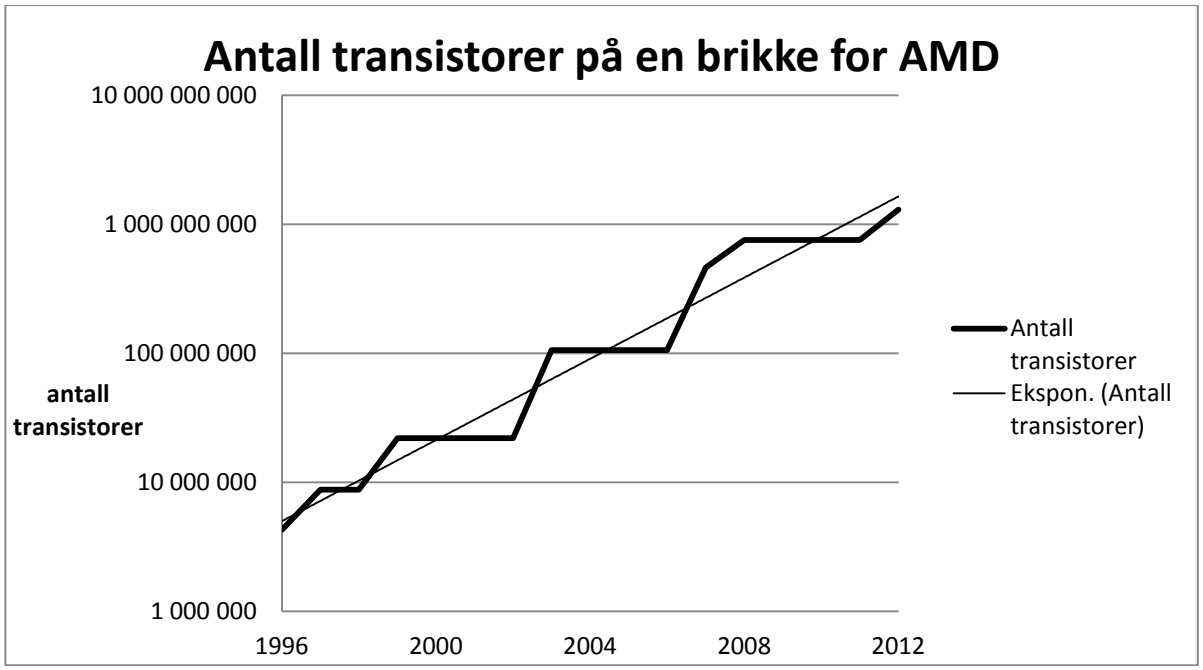
2.2.3 transistorutvikling for AMD

For utviklingen til AMD[21] starter vi fra den første prosessoren kun laget av AMD, nemlig AMD K5 i 1996 til AMD Trinity i 2012:

År	Navn	Transistorer (i millioner)
1996	AMD K5	4
1997	AMD K6	9
1999	AMD K6-III	21
1999	AMD K7	22
2003	AMD K8	106
2007	AMD K10 quad-core 2M L3	463
2008	AMD K10 quad-core 6M L3	758
2012	8-Core AMD Bulldozer	1200
2012	Quad-Core + GPU AMD Trinity	1303

Tabell 2: *Transistorutvikling for AMD prosessorer*

Hvis vi lager samme type graf, som ble gjort for Intel, får vi følgende.



Graf 2: *transistorutviklingen for AMD prosessorer. Y-aksen er logaritmisk skalert*

Dersom vi sammenligner funksjonene til Intel og AMD får vi følgende.

$$Intel = 13\,616e^{0,3283x}$$

$$AMD = 3\,488\,015e^{0.3622x}$$

Vi kan se at eksponenten til begge er ganske like, der Intel har 0,3283 mens AMD har 0,3622. Her starter x fra 0 og ikke årstallet. Tar vi utgangspunkt i disse dataene vil vi få en fordobling av antall transistorer på en brikke, for hver av produsentene, slik:

Hvor lang tid det tar før fordobling	
Produsent	**Antall år**
Intel	2,1
AMD	1,9

Tabell 3:*hvor mange år det tar for Intel og AMD for å fordoble antall transistorer på en brikke.*

2.2.4 Multikjernet datamaskinsarkitektur

En multikjernet arkitektur vil si at vi har flere prosesseringsenheter som arbeider uavhengig av hverandre. Dette gjør at parallellitet blir en praktisk idé på én datamaskin, med flere kjerner. I ideen kan man få dobbel hastighet for to kjerner, fire ganger så bra hastighet for fire kjerner også videre. Dette kan vi si fordi alle kjerner prosesserer samtidig, som vil føre til at når én er ferdig vil alle være ferdig. Derfor vil vi få en hastighet, som blir bedre basert på antall kjerner vi har tilgjengelig. Dette er selvfølgelig bare i ideen, og i realiteten støter vi på flere problemer som gjør at vi nødvendigvis ikke denne hastighetsøkningen vi ønsker, som vi så ved Amdahls lov i kapittel 2.1.4. Av prosessorene i Tabell 1, begynte Intel med hyperthreading med Pentium 4E, og er i dag en mye brukt teknologi som brukes i de fleste CPU arkitekturer. Min egen Intel Core I7 870 prosessor har fire fysiske kjerner, som er hyperthreaded, slik at de fungerer som åtte.

2.2.5 Varmeproblemer

Det er flere grunner til at utviklingen av prosessorer har rettet seg mot en multikjernet arkitektur istedenfor kun én raskere kjerne, og en av de største er varmeproblemet. Man har kommet til et punkt hvor flere transistorer vil føre til mer varme enn vi klarer å kjøle ned. Når man tilfører strøm til en prosessor, vil den frigjøre denne elektriske energien i form av varme. Det vil bli mye varmere ettersom vi legger til flere transistorer. Jo mer strøm en prosessor trenger, desto mer varme vil den gi fra seg. Formelen for hvor mye energi en prosessor bruker er som følgende[8]:

$$P = CV^2 \times f$$

Her er *P* den brukte energien, *C* er brikkens kapasitet, *V* er driftsspenningen og *f* er klokkefrekvensen på prosessoren. Dette er et problem man ikke kan ignorere, slik at man har kommet på dette konseptet om å legge inn flere kjerner enn kun én med enda flere transistorer. Dermed vil vi ikke nå høyere temperaturer. Framover i tid er håpet at vi finner opp en ny type teknologi eller type materialer som kan løse varmeproblemet vårt, slik at vi kan få enda raske kjerner. På "International Solid-State Circuit Conference" i 2001 sa Intels visepresident Patrick Gelsinger dette om varmeproblemet og tiden fremover [18]:

> *"If scaling continues at present pace, by 2005, high speed processors would have power density of nuclear reactor, by 2010, a rocket nozzle, and by 2015, surface of sun."*

2.3 Minnehierarki

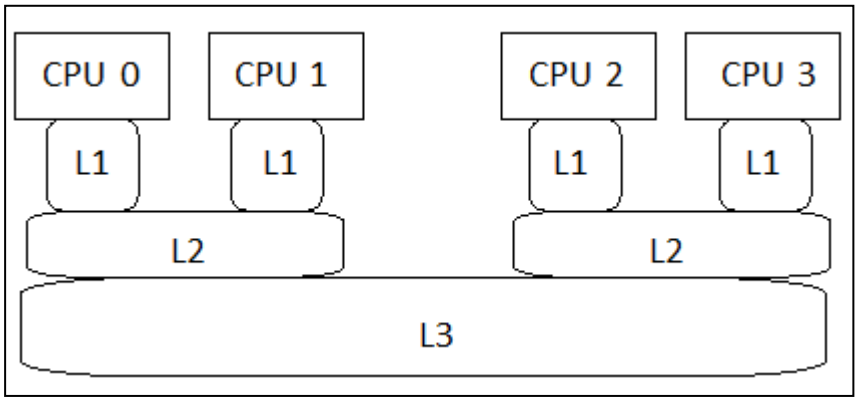
2.3.1 Multikjernet datamaskin

I en multikjernet datamaskin er det flere kjerner, som deler et felles minnehierarki. Ettersom flere kjerner gjør mye av sine lagringer på samme sted, må det unngås at de overskriver samme data. Dette kan kreve mye frem og tilbake med oppdateringer av data, synkronisering mellom flere eksekveringer og så videre. I en prosessor med én kjerne vil minnehierarkiet se slikt ut:

1. CPU registre
2. L1-L3 cacher
3. Hovedminne
4. harddisk

Forskjellen mellom disse ligger i prisen per byte og plassen. Jo høyere opp man kommer i minnehierarkiet desto raskere kan man aksessere data, men det er mindre lagringsplass. På de lavere nivåene er det mer plass, men det tar lang tid og aksessere data. Hastigheten mellom nivåene, spesielt med tanke på harddisk, er betydelige og man burde unngå å gå ett nivå ned hvis det ikke er helt nødvendig.

I en multikjernet arkitektur vil de tre forskjellige cachenivåene bli utnyttet på følgende måte. L1 cachen er en prosessors egen plass for lagring, mens L3 vil kunne deles mellom flere. Dette vil også L2, men i mindre grad enn L3. Et eksempel på et minnehierarki der vi har 4 prosessorer er som følgende:



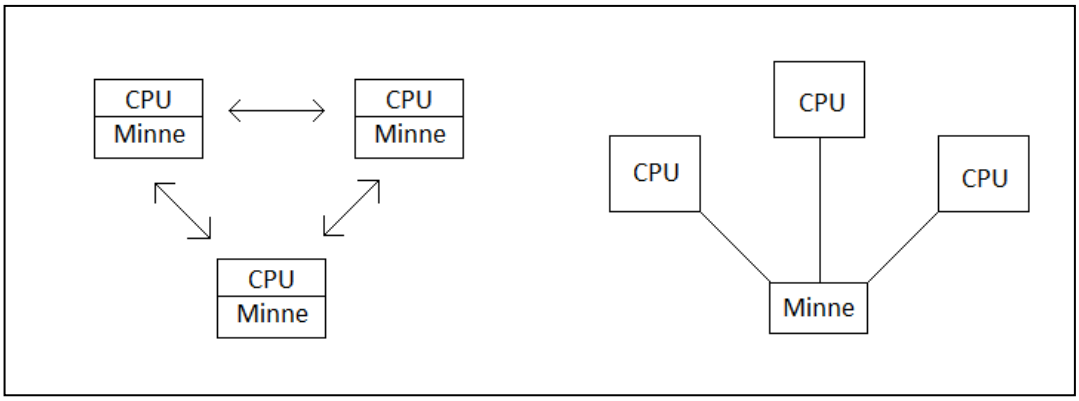
Figur 1: *Et typisk minnehierarki for en prosessor med fire kjerner.*

Det betyr at hvis flere kjerner jobber med samme data er det uunngåelig at data sendes videre til delte lagringsnivåer hvis flere prosessorer skal arbeide på samme data. Derfor må vi unngå at vi skriver og leser samme verdier om hverandre, da dette vil skape inkonsistens i minnet. En viktig del av parallellisering er at vi må synkronisere rundt felles data om nødvendig. Dette vil skape en dårligere speedup enn vi hadde håpet på, men det viktigste er at resultatet av et program blir riktig. Det kan til og med føre til at kjøretidene blir verre enn de sekvensielle versjonene.

2.3.2 Distribuerte systemer

En forgjenger til parallelle eksekveringer på én maskin, er distribuerte systemer. Dette vil si at vi kjører flere maskiner, som tilsynelatende fungerer som én maskin. Det virker på en slik måte at vi fordeler arbeidet på forskjellige maskiner, som alle gjør sin del, før det kommer tilbake igjen. Ganske likt det som skjer i en multikjernet arkitektur. Internett har gjort dette til en mulighet fordi vi kan skape kommunikasjon mellom store avstander. Sett utenfra kan det se ut til å være liten forskjell mellom eksekvering på en parallell maskin og en eksekvering på distribuerte systemer, men det finnes noen forskjeller, spesielt innenfor strukturen på minnet:

- I distribuerte systemer finnes det ikke noe form for delt minne, da det er snakk om flere maskiner på forskjellige steder. Dette betyr at vi må bruke andre mekanismer, som for eksempel *message passing interface (MPI)*[2], for å skape kommunikasjon om samme variabler mellom maskinene. MPI er en måte for meldingsutveksling mellom forskjellige maskiner. En multikjernet maskin deler data via delt minne. En fordel med distribuerte systemer er at hver maskin har enerett på sitt eget minne.
- Siden kommunikasjonen skjer via et nettverk vil distribuerte systemer påvirkes hvis nettverket er tregt. Dette gjør at distribuerte systemer har enda et lag med problemer knyttet til seg. En multikjernet datamaskin trenger kun minnehierarkiet for at kommunikasjonen skal eksistere. Det at kjernene ligger nærme hverandre gjør det til mer pålitelig og raskere kommunikasjon. En ulempe ligger i at flere kjerner arbeider på samme minne fører til egne problemer, som for eksempel uønsket overskrivning av data.



Figur 2: *Forskjeller mellom et distribuert system(til venstre) og en multikjernet maskin i forhold til minne.*

3. Parallellitet og tidtagning i Java

3.1 Parallellitet i Java

3.1.1 Tråder i Java

Java er et programmeringsspråk, som gir oss muligheten til å programmere parallelle programmer. Dette vil si at vi kan bruke Java til å starte og håndtere tråder, slik at de utfører egne oppgaver i parallell. Et Java program starter derimot ikke med flere tråder, slik at alle programmer vil starte sekvensielt. Java vil starte med en maintråd, som igjen kan starte opp andre tråder.

Java, som lager objektorienterte programmer, må skape spesielle objekter før det kan starte nye tråder for parallell eksekvering. Alle tråder vi ønsker å starte må være et objekt av klassen klassen *Thread*. Denne klassen implementerer interfacet *Runnable*, som har bare metoden *run()* i seg. Denne metoden er viktig for tråder i Java, fordi denne definerer hva en tråd skal gjøre uavhengig av andre tråder. Thread inneholder mange andre trådoperasjoner også, blant dem er *start()*, som setter i gang trådens run() metode.

Det er to måter vi kan lage trådobjekter i Java. Den ene er å implementere *Runnable*. Å lage et Runnable objekt vil ikke automatisk starte en ny tråd slik at vi kan kjøre kode i parallell. Ved å lage et slikt objekt har vi kun definert hva en tråd skal gjøre i det tilfelle hvor vi velger å starte en. For at vi skal starte en tråd, basert på et slikt objekt, må den instansieres som et Thread objekt. Deretter kaller vi på start() metoden til dette Thread objektet. En alternativ måte å lage trådobjekter på er å lage en subklasse direkte av Thread.

Forskjellen mellom disse to måtene å lage trådobjekter på ligger i hvor vi ønsker å ha kontroll over trådene våre. Dersom vi lager en klasse, som implementerer Runnable, må vi instansiere det som et objekt av Thread der vi ønsker å starte tråden. Dette betyr at vi har kontroll over tråden utenfor vår trådklasse. Om vi lager en subklasse av Thread, vil vi også kunne håndtere trådene i selve klassen, da vi har tilgang til alle trådoperasjonene her. Vi kan altså la tråden håndtere seg selv i sin klasse, eller begrense oss til der vi starter tråden. Fordelen ved å bruke interfacet Runnable ligger i at Java ikke tilbyr oss muligheten til å arve fra flere klasser. Dette fører til at hvis vi

velger å lage en klasse som en subklasse av Thread, vil vi ikke kunne arve fra andre klasser. Følgende kode viser hvordan man starter et en tråd ved bruk av Runnable interfacet.

```
class RunnableTraadStart{
    public static void main(String[] args){
        RunnableTraad rt = new RunnableTraad(50);
        //instansierer runnable objektet vårt med Thread
        Thread t = new Thread(rt);
        //starter ny tråd
        t.start();
        //main tråden fortsetter her
    }
}

class RunnableTraad implements Runnable{
    int tall;
    int antall = 0;

    public RunnableTraad(int tall){
        this.tall = tall;
    }
    public void run(){
        //ny tråd starter her og finner antall
        //partall fra 1 til vi treffer variabelen tall
        for(int i = 1; i <= tall; i++){
            if( (i % 2) == 0) antall++;
        }
    }
}
```

Noen ganger er det ikke ønskelig å la en tråd avslutte med en gang den er ferdig med sin egen oppgave. Det kan tenkes at en tråd er en del av en større gruppe tråder, som alle jobber med hver sin del av en større oppgave. I slike situasjoner vil man ofte heller la tråder vente til alle er ferdig. I Java får vi muligheten til dette med metodene *sleep()* og *join()*. Metoden sleep() gir oss muligheten til å stoppe en tråds kjøring for en gitt tid, bestemt av metodens parametre. Dette kan være nyttig i situasjoner hvor en tråd ikke har noe nyttig å gjøre i nåværende øyeblikk, men kanskje den kan fullføre sine oppgaver på et senere tidspunkt. På denne måten vil den ikke stjele dyrebar kjøretid fra andre tråder som gjerne vil gjøre sitt arbeid på dette tidspunktet. Metoden join() gjør at vi kan få tråder til å vente på hverandre. For eksempel hvor flere tråder må samarbeide om og en større oppgave, må de alle vente på hverandre før vi kan gi et ferdig svar.

Man kan også la tråder vente i en kø kalle på *wait()*, ofte ved når en logisk setning inntreffer. Når en annen tråd kaller *notify()* vil den starte første tråd i køen. Eventuelt kan den kalle *notifyAll()* for å starte alle trådene i køen. På denne måten kan tråder samarbeide, slik at når en har oppdatert felles data sier den i fra til andre som venter. Til tross for at wait() kan være en god løsning i mange situasjoner, har den noen ulemper. Et problem er at det finnes kun en felles kø for alle wait() setningene i et objekt. Det betyr at notifyAll() vil starte tråder, som kommer fra forskjellige wait() kall. En bedre løsning vil ha vært og hatt forskjellige køer for hver wait() setning. Med riktig kode, vil det ikke føre til problemer med resultatet av programmet, men det vil ta lang tid hvis mange tråder må sjekke om det er deres tur, til tross for at objektets

tilstand er uendret i forhold til deres perspektiv og vil uansett sette seg tilbake i køen. Dette kan føre til at programmet tar lengre tid å kjøre.

Java tilbyr oss mye for å programmere i parallellitet, men det er vanskelig for oss å oppnå bedre enn sublineær speedup. Mye av dette kommer nettopp av at vi må starte trådene våre selv etter at programmet er startet, og selve oppstarten tar av tråder kan ta betydelig tid.

3.1.2 Synkronisering i Java

Det å synkronisere tråder er ofte et nødvendig steg i parallell programmering, til tross for at det ofte fører til dårligere speedup. Det er ganske vanlig at tråder arbeider på felles data. Vi kan oppleve at når vi kjører flere tråder, at vi kan få veldig mange forskjellige kjøringer fordi de arbeider om hverandre. Dersom vi tar et enkelt eksempel kan vi se på hvordan dette vil se ut i en maskin:

$$i = i + 1;$$

Denne programsetningen kan deles opp i 3 atomiske deler.

- Vi leser variabelen *i*.
- Adderer med 1.
- Skriver tilbake til verdien *i*.

I mellom hver av disse delene kan en ny tråd komme og gjøre noe av sitt arbeid. Det vil være arbeid som å lese og/eller skrive til variabelen *i*. Dette gjør at vi kan få flere tråder som leser eller skriver samme verdier om hverandre, som gjør at resultatene blir feil. Trådene må kunne samarbeide, det vil si at de må være enige om hvem som skal få tilgang og hvem som må vente i en egen kø til de kan få tilgang, og dermed holde et konsistent minne.

Synkronisering kommer ofte til en høy pris tidsmessig. Der vi før, med et sekvensielt program, kunne lagre i nærmeste ledige cache må vi nå lagre på nærmeste delte lagringsnivå mellom alle våre tråder. Dette er måten vi kan dele på våre variabler, slik at alle trådene vet hvordan tilstanden til programmet er i kjørende tidspunkt. I 2.3.1 ble det nevnt at det er dyrere tidsmessig å gå lavere ned i minnehierarkiet, slik at vi vil helst unngå det. Med synkronisering er det ofte nødvendig å gå lengre ned, til tross for at vi har plass nærmere prosessoren.

En vanlig form for synkronisering er å låse operasjoner på felles data slik at bare én tråd kan arbeide på dem om gangen. I Java fungerer synkronisering slik at det skapes en lås, som er knyttet til et objekt. Man kan skape en slik lås ved å bruke nøkkelordet *synchronized* foran et metodenavn. Dette vil føre til at koden i denne metoden kan kun bearbeides av en tråd om gangen. En kø skapes for å hindre at flere går inn i metoden, slik at tråder kan ende med å vente med sine oppgaver til andre tråder har fullført sitt først. Når en tråd er ferdig med den synkroniserte metoden, og er klar for å gi fra seg objektlåsen, vil den alltid oppdatere dataene den har endret i objektet. Dette gjør det mulig for neste tråd som får tilgang til metoden til å være oppdatert med objektets nåværende tilstand. Denne måten å synkronisere på gir ansvaret for opprettholde de delte dataene over på objektet. Det vil si at det er objektet som bestemmer selv hva som er synkronisert eller ikke. De som kaller på den synkroniserte metoden trenger ikke å bestemme om den trenger full enerett over metoden og objektets data. Det er

muligheter i Java å gjøre dette og, ved å sette *synchronized* ved objektet man ønsker å arbeide i.

Synkronisering er av og til et nødvendig onde i parallell programmering, men minst mulig bruk av dette kan være essensielt for forbedre speedupen. Hvis man klarer å la trådene jobbe for seg selv med bare lokale variabler, for å så senere sammenligne hva alle trådene kom fram til, kan vi merke bedre kjøretider. Mye på grunn av at vi ikke trenger å jobbe lengre ned i minnehierarkiet enn nødvendig. Dersom vi har situasjoner der trådene ikke køer seg opp i den synkroniserte metoden, tar det ikke så lang tid å bruke slike metoder. For eksempel, kan det være en slik metode, som kalles sjeldent av trådene.

3.1.3 Java.util.concurrent

Dette er et bibliotek i Java, som tilbyr oss bedre og enklere måter å lage parallelle programmer. Mange vanskelige temaer innen parallellitet blir presentert som enkle klasser, interfacer og metoder slik at vi, som programmerere, kan bruke mer tid hva vi faktisk vil oppnå med parallell programmering.

En viktig del av dette biblioteket er *executor*. Dette gir oss større muligheter til å kontrollere trådene våre. Vi kan lage *pools* med tråder, det vil si at vi lager et gitt antall tråder som vi vil være tilgjengelig å bruke flere ganger igjennom programmet. Vi kan se på trådene våre på en slik måte at ikke bare blir satt til å gjøre en oppgave for deretter å avslutte, men heller at vi kan sette tråder i en slags sovetilstand. Dette gjør at vi kan gjenbruke allerede oppstartede tråder til nye oppgaver på et senere tidspunkt. Det å gjenbruke tråder fører til at vi ikke trenger å starte nye tråder så ofte, som gjør at minimerer overhead og programmet blir mer effektivt.

I tillegg til at vi kan bestemme antall tråder vi vil ha og gjenbrukingen av tråder, kan vi spesifisere hvordan vi vil starte trådene. Det å starte tråder er litt annerledes i dette biblioteket enn normalt. Ved å bruke vanlig Java vil vi starte nye tråder med kallet *Thread-objekt.start()*. Via *execute(Runnable r)* i dette biblioteket ser vi mindre på oppstart av tråder, men mer på legge til flere oppgaver i vår samling av parallelle oppgaver, som skal gjøres. Et eksempel hvor dette er fordelaktig er når vi ikke har flere tilgjengelige tråder, grunnet at alle i vår pool er i arbeid. I dette tilfellet kan vi heller sette opp en kø av arbeidsoppgaver, slik at når en tråd er klar for nytt arbeid kan vi sette i gang første oppgaven i køen.

Ikke bare har vi muligheten til å kontrollere måten vi vil starte tråder, men vi har også flere alternativer til måten vi vil avslutte dem, via *ExecutorService*. Måten vi kan avslutte vår pool av tråder på avhenger av oppgavene som er laget til dem. Vi kan nemlig velge å kalle metoden *shutdown()*, som gjør at vi vil vente på at alle oppgavene som eksisterer ved dette punktet til å bli ferdig. Samtidig vil vi ikke godta flere nye oppgaver. Dersom vi ikke ønsker å fullføre de resterende oppgavene kan vi kalle på *shutdownNow()*. En annen nyttig funksjon, sammen med shutdown() er *awaitTermination(...)*, der parametrene består av hvor lenge den skal vente. Den vil stoppe når den første av to situasjoner har inntreffer. Den ene er at alle oppgavene er fullført, mens den andre er når tiden, angitt av parametrene, går ut. Denne funksjonen er nyttig når vi vil vente med å fortsette programmet til alle oppgavene er ferdig.

Biblioteket tilbyr oss også ferdige metoder og klasser for å synkronisere tråder. En viktig er *CyclicBarrier*, der vi kan sette opp et barrierepunkt, som tråder ikke kan passere før alle har nådd dette punktet. Dette er nyttig i situasjoner hvor tråder må sette sammen sine egne resultater for et større, samlet resultat, som er essensielt for videre beregninger. Fra navnet ser vi at barrieren er syklisk, som vil si at vi kan bruke den på nytt etter at alle trådene har nådd punktet og blir frigjort. Et annet godt synkroniseringsverktøy, som dette biblioteket tilbyr oss, er *Semaphore*. Semaphorer virker som en låsemekanisme for delte data, der vi baserer oss på å inkrementere og dekrementere et tall. Når en tråd spør etter låsen, vil den få tilgang dersom tallet er over 0. Ellers vil den bli satt i kø til å vente. Dette vil si at flere kan ta låsen dersom tallet er over 1, som i noen situasjoner kan være nyttig. Dette fungerer også i situasjoner hvor tallet blir negativt, da det vil bli flere som setter seg i kø ventende på tilgang.

Java.util.concurrent gir oss mange muligheter, men mye av dette kunne vi ha programmert selv. Fordelen ligger i at dette forenkler prosessen, siden mye av dette kan være ganske komplisert kode og tidkrevende å lage. Dette betyr at det kan være nyttig å bruke biblioteket for parallell programmering for å unngå feil, som dette allerede har blitt håndtert og optimalisert, og vi kan oppnå resultater raskere om vi kan konsentrere oss om problemstillingen vår.

3.2 Tidtagning i Java

3.2.1 Ingen gjennomkjøring er like i parallell programmering

Det er mange grunner til at parallellisering av kode ikke er en lett oppgave. En av disse grunnene er nettopp det at ingen gjennomkjøring er lik den forrige. Dette har alt med når en tråd får kjøretid hos prosessoren. Fra programmererens perspektiv er det umulig å forutsi hvilken rekkefølge flere tråder vil fullfører sine oppgaver på. Det kan, for eksempel, hende at første tråden vi kjører i gang får mye tid av prosessoren til å gjøre sitt arbeid. For å illustrere dette har jeg laget et ganske enkelt og parallelt program. En tråds oppgave er å telle til en million før den avslutter. Vi måler hvor lang tid det tar for tråden å telle.

Vi ønsker å starte tre tråder fra vår main() metode, som vil kjøre denne run() metoden. Det vi vil finne ut er både rekkefølge på når trådene fullfører og tiden det tar for hver tråd å gjøre seg ferdig. Resultatene for gjennomkjøring er i tabellen under. Programmet er kjørt 3 ganger, der hver kjøring setter i gang 3 tråder for å kjøre run() metoden. Hver tråd har et identifikasjonsnummer fra 1 til 3, og trådene settes i gang i denne rekkefølgen.

Termineringsrekkefølgen til trådene	ms
Forsøk 1	
3	3,6
2	3,9
1	4,1
Forsøk 2	
3	4,5
2	4,7
1	4,9
Forsøk 3	
3	4,3
1	4,9
2	4,5

Tabell 4: *Et enkelt program med 3 tråder og 3 gjennomkjøringer, og som viser kjøretid og termineringsrekkefølgen. Disse kjøretidene er basert på en Intel core I7 870 prosessor med en klokkefrekvens på 2,93 GHz, med 8 kjerner (4 fysiske med hyperthreading).*

Her ser vi at for hver gjennomkjøring tar hver tråd forskjellig tid for å beregne seg fram, og det er ikke nødvendig at de i det hele tatt kommer i en lik rekkefølge, som vi ville regnet med hadde det vært et sekvensielt program. Hadde dette vært tilfelle skulle vi fått en rekkefølge 1, 2 og 3 hver gang.

Det å programmere i parallell byr på sine utfordringer, så det hadde vært bedre om parallellisering hadde vært automatisk prosess, det vil si at en kan skrive en vanlig sekvensiell kode for å generere en parallell versjon etterpå slik at vi muligens kan oppnå en raskere eksekvering.

3.2.2 Tidsmålinger i Java

Tidsmålinger vi tar ønsker vi skal gi oss en god representasjon av hvor lang tid dette programmet tar å kjøre. Det vi vil ha er et tall som kan gi oss en god indikasjon på hvordan dette programmet fungerer i forhold til andre, som gjør samme arbeid, men på en annen fremgangsmåte. Vi er interessert i er på hvilken måte kan vi parallellisere best på, slik at vi kan oppnå en god speedup på problemer vi ønsker løse. Derfor må vi representere alternativene på en fornuftig måte.

Om man lager et program, som løser en oppgave om og om igjen i samme kjøring, vil vi legge merke til at det er store forskjeller i hvor lang tid det tar å løse oppgaven etter hvert som kjøringene går. I Java kan vi se at programmer optimaliserer deler av programmet ettersom programmet går. Grunnen til at vi oppnår bedre tider etter hvert[13], som kjøringene går, kommer blant annet av "Just-in-time" (JiT) kompilering. Javas kompilering begynner slik:

- Starter med at det lages en *.class* fil, som er en bytekode. Dette blir interpretert når programmet kjøres.
- Javas virtuelle maskin (JVM) kan JiT kompilere bytekoden om til maskinkode, dersom den legger merke til at samme kode kalles flere ganger.
- Hvis JVM ser at koden kalles flere ganger kan den gjøre andre optimaliseringer også. I tillegg, kan det også være tilpasninger i cachene. På denne måten håndteres dataene mer effektivt etter hvert, som man kjører programmet, og dermed blir kjøretiden bedre.

For å vise et eksempel på JiT kompilering tar vi utgangspunkt i et program, som finner summen av en mengde. Programmet genererer 8000 uniformt fordelte tall og finner summen av disse. Vi måler hvor lang tid det tar for programmet å summere disse tallene i nanosekunder, men fremvist i tabellen under i mikrosekunder for lesbarhet. Vi gjør dette 30 ganger, for å sette i gang JiT kompileringen. Statistikken vi får er som følger:

Iterasjonsnr	µs
1	287
2	235
3	234
4	267
5	234
6	234
7	147
8	152
9	180
10	30

Iterasjonsnr	µs
11	31
12	28
13	31
14	38
15	28
16	28
17	31
18	79
19	27
20	26

Iterasjonsnr	µs
21	4
22	3
23	3
24	4
25	3
26	3
27	3
28	3
29	21
30	2

Tabell 5: *kjøretider over flere iterasjoner av programmet, som finner summen av en mengde. Disse kjøretidene er basert på en Intel core I7 870 prosessor med en klokkefrekvens på 2,93 GHz, med 8 kjerner (4 fysiske med hyperthreading).*

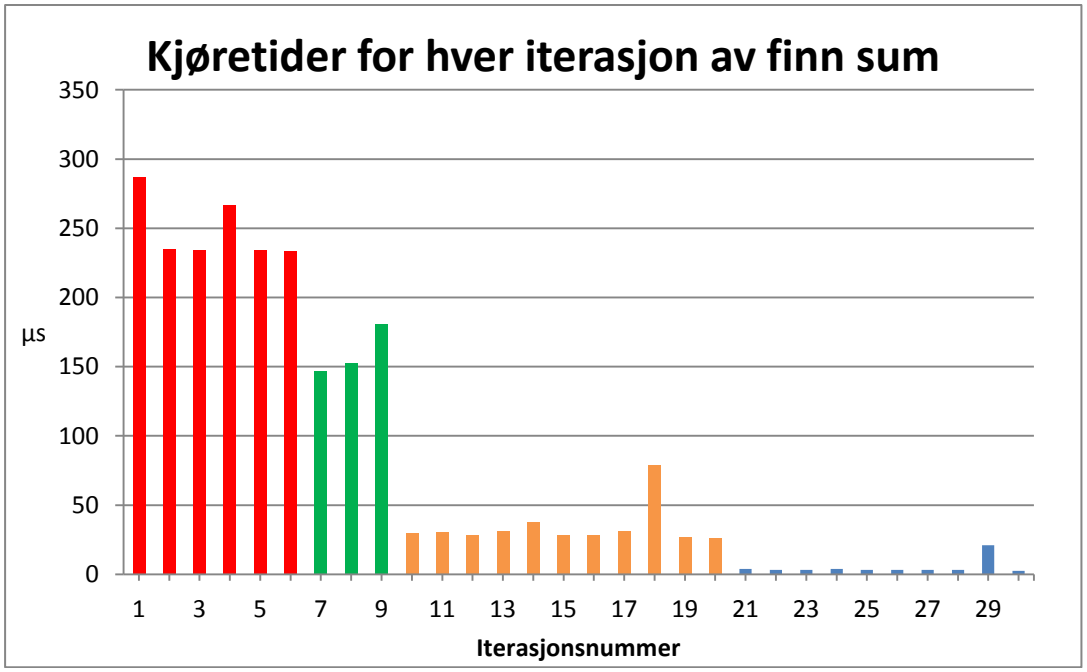


Diagram 1: *Kjøretider over flere iterasjoner av programmet, som finner summen av en mengde (8000 tall). Tallene baserer seg på Tabell 5. Fargene viser at JiT kompileringen slår til, og vi får bedre og bedre tider etter hvert. Rød er bytekode, grønn er maskinkode, oransje er første optimalisering og blå er andre optimalisering. Disse kjøretidene er basert på en Intel core I7 870 prosessor med en klokkefrekvens på 2,93 GHz, med 8 kjerner (4 fysiske med hyperthreading).*

Dette gir oss et ganske klart bilde på hvor stor forskjellen er mellom kjøringene. Den raskeste kjøringen er omtrent 117 ganger raskere enn den tregeste. Dette gjør at vi må gå riktig frem når vi skal representere tidsmålingene våre.

Det vi er ute etter er en god måte å ta en rekke målinger av et program og representere dette med ett tall. Som vi har sett er vår største utfordring altså å takle de første målingene som viser seg å variere stort i forhold til resten. Naturlige måter å samle numeriske data til et tall er gjennomsnitt og median. Begge har sine styrker og svakheter. Problemet med gjennomsnitt er måten den sliter med uteliggere på. Ifølge David S. Moore og George P. McCabe[15] kan vi si dette om gjennomsnitt:

"The mean is sensitive to the influence of a few extreme observations"

De første utslagene vi har i gjennomkjøringen vi hadde over kan vi si er ekstreme observasjoner. Median derimot har ikke dette problemet, da den ikke regner seg fram til hva midterste verdien er, ved å bruke alle verdier, men heller velger midterste verdi basert de numeriske dataene i en sortert rekkefølge. Derfor er ikke disse ekstreme observasjonene så innflytelsesrike, som de er for gjennomsnitt. For å vise den praktiske forskjellen kan vi se på gjennomsnitt og median for gjennomkjøringene vi hadde over.

Gjennomsnitt	Median
80	29

Her ser vi at medianen representerer verdiene våre mye bedre da de fleste verdiene er nærmere 29, og ikke 80.

4. Implementasjon av Java PRP for én rekursjonsmetode

4.1 Bakgrunn for Java PRP

4.1.1 Hva er Java PRP

Java PRP er et prosjekt som har blitt gått igjennom på forskjellige måter gjennom andre masteroppgaver. Grunnlaget for Java PRP er at den tar for seg parallellisering av sekvensielle programmer med en rekursjonsmetode. Tidligere versjoner har vært alt fra distribuerte systemer i C til programmering på multikjernemaskiner i Java. Poenget har ligget i at den genererte, parallelle versjonen skal gi oss en speedup samtidig som det skal gjøre parallellisering mer brukervennlig. Ved å følge et sett med regler og bruk av nøkkelord i form av kommentarer har man oppnådd dette. I masteroppgaven til Kristoffer Skaret[19] ble disse nøkkelordene brukt:

- /* PRP_PROC */
  Denne linjen settes over rekursjonsmetoden for å markere at for denne metoden er det ønskelig å parallellisere.
- /*PRP_CALL*/
  Dette settes over det faktiske kallet som setter i gang rekursjonen.
- /*PRP_FF*/
  Dette markerer typisk "full fanout" altså at alle rekursjonenskallene skal komme fra det første kallet. I en trerepresentasjon av rekursjonen vil dette si at alle andre noder enn roten kommer direkte fra roten. Denne setningen setter man over selve klassen.

4.1.2 Mellomledd mellom ett sekvensielt program og kompilatoren

Å skrive et sekvensielt program er lettere enn å skrive et parallelt program. Vi trenger ikke å tenke på hvilken oppgave som er heldig nok å få prosessorens oppmerksomhet, som fører til tilfeldige gjennomkjøringer. En sekvensiell kode går i den rekkefølgen den er kodet i. Programmere parallelle programmer fører også med seg mye administrativ kode for å sette i gang tråder. Derimot vet vi at et parallelt program kan gi oss flere fordeler, som for eksempel en mulig raskere kjøretid, som vi gjerne skulle utnyttet. Det vi ønsker oss er at vi programmerer med en sekvensiell tankegang, så lar vi et annet program ta seg av parallelliteten. Det er her Java PRP kommer inn. Det vil

fungere som et mellomledd mellom kildekoden vi skriver og kompilatoren som skal lage et kjørbart program. En slags oversetter fra det sekvensielle til det parallelle. Ved å skape et mellomledd mellom et sekvensielt program og kompilatoren, kan vi preprosessere programmet før vi kommer til kompilatoren slik at den kun ser et parallelt program.

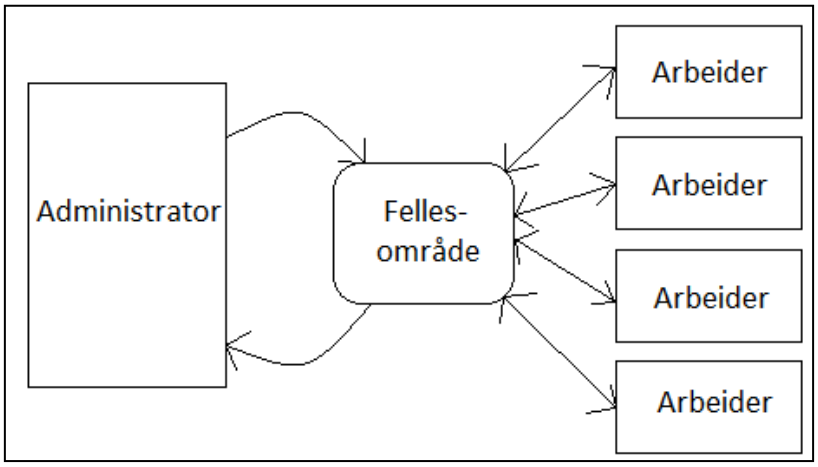
4.1.3 Fordeler ved automatisk parallellisering

Parallellisering medfører komplikasjoner som er tidkrevende å finne ut av og rette på, siden vi ikke får samme gjennomkjøring hver gang vi tester. Derfor hadde det vært bedre dersom vi kunne automatisere denne endringen fra et sekvensielt program til et parallelt. Det eneste man trenger å gjøre er å bruke litt ekstra tid på og preprosessere kildekoden slik at vi sender en parallell versjon til kompilatoren. Resultatet av dette er at vi utnytter alle våre eventuelle prosessorkjerner og kan få en speedup.

En annen god grunn til å automatisere parallelliseringsprosessen, er brukervennligheten. Istedenfor at vi skal lage en unik konstruksjon av parallellitetskode for hver gang vi ønsker å utnytte vår maskinvare bedre, kan vi bruke en preprosessor for å gjøre dette for oss. Dette gir oss mer tid til å løse selve oppgaven, enn å bruke tid på å løse problemer, som parallellisering introduserer. Alt vi trenger å gjøre er å lære oss nøkkelordene til Java PRP, og dermed parallellisere koden vår automatisk.

4.1.4 Administrator/arbeider-modell

Java PRP har tidligere bestått av administrator/arbeider-modell for å gjøre programmene parallelle. Denne modellen er god for å utnytte flere kjerner, da vi deler et større problem inn i mindre og uavhengige oppgaver. Dette betyr at vi skaper en administrator, som lager oppgaver og gir disse ut til flere arbeidere.



Figur 3: *administrator/arbeider modell*

Bildet ovenfor viser hvordan dette fungerer der en administrator legger ut oppgaver i et fellesområde. Arbeiderne tar for seg oppgaver og løser dem før de legger svarene tilbake og eventuelt tar en ny oppgave. Administratoren henter løsninger, som har blitt lagt i fellesområde av arbeiderne.

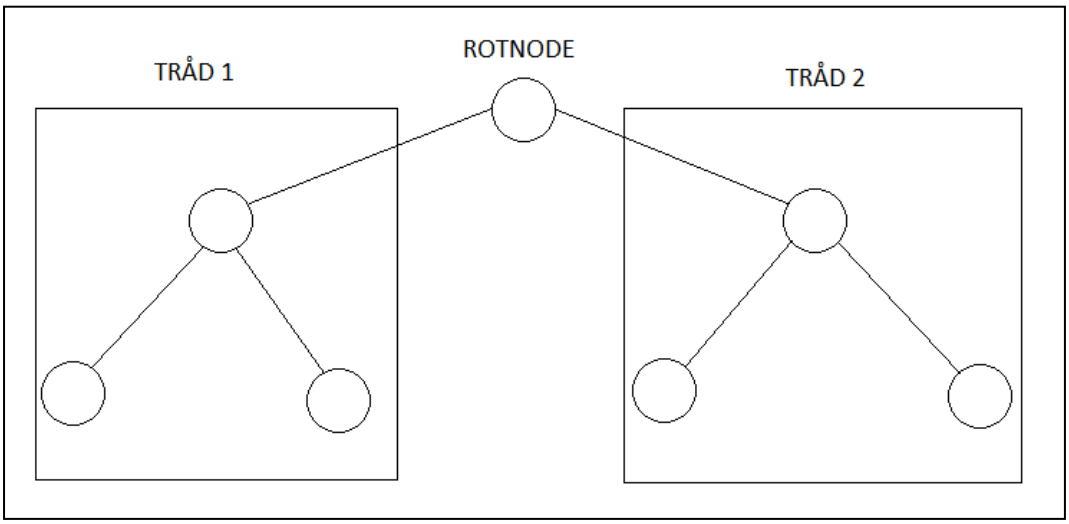
4.2 Rekursjon og parallellitet

4.2.1 Rekursjon og trær

Vi baserer oss på at rekursjonsmetodene inneholder dette:

- Et basistilfelle, hvor vi kan slutte rekursjonskallene og gi et resultat.
- Flere rekursjonskall, som fører beregningene videre.

Trestrukturer er en velkjent form å organisere data i programmer. Det at en informasjonsnode kan gi utløp til mer enn én node, som for eksempel en enkelt lenket liste, gir oss ofte muligheten til å prosessere data fortere. For datastrukturer, som inkluderer en form for trær, baserer mange av metodene seg på rekursjon for traversering. For å parallellisere metoder er det lettere dersom vi har en fanout, det vil si antall noder som kan springe ut fra en annen, større enn 1. Dette gjør det mulig å utnytte en multikjernet datamaskinsarkitektur bedre fordi vi kan la forskjellige kjerner arbeide i hver sine deltrær. Følgende bilde viser hvordan et tre med fanout lik 2 kan utnytte to kjerner via to tråder.



Figur 4: *Et tre med fanout lik 2, som operer med 2 tråder.*

4.2.2 Hvorfor rekursjon kan lett bli parallellisert

Rekursjonsmetoder egner seg bra til parallellisering fordi vi kjører kode, som gjentar seg flere ganger, slik at vi kan tilegne flere tråder til å jobbe med samme metode bare i forskjellige grener av rekursjonstreet. Rekursive metoder får, som oftest, alle sine data fra sine parametre, og returnerer et svar ved slutten av metoden.

Problemet vårt ligger derimot ikke nødvendigvis om det er lett å parallellisere, men om vil vi oppnå raskere kjøretider ved å parallellisere en rekursiv metode. Vi må unngå å parallellisere på mindre gunstige måter, som for eksempel at vi mye synkronisering, som fører til mye venting mellom trådene. Løsningen kan ligge i bedre måter å parallellisere på.

4.2.3 Java PRP eksempel

Et eksempel på hvordan Java PRP, laget av Kristoffer Skaret, vil ta en sekvensiell rekursjonsmetode, som summerer en rekke tall slik:

```
/*PRP_PROC*/
public int sumArray(int[] heltall, int start, int end){
    int length = (end - start);
    if(length < k){
        int sum = 0;
        for(int i = start; i < end; i++){
            sum += heltall[i];
        }
        return sum;
    }
    int half = start + ((end-start)/2);
    /*PRP_CALL*/
    int sum_left = sumArray(heltall,start,half);
    /*PRP_CALL*/
    int sum_right = sumArray(heltall,half+1,end);
    return sum_left+sum_right;
}
```

Her markerer vi følgende:

- Vi setter PRP_PROC over metoden for å vise Java PRP at dette er en rekursjonsmetode, som skaper grunnlaget for det parallelle programmet.
- Vi markerer selve rekursjonskallene med PRP_CALL for å markere at her starter et rekursjonskall.

Herfra vil vi starte genereringen av oppgavene, basert på rekursjonsinstansene. Flere arbeidere vil deretter bli skapt for å løse disse før de gir tilbake svarene de har fått.

4.3 Implementasjonen av denne oppgavens Java PRP

4.3.1 Kodetekst som parameter

Java PRP tar imot et kjørbart, sekvensielt program. Dersom programmet ikke er kjørbart vil heller ikke resultatet av Java PRP kjøre. Programmet kommer i et tekstlig format, altså String i Java. Dette kan Java PRP lese og bruke for å skape en parallell versjon. Ved å operere på koden på et tekstlig format kan vi kopiere og endre forskjellige deler av kode. For eksempel, vil vi at den rekursive metoden skal være like i det parallelle programmet, laget av Java PRP, som det sekvensielle programmet, slik at vi oppnår samme resultat.

For at programmet skal kunne parallelliseres må Java PRP få inn noen få nøkkelord, som fungerer som kommentarer i det originale programmet. Disse nøkkelordene gjør det mulig for Java PRP å identifisere noen viktig elementer i koden, som den er interessert i når det gjelder å parallellisere. Det brukes to nøkkelord:

- Det første nøkkelordet er /*FUNC*/, og vi vil sette denne over rekursjonsmetoden, som skal parallelliseres. Dette gjør det lettere for Java PRP å fokusere kun på den koden vi vil parallellisere.
- Det siste nøkkelordet er /*REC*/ som står over de rekursive kallene inne i metoden. Den vil forekomme én gang per rekursive kall.

4.3.2 Klassestruktur på det parallelle programmet

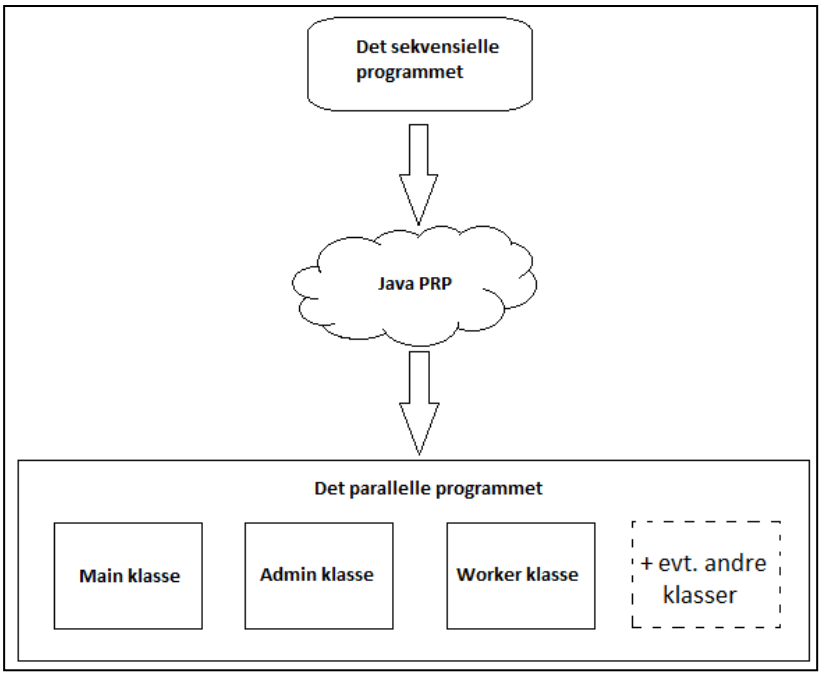
Uavhengig av klassestrukturen til det sekvensielle programmet, blir det laget et program basert på tre klasser. Den første er en klasse, der Javas mainmetode holder til, og har navnet lik det sekvensielle programmets navn med en ”Para” hengende etter. For eksempel vil et program ved navn ”*finnElement.java*” hete ”*finnElementPara.java*”. Det eneste denne klassen gjør er å kalle den neste klassen *Admin*, som vil starte alt arbeidet. Mainklassen har en fordel ved at den gjør filnavnet mer gjenkjennelig for brukeren.

Administratorklassen er den som kontrollere det meste administrative til trådene, alt fra oppstart og håndtering av arbeidere til å levere et resultat til bruker. Dens oppgave er å sette opp arbeidere, i form av tråder, og datastrukturen for disse. Den skal passe på at oppgavene blir løst og sender arbeidere når det trengs. Den har også i oppgave å organisere oppsamlingen av svar fra alle arbeiderne å gi et endelig svar til brukeren.

Siste klassen, som blir laget av Java PRP, er Worker. Denne klassen utfører selve arbeidet gitt av den rekursive metoden. Det er denne metoden som implementerer Runnable, som tilsier at det blir mange av denne klassen, som skal kjøre parallelt.

Ellers vil eventuelt andre klasser, som er med i det sekvensielle programmet, bli inkludert direkte inn i det parallelle programmet. For eksempel, kan man ha klasser, som gjør noen oppstartsberegninger.

Dette tilsvarer en klassestruktur, som gir oss en administrator/arbeidermodell, der administrator styrer alle administrative oppgaver, mens arbeiderne står for grovarbeidet, det vil si utføre de faktiske, rekursive kallene. Denne modellen gir oss derfor en struktur, som er godt egnet for parallelle programmer.



Figur 5: *Fra det sekvensielle programmet, gjennom Java PRP og til den parallelle versjonen. Det nye, parallelle programmet vil inneholde en klasse for main, Admin, Worker pluss eventuelt andre klasser fra det sekvensielle programmet, som ikke er en del av parallelliseringen.*

4.4 Admin

4.4.1 Administratorens oppstart

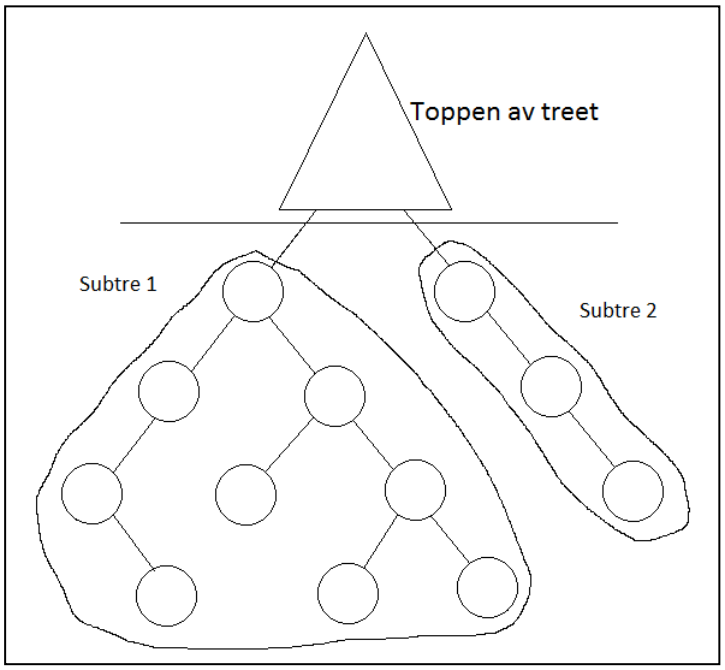
Administratoren har en rekke oppgaver den må fullføre. Det aller første den må gjøre er å samle opp relevant kode, som oppstår utenfor den rekursive metoden. Klassen Admin må inneholde koden, som forekommer før det sekvensielle programmet treffer kallet til den rekursive metoden. For eksempel, kan det hende at den må skape et array av elementer som den rekursive metoden skal arbeide på.

4.4.2 Administratorens visualisering av den rekursive metoden

Etter at vi har klart å få over annen kode kommer vi til det første sentrale punktet til administratoren. Vi ønsker i administratoren å ha en datastruktur for arbeiderne vi starter opp. Måten Java PRP går frem på er å visualisere seg trestrukturen til den rekursive metoden. Stegvis vil vi opprette datastrukturen vår slik:

- Vi har alltid en initial tilstand, som er første kallet til metoden, og vil derfor tilsvare roten i treet. Alle rekursive kall inne i metoden vil tilsvare barnenoder til roten vår. Dette gir grunnlaget til at vi ønsker en fanout større enn 1, altså at vi har flere rekursive kall enn 1. Er det bare ett rekursivt kall ender vi opp med en lenket liste istedenfor en trestruktur.
- Vi går dypere i treet, breddeførst, til vi ender med antall grener, i nederste nivå, lik antallet vi ønsker å utføre i parallell, som vil basere seg på antallet kjerner vi har tilgjengelig. Det er viktig å poengtere at vi traverserer breddeførst slik at vi ender med et mest mulig balansert tre når vi treffet et ønskelig antall subtrær.

- De subtrærne vi ender opp med trenger ikke å opprette flere barnenoder. På dette punktet har vi oppnådd så mange subtrær som vi ønsker å parallellisere, slik at disse vil begynne å utføre sin kode parallelt med hverandre. De i treet over dette skillet ligger på vent til de subtrærne i arbeid er ferdig.



Figur 6: *Visualisering av treet der vi ønsker å parallellisere med to tråder. Toppen av treet tilsvarer breddeførst traversering til vi ender på to subtrær. Disse to subtrærne vil traverseres dybdeførst.*

4.4.3 Datastrukturene for arbeiderne

Med denne trestrukturen som grunnlag, kan vi begynne med datastrukturen til vårt parallelle program. Vi oppretter to typer lister for arbeiderne våre, som i treet tilsvarer nodene. Én for de arbeiderne som ligger i toppen av treet og venter på svar fra sine barnenoder og en annen struktur for de arbeiderne, som representerer subtrærne, som skal utføre selve arbeidet.

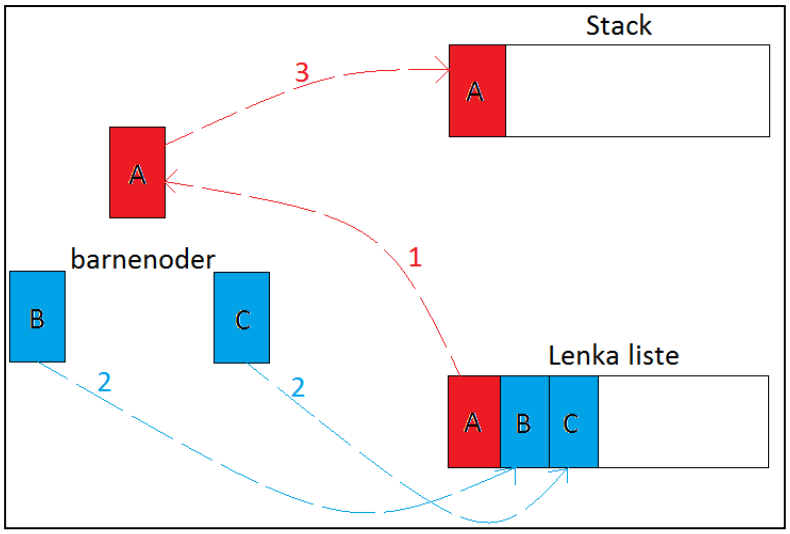
For de arbeiderne, som ligger på vent, vil vi prosessere dem i en LIFO liste, som i en stack. Dette kommer av at den aller siste arbeideren vi lager som skal vente på svar, må være den første til å få sitt svar. De arbeiderne over må vente på denne arbeideren igjen. Rotnoden, som vil legges inn først, må vente på alle.

For arbeidere, som faktisk skal utføre den rekursive metoden, lager vi en FIFO liste, som en lenket liste, fordi her kan de hentes ut som de kom, som støtter rettferdigheten med at første arbeider, som gjorde seg klar til å utføre sine kalkulasjoner, får levere sitt svar først. Dette vil også stemme overens med stacken over ventende arbeidere. Arbeideren på toppen av stacken venter på første arbeider i den lenka lista.

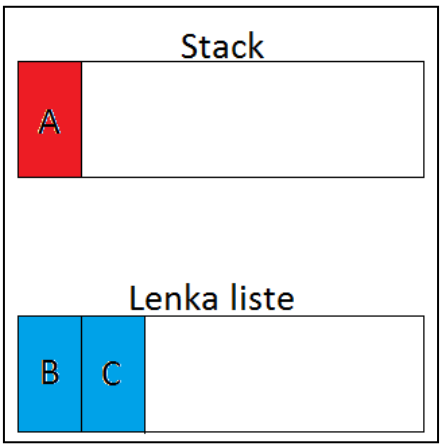
Datastrukturene er tomme ved oppstart, bortsett fra at rotnoden legges i den lenka lista. Figur 7 viser måten vi deler opp arbeiderne i disse to datastrukturene:

1. Hente første arbeider, node A, i den lenka lista
2. Finne denne arbeiderens barn, nodene B og C, for å legge disse på den lenka lista

3. Legge første arbeider, node A, på stacken.
4. Begynne på steg 1 igjen til vi har nok elementer på den lenka lista. Dette er da antallet arbeidere som skal arbeide parallelt. Da er vi ferdig med å opprette datastrukturen for arbeiderne.



Figur 7: *Administratoren oppretter datastrukturen til arbeiderne.*



Figur 8: *Datastrukturen etter at Figur 7 er ferdig.*

4.4.4 Oppstart av trådene

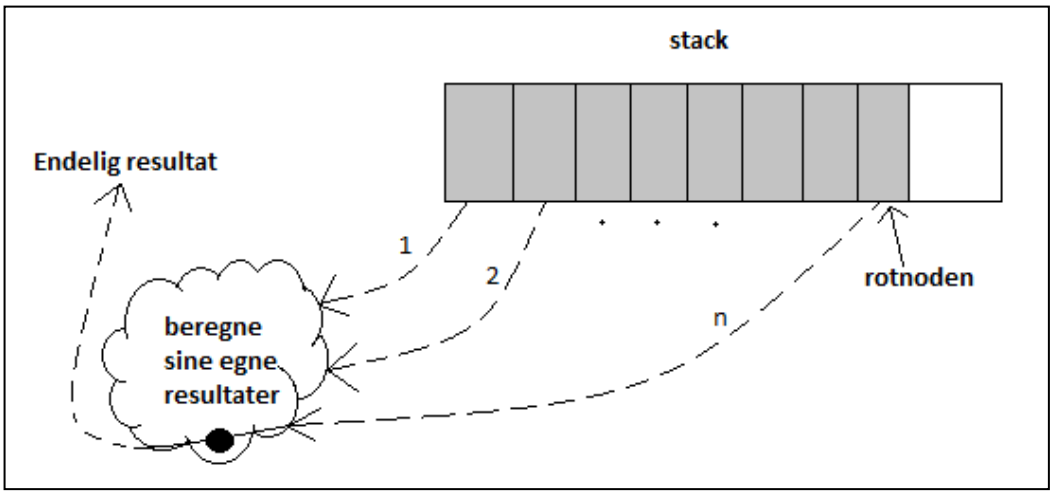
Nå som vi har gjort klart all datastruktur for arbeiderne er det på tide å sette i gang de arbeiderne som skal utføre den rekursive metoden. Det vil si at vi skal begynne med å starte opp arbeidertrådene våre. Dette gjør vi ved å bruke *ExecutorService*, fra Java.Util.Concurrent, der vi setter av plass til antall tråder som tilsvarer antallet kjerner tilgjengelig. Deretter setter vi i gang med arbeiderne i vår lenka lista, slik at de kan nå begynne å utføre sin kode i egen tråd, slik at vi nå faktisk kan begynne å arbeide i parallellitet. Dermed setter vi oss på vent til alt arbeid er ferdig, ved å bruke en syklisk barriere.

4.4.5 Kjøre sekvensielt for små kjøringer

Noen programmer har rekursjonsmetoder, som oppfyller kravene til Java PRP, men klarer ikke å skape nok oppgaver til for at Java PRP skal kunne parallellisere programmet. I disse tilfellene vil Java PRP kjøre rekursjonen sekvensielt. Ideen er at dersom programmet ikke klarer å generere nok oppgaver, er den mest sannsynlig ikke god nok for å parallelliseres. Dette kommer av parallelliseringens oppstartsfase, som for eksempel å starte flere tråder. Dette vil forekomme oftest for gjennomkjøringer av programmer med små mengder data å løpe igjennom.

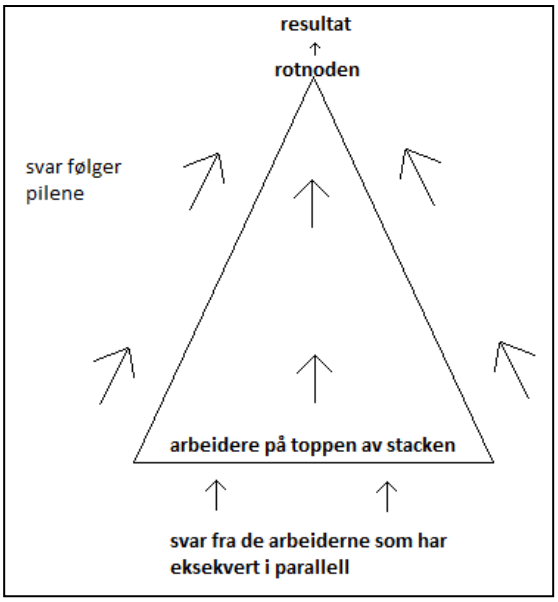
4.4.6 Oppsamling av resultater

På dette punktet våkner administrator til live igjen fordi alle arbeiderne i den lenka lista er ferdig med sine oppgaver. Neste oppgave nå er å la arbeiderne i stacken samle opp sine resultater. Arbeideren på toppen av stacken må først generere sine svar basert på sine barnearbeidere, for og deretter gå av stacken så neste arbeider kan gjøre det samme. Administratoren jobber seg igjennom stacken over ventende arbeidere, slik at de kan beregne seg fram til sine resultater. Til slutt ender vi opp i rotarbeideren der vi kan skaffe oss et konkluderende resultat og administratoren kan anse seg selv ferdig.



Figur 9: *Arbeiderstacken beregner seg innover til roten.*

Dette vet vi fungerer fordi de arbeiderne på toppen av stacken er de som først har barnenoder som er ferdig med kalkuleringene sine. De arbeiderne lengre ned i stacken må uansett vente på de på toppen. Dette kan vi vise ved å representere innsamling av resultatene som et tre.



Figur 10: *stackarbeiderne lager resultater representert via en trestruktur.*

4.5 Worker

4.5.1 Arbeidernes arbeidsoppgaver

Hovedoppgaven til en arbeider er å utføre en rekursjonsmetode. Denne metoden vil Java PRP finne via nøkkelordene, som brukeren har satt opp i det sekvensielle programmet. Der Admin klassen står for det meste sto for administrative arbeidet for å parallellisere, står Worker klassen for grovarbeidet med å kjøre den faktiske koden for å generere et resultat tilbake til bruker.

En arbeider har tre forskjellige oppgaver den kan utføre.

1. Den ene er å generere sine egne barnenoder. Dette er en oppgave administratoren vil bruke for å skape sine datastrukturer basert på den rekursive metoden.
2. Den neste oppgaven, som er den viktigste til enhver arbeider, er å utføre den rekursive metoden.
3. Den aller siste er å gi et svar basert på denne arbeiderens oppgave.

4.5.2 Forskjell mellom arbeiderne

Det som skiller alle arbeiderne er parametrene de arbeider på. Dette gjør at arbeiderne som eksekverer sin kode i parallell vil arbeide på forskjellig data. Dette er fordelaktig da det ikke gir mening at vi har flere arbeidere som gjør det samme. Et eksempel er oppgaven med å finne største tall i ett array, der vi bruker rekursjon til å dele opp arrayet i to for hvert kall. Da vil løvnodene i dette rekursjonstreet sitte med hver sin del av arrayet.

Det er også en forskjell mellom arbeidere i forhold til hvor de ligger i administratorens rekursjonstre. Det er visse arbeidere som ikke begynner sitt arbeid, men venter på sine barnenoder til å fullføre sitt arbeid. Dette er arbeiderne som blir plassert i stacken hos administratoren og ligger på toppen av rekursjonstreet. Disse vil ikke utføre rekursjonsmetoden på samme måte som arbeiderne lengre nede på rekursjonstreet, men sitter på vent til de får svar. Deretter kan de fullføre sin kode, der de spør etter barnenodene sine svar istedenfor å rekursere videre.

4.5.3 Den rekursive metoden

Den rekursive metoden er kopiert fra det sekvensielle programmet og inn i Worker klassen. Det er derimot noen forskjeller for å gjøre utførelsen lettere for det parallelle programmet. Den største forskjellen, i forhold til syntaks, er at at metodenavnet er endret for at det blir lettere å bruke i større skop for Java PRP. Slik at metoden har fått navnet "*mySolution(...)"*, og dette endrer også kallene på metodene, både den som starter det hele fra administratoren og de rekursive kallene inne i metoden. Den fikk dette navnet da metoden kun gir ett fullstendig svar fra rotnoden, mens alle arbeiderne bare kan komme fram til et midlertidig svar basert på sitt eget parametersett.

4.5.4 Arbeidernes administrative oppgaver

Til tross for at det meste administrative blir gjort i Admin klassen, blir noe overlatt til arbeiderne. Dette er arbeid som er naturlig for en arbeider å gjøre, men som oftest blir startet opp av Admin klassen. For det er naturlig at en arbeider, som vet mest om sine parametere, kan skape korrekte barnenoder. Derfor er dette oppstartsfasen til mange av arbeiderne, og blir startet opp av administratoren i dens oppstartsfase når den skaper sine datastrukturer. Grunnen til at dette ikke er oppstartsfasen til alle arbeidere er fordi det fins arbeidere som ikke skal skape flere noder, nemlig arbeiderne som skal utføre sin kode i parallelt med andre. Dette er kun en oppgave til de arbeiderne i toppen av treet som kommer til å vente på svar i tillegg til å ha utført rekursjonskoden.

Arbeiderne har derimot en felles administrativ oppgave. Nemlig så vil de generere svar til foreldrenoden. Fordi vi ser på den rekursive metoden, som et tre, vil hver node generere svar basert på sine parametere. Dette betyr at vi må til slutt levere svar oppover helt til vi treffer rotnoden som leverer et konkluderende resultat til brukeren. Dette er en egenskap alle arbeiderne deler, både de som utfører kode rekursivt og de som ligger på vent på at barnenoder skal fullføre.

4.5.5 Generere barnenoder

Vi gjenbruker den rekursive metoden, med noen få endringer, for å oppnå genereringen av nye arbeidere, nettopp barnenoder til nåværende arbeider. Grunnen til at vi vil bruke denne metoden for å skape koden som vil genereringen er basert på to essensielle ideer:

- Vi vet at antall nye rekursive kall er likt antall barnenoder vi ønsker å generere.
- Parametersettet til barnenodene skal være generert slik som det ville vært om det var faktiske rekursive kall, slik at den rekursive metoden lager nye parametersett akkurat slik vi ønsker det.

Dette skaper grunnlaget for at vi vil bruke den rekursive metoden. For det første er å endre de rekursive kallene om til å lage en arbeider per kall og lagrer dem i en lenka liste. For det andre er vi ikke interessert i koden, som etterfølger de rekursive kallene og vanligvis inneholder en form for behandling av svarene fra de rekursive kallene for deretter å returnere et svar fra denne instansen. Det vi derimot ønsker er å returnere lista over barnenodene til administratoren, samtidig som arbeideren bevarer en kopi selv.

4.5.6 Svargenereringsmetoden

Svargenerering fungerer veldig også likt som den rekursive metoden. Mye av koden hentes nemlig fra den rekursive. Forskjellen ligger i at den ikke vil rekursere videre, men heller hente svar fra sine barnenoder. Vi kan fort lete frem hvor mange barnenoder vi har basert på hvor mange rekursive kall som fantes i den originale metoden. Dermed kan vi erstatte alle kallene, som vi finner ved nøkkelordet /*REC*/ med å hente løsninger fra hver av barnenodene. Dermed håndterer vi dataene nøyaktig som den rekursive metoden hadde håndtert sine data fra de rekursive kallene.

4.6 Oppsummering for implementasjonen av Java PRP

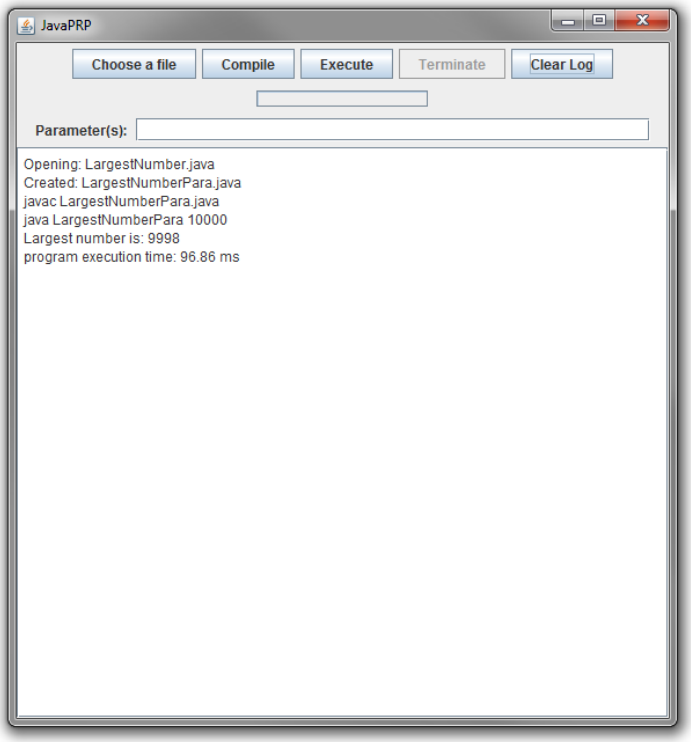
4.6.1 Java PRP for programmer med én rekursjonsmetode

Det vi har kommet frem til nå er et program, som klarer å parallellisere en rekke sekvensielle programmer. Dette baserer seg på at disse programmene følger Java PRP sine krav for hvordan koden skal oppbygges. Det vi får ut er et program, som kan kjøre parallelt med hensyn til antall kjerner tilgjengelig i datamaskinen, og har et fokus på speedup fra det sekvensielle programmet.

4.6.2 Grafisk grensesnitt

For å øke brukervennligheten til programmet er det laget et grafisk grensesnitt via *swing* i Java. Det er et ganske enkelt vindu, men gir brukeren lettere tilgang til å bruke Java PRP.

Av funksjoner, tilbyr dette grensesnittet naturligvis en måte å parallellisere en valgt fil. Det er også mulig å kompilere og kjøre programmet direkte i grensesnittet. Via en logg vil man få samme tilbakemeldinger, som en standard terminal. Loggen vil også skrive ut kjøretiden av det parallelle programmet i millisekunder, ettersom vi er ute etter effektivitet. Det vil også være en liten animasjon, i tilfelle kjøringen av programmet tar lang tid. Dette er for at brukeren ikke skal tro at Java PRP har låst seg fast. Det blir også gitt muligheten til å terminere, et program dersom det er ønskelig. Det er også lagt til en liten tekstboks for parametre til kjøringen.



Figur 11: *Et bilde av Java PRP sitt grafiske grensesnitt, der vi har parallellisert et program og kjørt dette.*

5. Delte variabler i Java

5.1 Globale og delte variabler

5.1.1 Global tilgang til de delte variablene

I et parallelt perspektiv, vil flere tråder kjenne til de delte variabler i programmet. Derfor vil delte variabler ofte være blant de globale variablene i programmene, da de er kjent for større deler av programmet. For at man skal parallellisere på en god måte, spesielt for Java PRP, som skal gjøre det på et generelt grunnlag, må man ta hensyn til hvordan man behandler slike variabler. Derfor er det nyttig å se på hvordan de globale variablene fungerer i Java.

5.1.2 Skop

Java er et blokkstrukturert programmeringsspråk[11], slik at kjennskap til andre variabler kommer an på hvilken blokk man tilhører. Man kjenner til variablene i sin egen blokk og blokkene som omringer sin selv. Dette betyr at mange indre blokker deler flere ytre blokker, som vil si at vi disse ytre variablene fungerer på mange måter som globale variabler, mens de ytre blokkene ikke kjenner til de indre variablene. I standard blokkstrukturerte programmer vil variablene i den ytterste blokka være helt globale variabler.

Declaration before use er en annen form for skop. Istedenfor at vi organiserer data i blokker, sier vi at alle variabler vi kjenner til må være deklarert over der det blir brukt i programteksten før de kan brukes. Det vil si at være globale variabler vil være på toppen av programmet. Dette er brukt i språk som C og Pascal.

Java, som et objektorientert språk, kan utvide blokkstrukturen ved objektspekere, slik at vi kan ha kunnskap om andre variabler enn det blokkstrukturen tillater. Dette gjør at vi kan ha globale variabler i andre objekter.

5.1.3 Statiske variabler

Vi kan definere metoder og variabler som *static* i Java, slik at de kan brukes av hele programmet uten å ha en peker til et objekt av klassen de tilhører. Man kan si at en statisk variabel tilhører en klasse, men ikke objekter av den. Det vil si at vi finner variabelen direkte via klassenavnet, og ikke via en objektspeker. Dette gjør at de fungerer globale variabler for hele programmet. For eksempel kan vi ønske å ha

konstanter i programmet som gjelder for alle objekter, slik som tallet pi. Dette kan vi legge inn i koden slik[20]:

$$static\ final\ double\ PI = 3.1415926535897932;$$

Her har vi også lagt til ordet *final* som gjør at variabelen ikke endrer seg i programmet. Dersom man har en statisk variabel, som ikke er final, vil vi kunne endre på denne variabelen over hele programmet, altså i alle objekter. Dette kan føre til at det er vanskelig å kontrollere tilstanden til variabelen.

5.1.4 Tråders perspektiv på variabler

Et Java program vil alltid starte med en tråd, nemlig maintråden[17]. Slik som det meste av prosesser i et operativsystem, blir tråder gitt en prioritet. Herfra kan vi starte nye tråder, som vil få samme prioritet som tråden som skapte den. Alle tråder i et program er en del av den samme prosessen. Dette gjør at de deler variablene som er laget i programmet som kjøres. Vi må derfor passe på variablene våres dersom vi skal bruke flere tråder samtidig.

For å opprettholde variablene riktig, så trenger vi å ha med *happens-before* forholdet. Det vil si at en endring fra en tråd er synlig for en annen. På denne måten vil vi ikke ha overskrivninger fra forskjellige tråder samtidig.

5.2 Programmere med delte variabler

5.2.1 Synkronisering, et nødvendig onde?

Synkroniserte metoder er en vanlig måte å bevare riktig tilstand til delte variabler. Dette er tilbudt til oss, som programmerere, av Java. Det er også forholdsvis enkelt å forstå, slik at dette hjelper på å gjøre parallelliseringen litt enklere.

Problemet med slike synkroniserte metoder er at de viser seg å være trege. Mye av grunnen til dette, kommer av at det dannes køer for å kunne få tilgang til metoden. Dersom flere tråder prøver å kalle på en slik metode vil det bli kun én tråd, som kan eksekvere koden om gangen. Dette betyr at tråder må vente i en kø, og dermed vil vi ikke oppnå den fulle, parallelle effekten. Det vil også være ekstra arbeid i forhold til å opprettholde strukturen til køen. Det vil også være en ekstra kø dersom den synkroniserte metoden inneholder et kall på *wait()*. Dette gir oss dårligere kjøretider når programmet kjører, dersom vi bruker dette mye i en kode. Ofte vil vi få dårligere tider enn om vi hadde kjørt dette sekvensielt.

Et alternativ er å la tråder arbeide mer lokalt, for og senere sammenligne sine resultater for å skape et svar til slutt. Det å arbeide lokalt når selve beregningene gjøres fører til at ingen tråder jobber på samme data før helt på slutten. Det er akseptabelt å ha antall kall på synkroniserte lik antall tråder, dersom vi har en tråd per kjerne, i motsetning til antall elementer vi arbeider på. Videre vil vi se på et eksempel på forskjellige måter å håndtere delte variabler på.

5.2.2 Kritisk kode

På en side av parallell programmering, nemlig *concurrent programming*[3], er det viktig at trådene kommuniserer for at kritisk kode i programmer, der man ofte håndterer delte variabler, ikke skal eksekveres samtidig. Slik synkronisering kommer ofte i form av *mutual exclusion*, der man tar seg av at såkalte kritiske seksjoner, altså kode man ikke vil at skal kunne kjøres i parallell. En annen form er *conditional synchronization*, der man vil utsette en del av et program til en gitt betingelse slår til.

Det er nemlig en slik tankegang man må følge for å bevare tilstandene til objekter. Synkroniserte metoder vil ofte komme på bekostning av kjøretiden. Dette fører ofte til at man vil heller ha den sekvensielle versjonen, fordi den kan vise seg å være mye raskere enn den parallelle koden med mye synkroniseringer. Det finnes mange former for synkroniseringer, men dersom koden krever mye av dette, vil det sjeldent være bedre enn en sekvensiell kjøring. For slike programmer blir synkronisering et nødvendig onde, for å kunne opprettholde at verdiene blir oppdatert og lest riktig.

5.2.3 Rekursjonsmetoder og delte variabler

De nærmeste variablene knyttet til rekursjonsmetoder, bortsett fra sine egne variabler, kan vi si kommer fra to steder. Den ene er parametersettet, og den andre er variablene i klassen den tilhører.

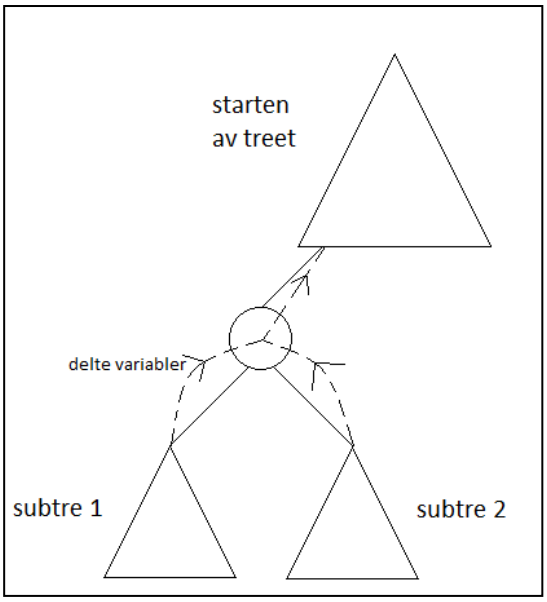
Verdiene til parametersettet er ofte forskjellig for hver instans av rekursjonsmetoden. Dette vil si at variablene ofte skrives over fordi vi ønsker at de neste instansene skal ha en annen gjennomkjøring metoden. For arbeiderne i Java PRP vil de skilles på hver sine parametersett.

Felles for alle instansene er klassevariablene som rekursjonsmetoden befinner seg i. Det betyr at vi har globale variabler i form av disse variablene. Dersom disse brukes i rekursjonsmetoden vår, er det mulig vi må synkronisere disse. Det kan være mange instanser, som ønsker å bruke variablene samtidig, slik at i et parallelt perspektiv kan vi oppleve at disse variablene overskrives på måter vi ikke ønsker.

5.2.4 La rekursjonstreet håndtere de delte variablene

Vi kan utnytte at rekursjon ofte representeres som et tre til å håndtere delte variabler, ved å la foreldrenoder samle resultatene til sine barnenoder. Vi tenker oss at vi lar rekursjonen kjøre sekvensielt i en breddeførst traversering til vi treffer nok antall subtrær som matcher antall tråder vi har ønsker. Dermed lar vi hver tråd kjøre disse subtrærne i parallell. Alle instansene har sine egne lokale varianter av de delte dataene, og leverer disse til foreldrenoder. Alle noder i rekursjonstreet som fører rekursjonen videre, vil sammenligne resultatene til sine barnenoder og levere sine oppover. Rotnoden sitter dermed med det endelige resultatet.

Ved å la rekursjonen ta for seg de delte variablene, vil vi ikke nødvendigvis trenge å synkronisere alle trådene, men vi samler opp resultater av de delte variablene mens vi returnerer oppover i rekursjonstreet.



Figur 12: *Et utsnitt av et rekursjonstre. Subtre 1 og subtre 2 gir fra seg sine resultater av de delte variablene til foreldrenoden, som igjen gir fra seg et samlet resultat til sin foreldrenode.*

En ulempe med denne ideen er at vi må kjøre metoden sekvensielt til vi har ønsket antall subtrær som vi kan kjøre i parallell. Generelt sett ønsker vi minst mulig sekvensielt for å starte opp rekursjonen når vi arbeider i et parallelt perspektiv. Samtidig kan vi se at når vi arbeider på store mengder data vil rekursjonstreet bli veldig stort. Dette betyr at delen vi gjør sekvensielt er liten i forhold til hva vi gjør i parallell. Parallellitet er best når vi arbeider på store mengder data, slik at det å optimalisere for disse forholdene kan være fordelaktig. Dette gjøres i Java PRP.

5.3 Alternativer for håndtering av delte variabler

5.3.1 Sende delte variabler som parameter

Dersom all data til rekursjonsmetoden kommer fra parametrene, kan vi utvide parametersettet slik at de inkluderer de delte variablene. Dette gjør at vi har bedre kontroll over hva som er de delte variablene, og vi vet at alle instansene av metoden vår ser på samme variabel. Derimot må vi passe på at vi har en form for synkronisering når vi skal bruke disse delte parametrene.

5.3.2 Egen klasse med delte variabler

Det å separere de delte variablene i en egen klasse kan være nyttig. Alle instanser av vår parallelle rekursjonsmetode deler kun én objektspeker, samtidig som vi gjør all kritisk arbeid i et eget sted.

I denne klassen kan vi også legge til alle metoder vi trenger for å håndtere de delte variablene, som for eksempel synkroniserte metoder for aksess til de delte variablene. Alle deler en peker til en delt klasse objekt.

Denne ideen om en egen klasse kan kombineres med det vi så i 5.3.1, der vi ville ha de delte variablene som parametre. Vi kan sende med en objektspeker til rekursjonsmetoden via parametrene.

5.3.3 Lese og skrive rettigheter til de delte variablene

Det er en måte trådene kan bruke de delte variablene våre uten å synkronisere. Dersom de delte variablene kun leses, vil dette bety at vi ikke vil få noen overskrivninger mellom trådene. Slike variabler vil aldri endre sin initialtilstand. Med en gang en delt variabel kan skrives til, kan vi verken tillate lesing eller skriving, uten at vi innfører en form for synkronisering.

5.4 Et eksempel på forskjellig bruk av delte variabler

5.4.1 Beregne summen av en mengde tall

For å illustrere hvordan bruken av delte variabler, i et parallelt program, kan påvirke kjøretidene, vil vi vise et eksempel. Dette eksempelet baserer seg på et program, der vi finner summen av en mengde tall. Det vil bli presentert tre versjoner av programmet. Det første er det sekvensielle programmet. De to andre er parallelle, der de er kun forskjellige på måten de synkroniserer rundt den delte variabelen, som representerer den totale summen. Disse vil vi sammenligne i forhold til kjøretidene.

5.4.2 Sekvensiell kode

Den sekvensielle koden baserer seg på en ganske enkel løkke som går igjennom alle tallene og legger til en *sum* variabel mens den holder på.

```
long summer(int[] arr){
    long sum = 0;
    for(int i = 0; i < arr.length; i++){
        sum += arr[i];
    }
    return sum;
}
```

5.4.3 Parallell kode 1: Mye bruk av synkronisering

Denne koden vil være ganske lik den sekvensielle, men her vil vi begynne å bruke flere tråder. Her vil vi ha en global sum variabel som vi kan legge på tall ved å bruke en synkronisert metode.

Koden for å summere for hver tråd ser slik ut:

```
void summer(){
    for(int i = start; i < end; i++){
        p1n.add(arr[i]);
    }
}
```

Der den synkroniserte metoden *add(...)*, som vi når via pekeren *p1n*, ser slik ut:

```
synchronized void add(long val){
    sum += val;
}
```

I denne implementasjonen går man inn i den synkroniserte metoden for hvert tall vi legger til.

5.4.4 Parallell kode 2: Mer lokalt arbeid

I motsetning til den første parallelle koden, vil vi ikke bruke en synkronisert metode, men heller la alle trådene beregne sin sum basert på sine tall, for og deretter senere legge alle summene sammen.

For denne versjonen er *summer()* koden litt annerledes:

```
void summer(){
    long sum = 0;
    for(int i = start; i < end; i++){
        sum += arr[i];
    }
    p2n.add(sum);
}
```

Den synkroniserte metoden er identisk til hva den var i den første parallelle koden i 5.4.3. Vi legger sammen en sum for hver av trådene, før vi helt til slutt legger disse delsummene til den totale. Dersom vi har 8 tråder og 10 000 tall så vil den første parallelle koden synkronisere 10 000 ganger, mens den andre vil gjøre det kun 8.

5.4.5 Kjøretidstester

Det vi ønsker å vise med disse testene er at synkronisering ikke er det beste alternativet, men samtidig finnes det andre løsninger, som å jobbe mer lokalt. Testene baserer seg på at vi kjører kodene i en løkke på 11. Dette gir andre resultater enn om vi hadde kjørt kodene 11 ganger etter hverandre ettersom Java JiT kompilerer og optimaliserer koden som kjøres. Vi valgte dette tallet fordi da får vi en ekte median, og ikke et snitt av to tall. Tallene er uniformt fordelte tall mellom 0 og n-1, der n er antall tall vi summerer. Tidene blir målt i nanosekunder i programmet for nøyaktighet, men framvist i tabellen som millisekunder for lesbarhet. Tabellen skriver K for tusen og M for million. Tallet framvist i tabellen er medianen av de 11 gjennomkjøringene. Antall tråder vi ønsker å bruke er lik antall kjerner vi har tilgjengelig. Testene baserer seg på en Intel core I7 870 prosessor med en klokkefrekvens på 2,93 GHz. Den har 4 fysiske kjerner, som er hyperthreaded slik at vi vil kunne utnytte dette som 8 kjerner.

5.4.6 Resultater

Resultatene vi får er som følgende:

Antall tall	Kjøretider(ms) for		
	Sekvensiell "finn sum"	Parallell 1 ("finn sum" med mye synkronisering)	Parallell 2 ("finn sum" med lite synkronisering)
10K	0,03	0,9	0,9
50K	0,02	6	0,9
100K	0,04	12	1
500K	0,2	63	1
1M	0,4	125	1,5
5M	3	625	3
10M	5	1239	4
50M	25	6286	15
100M	52	13269	29

Tabell 6: *Resultatene av kjøretidstestene av en sekvensiell og to parallelle versjoner av "finn summen av et array av tall". Disse kjøretidene er basert på en Intel core I7 870 prosessor med en klokkefrekvens på 2,93 GHz, med 8 kjerner (4 fysiske med hyperthreading).*

Vi kan se at Parallell 1 er den store taperen i denne testen. Dette er den parallelle, som bruker synkroniserte metoder mye, og dette gjør at kjøretidene blir dårligere. Ved dette eksempelet kan vi si parallellisering i seg selv ikke nødvendigvis er bra. Mye ligger i hvordan vi velger å parallellisere. Bruke mindre synkroniserte metoder kan gjøre at vi kan oppnå bedre kjøretider. Til sammenligning ser vi at Parallell 2 med lite synkronisering gjør det bedre enn den sekvensielle for større antall tall.

5.5 Delte variabler i Java PRP

5.5.1 Inkludere disse ideene i Java PRP

Bortsett fra kompleksiteten rundt dette med å generalisere delte variabler, kommer vi også bort i brukervennligheten. Nemlig hvor mye kan vi kreve at brukeren av Java PRP skal kunne om delte variabler. Det vil være nyttig for Java PRP om vi hadde inkludert en form for kommentar som sier at her kommer en delt variabel.

Vi ønsker samtidig at Java PRP skal bruke minst mulig synkroniserte metoder. Dersom vi kan skape lokale varianter av de delte variablene, kan vi også muligens oppnå en speedup med programmet vårt. Bare det å inkludere synkronisering vil ikke gi oss det vi vil. Java PRP vil ikke generere egne synkroniserte metoder for de delte variablene.

I Java PRP vil hver tråd, som arbeider på et subtre i rekursjonen, basere seg på parametersettet sitt. På denne måten vil vi oppnå at hver tråd arbeider på sine lokale kopier av variablene sendt til metoden. Dersom man bruker globale variabler, utenfor selve rekursjonsmetoden, må dette være variabler, som ikke endres i løpet av programmet.

5.6 Bug funnet i Java

5.6.1 Bug med System.nanoTime()

Mens jeg lagde dette eksempelet, om å finne summen av en mengde, møtte jeg et problem med tidsmålingene. Jeg oppnådde 0 nanosekunder i kjøretid, etter hvert som Java optimaliserte kjøringene, som virket merkelig da jeg kjørte en måling med *System.nanoTime()* før og etter kallet på rekursjonsmetoden, som beregnet summen. Et eksempel på dette er et "finn sum" program der vi har en mengde på 9000 tall:

Kjøringsnummer	kjøretid(ns)
1	241040
2	261301
3	270733
4	253616
5	236149
6	17816
7	34933
8	31091
9	4890
10	4541
11	4891
12	4542
13	4192
14	4192
15	0
16	0
17	0
18	0
19	0
20	0

Tabell 7: *Kjøretider i nanosekunder for "finn største" program, for en mengde på 9000 elementer. Ved den 15. kjøringen vil vi få 0 nanosekunder, som kjøretid. Dette vil være en feil. Disse kjøretidene er basert på en Intel core I7 870 prosessor med en klokkefrekvens på 2,93 GHz, med 8 kjerner (4 fysiske med hyperthreading).*

Tidsmålingene ble gjort slik:

```
long start = System.nanoTime();
long sum = new FinnSum().summer(arr);
long timeTakenNS = System.nanoTime() - start;
```

Etter at min veileder, Arne Maus, rapporterte dette til Oracle, fikk han følgende tilbakemelding:

Emne: Your Report (Bug ID: 9006037) - Call to System.nanoTime() optimized away ?
Dato: Fri, 16 Aug 2013 03:32:13 -0600 (MDT)
Fra: Bug-Report-Daemon_WW@ORACLE.COM <Bug-Report-Daemon_WW@ORACLE.COM>
Til: arnem@ifi.uio.no

Dear Java Developer,

Thank you for reporting this issue.

We have determined that this report is a new bug and have entered the bug into our bug tracking system under Bug Id: 9006037. You can look for related issues on the Java Bug Database at http://bugs.sun.com.

We will try to process all newly posted bugs in a timely manner, but we make no promises about the amount of time in which a bug will be fixed. If you just reported a bug that could have a major impact on your project, consider using one of the technical support offerings available at Oracle Support.

Thanks again for your submission!

Regards,
Java Developer Support

Det viser seg at dette faktisk er en bug i Java. Uten at vi har fått svar tilbake om hva eksakt problemet er, eller om det er løst, kom vi frem til at det muligens kan være at kallet på *System.nanoTime()* ble optimalisert bort, slik at startkallet og sluttkallet kom rett etter hverandre, istedenfor å inkludere kallet på rekursjonsmetoden vår. Vi klarte å fikse problemet ved å legge til en ekstra programsetning i mellom startkallet og sluttkallet på *System.nanoTime()*, som for eksempel en printsetning eller legge til en variabel som bruker verdien vi fikk tilbake fra rekursjonsmetoden.

I tiden før masteroppgaven ble ferdiggjort viste det seg at en løsning kom. Etter omtrent et halvt år senere, fra dette ble oppdaget, ble denne feilen tilsynelatende rettet opp av Oracle, i Java 8. Det skal nå være mulig å kjøre dette programmet korrekt.

6. Parametersett i Java

6.1 Rekursjon er grunnlaget for Java PRP

6.1.1 Parametre er en viktig del av rekursjon

Siden Java PRP baserer seg på rekursjonsmetoder, vil vi i dette kapittelet se hvordan parametre oppfører seg i rekursjon. Parametre kan oppføre seg forskjellig basert på hvilke type variabler de er. Rekursjonsmetoder vil ofte ha en progresjon gjennom parameteroverføring, instansene skilles i parametrene. Vi ønsker å se på hvordan de forskjellige instansene av rekursjonen påvirker hverandre, i forhold til parametre.

6.2 Forskjellige typer parametre

6.2.1 Mekanismer for parameteroverføring

Boka *Compiler Construction - Principles and Practice*[10] gir oss et godt innblikk i hvordan parametre fungerer. Parametre er verdier, som overføres til en metode eller en konstruktør, og spiller en viktig rolle for hvordan Java PRP programmet skal håndtere delte variabler. Parametre kan tolkes på flere forskjellige måter. De to vanligste er:

1. *pass-by-value*
   Dersom parameterøverføringen følger en slik mekanisme, vil det si at disse parametrene fungerer som kopier av de originale verdiene. Dette betyr at endringer i disse verdiene ikke påvirker de originale verdiene utenfor metoden.

2. *pass-by-reference*
   Denne mekanismen er på mange måter den motsatte varianten av *pass-by-value*. Her sendes parameteren, som en adresse til hvor parameteren er lagret i minnet. Dette betyr at endringene i metoden lagres i denne adressen. Dette fører til at endringer i metoden varer utover metoden.

Ofte er det slik at et programmeringsspråk gjør det mulig å velge hvilken mekanisme man vil ha. For eksempel, kan man velge pass-by-reference i C++, ved å sette en '&' foran parameternavnet, slik:

$$void\ minFunksjon(int\ \&x);$$

I noen språk er kun én mekanisme mulig. I FORTRAN77 brukes kun pass-by-reference. Siden Java PRP er laget av, og for, Java programmer, vil det være mest nyttig å se hvordan Java håndterer parametre.

6.2.2 Primitive typer i Java

Oracle[16] har selv forklart hvordan de primitive typene fungerer. I metoder, som tar i mot primitive typer som parametre, vil disse bli overført med *pass-by-value*. Det vil si at endringer på en slik verdi vil kun være synlig inne i metodeblokken, og ikke ha en påvirkning på verdien utenfor.

6.2.3 Objektspekere og arraypekere i Java

Videre, hos Oracle[16], har de også informasjon om objektspekere og arraypekere. Objektspekere er også overført med pass-by-value. Dette betyr at endringer av pekeren i metoden vil ikke påvirke den originale pekeren, som ble sendt som parameter i kallet.

Derimot skiller objektspekere seg fra de primitive typene ved at endringer i objektets elementer kan endres i metoden og være persistente igjennom metoden og tilbake til kallet. Dette vil si at endringer på pekeren varer ikke utover metoden, men endringer i objektets innhold vil vare. En eksempelmetode som viser forskjellen:

```
/**
*En eksempelmetode som tar imot en peker
*av type Eksempel, som har to tallverdier (verdi1
*og verdi2). Metoden setter disse to verdiene
*og deretter endrer pekeren e til et nytt objekt
**********/
void endreEksempel(Eksempel e){
    e.verdi1 = 2;       //vil vare utover metoden
    e.verdi2 = 3;       //vil vare utover metoden
    e = new Eksempel();  //vil ikke vare utover metoden
}
```

Arraypekere i Java fungerer på en lignende måte som andre objektspekere. Arrayer er objekter, som inneholder en gitt mengde av én type objekt eller primitiv type. På samme måte som andre objektspekere vil ikke endringer i arraypekeren vedvare, men man kan endre på hva de forskjellige verdiene. Dersom det er et array av objektspekere kan vi endre disse pekerne.

Slik kan vi se at Java bruker alltid pass-by-value som overføringsmekanisme. Java har heller ikke en variant, som C++, slik at vi kan velge å bruke en annen form dersom vi ønsker dette.

6.3 Parametre og delte variabler

6.3.1 Delte primitive typer

For metoder, der parametrene består av primitive variabler, ser vi at de fungerer som kopier. Dette vil også gjelde i et parallelt perspektiv. Dersom vi har en metode, som vi kjører i parallellitet med flere tråder, vil hver tråd skrive til sine egne kopier av disse primitive parametervariabler. Alle endringer, som gjøres i metoden, vil ikke vare utover metodeblokken.

6.3.2 Delte pekere

Det som gjelder for primitive typer gjelder for pekere også. Dersom en metode, som endrer en objektspeker, kjøres av mange tråder samtidig vil ikke endringen av pekeren påvirke de andre trådene.

Derimot vil endringer i objektenes innhold vare utover metoden. Dette gjør at objektspekere, som en del av parametrene, kan føre til at tråder overskriver variabler. Derfor må vi beskytte disse variablene for at parallelle eksekveringer ikke tilegner verdier til variabler, som den ikke skal ha. For eksempel, at en tråd leser utdaterte verdier, grunnet at en annen tråd skriver til samme variabel rett etter lesingen. Denne beskyttelsen får vi ved, for eksempel, å bruke synkroniserte metoder.

6.4 Parametre i rekursjon

6.4.1 Parametre håndteres forskjellig

For primitive typer vil det opprettes kopier for hver instans. Det vil si at variabelen ikke er avhengig av andre instanser i rekursjonstreet. Dette kan være fordelaktig for programmer der vi vil ha mer lokalt arbeid for hver tråd, og dermed mindre samarbeid med andre tråder. For programmer der vi ønsker at de skal oppføre seg som samme variabel, må vi legge til ekstra kode for å oppnå denne effekten.

Objektspekere kan være avhengig av andre instanser i rekursjonstreet, dersom det gjøres endringer i objektets innhold. Dersom man sender en objektspeker som en parameter, er det stor sannsynlighet for metoden skal bruke innholdet til objektet.

7. Egnede metoder for Java PRP

7.1 Begrensninger av Java PRP

7.1.1 kommunikasjon mellom tråder

I flere forskjellige typer parallelle metoder kreves det kommunikasjon mellom trådene. For eksempel, gjelder dette der alle tråder arbeider på felles data. Dette gir et godt utgangspunkt for å finne ut hvilke type rekursjonsmetoder, som er egnet til å parallellisere automatisk ved å bruke Java PRP. Ofte finner man ut at parallellisering ikke nødvendigvis er den riktige veien å gå dersom man vil ha en raskere eksekvering av sitt program. Dette gjør at vi kan begrense omfanget til Java PRP til visse metoder som følger gitte strukturer og er fornuftige å parallellisere.

7.1.2 Rekursjonsmetoder uten delte variabler

Noen rekursjonsmetoder er godt egnet til parallellisering fordi de ikke trenger å dele variabler når flere tråder gjennomkjører. Dette betyr at alle trådene kan prosessere for seg selv til de er ferdig, uten å måtte tenke på andre tråder og hvordan de opererer på sine variabler. Det er altså ingen krav til kommunikasjon mellom trådene mens de gjør sine oppgaver.

Et eksempel på en slik rekursjonsmetode er Quicksort. Her vil rekursjonen dele opp i to nye instanser som er uavhengig av hverandre. Den ene instansen vil ta seg av de mindre verdiene, og den andre vil ta seg av de større. Dette gjør at en tråd trenger kun å tenke på sine egne tall, og ingen andres slik at vi ikke trenger å synkronisere med andre tråder for å komme frem til en sortert mengde tall.

Slike metoder er gode kandidater til å parallelliseres av Java PRP. Dette er på grunn av at de både egner seg til parallellisering, og de er ganske enkle å lage med tanke på at de ikke trenger noen synkroniseringmetoder for å fungere.

7.1.3 Noen metoder egner seg ikke til parallellisering

Det finnes metoder som krever full tilgang til datastrukturene i programmet til enhver tid. Dette gjør det vanskelig å få til en god parallellisering av programmet fordi alle tråder må sjekke de delte variabler ofte. Det at vi må bruke synkroniseringsmetoder ofte vil føre til veldig dårlig kjøretid, slik at vi kan bruke den sekvensielle versjonen istedenfor.

Dette fører til konklusjonen om at ikke alt er nødvendig for Java PRP å ta for seg. Dersom parallellisering av denne typen metoder ikke fører til bedre kjøretider, er det bedre å la være. Derimot er det ikke slik at et program med delte variabler er umulig å parallellisere effektivt. Det finnes former for metoder slik at vi har delte variabler der vi har muligheten til å optimalisere bruken av synkronisering. For eksempel, er det metoder der vi ønsker at trådene samkjøres etter visse faser er over. Tråder arbeider selvstendig i en fase, for å samarbeide med de andre etterpå.

Java PRP kan ta for seg noen varianter av rekursjonsmetoder som inneholder delte variabler. Dette vil hovedsakelig gjelde metoder som har muligheten til å forbedres ved parallellisering. Disse variantene må også være mulig å formidle videre til en bruker, slik at den personen vet hvilke metoder som kan, og ikke kan, parallelliseres via Java PRP.

7.1.4 Kjennetegn til egnede metoder

Java PRP kan ta for seg metoder som ikke bruker delte variabler, som Quicksort nevnt i 7.1.2. Her vil instansene av rekursjonsmetoden ikke måtte kommunisere med hverandre for å gi fra seg et svar. Instansene kan jobbe på egne data, for og gi fra seg sitt svar til instansen over i rekursjonstreet.

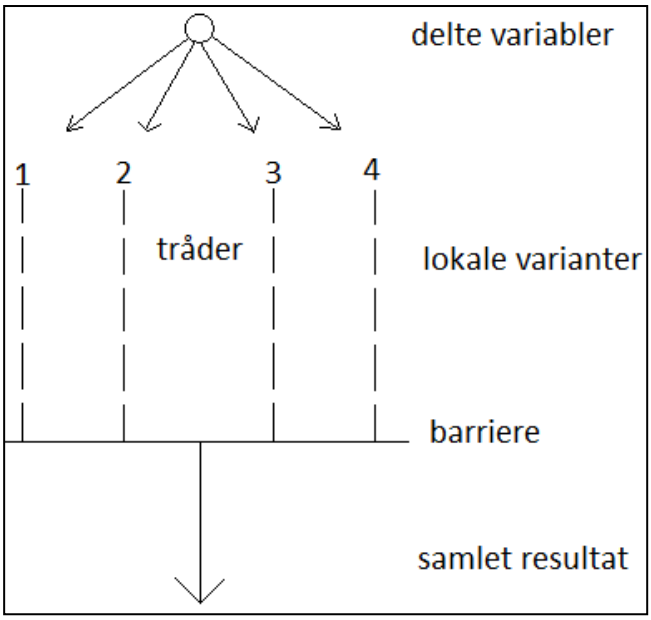
For å ta et eksempel på en type metode, med delte variabler, som kan egne seg til parallellisering, er de som vi kan dele opp i faser. Det vil si at vi lar tråder arbeide selvstendig i forskjellige faser, og vi kan deretter sette sammen deres lokale resultater mellom disse fasene sekvensielt. Dersom vi ikke har mange faser kan vi oppnå en speedup ved parallellisering.

7.2 Former for egnede metoder med delte variabler

7.2.1 La trådene arbeide lokalt

Vi kan se på de delte variablene som lokale kopier for hver tråd inntil et visst punkt. Senere i eksekveringen må vi ende opp med en samlet verdi for de delte variablene. Det vil si at vi må bruke alle trådenes lokale varianter for å komme frem til et samlet resultat. Dermed kan vi si at denne løsningen går i to faser.

1. Den første går ut på at vi kjører programmet parallelt hvor de delte variablene er gitt som lokale varianter til hver tråd. Her kreves det ingen kommunikasjon mellom trådene.
2. Ved for eksempel en barriere, kan vi finne ut når trådene er ferdig med første fase. Nå kommer den sekvensielle delen hvor vi kommer frem til et samlet resultat basert på de lokale variantene.

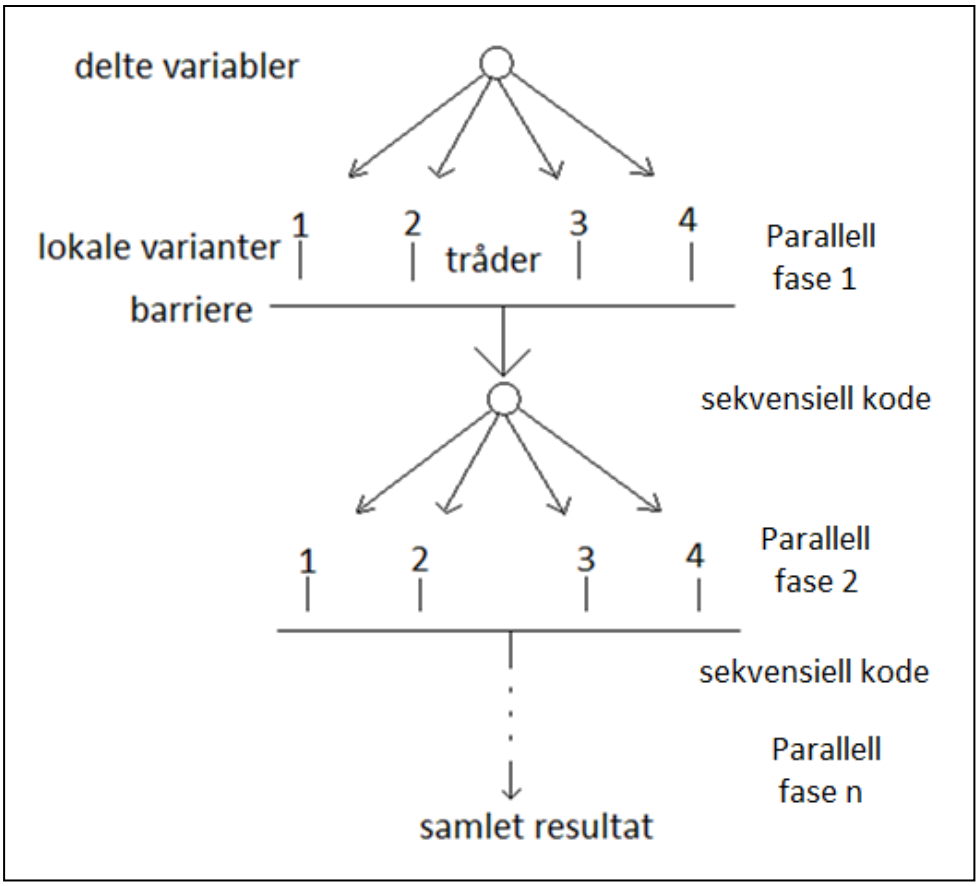


Figur 13: *Et program med delte variabler der hver av de fire trådene får lokale varianter av disse variablene. Hver tråd arbeider selvstendig i parallellitet. Når alle har truffet barrieren kan alle trådene gi fra seg et samlet resultat basert på sine lokale varianter. Dette vil gjøres sekvensielt.*

Å lage parallelle programmer slik, framfor å synkronisere mens trådene holder på, er fordelaktig. Her vil vi la trådene arbeide selvstendig, slik at vi kan oppnå bedre kjøretider. Ved å ha en sekvensiell fase helt på slutten har vi eliminert mye av problemstillingen ved delte variable. I denne sekvensielle fasen har vi mulighet til å ordne de delte dataene. Dessverre fungerer ikke dette på alle programmer, som inneholder delte variabler.

7.2.2 Arbeide parallelt i faser

Vi kan utvide ideen i 7.2.1 dersom vi har metoder, som krever større innsyn i de delte dataene mens eksekveringen holder på. Ved å legge til egne faser for parallellisering, kan vi la trådene arbeide selvstendig slik at vi utnytter parallelliseringen mest mulig. Dette gjøres så lenge metoden tillater det, men til et visst punkt må vi synkronisere de delte dataene mellom trådene. Dette kan gjøres i en sekvensiell fase, som etterfølger hver parallelle. Når vi er ferdig med dette kan vi fortsette til neste parallelliseringsfase. Slik fortsetter det til vi er ferdig med programmet og kan gi fra oss et svar.



Figur 14: *Et program, som kjøres parallelt i flere faser. I en parallelliseringsfase eksekverer trådene uavhengig av de andre, og disse fasene er alltid etterfulgt av en sekvensiell fase.*

Dette gjelder bare noen former for metoder og er avhengig av at mesteparten av arbeidet gjøres inne i parallelliseringsfasen. Dersom vi ender med mange små faser ender vi med mange sekvensielle faser og resultatet er dårlige kjøretider. Derfor er denne formen for parallellisering avhengig av at det ikke er nødvendig med mange sekvensielle faser. Det kreves også at trådene har muligheten til å arbeide selvstendig i parallelliseringsfasene. En fordel med slik faseoppdeling er at man kan la resultatene fra en fase leses, men ikke skrives til, av de neste fasene. Dette gjør at man legger opp til at man kan fortsette beregningene basert på hva man allerede har funnet ut. Man tillater også at alle trådene kan lese disse resultatene uten å synkronisere mellom hverandre, fordi de ikke vil skrives til lenger.

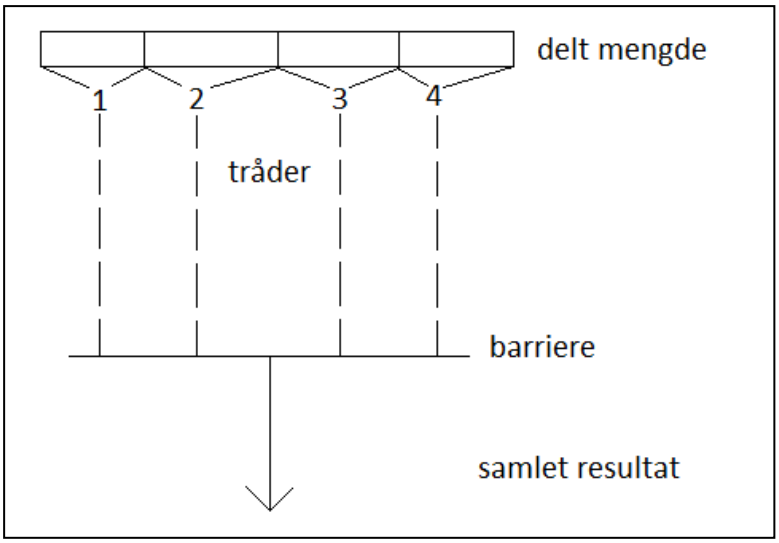
7.2.3 Delte variabler i form av mengder

Istedenfor at de delte variablene kommer som enkle typer, som tall, strenger og så videre, kan de ofte ende opp som hele mengder. Dette gjør det vanskelig å gjøre som i 7.2.1, fordi vi vil at trådene skal arbeide på hver sin del av mengdene, ikke arbeide på lokale varianter av hele mengden. Derfor kan det være fordelaktig å la hver tråd få sin del av mengden, og kun operere på denne.

I slike situasjoner, hvor trådene arbeider på samme mengde og hver tråd er begrenset til hver sin del av den, er det viktig å tenke på skrive- og lesesettet til hver tråd. Dersom hver tråd har muligheten til å skrive til alle elementene i sin del av mengden, kan vi ikke la andre tråder lese eller skrive på denne delmengden. Det vil ikke være nødvendig å ha noen form for kommunikasjon mellom trådene her.

Slik som vi hadde i 7.2.1, vil det her være to deler av gjennomkjøringen av programmet.

1. Hver tråd får gitt sitt intervall i den delte mengden. Innenfor denne mengden har de rettighetene til både å skrive og lese til elementene. De kan derimot ikke arbeide på andre tråders delmengder.
2. Ved å bruke en barriere kan vi senere gi et samlet resultat, enten dette er mengden i seg selv eller kalkulasjoner basert på denne.



Figur 15: *Et program der de fire tilgjengelige trådene arbeider på forskjellige deler av den delte mengden. Når alle trådene treffer barrierepunktet kan de alle lese hele mengden, men ikke skrive. De gir også fra seg et resultat basert på sine egne kalkulasjoner.*

Slik som vi gjorde i 7.2.2 kan dette utvides til flere faser. Det vil si at vi lar disse to punktene nevnt ovenfor være en fase av flere. Samme konklusjon vil vi ende opp med her, altså at for at dette skal fungere trenger vi å ha minst mulig synkronisering, og lengre parallelliseringsfaser.

7.2.4 Lage en sekvensiell kallmetode

De foregående eksemplene i 7.2.1 til 7.2.3 tar for seg hvordan den rekursive metoden kan være for en parallell fase. Det kan også vise seg nyttig å se på selve oppstarten av disse fasene. Å lage en egen metode for kallsekvensen kan gi oss flere fordeler.

Vi kan bruke denne metoden som en sekvensiell oppskrift på hvordan utførelsen av det ellers parallelle programmet vil fungere. I programmer der vi arbeider i faser vil vi kontrollere flyten i denne metoden. Et eksempel på en slik kode er:

```
/*
*Metode som kontrollerer
*kallseksenvensen til
*det parallelle programmet
************************/
void kallSekvens(){
    <start parallell fase 1>
    <start sekvensiell fase 1>
    <start parallell fase 2>
    <start sekvensiell fase 2>
    ...
    <start parallell fase n>
    <start sekvensiell fase n>
}
```

Å bygge programmet vårt rundt en slik metode kan også øke brukervennligheten til Java PRP. Siden metoden viser programmets sekvensielle gjennomgang, vil det være lettere å konstruere en slik selv om man ikke har mye erfaring med parallellisering.

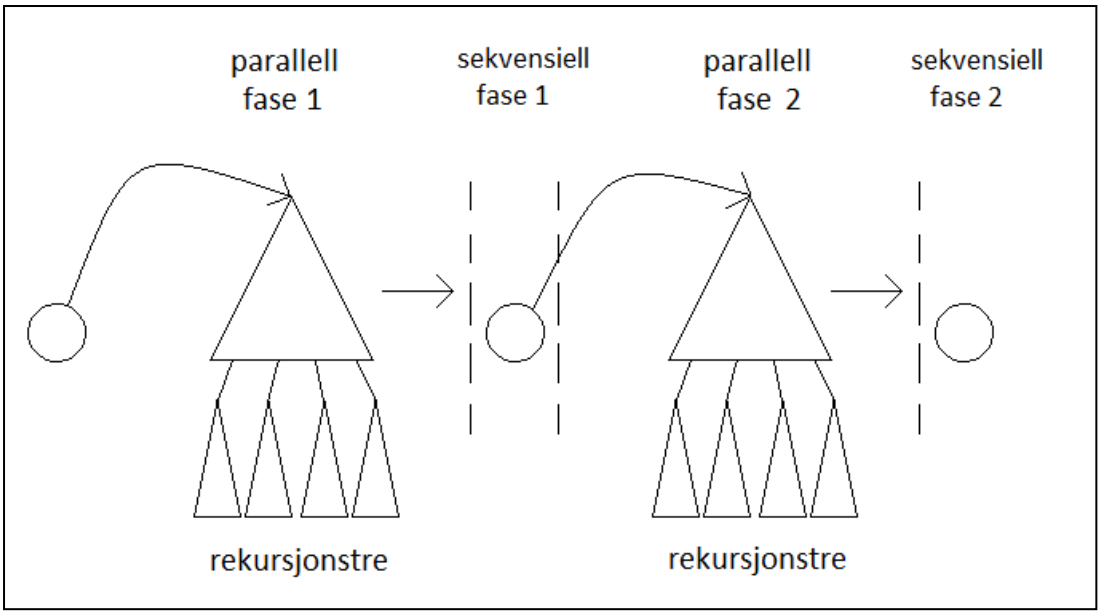
7.3 Inkludere formene i Java PRP

7.3.1 En faseoppdelt Java PRP

Av ideene, som vi la frem i 7.2, kan vi kombinere de til en form for en egnet metodestruktur. Denne formen vil bestå av en faseoppdeling, som betyr at Java PRP vil kunne kjøre parallelle oppgaver i flere faser, med sekvensielle faser i mellom dem. De parallelle fasene består av rekursjonsmetoder, der vi kjører et antall subtrær av rekursjonstreet i parallell, altså de er ganske like standard Java PRP for én rekursjonsmetode. Det viktige er at vi kan kjøre hver tråd helt uavhengig av hverandre. Det vil være her vi eventuelt oppnår en speedup.

Vi fortsetter også på ideen med å arbeide på mengder, som vi så i 7.2.3, fordi dette både er vanlig for mange metoder og dette passer inn i en faseoppdelt Java PRP. Det vil si at trådene arbeider på hver sin del av mengden. Her er det viktig at hver tråd har kun lese- og skriverettigheter til sin egen delmengde, samt at alle har leserettigheter til data, som ikke endres av noen av de andre trådene. For eksempel, resultatene fra en fase kan leses av videre faser, men ikke skrives til. I rekursjonsmetoder blir det vanlig at parametre setter grensene for hvor nåværende instans kan arbeide på i mengden, for eksempel ved to indekser som angir start og slutt på delmengden. Dette gjør at metoden i seg selv tar seg av oppdelingen av mengden, slik at dette blir abstrahert fra faseoppdelingen.

Etterfulgt av hver fase er en sekvensiell fase. Det er bare maintråden, som kjører kode i denne fasen, slik at det er ingen parallelle oppgaver her. Trådene ligger på vent til neste parallelle fase. En slik sekvensiell fase kan etterfølges av en ny, parallell fase, eventuelt at programmet er ferdig, og vi er klare til å gi fra oss resultatet av hele programmet. Det er tilfeller hvor vi ikke trenger en sekvensiell fase. Det skal være mulig å unngå en sekvensiell fase om det er ønskelig.



Figur 16: *Flytdiagram for et faseoppdelt program med to faser. Fasene starter med en parallell fase, som inneholder en rekursjonsmetode, som deler rekursjonen i fire subtrær for trådene. Dette etterfølges av en sekvensiell fase.*

7.3.2 Innholdet i de parallelle fasene

Det er flere måter vi kan se på hva som skal være med i hver parallelliseringsfase. Det viktigste er at trådene må få arbeide selvstendig. Dette er på grunn av at dette gir best utnyttelse av parallelliseringen. Dette betyr også at en mulig forbedring, i kjøretid, med et parallelt program kommer an på hvor mange faser vi har. Jo flere faser vi har, jo mer må vi bryte opp parallelliseringen.

Hver parallelliseringsfase vil basere seg på hver sin rekursjonsmetode. Her vil hver metode kunne ha flere tråder i arbeid. Dette vil ligne det vi allerede har implementert i Java PRP, nemlig de problemene som er enkelt parallelliserbart. Dette vil være en utvidelse ved at vi gjør arbeidet i flere faser, der de delte variablene kan settes sammen mellom hver fase. Dette ligner på Figur 16, fordi vi lar hver rekursjon fullføre før vi fortsetter videre til den sekvensielle fasen. Slike programmer kan lettere gi en speedup, fordi her vil alle rekursjonsmetodene kunne arbeide selvstendig, da dette vil være et krav for Java PRP. Det legges opp til at resultatet fra hver fase er kun lesbart for de videre fasene.

7.3.3 Overgang til den sekvensielle fasen

For å inkludere sekvensielle faser trenger vi en form for barriere slik at vi vet når vi skal vi kan begynne denne fasen. For eksempel, kan dette gjøres ved at vi fører et barriereobjekt til hvert arbeiderobjekt. På denne måten vil vi kunne la trådene si ifra når de er ferdige. Slik gjøres det allerede i Java PRP for programmer med én rekursjonsmetode, sånn at vi vet når rekursjonens parallelle del er ferdig, og vi kan fortsette til roten av rekursjonen.

Slik som i 7.2.4, kan vi inkludere en metode som står for den sekvensielle gjennomgangen av det parallelle programmet. Denne kan vi utvide slik at den fungerer som en administrator for hele programmet. Det vil si at denne metoden bestemmer når den neste fasen starter. Barrieresynkroniseringen vil lede tilbake igjen til denne metoden, som vil starte opp den sekvensielle fasen. En slik metode kan også gjøre det lettere for en bruker av Java PRP, til å se hvordan utførelsen i det parallelle programmet er. Dette vil bli et krav, da det blir brukeren som skriver denne sekvensielle "oppskriften" på programmet.

8. Utvidelse av Java PRP: faseoppdelte programmer

8.1 Programstruktur

8.1.1 Et faseoppdelt program

Java PRP vil kunne parallellisere flere rekursive metoder i et program, i flere faser. Hver slik overordnet fase er todelt, da vi har en parallell fase og en synkroniseringsfase tilknyttet en rekursjonsmetode.

Den parallelle fase er der selve rekursjonsmetoden blir utført. Det vil her bli satt tråder til å gå igjennom metoden, slik at vi får den parallelle eksekveringen vi ønsker. Disse parallelle fasene er veldig like det vi har gjort tidligere i Java PRP, der vi har parallellisert kun én rekursiv metode per program.

For at vi skal unngå å måtte synkronisere i selve metoden, legger vi opp til at man kan håndtere delte variabler etter denne parallelle fasen. Dette gjør at vi unngår å måtte stoppe opp midt i den parallelle eksekveringen. Denne fasen blir kjørt sekvensielt av maintråden og dermed ikke i parallellitet. Dette er en fase, som vi kan bruke til, for eksempel, å spre lokal informasjon funnet i rekursjonen til global informasjon.

8.1.2 Administratormetode

Det grunnleggende bak faseoppdelingen ligger i en metode, som fungerer som en administrator for programmet. Denne metoden starter opp de parallelle og de sekvensielle fasene. Metoden vil avslutningsvis returnere resultatet til hele programmet, som fasene arbeider seg frem til.

Denne metoden må brukeren av Java PRP skrive, og er et krav for at Java PRP skal kunne generere et faseoppdelt og parallelt program. Det følger visse krav til hvordan man skal lage en slik metode. Det første er at metoden må ha en returverdi, som vil være det faktiske resultatet av faseoppdelingen. Den andre går ut på hvordan vi syntaksmessig setter opp kallene til de forskjellige fasene i programmet. Java PRP prosesserer disse kallene ved å se på to linjer om gangen i denne metoden. Den første vil være kallet på rekursjonsmetoden, og neste vil være kallet på metoden, som representerer den sekvensielle fasen. Dette gjøres for alle fasene til vi returnerer

svaret. Dette gjør at denne administratormetoden må følge en ganske fast struktur med to og to linjer per fase.

Administratormetoden gir et løfte tilbake til brukeren. Den lover nemlig at denne metoden vil kjøres sekvensielt. Den vil starte opp parallelle faser, men vil ikke fortsette til neste fase før forrige er ferdig. Dette gjør at brukeren har større kontroll over programflyten, samtidig som vi kan utnytte effektiviteten vi kan få fra en parallell eksekvering.

Et eksempel på en administratormetode:

```
/*ADMIN*/
public int minAdminMetode(...){
    int svar = rekursivMetode1(...);
    sekvensiellFase1();
    svar = rekursivMetode2(...);
    sekvensiellFase2(...);
    svar = rekursivMetode3(...);
    sekvensiellFase3(...);
    svar = rekursivMetode4(...);
    sekvensiellFase4(...);
    return svar;
}
```

Eksempelet ovenfor gir en administratormetode, som tar for seg fire faser, der alle fasene inneholder en parallell fase og en sekvensiell fase.

Av og til hender det at vi ikke trenger en sekvensiell fase. Vi kan utnytte ideen med at sekvensielle faser er ubehandlet av Java PRP ved å la linjen, som representerer synkroniseringsfasen, stå tom eventuelt med en kommentar dersom man ikke trenger en slik fase. Ved å endre forrige eksempel, kan vi få en administratormetode, der vi ikke trenger en sekvensiell fase for andre og fjerde fase, til å se slik ut:

```
/*ADMIN*/
public int minAdminMetode(...){
    int svar = rekursivMetode1(...);
    sekvensiellFase1();
    svar = rekursivMetode2(...);
    //ingen sekvensiell metode nødvendig
    svar = rekursivMetode3(...);
    sekvensiellFase3(...);
    svar = rekursivMetode4(...);
    //ingen sekvensiell metode nødvendig
    return svar;
}
```

8.1.3 Nye nøkkelord

For at Java PRP skal kunne starte og generere et faseoppdelt program trenger vi å utvide settet med nøkkelord. De nye vil være:

- /*ADMIN*/
  Denne gjør at det er mulig for Java PRP til å finne ut hvor administratormetoden vår befinner seg. Den brukes også til å finne ut om dette er et faseoppdelt program eller ikke.

- /*FUNC x*/
  Tidligere har vi hatt nøkkelordet /*FUNC*/ for å finne den rekursive metoden, som brukeren ønsker å ha parallellisert. Dette må nå utvides på grunn av at vi nå har flere rekursive metoder. Derfor må dette nøkkelordet si hvilken fase den er del av. Det vil si at den første rekursive metoden, som skal parallelliseres av Java PRP, vil få nøkkelordet /*FUNC 1*/, og neste /*FUNC 2*/ og så videre.

Bortsett fra at endringen av /*FUNC*/ vil de andre nøkkelordene fortsatt bestå slik de har gjort tidligere, for løsningen av én rekursjonsmetode.

8.1.4 Flere arbeiderklasser

De parallelle programmene Java PRP genererer, som ikke er faseoppdelt, vil alltid ha kun én arbeiderklasse. Dette kommer av at vi har kun én rekursiv metode å parallellisere. Vi trenger flere arbeiderklasser, fordi hver parallelle fase gir opphav til en ny arbeiderklasse. Disse klassene vil ha ganske lik struktur, nemlig at de har en metode for å generere subtrær i rekursjonen, samle inn svar, en run()-metode også videre.

Som et resultat er det ikke nok å kalle en klasse for *Worker* lenger, da vi har flere arbeiderklasser. De nye klassene vil få et nytt navn, basert på det originale klassenavnet i det sekvensielle programmet, nemlig *<originalt klassenavn>_worker*. Dette kan også ha en fordel av at brukeren gjenkjenner mer av sitt eget.

8.1.5 Muligheten til å la rekursjonsmetoder starte fra flere hold

Noe som ikke er direkte relatert til et faseoppdelt program, men viste seg å være nyttig er å la en rekursjonsmetode ha flere utgangspunkt. Til nå har Java PRP kun håndtert rekursive metoder der rekursjonstreet har én rot, slik at vi kan starte metoden fra ett kall. Av og til har vi oppgaver der vi ønsker å starte rekursjonen fra flere hold. Dette gjør at vi ikke trenger mer kompliserte rekursjonsmetoder for å oppnå dette, men heller abstrahere en annen type oppstart fra resten av rekursjonen. Mye av kravene for rekursjonsmetodene vil også gjelde for disse oppstartsmetodene, som for eksempel at kallene på rekursjonsmetoden må ha en /*REC*/ kommentar over seg.

Dersom en bruker ønsker å bruke dette, trenger ikke Java PRP å vite dette på forhånd. Dette kommer av at når administratormetoden kommer til et kall på en ny, parallell fase, har den informasjonen om hvilke kall den forventer skal komme. Denne informasjonen får den fra det nye nøkkelordet over de rekursive metodene. Disse sier nemlig hvilken fase den har kommet til. Dermed kan den sammenligne forventet navn med faktisk navn på den forventede metoden. Dersom disse er ulike, vet Java PRP at den må innom en oppstartsmetode først, og håndterer oppgaven deretter. Dette gjelder

kun for faseoppdelte programmer. Denne oppstartsmetoden må også befinne seg i samme klasse, som administratormetoden.

8.2 Faseoppdelt adminklasse

8.2.1 Arbeidsoppgaver

Den faseoppdelte adminklassen vil erstatte mainklassen til det sekvensielle programmet. Den tar for seg eventuell oppstart av programmet, for eksempel dersom man ønsker å generere noen datastrukturer eller lignende, før den setter i gang administratormetoden. Det er i denne klassen vi har administratormetoden, slik at selve programflyten gjennom fasene styres her. Mye av arbeidet til Java PRP i denne klassen er å lete etter gamle pekere til de gamle klassene til de forskjellige fasene, og erstatte med nye, eventuelt fjerne dem. For rekursjonsmetode, som krever en oppstartsmetode for å settes i gang, vil slike oppstartsmetoder befinne seg her.

8.3 Faseoppdelt workerklasse

8.3.1 Arbeidsoppgaver

Alle arbeiderklassene vil strukturmessig være veldig like. De skilles kun av hvordan selve rekursjonsmetoden ser ut. De inneholder en metode for å skape barnenoder, selve rekursjonsmetoden, en metode for å generere svar fra denne instansen av rekursjonstreet samt resten av den originale klassen, som inkluderer dens metoder og variabler. Like viktig her er at hver instans i rekursjonstreet må kunne arbeide uavhengig av andre instanser, da det genererte Java PRP programmet baserer seg på at vi ikke skal ha synkroniserte metoder med. Dersom man trenger en form for samling av data, er de sekvensielle fasene en mulighet.

9. Java PRP eksempel 1: Finn største tall i en mengde

9.1 Finne største tallet i en mengde

9.1.1 "Finn største tallet" i form av rekursjon

Å finne største tallet er ofte representert i en løkkeform. Det vil si at vi går igjennom fra start til slutt, og finner største tallet i mengden. For at dette skal kunne gjøres i Java PRP må koden skal være rekursiv. Dette kan gjøre koden mindre effektiv enn normalt, men vi ønsker her heller å ha et enkelt eksempel for Java PRP.

9.1.2 Sekvensiell kode

Til tross for dens enkle natur viser den oss at kompleksiteten rundt parallellitet ikke nødvendigvis varierer basert på kompleksiteten til en kode. Den rekursive, sekvensielle koden baserer seg på at man skaper et array av en gitt lengde *n*, slik at alle verdiene er en uniformt fordelt mellom 0 og n-1.

Kodens fanout er to, slik at neste nivå i enhver rekursjon blir delt i to mindre problemer, som her vil være å dele mengden av tall som skal letes igjennom i to. Dersom vi treffer basistilfellet, som her vil være når vår mengde er mindre enn en global verdi *LIMIT*, som er lik 5, vil vi stoppe å rekursere videre. Vi vil på dette tidspunktet lete igjennom de få tallene vi har og returnere den største av dem.

```
/*FUNC*/
int find(int[] arr, int start, int end){
    int largest = 0;
    if((end-start) < LIMIT){
        for(int i = start; i < end; i++){
            if(arr[i] > largest) largest = arr[i];
        }
    return largest;
    }
    int half = (end-start) / 2;
    int mid = start + half;
    /*REC*/
    int leftVal = find(arr,start,mid);
    /*REC*/
    int rightVal = find(arr,mid+1,end);
    if(leftVal > rightVal) return leftVal;
    return rightVal;
}
```

9.2 Tester

9.2.1 Hvordan vi tester

For at Java PRP skal være brukbart, må den kunne gi oss bedre kjøretider enn det sekvensielle programmet. Derfor ønsker vi å kjøre noen tester. Testene baserer seg på at vi kjører en løkke i programmene på 11 ganger. Dette gjør at vi får flere tall å sammenligne med. Resultatet blir medianen av disse. Ved å kjøre alle 11 i en gjennomkjøring, vil vi kunne oppnå Javas JiT kompilering og optimalisering av koden. For alle tallene vil vi ha uniforme og tilfeldige verdier mellom 0 og n-1. I programmet blir det målt i nanosekunder, men det blir framvist i millisekunder. Testene baserer seg på en Intel core I7 870 prosessor med en klokkefrekvens på 2,93 GHz. Den har 4 fysiske kjerner, som er hyperthreaded slik at vi vil kunne utnytte dette som 8 kjerner.

9.3 Test 1: Java PRP

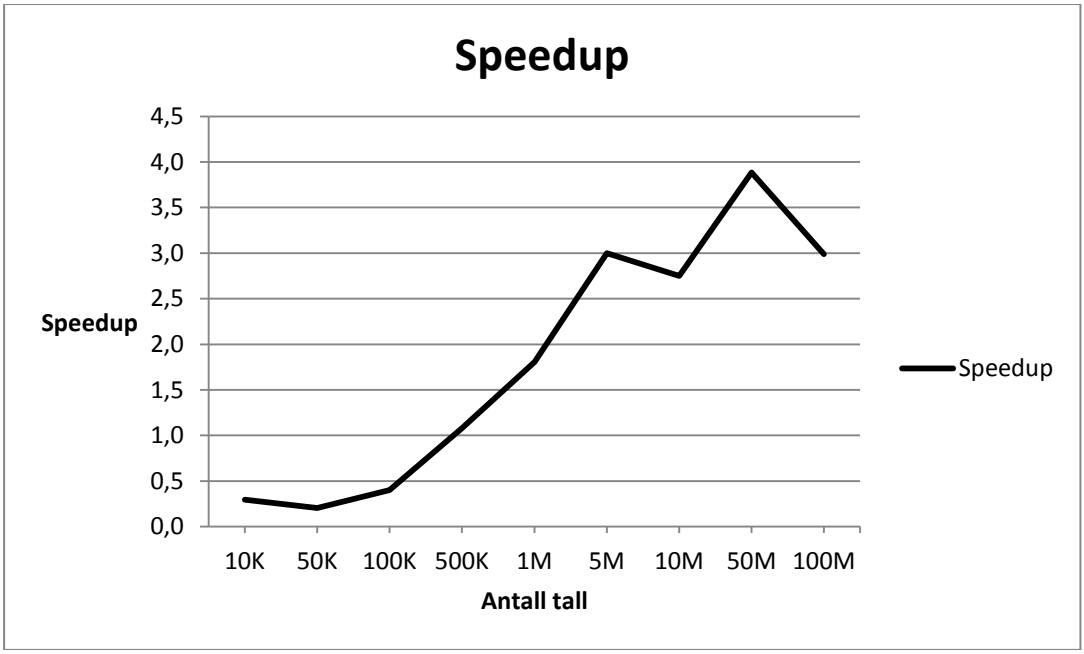
9.3.1 Resultater

Videre er tabellen som er resultatene for kjøringene av programmene. I tabellen vil K tilsvare tusen og M en million.

Antall tall	Kjøretider(ms) for sekvensiell "Finn største tall"(ms)	Kjøretider(ms) for Java PRPs parallelle "Finn største tall"(ms)	Speedup
10K	0,3	1,0	0,3
50K	0,2	1,1	0,2
100K	0,4	0,9	0,4
500K	2	1	1,1
1M	4	2	1,8
5M	15	5	3,0
10M	27	10	2,8
50M	211	54	3,9
100M	377	126	3,0

Tabell 8: *kjøretider til sekvensiell og parallell "finn største tall" i millisekunder. Disse kjøretidene er basert på en Intel core I7 870 prosessor med en klokkefrekvens på 2,93 GHz, med 8 kjerner (4 fysiske med hyperthreading).*

9.3.2 Speedup i en graf



Graf 3: *speedupen ved å bruke et parallelt "finn største tall" program, generert av Java PRP. Disse kjøretidene er basert på en Intel core I7 870 prosessor med en klokkefrekvens på 2,93 GHz, med 8 kjerner (4 fysiske med hyperthreading).*

Graf 3 forteller oss at den parallelle versjonen gjør det dårligere til vi er på mengder rundt over 500 000. Etter dette punktet blir programmet, som er generert av Java PRP, bare bedre og bedre.

9.3.3 Tolkning av resultatene

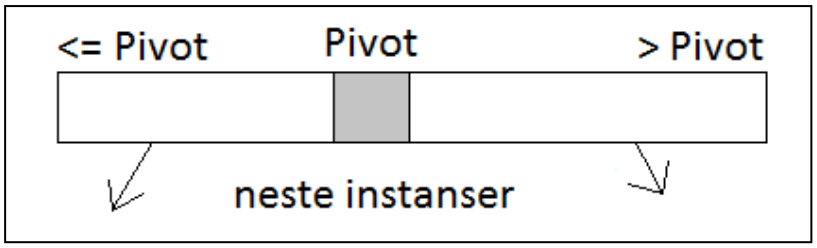
Av Tabell 8 kan vi kan se at Java PRP sin parallelle versjon blir bedre og bedre for høyere verdier, men på mindre verdier er den ikke bedre enn den sekvensielle. Grunnen til at den parallelle versjonen presterer dårligere for lavere verdier er den ekstra tiden, som følger Java PRP da vi må opprette tråder, generere arbeidere og sette dem i gang samtidig som vi har en sekvensiell resultatgenerering. Ettersom vi får flere verdier, kan vi parallellisere enda flere beregninger. Dette er grunnen til at den sekvensielle delen blir mindre og mindre i forhold til den parallelle delen jo flere verdier vi legger til i mengden tall. Derfor vil den parallelle versjonen bli bedre og bedre. Dette ser vi i dette tilfellet, hvor vi når opp mot 3,9 i speedup ved å bruke parallellitet for 50 millioner tall.

10. Java PRP eksempel 2: Quicksort

10.1 Quicksort algoritmen

10.1.1 Hvordan Quicksort fungerer

Quicksort er en sammenligningsbasert sorteringsalgoritme. Den fungerer slik at vi begynner med å velge et element i mengden vår, som vi kaller pivotelementet, og sorterer resten av mengden basert på dette elementet. Resultatet av dette er at vi ender med to delmengder på hver sin side av pivotelementet. Den venstre siden inneholder elementer mindre enn pivotelementet, mens den høyre inneholder større elementer enn pivotelementet. Disse delmengdene er ikke nødvendigvis sorterte, men vi kan gjøre samme operasjon på nytt for hver av disse mengdene igjen slik at vi til slutt ender på små mengder å sortere. Dette oppnår vi med rekursjon.



Figur 17: *En instans av Quicksort. Vi har valgt et pivotelement og sortert basert på dette elementet. Pilene indikerer neste instanser.*

10.1.2 Parallell Quicksort

Quicksort egner seg godt til å parallelliseres. En av grunnene kommer av at vi ikke får felles data. Dette betyr til at vi slipper å synkronisere i koden vår. Det å parallellisere Quicksort er ikke en ny idé. I boka "Rett på Java"[1] er det blitt gjort tester, som sammenligner kjøretider til en sekvensiell og parallell Quicksort. Den sekvensielle koden baserer seg på samme prinsipp, som forklart i 10.1.1. Først lager de et pivotelement og deler opp mengden basert på denne. Den parallelle koden er ganske lik, men de har lagt til kode for å starte nye tråder. For hver instans av Quicksort sjekker den om mengden den skal sortere er større enn en variabel "PARA_LIMIT", som tilsvarer 50 000. Dersom mengden er større starter instansen to nye tråder for de neste rekursjonskallene, slik at de kan arbeide parallelt.

I boka ser vi to tabeller, som viser forskjellen mellom den sekvensielle og parallelle Quicksort i kjøretider. Det er to tabeller fordi de ønsker å teste programmene på henholdsvis 8 og 64 kjerner. Basert på disse tabellene ser vi at de oppnår en god speedup ved å parallellisere Quicksort.

	10	100	1 000	10 000	100 000	1 mill.	10 mill.	100 mill.
Sekvensiell	0,01	0,05	0,70	13	18	89	963	11 196
Parallell	0,01	0,06	0,84	16	24	67	356	3579

Tabell 9: *"Antall millisekunder det tar å sortere en array med tilfeldig innhold med den sekvensielle og parallelle versjonen av Kvikksort (parallelt for deler som er lengre enn 50 000). Kjørt på en Intel i7 870, 3GHz klokke med 8 kjerner." [1]*

	10	100	1 000	10 000	100 000	1 mill.	10 mill.	100 mill.
Sekvensiell	0,01	0,05	0,81	18	21	197	1413	16 497
Parallell	0,01	0,06	0,99	21	56	106	1332	5025

Tabell 10*: "Antall millisekunder det tar å sortere en array med tilfeldig innhold med den sekvensielle og parallelle versjonen av Kvikksort (parallelt for deler som er lengre enn 50 000). Kjørt på en maskin med 4 Intel Xeon prosessor L7555, 1,87 GHz klokke, totalt 64 kjerner"[1]*

10.1.3 O-notasjon til Quicksort

I gjennomsnittlig O-notasjon bruker quicksort:

$$i\ gjennomsnitt = O(n \log n)$$

Første *n* kommer av at for hver instans av quicksort må vi sortere *n-1* tall basert på et pivotelement. Den resterende delen *log n* kommer av antall nivåer av quicksort vi må, gjennomsnittlig, løpe igjennom. I snitt vil pivotelementet havne nær midten av arrayet, slik at vi ofte retter oss inn mot to nesten like mengder, som vi fører videre i rekursjonskallene. Dybden til treet vil derfor rette seg mot *log n*. Dette gjør at i gjennomsnitt får $n \log n$ i O-notasjon. I verste tilfelle vil quicksort derimot være:

$$verste\ tilfelle\ = O(n^2)$$

Dette kommer av at vi kan være uheldig med valget av vårt pivotelement vil ende på starten eller slutten av den sorterte mengden. Dersom dette repeterer seg gjennom hele rekursjonen vil vi ende med *n-1* videre kall nedover. Dette kommer av at vi ikke deler rekursjonen i to lenger, men sender alle elementene, minus pivotelementet, videre ned. Vi sender altså ned *n-1* elementer for hvert rekursjonskall i dette tilfelle. Dette fører til at vi gjør $n \times (n-1)$ oppslag i algoritmen, som i O-notasjon blir n^2.

10.1.4 Basistilfellet til Quicksort

Quicksort er kjent for å være en god sorteringsalgoritme for middels store mengder. For mindre mengder har man ofte foretrukket andre sorteringsalgoritmer, som innstikksortering. Det vil si at vi kan sortere basert på innstikksortering når vi kommer på endene av rekursjonen, når mengdene er små. Dette vil være vårt basistilfelle for Quicksort algoritmen.

Et langsommere alternativ til å bruke innstikksortering er å la Quicksort arbeide helt til mengden elementer den skal lete i er på 0 eller 1 element. Dette fungerer også helt fint, fordi man alltid kan være sikker på at en mengde som har 0 eller 1 element i seg vil alltid være sortert.

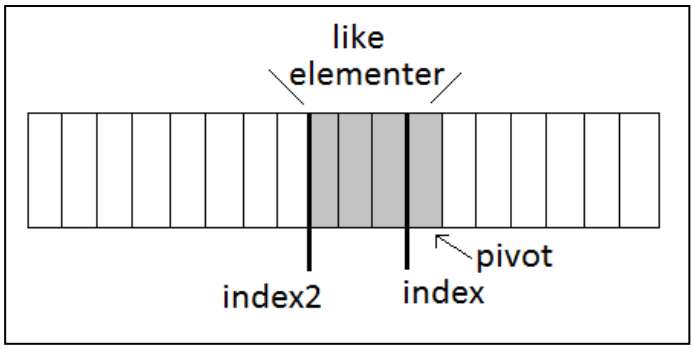
10.1.5 Den sekvensielle Quicksortkoden

Slik ser Quicksortmetoden ut:

```
/*FUNC*/
int[] quicksort(int[] array, int left, int right){
    if (right-left < INSERT_LIM){
        return insertSort(array,left,right);
    }else{
        int pivotValue = array[(left + right) / 2];
        swap(array, (left + right) / 2, right);
        int index = left;

        for (int i = left; i < right; i++) {
            if (array[i] <= pivotValue) {
                swap(array, i, index);
                index++;
            }
        }
        swap(array, index, right);
        int index2 = index;
        while(index2 > left && array[index2] == pivotValue){
            index2--;
        }

        /*REC*/
        array = quicksort(array, left,  index2);
        /*REC*/
        array = quicksort(array, index + 1, right);
        return array;
    }
}
```

Koden fungerer slik at den først sjekker om basistilfellet inntreffer, som her vil være om antall tall vi skal lete igjennom er mindre enn variabelen INSERT_LIM, som er 48. Denne variabelen hjelper oss å finne ut om vi nå har nådd det punktet at vi vil bruke innstikksortering for de siste tallene. Hver instans for ansvar for sin del av mengden, som er angitt av parametrene *left* og *right*.

Dersom vi ikke vil begynne med innstikksortering enda, velger vi oss et pivotelement og begynner å sortere basert på denne. Vi tar deretter for oss eliminering av elementer, som er lik pivotelement, ved å bruke to indekser til å dele opp mengden vår. Den første indeksen, her kalt *index2* fordi vi finner denne etterpå, markerer første gang vi treffer pivotelementets verdi og siste indeks markerer siste. Slik kan mengden se ut med indeksene:



Figur 18: *Vi eliminerer alle like tilfeller av pivotelementet.*

Deretter deler vi rekursjonen i en venstredel, som vil være til *index2*, og en høyredel, som vil starte fra *index*+1. Når disse rekursjonskallene kommer tilbake til oss returnerer vi vårt array tilbake til instansen som kalte på oss. Til slutt vil vi ende på et arrayet bestående av en sortert mengde tall.

10.2 Kjøretidstester

10.2.1 Testing av quicksort

For at det skal være nyttig å kunne kjøre et slikt program ved å bruke parallellitet må vi kunne oppnå bedre resultater, i forhold til kjøretider, enn den originale koden. For å finne ut om vi har oppnådd bedre resultater, ønsker vi å kjøre noen tester. Testene baserer seg på at vi kjører kodene i en løkke på 11. Tallene i mengdene baserer seg på uniformt fordelte mellom 0 og n-1, der n er antall tall, som skal sorteres. Tidene blir målt i nanosekunder i programmet for nøyaktighet, men framvist i tabellen som millisekunder for lesbarhet. For antall tall blir det skrevet K for tusen og M for million. Kjøretidene framvist i en er medianen av de 11 gjennomkjøringene, som kjøres.

Testene baserer seg på en Intel core I7 870 prosessor med en klokkefrekvens på 2,93 GHz. Den har 4 fysiske kjerner, som er hyperthreaded slik at vi vil kunne utnytte dette som 8 kjerner.

10.3 Test 1: Parallell Quicksort generert av Java PRP

10.3.1 Testing og resultater

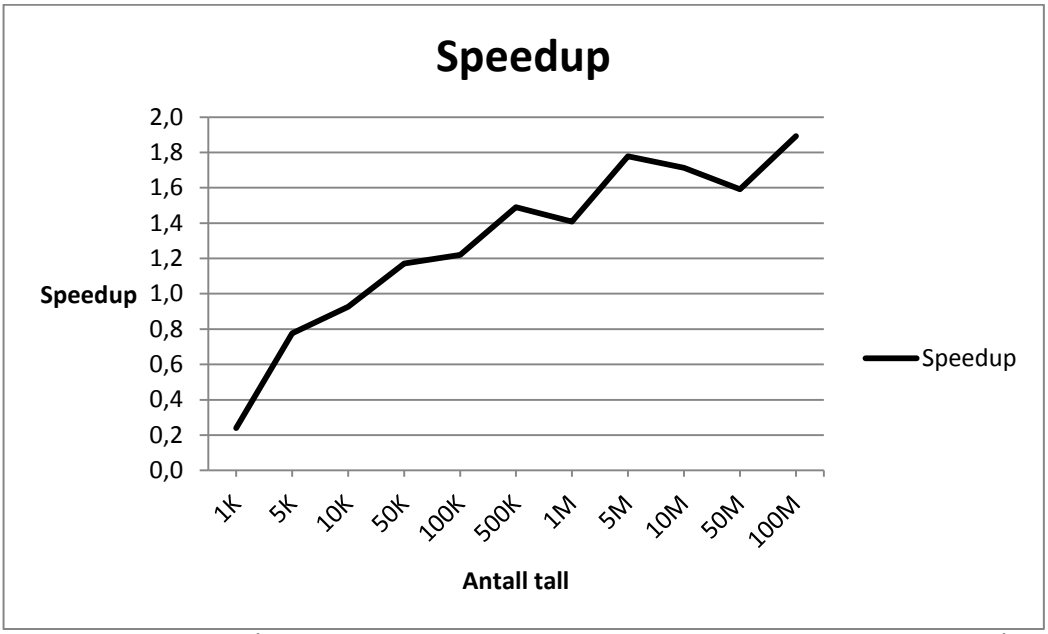
Denne testen vil vise oss forskjellen i kjøretider mellom sekvensiell Quicksort og den parallelle, som er generert av Java PRP.

Antall tall	Kjøretid(ms) for sekvensiell Quicksort	Kjøretid(ms) for Java PRPs parallelle Quicksort	Speedup
1K	0,7	3,0	0,2
5K	2	2,9	0,8
10K	5	5	0,9
50K	4	3	1,2
100K	6	5	1,2
500K	35	24	1,5
1M	76	54	1,4
5M	423	238	1,8
10M	901	526	1,7
50M	5 385	3382	1,6
100M	10996	5813	1,9

Tabell 11: *Kjøretider til sekvensiell og parallell, generert av Java PRP, Quicksort i millisekunder. Disse kjøretidene er basert på en Intel core I7 870 prosessor med en klokkefrekvens på 2,93 GHz, med 8 kjerner (4 fysiske med hyperthreading).*

10.3.2 Speedup vist i en graf

Vi kan benytte oss av følgende graf, for å vise hvordan speedup blir bedre ettersom mengden øker.



Graf 4: *Speedup ved å bruke Java PRPs parallelle Quicksort. Disse kjøretidene er basert på en Intel core I7 870 prosessor med en klokkefrekvens på 2,93 GHz, med 8 kjerner (4 fysiske med hyperthreading).*

10.3.3 Tolkning av resultatene

Tabellen og grafen viser en klar fordel ved å bruke den parallelle versjonen som Java PRP generer når vi kommer til større mengder. Vi kan se at for opp til 10 000 elementer vil den sekvensielle vinne, som kan komme av at den parallelle har en del administrativt den må sette opp for å kunne starte parallelliseringen. Dette, sammen med oppsamling av resultater, vil være sekvensielt i det programmet vi får generert av Java PRP. Dette fører til at vi har en del av programmet som uansett må kjøres sekvensielt til tross for at det er en parallell versjon av Quicksort. Ellers ser vi at den parallelle versjonen slår den sekvensielle med en speedup på 1,9 ved 100 millioner elementer, som viser oss at Quicksort kan forbedres ved å bruke parallellitet.

10.4 Test 2: Parallell Quicksort med lastbalanse

10.4.1 Lastbalanse

Det som er av interesse å teste er om antall tråder påvirker kjøretidene til det parallelle Quicksort programmet. I Java PRP tar den og starter like mange tråder, som det er tilgjengelige kjerner i datamaskinen det kjøres på. Med enda flere tråder får vi lastbalanse i programmet, altså vi lager flere og mindre parallelle oppgaver. På denne måten vil det være mindre sjanse for at en tråd får en stor oppgave, mens en annen får en liten oppgave slik at den blir sittende å vente. Slik vil vi få en jevnere arbeidsfordeling mellom trådene. I Quicksort tilfellet er dette ønskelig, på grunn av at de forskjellige instansene av Quicksort kan variere i størrelse. Dette kommer av at pivotelementet sjeldent er medianen i mengden vi leter i hver gang, som gjør at et subtre i rekursjonstreet kan variere i størrelse og dermed får vi en skjev fordeling av arbeidet. Derfor gir vi alle trådene flere og mindre instanser, fordi vi ønsker og mest mulig å unngå at en tråd venter lenge på andre.

Denne versjonen av Quicksort vil være lik til den, som er generert av Java PRP, med et unntak. Vi øker antall tråder til å være antall kjerner ganget med 10. Slik at min maskin med 4 kjerner, altså 8 med hyperthreading, vil nå sette i gang 80 tråder.

10.4.2 Testing og resultater

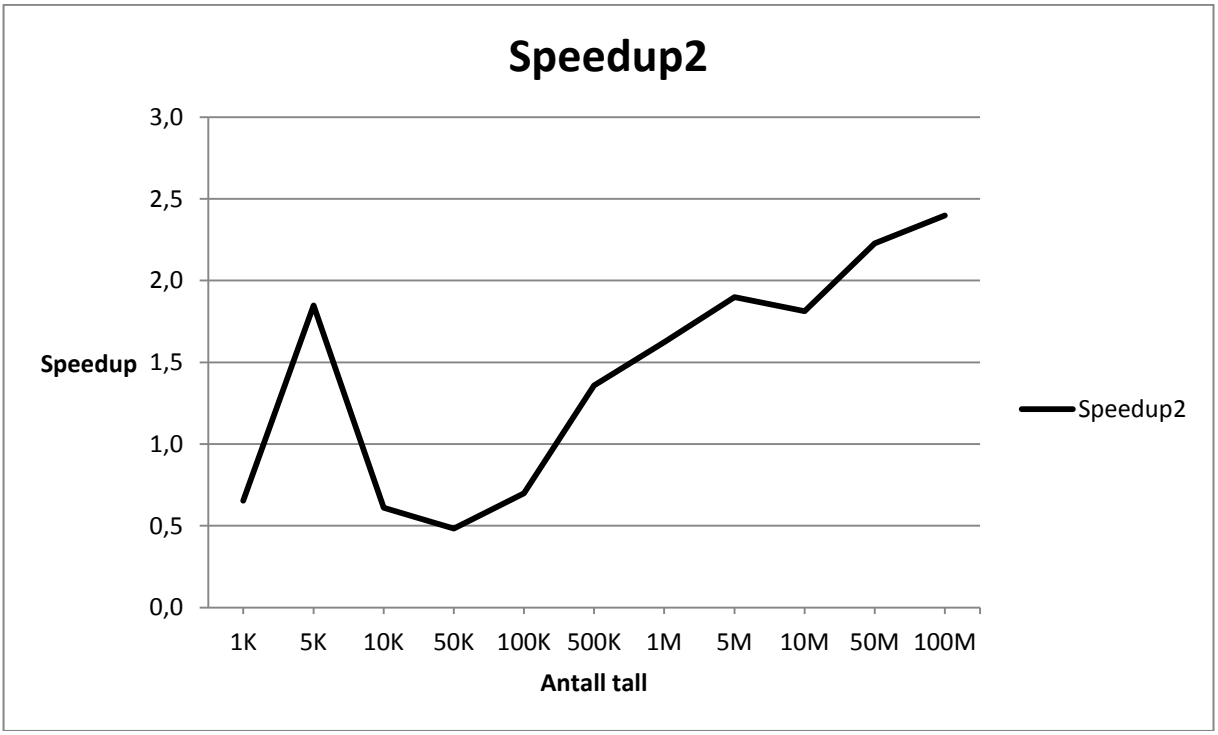
I denne testen vil vi se forskjellen mellom den sekvensielle Quicksort og en parallell Quicksort, som er lik den, som er generert av Java PRP bortsett fra at den har 10 ganger flere tråder. På en måte er det Java PRP med lastbalanse. Her vil vi ha *Speedup2*, da den tar for seg speedup vi får med parallell Quicksort med lastbalanse.

Antall tall	Kjøretid(ms) for		Speedup2
	Sekvensiell Quicksort	Parallell Quicksort med lastbalanse	
1K	0,7	1,1	0,7
5K	2,3	1,2	1,8
10K	5	7	0,6
50K	4	7	0,5
100K	6	9	0,7
500K	35	26	1,4
1M	76	47	1,6
5M	423	223	1,9
10M	901	497	1,8
50M	5 385	2418	2,2
100M	10996	4584	2,4

Tabell 12: *Kjøretider til sekvensiell og parallell Quicksort, med lastbalanse, i millisekunder. Disse kjøretidene er basert på en Intel core I7 870 prosessor med en klokkefrekvens på 2,93 GHz, med 8 kjerner (4 fysiske med hyperthreading).*

10.4.3 Speedup vist i en graf

Ved å vise resultatene i en graf kan vi tydeligere vise hvordan speedup for Quicksort med lastbalanse er.



Graf 5: *Speedup vi får ved å bruke parallell quicksort, med lastbalanse. Disse kjøretidene er basert på en Intel core I7 870 prosessor med en klokkefrekvens på 2,93 GHz, med 8 kjerner (4 fysiske med hyperthreading).*

Vi ser at ved 5000 tall får speedup et noe uventet hopp, opp til 1,8. Dette kommer av at den parallelle Quicksort blir kjør sekvensielt i den parallelle versjonen grunnet lite data. Den genererer barnenoder, som løser oppgaven så dette blir et tilfelle hvor vi

sammenligner to sekvensielle versjoner av Quicksort, der den ene blir løst dybdeførst mens den andre blir løst breddeførst. Det er den "parallelle" versjonen, som løser den breddeførst. Det viser seg at for 5000 punkter er det fordelaktig å løse den på denne måten, i dette tilfellet.

10.4.4 Tolkning av resultatene

Vi ser en lignende trend her, som vi gjorde i 10.3, altså at den parallelle Quicksort oppnår bedre kjøretidsresultater enn den sekvensielle. Det er derimot et par forskjeller. Vi ser at speedup blir bedre først ved 500 000 her, i motsetning til 50 000 i Java PRP sin parallelle Quicksort. Når vi har flere tråder, trenger vi også flere oppgaver. Dette gjør at det er et behov for et større rekursjonstre, altså flere tall å sortere. Slik vil det ta større mengder for at Java PRP vil parallellisere og det er en større administrativ del av programmet når vi har flere tråder. Når vi kommer opp til de større mengder, legger vi merke til at vi får bedre tider med lastbalanse, enn uten. For 100 millioner tall å sortere klarte vi her å få en speedup på 2,4.

10.5 Sammenligning av de to parallelle Quicksort

10.5.1 Testing og resultater

Den neste tabellen viser kjøretidene til to versjoner av parallelle Quicksort, samt den sekvensielle versjonen. En er generert av Java PRP, mens den andre er lik bortsett fra at den kjører 10 ganger flere tråder slik at den har med lastbalanse. I dette tilfellet, kan vi vise en forbedring fra en parallell til en annen slik:

$$Forbedring = \frac{\begin{matrix} Kjøretid\ for\ Java\ PRPs \\ parallelle\ Quicksort \end{matrix}}{\begin{matrix} Kjøretid\ for\ parallelle \\ Quicksort\ med\ lastbalanse \end{matrix}}$$

Antall tall	Kjøretid(ms) for			Speedup for		forbedring med lastbalanse
	sekvensiell Quicksort	JavaPRP uten lastbalanse	Parallell Quicksort med lastbalanse	JavaPRP uten lastbalanse	Parallell Quicksort med lastbalanse	
1K	0,7	3,0	1,1	0,2	0,7	2,7
5K	2,3	2,9	1,2	0,8	1,8	2,4
10K	4,6	5	7	0,9	0,6	0,7
50K	3,6	3	7	1,2	0,5	0,4
100K	6,3	5	9	1,2	0,7	0,6
500K	35,4	24	26	1,5	1,4	0,9
1M	76,4	54	47	1,4	1,6	1,2
5M	423,0	238	223	1,8	1,9	1,1
10M	901,4	526	497	1,7	1,8	1,1
50M	5384,9	3382	2418	1,6	2,2	1,4
100M	10995,9	5813	4584	1,9	2,4	1,3

Tabell 13: *kjøretidene mellom sekvensiell Quicksort, Java PRPs parallelle Quicksort og en parallell Quicksort med lastbalanse. Disse kjøretidene er basert på en Intel core I7 870 prosessor med en klokkefrekvens på 2,93 GHz, med 8 kjerner (4 fysiske med hyperthreading).*

10.5.2 Tolkning av resultatene

Fra 1000 til 5000 tall å sortere vinner den parallelle med lastbalanse, i forhold til kjøretider. Dette kommer av at Java PRP sin versjon vil prøve å parallellisere, mens den med lastbalanse bruker en sekvensiell løsning. Dette kommer av at den ikke klarer å lage nok oppgaver til å parallellisere. Mellom 5000 og 500 000 ser vi at Java PRP sin versjon er bedre, som kan komme av at den ikke trenger å sette i gang så mange tråder, men enda utnytte kjøretidsforbedringene, som parallellitet gir oss.

Lengre ned i Tabell 13 kan vi derimot se at den parallelle med lastbalanse begynner å bli bedre og bedre i forhold til den andre parallelle, generert av Java PRP. Dette kan bety at lastbalanse fungerer når vi ender på store mengder data. Her unngår vi nemlig at noen tråder får store arbeidsmengder, mens noen har mindre. Dette kan føre til at noen tråder må vente på andre slik at vi ikke utnytter parallelliteten til det fulle. Fordelingen av arbeid kan vi si er mer rettferdig.

10.5.3 Lastbalanse i Java PRP

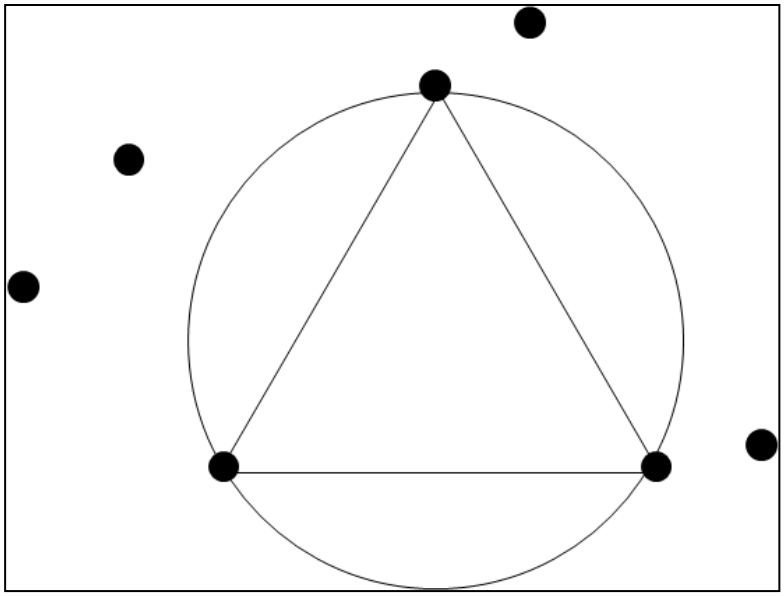
I dette tilfellet fikk vi bedre kjøretider ved å inkludere lastbalanse i implementasjonen av parallell Quicksort. Det setter derimot større krav for hvilke rekursjonsmetoder, som Java PRP kan parallellisere. Det krever nemlig at rekursjonstreet blir stort nok til å lage nok oppgaver til alle trådene. Det blir også krav til at man trenger store datamengder å prosessere, for og ytterligere gjøre rekursjonstreet større. Ettersom Java PRP skal parallellisere på et generelt grunnlag, vil det være bedre om den er mer brukervennlig. Dersom det er mindre krav til antall oppgaver, tillater vi flere rekursjonsmetoder, som ikke klarer å lage mange oppgaver.

11. Java PRP eksempel 3: Delaunay triangulering

11.1 Delaunay triangulering

11.1.1 Definisjon

Gitt et sett med punkter *P* i et kartesisk rom, ønsker vi å finne trianguleringen, slik at den oppfyller kravet om at den omskrevne sirkel rundt en gitt trekant i trianguleringen, basert på tre punkter i P, ikke inneholder noen andre punkter fra P. Dersom dette blir oppfylt for alle trekantene er dette en Delaunay triangulering[12]. For et fullstendig eksempel av en Delaunay triangulering for 100 punkter henvises det til Figur 32 senere i dette kapittelet.



Figur 19: *Gitt et sett med punkter, kalt P, så vil en trekant, basert på tre punkter i P, være en gyldig Delaunay trekant, dersom trekantens omskrevne sirkel ikke inneholder noen andre punkter P.*

11.1.2 Bruksområder

Det å finne Delaunay triangulering av en punktmengde har mange bruksområder. Det kan brukes til å gi ytterligere informasjon om geografiske kart, slik at vi kan modellere terreng basert på punkter i kartet. Det blir også mye brukt innen spillgrafikk, der forskjellige objekter blir tegnet opp først som figurer representert med punkter og trianguleringen basert på dem.

11.2 En faseoppdelt Delaunay triangulering

11.2.1 Oversikt over trianguleringen

For å lage et eksempel av Java PRPs parallellisering av faseoppdelte programmer vil det her bli presentert en måte å beregne seg frem til Delaunay trianguleringen av et sett med punkter i flere faser. Dette programmet vil vise et eksempel på hvordan man lager et sekvensielt, faseoppdelt program, som Java PRP kan tolke og parallellisere riktig.

Ved oppstart av programmet vil vi generere de nødvendige datastrukturene for programmet. Det viktigste vil være å generere punktsettet vårt. Punktene vil være tilfeldige heltallsverdier for x- og y-verdiene. Eneste unntaket er at de må være unike punkter, da det ikke gir mening å skape en triangulering der vi har flere punkter i samme posisjon. Deretter vil vi begynne med vår administrative metode, som setter i gang de forskjellige fasene.

Den første fasen vil bestå av å beregne den konvekse innhyllingen for punktsettet vårt. Dette vil gi oss et subsett av det totale punktsettet. Dette subsettet inneholder punkter, som representerer yttergrensene av punktsettet. Det vil si at vi nå kan finne ut hvilke punkter, som er på grensen av trianguleringen. Dette er nyttig informasjon i de senere fasene.

Den neste fasen vil finne de *k* nærmeste nabopunktene til hvert punkt, for en gitt verdi *k*. Ved å følge visse regler for hvilke to nabopunkter, som kan danne en gyldig Delaunay kant sammen, kan vi kanskje redusere mye av beregningene for å finne den totale Delaunay trianguleringen.

Den forrige fasen vil ikke kunne beregne hele Delaunay trianguleringen, kun basert på de *k* nærmeste naboene. Dette gjør at vi trenger en tredje fase, der vi gjør beregningene for resten av Delaunay trianguleringen. Fordi vi har gjort en del arbeid med punktsettet vårt i de forrige fasene, behøver vi ikke å beregne hele trianguleringen i denne fasen, fordi vi kan bruke informasjonen tidligere funnet for å gjøre denne fasen raskere.

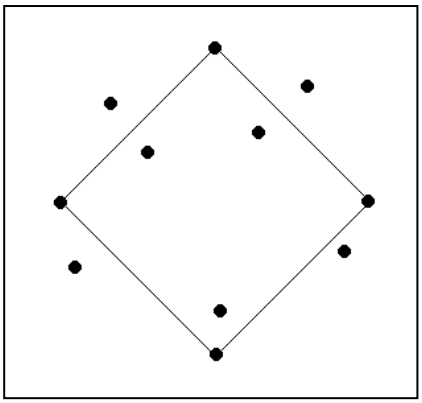
Når disse tre fasene er ferdig, har vi en fullstendig Delaunay triangulering basert på punktsettet vårt. Programmet er laget på en slik måte at vi kan nå sende dette til Java PRP, slik at vi får en faseoppdelt og parallell Delaunay triangulering, basert på det vi har gjort i det sekvensielle programmet.

11.3 Fremgangsmåten for å beregne den konvekse innhyllingen

11.3.1 En rekursiv metode for konveks innhyllingen

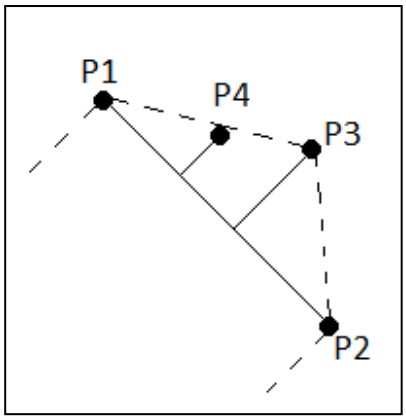
Java PRP har noen grunnleggende krav for å kunne parallellisere en metode, som gjør at vi må tilpasse vår metode for å finne den konvekse innhyllingen etter disse. Det første vil være å finne en rekursiv metode, som deler seg i to eller flere deler for hver instans av rekursjonen. Vi vil også at instansene skal kunne arbeide selvstendig.

Rekursjonsmetoden baserer seg på en algoritme presentert i "*Delaunay triangulation and the convex hull of n points in expected linear time"*[12]. Den tar utgangspunkt i at vi vet om noen spesielle punkter i punktsettet, som fra starten av, skal være med i den konvekse innhyllingen. Disse punktene er de som representerer de ytre grensene for x- og y-verdiene. For eksempel, punktet med den minste x-verdien blant alle punktene være med i innhyllingen. I de fleste tilfeller vil vi få fire punkter som skaper en diamantlignende firkant. I noen tilfeller vil et punkt representere flere av de ytre grensene, men dette har ingen betydning for videre beregning. Kun i noen spesielle tilfeller kan vi få to eller tre ytre punkter. Slik, som alle andre firkanter, får vi fire linjer mellom punktene. Det er med disse linjene vi begynner rekursjonen vår. Fra disse linjene vil vi arbeide oss utover til å finne de utenfor.



Figur 20: *Ut ifra et punktsett vet vi allerede fra starten at de punktene, som representerer de ytre grensene (feks, største x, minste y osv.), er med i den konvekse innhyllingen. Disse danner ofte en firkant, som er starten på rekursjonsmetoden vår for konveks innhylling.*

Hver rekursjonsinstans får inn to punkter, P_1 og P_2, som gir oss linjen vår, og et sett av alle punktene, som er over denne linjen. Blant disse punktene finner vi det punktet, ved navn P_3, som er lengst unna linjen. Denne vil være i den konvekse innhyllingen. Deretter deles rekursjonen i to, fra P_1 til P_3, og P_3 til P_2. Vi finner også punktene over disse linjene, som sendes til de neste instansene. Dersom en instans oppdager at det er ingen punkter utenfor sin linje, vet vi at det ikke er flere punkter, som er i den konvekse innhyllingen, å hente her.



Figur 21: *En instans i rekursjonen for å finne den konvekse innhyllingen. Instansen får inn punkt P_1og P_2, samt settet av punktene over linjen mellom P_1og P_2, nemlig P_4 og P_3. Vi finner punktet som er lengst unna linjen av P_4 og P_3. P_3 er lengst unna, slik vi deler rekursjonen i to deler, nemlig en instans med P_1og P_3, og en annen med P_3 og P_2. Vi gjør samme operasjon der.*

Denne metoden fyller vårt krav om en rekursiv metode som deler seg to deler per instans. Eventuelle tråder vil operere selvstendig, som betyr at det er en god metode for å kjøres i parallellitet. Hver instans ser bare på punktene som er utenfor sitt linjestykke mellom to punkter, og disse deles ikke med andre.

11.4 Geometriske hjelpemetoder for å finne den konvekse innhyllingen

11.4.1 Lengden fra et punkt til en linje

For at vi skal kunne beregne hvilket punkt utenfor en linje som er med i den konvekse innhyllingen, må vi finne punktet lengst unna. For Figur 21 vil det si at vi trenger en formel for å finne punkt P_3.

Det er to situasjoner som kan oppstå for rekursjonsmetoden vår. Den første forekommer når de to punktene som danner linjen er samme punkt. For eksempel, dersom punktet for største x-verdi er det samme som punktet for største y-verdi. Dette betyr at ingen punkter er over dette punktet, nemlig fordi at da ville disse representert de ytre grensene. Derfor vil det ikke være noen form for beregninger her.

Den andre situasjonen er slik at vi har en linje, og flere punkter utenfor. Med tre punkter, nemlig to punkter som danner linjen og et punkt utenfor, trenger vi noen beregninger for å komme fram til lengden. Den første vil være å komme fram til likningen til linjen vår basert på de to første punktene våre:

$$y - y_1 = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$$

Dette gir oss en likning på denne formen:

$$0 = ax + by + c$$

Dette bruker til å finne en form av lengden til punktet gitt ved (x_0, y_0) via følgende formel:

$$lengde2 = |ax_0 + by_0 + c|$$

Med form av lengden menes det at det kreves flere beregninger for nøyaktig lengde, men siden verdien kun blir brukt til sammenligning er det mulig å ignorere deler av beregningene. Alle punktene over linjen blir sjekket ifølge denne formelen og den lengste blir stående igjen som et medlem av den konvekse innhyllingen. Rekursjonen fortsetter med dette punktet som utgangspunkt.

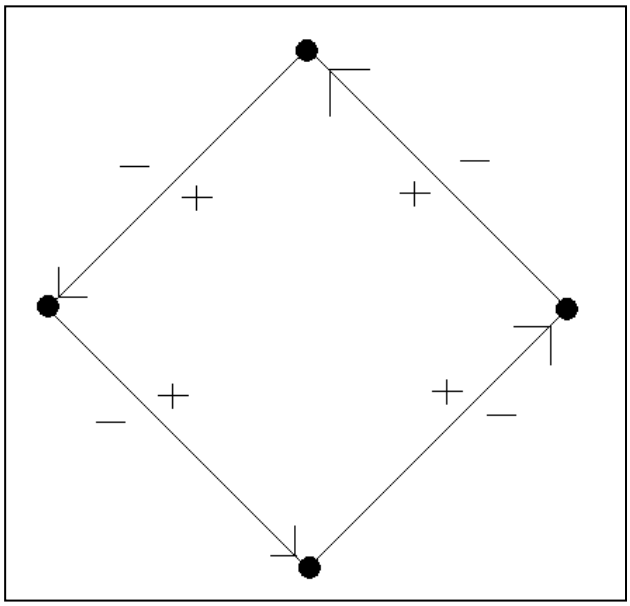
11.4.2 Finne ut hvilken side et punkt er i forhold til en linje

For at vi skal klare å finne den konvekse innhyllingen må vi også kunne vite hvilke side punktene befinner seg i forhold til en en linje. Det vil si at vi trenger en formel som sier hvor et punkt befinner seg i forhold til en linje. I Figur 20 så vi at det dannes en firkant av de fire ytterpunktene. Allerede her vet vi at punktene inne i firkanten ikke er med i den konvekse innhyllingen.

Vi følger en kode, som baserer seg på kryssproduktet mellom linjen P_1 og P_2 og punktet vi er ute etter P_3.

$$a = (x_2 - x_1) * (y_3 - y_1) - (y_2 - y_1) * (x_3 - x_1)$$

Dersom a er 0 vil punkt P_3 befinne seg på linjen. Ellers vil formelen vise hvilken side punktet er på avhengig av om *a* er et negativt eller positivt tall. Dette avhenger av retningen til linjen, altså er det linjen P_1 til P_2 eller er det P_2 til P_1. Vi ønsker å bruke samme formel for alle linjene, slik at det blir tatt høyde for retning. Figur 22 viser hva vi er ute etter når vi bruker denne formelen.



Figur 22: *Bildet viser den første firkanten, som representerer de fire ytterpunktene i innhyllingen. Slik som linjene er rettet mot, vil formelen, i 11.4.2, gi oss et positiv tall for punkter inni firkanten, og et negativt tall om den er utenfor. Dersom tallet er 0 befinner punktet seg på linjen.*

Man kan bruke formelen i 11.4.1 for å finne ut hvilken side et punkt er i forhold til en linje. Dersom man endrer den slik at man ikke tar absoluttverdien av distansen, vil den ene siden gi deg et negativt tall og den andre et positivt. Ulempen med denne er at man trenger å vite hvordan linjen ser ut for å vite hvilken side som er positivt og negativ. Det vil si om linjen er synkende, stigende, vertikal eller horisontal. Det trenger ikke kryssproduktsformelen. Den trenger heller ingen informasjon om linjelikningen. For den vil en linje, ut fra et punkt, ha den positive siden til venstre og den negative på høyre. Dette gjelder i 360° rundt punktet. Dette ser vi i Figur 22.

I tilfellet for den konvekse innhyllingen vet vi noe om hvordan linjene ser ut, basert på de fire hjørnene av startfasen. Derimot er det unntakstilfeller for horisontale og vertikale linjer, som kan oppstå uansett hvor vi er, men må håndteres spesielt basert på hvor i innhyllingen vi befinner oss i, slik at det blir vanskeligere. Senere i programmet må vi finne ut hvilken side et punkt er i forhold til generelle linjer, som vi ikke vet hvordan ser. I disse må vi også bruke kryssproduktet.

11.5 Inkludere metoden for å beregne den konvekse innhyllingen i Java PRP

11.5.1 Rekursjonen starter fra flere kanter

Denne formen for å finne den konvekse innhyllingen krever at rekursjonen starter fra fire kanter, basert på de ytre grensene av punktsettet. Dette betyr at det må settes i gang en oppstartsmetode der den setter i gang de fire delene av rekursjonen. Derfor lager vi en metode, som kaller på de fire kantene. I denne metoden settes det i gang rekursjonskall til de fire forskjellige kantene. Ytterpunktene har vi funnet tidligere, når vi laget punktene.

Vi lager først settet av punkter over linjen som dannes av to punkter, som representerer ytterpunkter. Deretter startes rekursjonen for å finne punktene i den konvekse innhyllingen mellom disse to startpunktene.

Ved å legge til en slik kallmetode for rekursjonen blir ikke den parallelle delen annerledes. Selve arbeidet for trådene baserer seg på selve rekursjonsmetoden, og modellen for å arbeide med rekursjonstreet er likt. Nemlig modellen der vi setter inn arbeiderinstansene breddeførst i starten, mens trådene arbeider dybdeførst.

11.5.2 Første fase av et større program for Delaunay triangulering

Å finne den konvekse innhyllingen er bare en mindre del av Delaunay trianguleringen. Som et faseoppdelt program, vil dette være første fasen. Dette vil skape grunnlaget for neste fase, som er begynnelsen av trianguleringen ved at vi finner de k nærmeste naboene.

Den parallelle fasen består av selve beregningen for å finne den konvekse innhyllingen. Hovedarbeidet ligger i rekursjonsmetoden, som er forklart i 11.3.1. Ved å følge administratormetoden, vil den parallelle fasen returnere tilbake til denne metoden før vi går i gang med den første sekvensielle fasen.

Siden det er punktsettet som er dette programmets felles data, vil vi her gjøre om på disse punktene gitt den nye informasjonen vi fikk fra den parallelle fasen. Til nå har vi kun et subsett av punktsettet, som er den konvekse innhyllingen. I den sekvensielle fasen vil vi sette kanter mellom dem i rekkefølge. Det vil si at vi skal kunne følge en linje rundt punktsettet, slik at punktene i den konvekse innhyllingen kommer etter hverandre.

11.6 De k nærmeste naboene

11.6.1 Dele opp punktene i bokser

Vi deler opp punktmengden i flere bokser, slik at vi kan lettere plukke et utvalg av punkter basert på deres posisjoner. Boksene er et todimensjonelt array, som vil deles utover koordinatsystemet.Boksen, som representerer første boks fra origo, inneholde punktene der koordinatene er mellom 0 og lengden på boksene i både x og y.

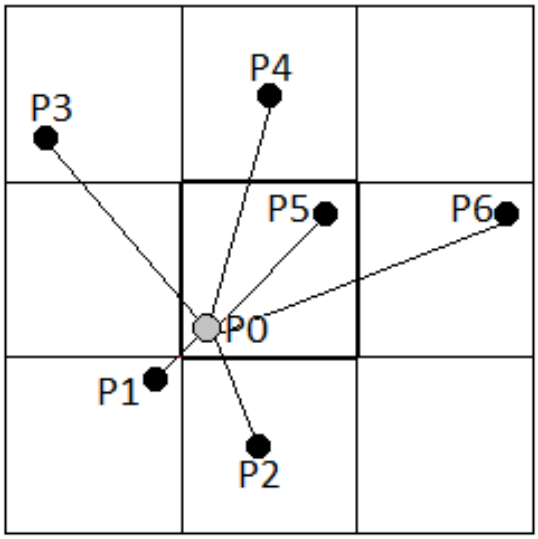
Ved å bruke bokser kan vi lettere finne frem til forskjellige punkter vi er ute etter. Dette vil hjelpe oss i situasjonen hvor vi ønsker å finne den aller nærmeste naboen, og de *k* nærmeste naboene. Å dele opp punktsettet vårt slik er en idé gitt fra "*Delaunay triangulation and the convex hull of n points in expected linear time"*[12].

11.6.2 Finne den aller nærmeste naboen

Arne Maus og Jon Moen Drangen[14] kom frem til denne alternative måten å finne kanter i en Delaunay triangulering. Ved å finne det nærmeste nabopunktet til et punkt kunne vi danne en gyldig Delaunay kant mellom disse. Teoremet er følgende:

T1: "Let P be a set of points $p_1, p_2, \ldots, p_n$ *in the plane. Then, for all* p *in P, where* b *in P is the unique closest neighbor to* p*, the line segment* pb *is an edge in the Delaunay triangulation of P."*

For å få til dette ønsker vi å bruke boksene. Vi bruker et lite antall bokser å lete igjennom istedenfor å lete igjennom alle punktene, for å sjekke hvilke punkt er nærmeste nabo. Dette gjør at vi betydelig reduserer antall punkter å gå igjennom. Alle punktene vet om hvilken boks de tilhører, slik at vi har et utgangspunkt for hvilken boks vi ønsker å lete etter. Derimot holder det ikke bare å sjekke sin egen boks, fordi vårt punkt kan ha et nærmere nabopunkt i boksen ved siden av. Dette gjør at vi sjekker alle boksene rundt vårt punkts boks. Dersom avstanden til et gitt punkt innenfor leteområdet vårt er større enn 1 bokslengde, må vi utvide vårt søk. Det blir dermed et søk i 3x3, 5x5, 7x7 osv. med vårt punkts boks som midterte.

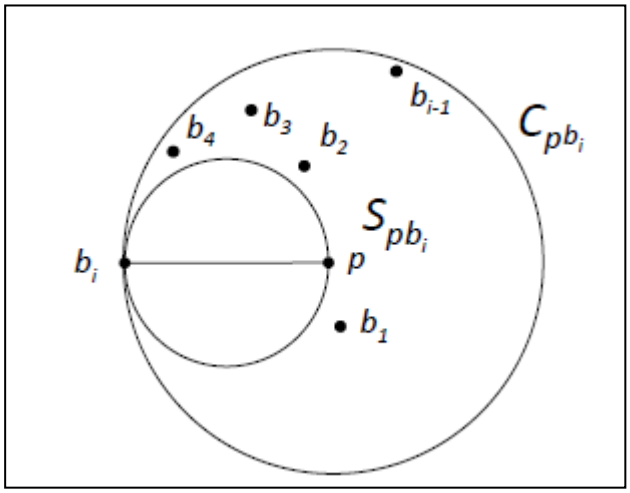


Figur 23: *Bilde av hvordan vi leter etter nærmeste nabo til punkt P_0. Vi deler opp koordinatsystemet i mindre bokser, som er lettere å lete igjennom. Istedenfor å lete igjennom alle punktene leter vi kun i boksene rundt P_0 sin boks. I dette tilfellet er nærmeste punktet P_1, som ikke er i samme boks som P_0.*

11.6.3 Finne naboer lenger unna

Forrige teoremet T1, i 11.6.2, kan utvides til å finne kanter i Delaunay trianguleringen blant de *k* nærmeste punktene. Teoremet er slik[14]:

T2: "Let P be a set of points $p_1; p_2, ..., p_n$ in the plane. Let $b_i (i = 1; 2, ..., k)$ in P be the k closest neighbors to a point p in P. Then pb_i is an edge in the Delaunay triangulation of P if none of the closer neighboring points $b_j (j = 1, ..., i - 1)$ are included in the circle S_{pbi} with pb_i as its diameter."



Figur 24: *Bildet[14] viser hvordan man kan finne kanter i Delaunay trianguleringen ved å sjekke de k nærmeste naboene. Så lenge sirkelen S_{pbi}, med linjen p-b_i som diameter, ikke inneholder noen andre punkter, er denne linjen en gyldig Delaunay kant.*

Vi finner de gyldige kantene, blant de *k* nærmeste naboene til vårt punkt, ved å gjøre som vist i Figur 24. Vi tenker oss at vi danner en sirkel basert på vårt punkt og et gitt nabopunkt, der disse to punktene er diameteren. For å finne ut punktene, som er innenfor sirkelens grenser, må vi først finne ut to elementer til denne sirkelen. Det første er sentrum, og det andre er radius til sirkelen. Alle punkter, som har en lengde

fra sentrum mindre enn radius, må være innenfor sirkelen. Dette gjør at vårt punkt og dette nabopunktet ikke danner en gyldig Delaunay kant.

11.6.4 Rekursjonsmetoden

Rekursjonsmetodens hovedfunksjon er å tildele rader av bokser til forskjellige tråder. Metoden deler opp det todimensjonale arrayet av bokser helt til vi har en rad i hver løvnode av rekursjonstreet. Deretter sender den hver rad videre til en annen metode, som vil gå igjennom hver av boks i denne raden en om gangen. Deretter arbeider den på hver boks slik som det ble vist i 11.6.2 og 11.6.3. Rekursjonsmetoden returnerer til slutt arrayet av punkter, der punktene er oppdatert med de *k* nærmeste naboene.

Årsaken til at vi deler opp i rader i rekursjonsmetoden, før vi går igjennom hver individuelle boks, er at vi kan vite hvilke bokser hver tråd kommer til å gå igjennom senere. Dette gjør at vi kan overskrive data i punkter, dersom de er i samme boksrad. Rekursjonsmetoden ser slik ut:

```
/*FUNC 2*/
Point[] findAllClosestNeighbours(Box[][] b, Point[] p, int start, int
end, int BOX_LEN){
    if((end - start) == 0){
        findValidNeighbours(b, p,start,k,BOX_LEN);
        return p;
    }
    int end2 = start + ((end-start)/2);
    int start2 = end2+1;
    /*REC*/
    p = findAllClosestNeighbours(b, p, start, end2,BOX_LEN);
    /*REC*/
    p = findAllClosestNeighbours(b, p, start2,end,BOX_LEN);
    return p;
}
```

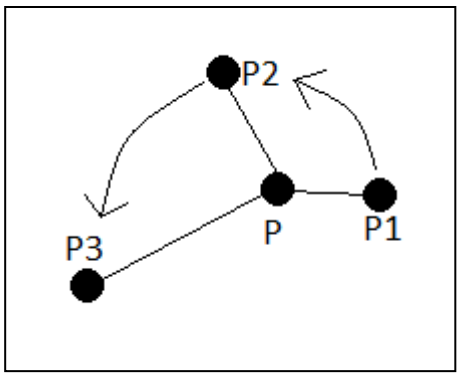
Denne rekursjonsmetoden kaller på metoden *findValidNeighbours(...)*, som deler opp radene i enkelte bokser og finner de *k* nærmeste, gyldige Delaunay nabopunktene. For å illustrere bedre er dette omrisset av metoden:

```
public void findValidNeighbours(Box[][] b, Point[] p, int row, int k,
int BOX_LEN){
    //alle boksene
    for(int i = 0; i < b[row].length; i++){
        //alle punktene i boksen
        for(int j = 0; j < b[row][i].points.size(); j++){
        //finne k gyldige Delaunay naboer for nåværende boks
        //...
        }
    }
}
```

11.6.5 Sortere et punkts nærmeste naboer

Et punkt har en liste over hvilke punkter den deler kanter med. Etter første fase vil denne lista bestå av to elementer for alle punkter, som befinner seg i den konvekse innhyllingen. Det første punktet i lista er før i innhyllingen, mens det neste punktet er etter i innhyllingen og vil være på plass nummer to i lista. Det at vi har punktet før i innhyllingen på første plass er noe vi opprettholder videre, fordi det gir oss informasjon om hvor de konvekse kantene befinner seg blant et punkts liste over sine Delaunay kanter.

Etter at vi har funnet de *k* nærmeste naboene til et gitt punkt, som tilfredsstiller teorem T2 i 11.6.3, vil disse punktene være sortert etter lengden til det gitte punktet. Det vil si at den aller nærmeste og gyldige punktet er først blant disse naboene, og den lengst unna til slutt. Deretter sorteres de annerledes før vi setter i gang med tredje fasen. Vi vil sortere dem slik at vi kan følge alle punktene i en sirkel. Følgende bilde viser et eksempel:



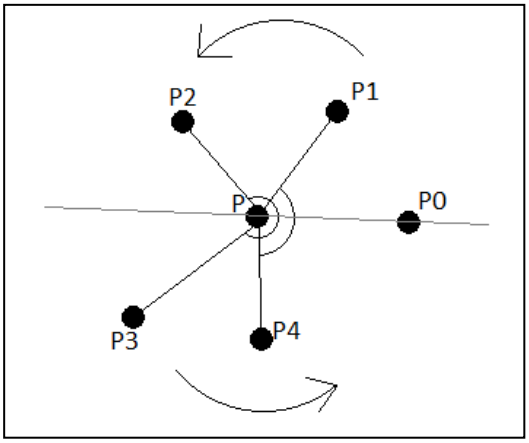
Figur 25: *Bildet viser punkt P, med sine tre nærmeste naboer. De blir sortert slik at vi kan følge rotasjonen mot klokka. Den sorterte lista over nabopunkter til P vil dermed være P1,P2 og P3.*

Dersom punktet vårt er med i den konvekse innhyllingen, vil første element i lista være punktet før i innhyllingen. Dersom punktet vårt ikke er med, vil første elementet være den aller nærmeste naboen. Denne måten å sortere på er bedre enn kun etter lengden til neste punkt, fordi neste fase kan nå sjekke neste punkt i lista om dette kan skape en ny DT (Delaunay trekant).

11.6.6 Måter å sortere punktene

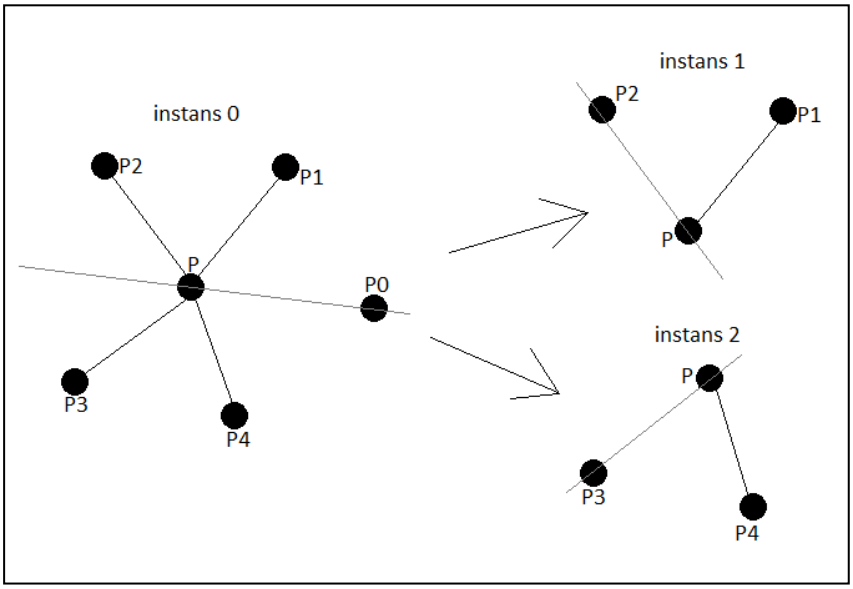
Det finnes flere alternativer for å sortere punktene rundt et punkt på, slik at vi oppnår rekkefølgen vi ønsket i 11.6.5. Siden denne operasjonen skal gjøres for hvert eneste punkt i punktmengden vår, vil vi at sorteringen skal være raskest mulig.

Et alternativ for å sortere punktene er å implementere en innstikksortering for de forskjellige vinklene, som baserer seg på vårt midtpunkt og dens nabopunkter. Det vil si at dersom vi har et midtpunkt P, med de gyldige Delaunay nabopunktene P_0, P_1 ... P_n, vil vi finne alle vinklene P_0PP_1, P_0PP_2 og så videre. Vi vil dermed alltid gi vinkelen basert på vårt midtpunkt, dens første nabo og et annet punkt i lista. P_0 vil alltid være først i lista over P sine nabopunkter. Dette gjør at punktene sorteres etter hvilken vinkel de førte til. Vi må også huske at vi må sortere over og under linja $P\text{-}P_0$ hver for seg, da vi bruker cotangens for å beregne vinkelen. Dette gjør at vi har to sorterte lister, nemlig nabopunktene over og de under linja $P\text{-}P_0$. Vi setter dem sammen, men vi reverserer den ene lista, da denne er sortert i omvendt rekkefølge.



Figur 26: *Bildet viser hvordan vi kan sortere rundt et punkt, ved å se på vinklene som dannes med linjen P-P_0 og et gitt annet nabopunkt til P. Vi bruker innstikkssortering for å sortere punktene basert på hvilke vinkler de gir P_0PP_i, der P_i er et gitt nabopunkt.*

En annen måte å sortere punktene på, er å bruke rekursjon til å dele opp punktene i mindre deler. Rekursjonsmetoden deler opp mengden nabopunkter i to for hver gang, nemlig punktene over og under en linje. Denne linjen dannes av midtpunktet vårt, P, og et gitt punkt, P_x, i mengden punkter denne instansen ser på. Når vi er nede på 1 eller 0 punkter over eller under linja vår, vet vi noe om posisjonen til P_x, for eksempel, at dette punktet er på slutten eller begynnelsen av nåværende punktmengde for denne rekursjonsinstansen.



Figur 27: *Et rekursivt alternativ for å sortere punktmengden vår på. Hver rekursjonsinstans, får inn en mengde punkter. Et av disse punktene, P_x, danner en linje med midtpunktet vårt, P. For hver instans, vet vi at P_x må være foran de punktene over linja, og etter de som er under. Bildet viser hvordan vi går fra en instans, til de to neste. I instans 1, vet vi at P_2 må være etter P_1. I instans 2, vet vi at P_3 må være før P_4.*

I dette program, er første alternativet slik vi sorterer nabopunktene på. Dette kommer av at vi har en metode laget for å finne cotangensverdien til en vinkel gitt av tre punkter, slik at vi kan gjenbruke dette i denne metoden.

11.7 Geometriske hjelpemetoder for å finne de *k* nærmeste naboene

11.7.1 lengden mellom to punkter

For å finne ut hvilke punkt som er det nærmeste bruker vi en variant av Pytagoras' setning. Denne brukes for hvert punkt i boksene, som er rundt vårt punkt. Den korteste lengden gir oss den nærmeste nabo. Siden vi bruker lengdene kun til sammenlignelse, kan vi fjerne noe av beregningen av den faktiske lengden, som å ta kvadratroten. lengden blir dermed beregnet slik:

$$lengde2 = (x_2 - x_1)^2 + (y_2 - y_1)^2$$

Dette vil vi bruke til å finne de k nærmeste naboene til et punkt, samt finne ut om det er noen punkter innenfor en sirkel.

11.7.2 Beregne størrelsen av boksene

Størrelsen på boksene vil si hvor mange koordinater skal hver boks dekke i både x- og y-aksen. Det vi ønsker er at denne størrelsen ikke skal være fast, for eksempel alltid 10, 100 eller 1000. Dette kan føre til at for små mengder punkter kan det være få punkter per boks, mens i store mengder punkter vil boksene være fulle av punkter. Det vil si at boksene ikke er proporsjonale basert på *n* punkter. Derfor ønsker vi at størrelsen på boksene skal variere på *n*.

Det er flere måter man kan bruke n for å komme frem til en størrelse. Måten dette programmet gjør det er å finne fram til den størrelsen slik at vi, i snitt, har 4 punkter per boks. Vi vet at vi vil ha koordinatene våre vil basere seg fra 0 til maksimalverdi av x og maksimalverdi av y, som er tilnærmet lik n, og at det vil være n punkter totalt. Slik ser formelen ut for å finne fram til størrelsen:

$$boksstørrelse = \frac{n}{\sqrt{\frac{n}{4}}}$$

11.8 Andre fase av Java PRPs faseoppdeling for Delaunay triangulering

11.8.1 Den parallell fasen

Dette vil være andre del av faseoppdelingen til Delaunay trianguleringen. Her vil forutsetningen være at hele den konvekse innhyllingen er funnet. Det kommer altså etter den sekvensielle fasen til beregningen av den konvekse innhyllingen.

Den parallelle fasen for å finne de nærmeste naboene vil bestå av rekursjonsmetoden gitt i 11.6.4. Denne metoden gir utgangspunktet for det som kommer til å bli en ny, parallell eksekvering, som vil gi oss de *k* nærmeste naboene for hvert punkt.

11.8.2 Den sekvensielle fasen

Siden punktsettet er felles data, må vi passe på at vi ikke endrer de samme punktene samtidig. Hver løvnode i rekursjonstreet behandler hvert sin rad av bokser, slik at de kun skriver til denne. De leser andres x- og y-verdier, men disse endres aldri, slik at det ikke blir misforståelser på grunn av punktenes posisjoner. Hver av disse løvnode skriver informasjon til sine egne punkter, nemlig hvilke punkter vi kan danne en gyldig Delaunay kant til.

Kantinformasjonen er kun gitt til punktet, som blir behandlet, det vil si hvis vi er på punkt P_1, og har funnet en kant til punkt P_2, vet ikke nødvendigvis punkt P_2 om denne kanten. Det at denne informasjonen går begge veier, vil hjelpe for neste fase. Neste fase vil nemlig være å fullføre Delaunay trianguleringen. Dette vil si at den sekvensielle fasen vil bestå av å gjøre all naboer kjenner til hverandre begge veier, samt at vi må sortere punktene, som et punkt danner kant med, slik som det blir beskrevet i 11.6.5.

11.9 Resten av Delaunay trianguleringen

11.9.1 Fullføre trianguleringen

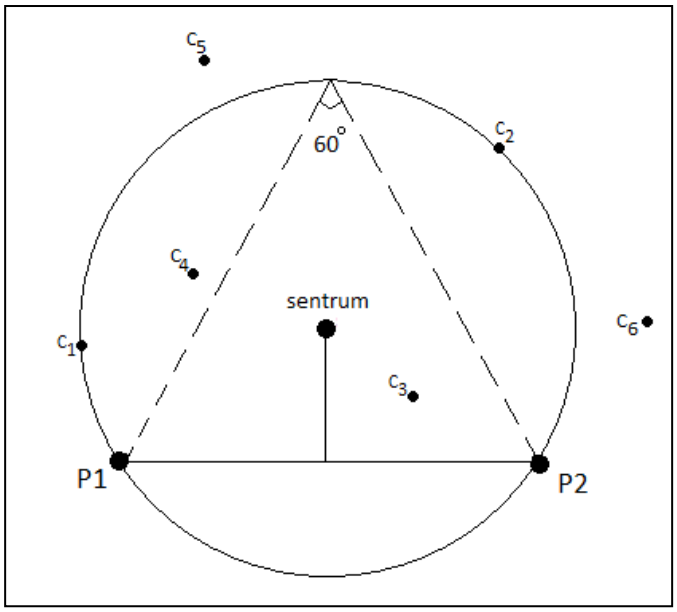
Basert på arbeidet gjort til nå, nemlig funnet den konvekse innhyllingen og funnet de *k* nærmeste naboene til hvert punkt, har vi enda ikke fullført oppgaven med å finne den totale Delaunay trianguleringen. Fordi det kan være kanter som ikke er oppdaget allerede, må vi finne disse ved å bruke andre metoder. Derimot har vi flere fordeler ved å ha funnet mange av kantene allerede.

11.9.2 Egenskaper ved kantene allerede funnet

Vi vet at alle punkter har en Delaunay kant til minst ett annet punkt, nemlig dens nærmeste nabo. Dette gjør at vi alltid kan bygge de siste kantene med minst én kant, som grunnlag. På denne måten kan vi finne hele Delaunay trianguleringen for et punktsett. Det, som forklares videre, vil kun gjelde for kantene som vi allerede har funnet ved å lete etter de nærmeste naboene.

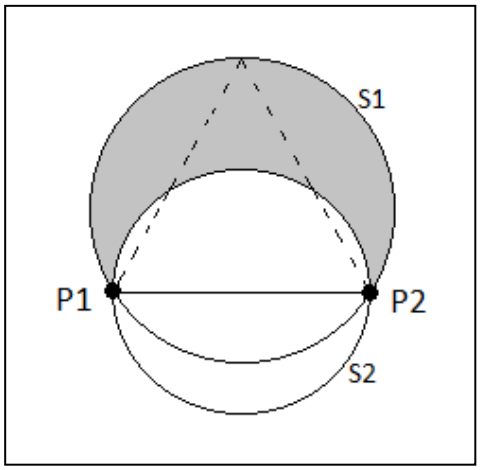
Dersom vi bruker en allerede funnet Delaunay kant ved at de er blant de nærmeste naboene til hverandre, P_1P_2, og sjekker opp mot et annet punkt, C, vet vi at vinkelen P_1CP_2 vil være mindre eller lik 90°. Dette kommer av ingen punkter befinner seg i en sirkel der kanten vår er diameter, da dette var en gitt regel for kanter funnet i fase to. Et annet punkt, C_x, på denne sirkelen vil føre til at vinkelen $P_1C_x\ P_2$ vil være 90°, da dette blir periferivinkler.

Vi baserer oss på at vinklene i Delaunay triangulering er i snitt lik 60°[9], da det er omtrent 6 Delaunay kanter ut fra et gitt punkt. Derfor kan vi i utgangspunktet se på sirkelen, som inneholder en trekant med Delaunay kanten P_1P_2 og et annet punkt C_x slik at vinkelen $P_1C_x\ P_2$ er 60°. Dette fører til at punkter innenfor sirkelen, som dannes av denne likesidede trekanten, vil være kandidater for å inngå i en DT med den eksisterende kanten.



Figur 28: *Bildet viser en del punkter, og kanten P_1P_2 allerede funnet som riktig kant. Punktet "sentrum" gir sentrum til sirkelen der vi får vinkler med 60°. C_1 og C_2 er på sirkelen og C_3 og C_4 er innenfor sirkelen, slik at disse er kandidater for å danne en DT med kanten P_1P_2. Punktene C_5 og C_6 er utenfor sirkelen, og kan derfor ikke danne en DT med kanten P_1P_2.*

Det eventuelle punktet vi finner må oppfylle definisjonen, gitt i 11.1.1, for å være en gyldig kant. Dette betyr at ingen punkter kan være inne i sirkelen, som dannes av trekantens tre punkter, for at det skal være en gyldig DT. Bildet under viser hvor punktene vil befinne seg dersom kanten P_1P_2 er nærmeste naboer:



Figur 29: *Bildet viser hvor vi leter etter punktene for å danne en DT, med punktene P_1 og P_2 allerede satt, som nærmeste naboer, nemlig den grå delen av sirkelen S_1. S_1 er slik at P_1 og P_2 er med i en trekant der det siste punktet på sirkelen, kalt C, vil gi P_1C P_2 en 60° vinkel, mens sirkelen S2 er der linjen P_1- P_2 er diameteren.*

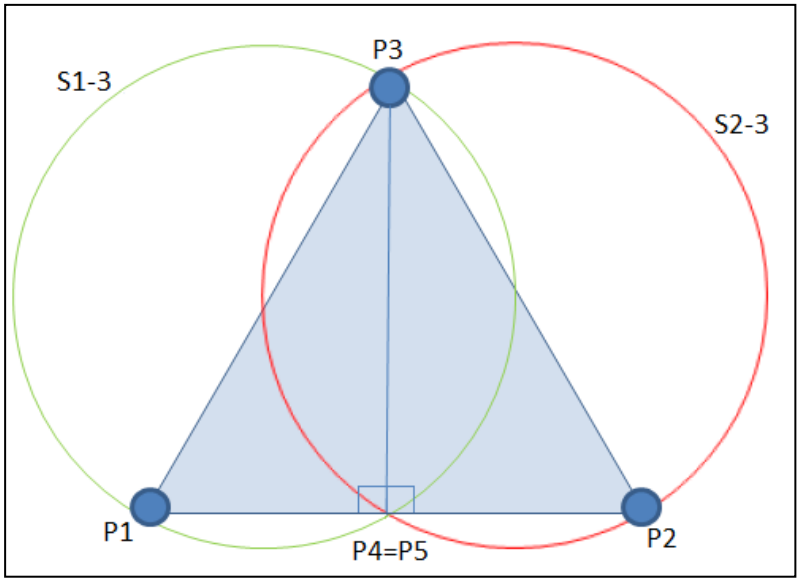
11.9.3 Finne DT basert på Delaunay kantene funnet fra andre fase

Fra fase to har vi informasjon om en mengde Delaunay kanter, og det er mulig at vi kan kombinere mange av disse kantene slik at vi faktisk har flere DT uten at vi trenger å gjøre utvidede søk i fase tre. Dette gjør det mulig for at vi kan unngå de vanlige beregningene for Delaunay triangulering i disse tilfellene, dersom vi har en måte å oppdage disse på. Derfor vil det være nyttig å finne ut hva vi kan bruke, av informasjonen funnet i fase to, for å oppdage DT.

Utgangspunktet vårt er at fase to, der vi har funnet Delaunay kanter fra de *k* nærmeste nabopunktene til hvert punkt, er ferdig, og vi er nå i fase tre slik at vi skal finne resten av Delaunay trianguleringen. Vi er i et gitt punkt, P_1, og har funnet en Delaunay kant til punkt P_2. Dette kan være en kant fra enten fase to eller tre. Nå gjelder det å finne ut hva er tilstrekkelig informasjon fra fase to, slik at vi kan konkludere med at vi allerede har en DT uten å gjøre søk etter en ny.

Ifølge T2 kan vi si at ingen punkter vil være innenfor en sirkel der en Delaunay kant, funnet i fase to, er diameteren. Dette vil vi bruke til å finne ut om disse sirklene vil overlappe med en trekants omskrevne sirkel. Dette var grunnlaget for at en trekant var en gyldig DT, slik det ble definert i 11.1.1. Dersom disse sirklene overlapper kan vi si at dette er en gyldig DT, uten å måtte beregne videre.

For at en trekant skal være en gyldig DT, må vi vise at sirklene fra T2 dekker trekantens areal. De delene av den omskrevne sirkel utenfor trekanten vil alltid dekkes da sirklene, der kantene fra fase to er diameterene, vil strekke lengre utover, samt at den siste kanten er allerede godkjent av fase tre, slik at det er ingen forstyrrende punkter utenfor den. Vi baserer oss videre på punktene P_1 og P_2, og at de deler punktet P_3, som et punkt funnet i fase to. Basert på T2, vil vi få to sirkler, som vi kaller vi $S_{1\text{-}3}$ og $S_{2\text{-}3}$.



Figur 30: *Dersom kantene P_1P_3 og P_2P_3 er funnet fra fase to vil vi kunne dekke hele trekantens areal med sirklene $S_{1\text{-}3}$ og $S_{2\text{-}3}$ der kantene er diameterne. Gitt at begge sirklene treffer kanten P_1P_2, i P_4 og P_5,slik at vinkelen linjesegmentet P_3 til denne kanten er 90°, så må de treffe i samme punkt. P_4 og P_5 er altså samme punktet. Dette betyr at sirklene dekker hele trekanten $P_1P_2P_3$. Dette vil føre til at vi kan si at trekanten $P_1P_2P_3$ er en gyldig DT.*

Ettersom begge sirklene har hver sin kant, som diameter, vil et gitt tredje punkt på sin sirkel gi opphav til en vinkel på 90°. Det vil si at dersom punkt P_1 og P_3 finner et tredje punkt, P_x, på sirkelen $S_{1\text{-}3}$, så vil vinkelen $P_1P_xP_3$ alltid være 90°. Det samme vil gjelde for punktene P_2 og P_3 og deres sirkel $S_{2\text{-}3}$. Dersom vi kaller punktet der $S_{1\text{-}3}$ krysser P_1P_2 for P_4, vil dette gi en vinkel $P_1P_4P_3$ med 90°. Det samme gjelder for den andre sirkelen, altså vi får et punkt P_5 der $S_{2\text{-}3}$ krysser P_1P_2 slik at vi får en vinkel $P_2P_5P_3$ med 90°. Ettersom punkt P_3 kun kan være vinkelrett med kanten P_1P_2 på et bestemt punkt,

vil dette si at sirklene $S_{1\text{-}3}$ og $S_{2\text{-}3}$ krysser kanten P_1P_2 i samme punkt, altså P_4 er lik P_5. Dermed kan vi si at $S_{1\text{-}3}$ og $S_{2\text{-}3}$ til sammen vil dekke hele trekanten $P_1P_2P_3$.

DT funnet på denne måten er ikke nødvendigvis basert kun på Delaunay kanter funnet i fase to. Dette kommer av at kanten P_1P_2 kan være funnet fra enten fase to eller tre. Dette gjør det derimot lettere for fase tre å finne nye trekanter, da den slipper å gjøre utvidede søk etter punkter.

Det vi gjør er å sjekke om den nødvendige informasjonen fra fase to er tilstede slik det er beskrevet ovenfor. Vi kan ignorere det vanlige søket etter neste punkt, og fortsatt til neste trekant, dersom dette slår til. Dersom vi lagrer informasjonen om et punkts nærmeste Delaunay naboer i en separat liste, kan vi anse denne informasjonen fast i fase tre, slik at vi kan sjekke for disse tilfellene utenfor en tråds punktsett i en parallell versjon.

11.9.4 Finne gyldige kanter ut i fra neste nærmeste nabopunkt.

Det er ikke alltid slik at to nabopunkter deler neste nabopunkt, men vi kan enda bruke informasjonen fra forrige fase. Med arbeidet vi allerede har gjort, kan vi se noen elementer vi kan legge som grunnlag for videre trianguleringen.

Alle punkter har minst én kant å bygge fra. Dette kommer av at vi har sett på alle punktenes k nærmeste naboer. Siden alle punkter vil ha ett nærmeste nabopunkt, kan vi ta utgangspunkt i minst én kant i søk etter nye DT.

Vi kan se at alle kanter, unntatt de som er med i den konvekse innhyllingen, i en Delaunay triangulering er en del av to trekanter. Dette kommer av en kant, som er midt i punktsettet, vil danne en trekant med et punkt over og under linjen. For de kantene som er med i den konvekse innhyllingen, vil de kun danne én trekant, fordi de alltid vil ha en side, som ikke har flere punkter utenfor. Denne informasjonen hjelper oss med å finne riktige punkter å lete igjennom, når vi ønsker å finne en DT, nemlig ved å lete etter punkter på kun en side av kanten om gangen.

Etter fase to, vet vi at for et gitt punkt så er alle punktene den danner en kant med sortert mot klokka, slik som det ble beskrevet i 11.6.5. Dette er fordelaktig for videre beregning i fase tre, da vi kan bruke dette, som et utgangspunkt, for hvor vi skal finne nye DT.

Et eksempel på dette er om punkt P_1 skal danne en trekant sammen med punktet P_2, som vi har tatt ut ifra lista om kantene til P_1. Vi vet at punktet, P_3, som kommer etter P_2 i P_1 sin liste, er en god kandidat. Dette kommer av måten naboene til P_1 sortert. Derimot kan det finnes punkter, som er bedre kandidater enn P_3, men disse vil ikke være en del av lista til P_1. Derfor må vi sjekke om det finnes punkter innen sirkelen, som dannes av trekanten $P_1P_2P_3$.

11.9.5 Finne gyldige kanter ut i fra et punkt

Det finnes tilfeller der et punkt ikke kan benytte seg av ideen gitt i 11.9.3. Dette kan for eksempel være der et punkt kun har funnet ett nabopunkt. Dette gjør det umulig å lete videre fra informasjonen fra fase to. Derfor må vi skape et søkeområde etter neste punkt, som utvides til vi finner et akseptabelt punkt.

Søkeområdet vårt baserer seg på ideen om at den gjennomsnittlige DT dannes ved vinkler rundt 60°. Vi danner dermed en sirkel basert på de to punktene, P_1 og P_2, som danner vår grunnlinje, og et gitt punkt P_3, slik at vinkelen $P_1P_3P_2$ ville vært 60°. Dersom vi ikke finner noen akseptable punkter innenfor denne sirkelen, utvider vi sirkelen. Slik fortsetter vi til vi finner minst et annet punkt. Et funnet punkt er definert, som det punktet P_x, som danner den største vinkelen $P_1P_xP_2$. Dette er fordi sirkelen, som dannes rundt disse tre punktene vil ikke inneholde noen andre punkter, slik at vi ikke bryter regelen for DT, slik som vi så i 11.1.1. Dette er en idé fra "*Delaunay triangulation and the convex hull of n points in expected linear time"*[12].

11.9.6 Opprettholde sorteringen fra fase to

For at DT vi finner i denne fasen skal kunne være lett å finne frem til er det fordelaktig å opprettholde sorteringen, som vi gjorde i fase to. Ved å la punktene være sortert mot klokka, vil trekantene være lett gjenkjennelig. Dette kommer av at to og to punkter i en liste for et gitt punkt vil være en del av en trekant med dette punktet. To punkter etter hverandre, i en slik liste, vil ikke ha punkter i mellom seg gitt denne sorteringen. Dette gjør at vi kan allerede etter fase tre, identifisere alle trekantene etter rekkefølgen til hvert punkts sorterte liste over Delaunay kanter.

11.9.7 Rekursjonsmetoden

Metoden ligner på den vi lagde i 11.6.4, da vi er ute etter mye av det samme, nemlig bokser og punktene inne i dem. Vi arbeider altså nede i løvnodene i rekursjonstreet, der vi har de forskjellige radene av bokser. Deretter sender vi hver rad videre slik at vi jobber på én og én boks om gangen.

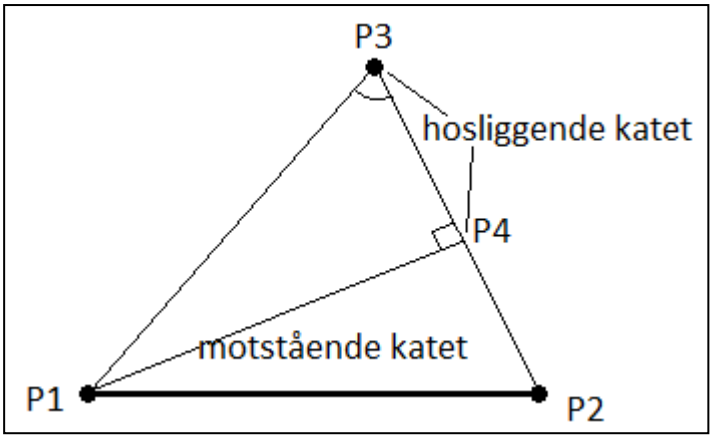
11.10 Geometriske hjelpemetoder for resten av Delaunay trianguleringen

11.10.1 Finne vinkel til et punkt

Vi tenker oss situasjonen der vi har en kant, og er interessert i vinkelen til det nye punktet, som vi danner en trekant med. Vi går igjennom noen få steg for å komme frem til dette. Videre eksempel baserer seg på at vår gyldige Delaunay kant er mellom punktene P_1 og P_2. Det tredje punktet er P_3. Det vi ønsker å gjøre er å la P_1 og P_3 være del av en rettvinklet trekant, slik at vi kan bruke trigonometriske funksjoner, som cotangens, for å finne en sammenligningsverdi for vinkelen til P_1P_3 P_2. Vi bruker den negative verdien fra cotangens fordi denne stiger fra -180° til 180°. På denne måten får vi en enkel verdi, som representerer vinkelen vi ønsker.

1. Først trenger vi likningen til kanten P_2P_3.
2. Deretter trenger vi kanten, som er den korteste lengden fra P_1 til linjen P_2P_3. Dette vil skape en 90° vinkel mellom kantene våre. Punktet, der kanten fra P_1 treffer kanten P_2P_3, kaller vi P_4.
3. Nå har vi en trekant mellom $P_1P_3P_4$, som er rettvinklet. Det betyr at vi kan bruke cotangens for å finne en sammenligningsverdi for P_3, slik at vi kan vite om dette er den punktet som gir størst for vår trekant. Siden Java ikke har cotangens innebygget må vi bruke noen andre metoder for å komme frem til riktig svar. Vi gjør verdien negativ fordi da gir cotangens oss en økende kurve.

$$-cotan\theta = -\frac{1}{tan\theta} = -\frac{1}{\left(\frac{motstående\ katet}{hosliggende\ katet}\right)} = -\frac{hosliggende\ katet}{motstående\ katet}$$



Figur 31: *Vi ønsker å finne vinkelen til $P_1P_3P_2$. Vi finner en rettvinklet trekant med P_1 og P_3, slik at vi kan bruke den negative cotangensverdien med kantene hosliggende katet og motstående katet.*

11.11 Inkludere denne trianguleringen, som en del av faseoppdelingen

11.11.1 Den parallelle fasen

Dette vil være den avsluttende fasen i Delaunay trianguleringen. Det vil si at vi her må kunne finne hele trianguleringen. Denne parallelle fasen vil inneholde rekursjonsmetoden vist i 11.9.7. Det vil føre til at vi arbeider med å finne resten av kantene, som er i hele punktsettet. Med denne rekursjonsmetoden følger alle de geometriske operasjonene, som for eksempel at vi finner største vinkel. Denne parallelle fasen vil komme etter den sekvensielle fasen til metoden for å finne hvert punkts k nærmeste naboer.

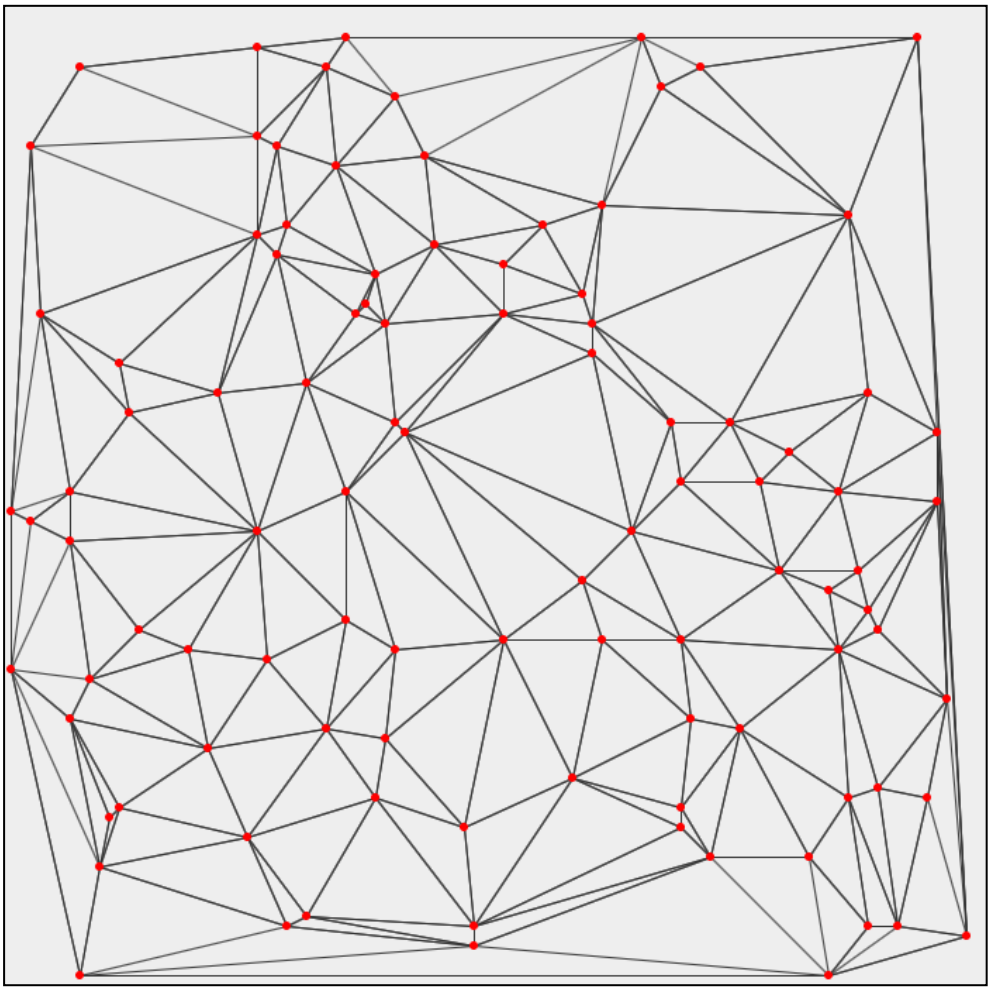
Forutsetningen for denne fasen er at alle punkter har minst én gyldig Delaunay kant til et annet punkt. Denne forutsetningen oppfylles fordi vi vet at alle punkter har fått en kant til sitt nærmeste nabopunkt. Det andre forutsetningen vi har er at den konvekse innhyllingen er funnet. Dette spiller en rolle der vi velger antall trekanter vi skal lete igjennom per kant, nemlig at kanter i den konvekse innhyllingen er med i kun én

trekant, mens resten er med i to. Dette ble gjort i første fase, som gjør at denne forutsetningen også er oppfylt.

11.12 Eksempel på en løst Delaunay triangulering

11.12.1 En triangulering på 100 punkter

For å illustrere et eksempel på hvordan en Delaunay triangulering av 100 uniformt fordelte, men unike, punkter vil se ut, er programmet utvidet slik at den kan vise punktene og kantene i et bilde. Dette viser hva vi faktisk er ute etter når vi lager et slikt program, som beregner seg fram til Delaunay trianguleringen av et punktsett.



Figur 32: *En Delaunay trianguleringen av 100 punkter, som er beregnet fram av programmet forklart i dette kapittelet.*

11.13 Parallell Delaunay Triangulering

11.13.1 Det genererte programmet fra Java PRP

Java PRP generer et parallelt program av Delaunay triangulering basert på det sekvensielle programmet beskrevet i dette kapittelet. Det vil være et faseoppdelt program, slik at den går igjennom flere parallelle faser for å komme frem til resultatet. Programmet baserer seg på at vi ikke har noen synkroniseringsmetoder for å opprettholde dataene våre. Dette kommer av at det sekvensielle programmet er laget på en slik måte at en parallell versjon skal ikke trenge dette.

11.13.2 Den administrative metoden

Kravet for at Java PRP skal kunne parallellisere en slik faseoppdeling, er at det sekvensielle programmet legger til grunn en administrativ metode, som har kontrollen over flyten til programmet. Løftet fra Java PRP til brukeren, som lagde dette sekvensielle programmet, er at denne administrative metoden vil eksekveres sekvensielt.

For vår triangulering vil programmet bestå av tre parallelle faser og to sekvensielle faser. De tre parallelle fasene er:

1. Finne den konvekse innhyllingen for punktsettet.
2. Finne de k nærmeste naboene for hvert punkt.
3. Beregne resten av Delaunay trianguleringen.

Vi har ikke i dette programmet et behov for en sekvensiell fase etter hver eneste parallelle fase. Vi har følgende sekvensielle faser:

1. Vi har en fase etter vi har funnet den konvekse innhyllingen, fordi vi ønsker at alle disse punktene etter hverandre i en bestemt rekkefølge. Dette gjør at når vi etter på ser på denne mengden punkter, skal vi kunne følge punktsettet vårt i samme vei.
2. Etter at vi har funnet de k nærmeste naboene trenger vi å spre denne informasjonen over til andre punkter involvert. Vi må også sortere punktenes liste over punkter de deler kant med.

Den tredje parallelle fasen, som er resten av Delaunay trianguleringen, trenger ikke en sekvensiell fase da den er ferdig sortert mens vi beregner og alle punkter vil finne alle sine kanter av seg selv. DT finner vi fordi vi opprettholder sorteringen fra fase to.

11.14 Kjøretidstester

11.14.1 Grunnlag for testing

Hele denne oppgaven baserer seg på at vi ønsker at Java PRP skal kunne generere en parallell versjon av Delaunay Triangulering, som gir oss bedre kjøretider enn den sekvensielle. For å finne ut om dette er sant, setter vi i gang noen tester, som vil gi oss kjøretider for forskjellige mengder punkter, for både den sekvensielle og parallelle Delaunay Trianguleringen.

Vi kjører koden i en løkke på 11 ganger, og finner medianen for disse kjøretidene. Punktene baserer seg på uniformt fordelte tallverdier i sine koordinater, bortsett fra at alle punktene må unike. Koordinatene vil variere fra 0 til n-1, i et koordinatsystem basert på største verdiene for x og y, det vil si verdiene er bestemt ut i fra punktene, som har høyest x- og y-verdi. Vi legger opp til at fase to av programmet leter blant sine ti nærmeste nabopunkter for å finne gyldige Delaunay kanter.

Tidene blir målt i nanosekunder for presisjon, og fremvist her i millisekunder for lesbarhet. Tabellen vil vise K for tusen, og M for million. Testene baserer seg på en Intel core I7 870 prosessor med en klokkefrekvens på 2,93 GHz. Den har 4 fysiske kjerner, som er hyperthreaded slik at vi vil kunne utnytte dette som 8 kjerner.

11.14.2 Testing og resultater

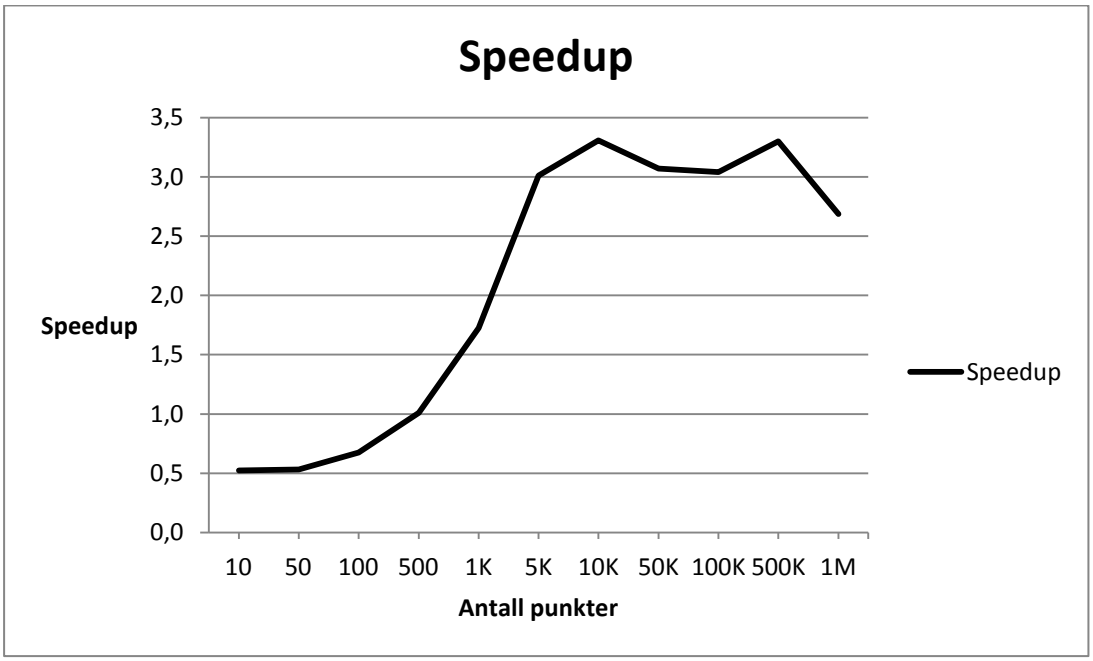
Vi vil her presentere resultatene av de forskjellige kjøretidene vi har fått fra de forskjellige kjøringene, for både sekvensiell og parallell Delaunay triangulering. Vi vil også vise speedup vi får med parallell kjøring.

	Kjøretider(ms) for		
Antall punkter	Sekvensiell Delaunay triangulering	Java PRPs parallelle Delaunay triangulering	Speedup
10	2	3	0,5
50	5	10	0,5
100	8	11	0,7
500	16	16	1,0
1K	27	16	1,7
5K	118	39	3,0
10K	215	65	3,3
50K	1 136	370	3,1
100K	2 280	750	3,0
500K	14 084	4 268	3,3
1M	30 515	11 359	2,7

Tabell 14: *Kjøretider til sekvensiell og parallell Delaunay triangulering, generert av Java PRP, i millisekunder. Fase to leter blant de 10 nærmeste nabopunktene. Disse kjøretidene er basert på en Intel core I7 870 prosessor med en klokkefrekvens på 2,93 GHz, med 8 kjerner (4 fysiske med hyperthreading).*

11.14.3 Speedup vist i en graf

Følgende graf viser hvordan speedup utvikler seg, når antallet punkter øker.



Graf 6: *Viser speedup vi får ved å beregne Delaunay triangulering i parallell, generert av Java PRP. Dataene er hentet fra Tabell 14 og baserer seg på en Intel core I7 870 prosessor med en klokkefrekvens på 2,93 GHz, med 8 kjerner (4 fysiske med hyperthreading).*

Allerede fra 1000 punkter er den parallelle versjonen 1,7 ganger bedre enn den sekvensielle. Vi kan også se at den grafen treffer et høydepunkt på rundt 3,3 i speedup. Vi går litt ned for 1 million punkter. Dette kan ha noe med at vi begynner å bruke hovedminne, istedenfor cachene våre, slik at det tar lengre tid å aksessere data.

11.14.4 Lastbalanse

Med Quicksort fikk vi bedre speedup ved å lage flere tråder enn kun antall kjerner tilgjengelig. Istedenfor for at vi skapte tråder lik antall kjerner, tok vi antall kjerner ganget med ti. På denne måten fikk man mindre, men flere oppgaver til trådene, slik at det ble mindre venting mellom tråder som ble fort ferdig kontra de som tok lang tid. For å teste hvordan dette programmet gjør det med slik lastbalanse sammenlignes det to versjoner der det kjøres tråder lik antall kjerner, altså uten lastbalanse, og den andre kjører ti ganger flere tråder. Ellers er programmene helt like. Vi beregner forbedringen mellom de to parallelle versjonene slik:

$$forbedring = \frac{kjøretid\ til\ Java\ PRP\ sin\ parallelle\ Delaunay\ triangulering}{kjøretid\ til\ parallell\ Delaunay\ triangulering\ med\ lastbalanse}$$

Antall punkter	Kjøretider(ms) for			Speedup for		Forbedring med lastbalanse
	Sekvensiell Delaunay triangulering	Java PRPs parallelle Delaunay triangulering	Parallell Delaunay triangulering m/ lastbalanse	Java PRPs parallelle Delaunay triangulering	parallell Delaunay triangulering m/lastbalanse	
10	2	3	3	0,5	0,5	1,0
50	5	10	9	0,5	0,6	1,1
100	8	11	9	0,7	0,9	1,3
500	16	16	25	1,0	0,6	0,6
1K	27	16	43	1,7	0,6	0,4
5K	118	39	201	3,0	0,6	0,2
10K	215	65	440	3,3	0,5	0,1
50K	1136	370	455	3,1	2,5	0,8
100K	2280	750	707	3,0	3,2	1,1
500K	14084	4 268	3940	3,3	3,6	1,1
1M	30515	11359	11060	2,7	2,8	1,0

Tabell 15: *Kjøretider til to parallelle og en sekvensiell Delaunay triangulering i millisekunder. Parallell med lastbalanse kjører i gang 10 ganger flere tråder enn den generert av Java PRP. Fase to leter blant de 10 nærmeste nabopunktene. Disse kjøretidene er basert på en Intel core I7 870 prosessor med en klokkefrekvens på 2,93 GHz, med 8 kjerner (4 fysiske med hyperthreading).*

I denne tabellen ser vi stor variasjon i hvilken versjon som er bedre i forhold til kjøretider. For kjøretidene der antall punkter er mellom 10 til 100, ser vi at versjonen med lastbalanse er bedre, fordi den ikke parallelliserer oppgavene da rekursjonsmetodene ikke klarer å skape nok rekursjonsinstanser, og for så få punkter er dette fordelaktig. Der antall punkter er mellom 500 og 50 000 er Java PRP uten lastbalanse ganske mye bedre. Dette kommer av at versjonen med lastbalanse fortsatt kjører den sekvensielle versjonen av fasene. Over 100 000 punkter er det såvidt bedre med lastbalanse. Ettersom Java PRP med lastbalanse ikke klarer å parallellisere før 100 000 punkter og over, er den ikke like god som uten lastbalanse.

Med Quicksort oppgaven var det ingen problem å skape nok rekursjonsinstanser, slik at vi hadde nok oppgaver i Java PRP. De forskjellige fasene i dette Delaunay trianguleringsprogrammet klarer ikke å generere like mange oppgaver. Dette fører til at en parallell versjon med slik lastbalanse, vil kun være fordelaktig for stor nok mengde av antall punkter. Dersom Java PRP ikke klarer å få nok oppgaver til sine tråder, vil den kjøre den sekvensielle versjonen av fasen istedenfor. Dette gjør at gjør at vi kommer frem til flere alternativer for å parallellisere dette Delaunay trianguleringsprogrammet i forhold til lastbalanse.

Dersom vi vil inkludere lastbalanse kan vi gjøre endringer i rekursjonsmetodene. Det vil si at vi må endre metodene, slik at de klarer å dele opp i flere rekursjonsinstanser, og dermed generere nok oppgaver til Java PRP. En endring vi kan gjøre i rekursjonsmetodene i fase to og tre er å dele opp i enkelte bokser istedenfor rader. Dette vil føre til flere rekursjonsinstanser, og vil vi med stor sannsynlighet klare å implementere en parallell versjon med en lastbalanse. Det som gjør denne ideen vanskelig er at en eventuell rekursjonsmetode må følge gitte former for å være akseptabel for Java PRP. Dersom vi skal dele opp en todimensjonal array, kan det ikke være ulike rekursjonskall basert på om vi traverserer radene eller kolonnene.

Et annet alternativ er å gjøre boksene mindre, slik at vi får enda flere rader. Dette gjør at de nåværende metodene vil dele seg opp enda mer. Boksstørrelsen er beregnet basert på hvor mange punkter vi vil ha i gjennomsnitt per boks. Den er nå på 4, slik at dersom vi reduserer dette antallet, kan det gå utover kjøretiden til diverse metoder, som står for søking etter punkter, da den oftere må utvide søket sitt til flere bokser. Dette gjør at det ikke nødvendigvis er bedre å gjøre boksene enda mindre.

Vi kan også endre det parallelle programmet slik at antall tråder den kjører i gang ikke baserer seg på en statisk verdi, men heller baserer den på en verdi, som sier noe om hvor mye rekursjonsmetoden kommer til å dele seg. For fase to og tre vil dette si at antall tråder vil baserer seg på antall punkter. På en slik måte kan vi implementere lastbalanse, som varierer basert på hvor mange punkter vi skal triangulere. Dette gjør at vi vil oppnå parallellitet for et ganske lite antall punkter, samtidig som vi kan ha lastbalanse for et større antall punkter. Det, som gjør dette problematisk, er at det parallelle programmet blir generert av Java PRP, som har mindre innsyn i programmet enn om vi hadde laget det parallelle selv. Det vil si at den ikke vet hvor informasjonen om antall punkter er. For at dette skulle fungert måtte vi ha implementert en versjon av Java PRP der brukeren kan markere en verdi, som forteller hvor mange rekursjonsinstanser den kan generere.

Valgene vi har er dermed å ha en litt bedre speedup for et stort antall punkter, eller om programmet vårt skal kunne parallellisere for mindre punktsett. I dette parallelle programmet er det ikke inkludert lastbalanse, da vi ønsker at trianguleringen skal være mest mulig parallell. Vi så også fra kjøretidene i Tabell 15 at vi ikke fikk så veldig stor forbedring ved å inkludere lastbalanse. I mange tilfeller var det også en ulempe selv for et stort antall punkter.

11.15 Konklusjon

11.15.1 Resultatene fra parallell Delaunay triangulering

Delaunay triangulering er eksempelet vårt for Java PRP sin generering av parallelle, faseoppdelte programmer. Kjøretidene vi fikk, der vi sammenlignet sekvensiell Delaunay triangulering med en parallell versjon generert av Java PRP, viste oss at vi fikk en god speedup med den parallelle. Dette forteller oss at slike faseoppdelte programmer har gode muligheter for å forbedres med parallellisering, dersom det blir gjort riktig. Ved å følge Java PRP sine krav for slike programmer, ser vi at det ble en bra forbedring, i forhold til kjøretider, for Delaunay triangulering.

Vi var også innom hvordan dette programmet fungerte dersom vi inkluderte lastbalanse i den parallelle versjonen av Delaunay trianguleringen. Ved å øke antall tråder vi startet, fikk vi en liten speedup for noen av kjøringene med større antall punkter, men den ville ikke parallellisere før vi nådde 50 000 punkter. Dette gjorde at det ikke virket fordelaktig, i dette tilfellet, å inkludere noe slikt i Java PRP. I 11.14.4 ble det diskutert problemene rundt lastbalanse og det parallelle Delaunay trianguleringsprogrammet, generert av Java PRP.

12. Delaunay triangulering og delte variabler

12.1 Delaunay Triangulering med hensyn til parallellitet

12.1.1 De delte variablene i koden

I programmet for Delaunay trianguleringen er punktene felles data. Bortsett fra koordinater, inneholder punktobjektene informasjon om forskjellige tilstander og relasjoner til andre punkter, som vi bygger opp mens går igjennom de forskjellige fasene til programmet. Eksempler, på slik informasjon, er en liste av gyldige Delaunay nabopunkter og en tilstandsvariabel som sier om punktet er en del av den konvekse innhyllingen.

Av disse variablene, innenfor hvert punktobjekt, er det nyttig å se hva vi kan lese og skrive til i programmets forløp. Det aller første vi kommer til er punktenes koordinater. Koordinatene er faste variabler, som opprettes ved oppstart og endres ikke igjennom programmet. Dette gjør at alle trådene kan lese alle punktenes x- og y-verdier.

Hvert punkts liste over gyldige Delaunay nabopunkter er ikke faste variabler, fordi disse vil endres flere ganger igjennom alle fasene av programmet. Dette betyr at hvert punkts liste kan kun endres av én tråd. Det vil si at det ikke kan endres av noen andre tråder, da disse listene vil endre seg i løpet av programmets levetid. Derimot kan en tråd se på flere punkter, dersom vi vet at samme tråd har arbeidet, eller kommer til å arbeide, på disse punktene.

Tilstandsvariabelen, for om punktet er med i den konvekse innhyllingen, blir satt på alle punktene det gjelder fra første fase. Dette betyr at trådene kan lese denne verdien i alle punkter fra og med den andre fasen.

Etter at punktsettet er opprettet i starten av programmet, lager vi også boksene, som vil inneholder de forskjellige punktene. Disse vil på samme måte, som punktene, være faste variabler. Det vil si at en boks vil ikke endre hvilke punkter den inneholder i løpet av programmet.

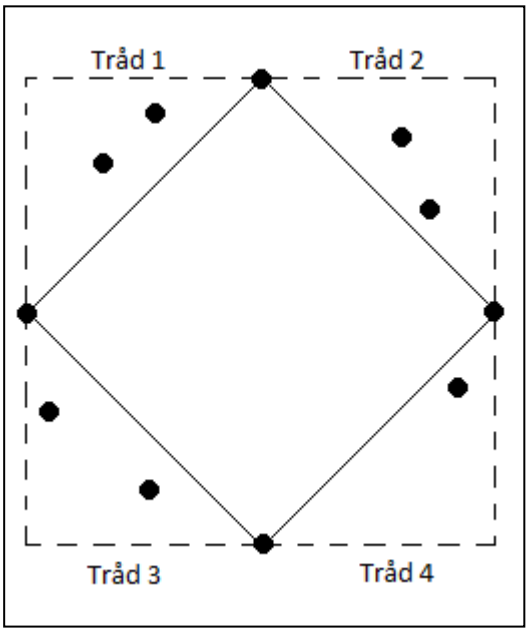
12.2 De enkelte fasene og delte variabler

12.2.1 Den konvekse innhyllingen

For å finne den konvekse innhyllingen av et punktsett, må vi passe på hvordan vi bruker de forskjellige punktene med hensyn til en parallell eksekvering. Løsning ble en rekursiv metode, som deler opp punktsettet i forskjellige deler, slik at hver eventuelle tråd arbeider på hver sin mengde av punkter alene. På denne måten vil hver tråd finne den konvekse innhyllingen av sin lille del av det totale punktsettet. Når metoden returnerer oppover i rekursjonen vil vi sette sammen delene av den konvekse innhyllingen, slik at vi til slutt ender opp med hele. Deretter sorteres de slik at de følger i en linje rundt punktsettet.

Metoden gir mulighet for hver tråd å arbeide uten å se på noen andre punkter for å komme frem til resultatet sitt. Dette fører at vi ikke trenger noen form for synkroniseringsmetoder for å opprettholde datastrukturene våre. Resultatet er en metode, som er god for parallell eksekvering. Figur 33 viser hvordan hver tråd har sine egne deler av punktsettet å arbeide på

.



Figur 33: *Et eksempel på starten av beregningen av den konvekse innhyllingen. Vi deler opp punktmengden, slik at de fire trådene i dette eksempelet arbeider på hvert sitt hjørne. Dette fører til at de arbeider selvstendig på sine punkter, og trenger ikke å se på noen andre. Dette vil gjelde videre i rekursjonen også.*

12.2.2 Søkingen etter de k nærmeste Delaunay kantene til punktene

Når vi søker etter potensielle Delaunay nabopunkter trenger vi både punktene og boksene. Dette gjør at vi må gjøre det klart hva en tråd kan gjøre med andre punkter, og hva den ikke kan.

Søkemetoden er en rekursiv algoritme, som deler opp i rader av bokser. Disse radene blir sendt til en annen metode der de deles opp i de individuelle boksene. Dette gir oss flere fordeler i en parallell kjøring av programmet:

- Ettersom vi vet at en tråd går igjennom alle boksene i en rad, vet vi noe om hvilke punkter tråden kommer til å arbeide på, og hvilke den har arbeidet på før. Dette gjør at vi kan lese og skrive til punkter, som er i samme rad, til tross for at de ikke er i samme boks.
- Selv om vi har informasjon om en rad, går vi igjennom én og én boks, hver for seg. Dette er i følge rekursjonsmetoden beskrevet i kapittel 11.6.4 om selve programmet for Delaunay triangulering.

I denne fasen er det noen deler av punktenes informasjon, som er tilgjengelig for oss. Alle punkters koordinater kan leses av alle trådene. Det samme gjelder tilstandsvariabelen om punktet er med i den konvekse innhyllingen, da denne kan sees på som en fast variabel etter første fase. Derimot kan ikke listen over de gyldige Delaunay nabopunktene leses, eller skrives til. Man kunne tenke seg at det hadde vært greit at hvis punkt A setter punkt B, som nabopunkt, burde vi også sette punkt A som nabo til punkt B. Dette kan føre til feil i programmet, da vi ikke vet om punkt B skrives til av en annen tråd på samme tidspunkt. Vi kan heller ikke lese listene til andre punkter, da vi ikke vet tilstanden til lista, som for eksempel om den er ferdiglaget, ikke påbegynt enda eller at en tråd arbeider på nåværende tidspunkt.

Metoden er laget med fokus på delte variabler, slik at vi ikke behøver synkroniseringsmetoder. Dette er på samme måte som første fase, og vil føre til at en parallell eksekvering vil ende opp med bedre kjøretider. Når den parallelle fasen er over og vi går over til den sekvensielle, kan vi sortere Delaunay nabopunktene. Vi setter også alle nabopunktene i alle sine lister. Det vil si at dersom punkt A er Delaunay nabo til punkt B, setter vi også at B er Delaunay nabo til A.

12.2.3 Beregningen til resten av Delaunay trianguleringen

Framgangsmåten til rekursjonsmetoden, som beregner resten av trianguleringen, er ganske lik den vi fikk i 12.2.2. Den baserer seg på mye av samme måte å komme frem til hvert enkelte punkt, nemlig at vi først deler opp i rader og deretter tar en annen metode og deler opp i enkelte bokser.

Vi kan bruke informasjonen vi fikk fra den andre fasen, til og finne Delaunay trekanter slik at vi unngår å bruke standard søking hver gang. Vi lar informasjonen om ett punkts Delaunay kanter funnet blant de k nærmeste naboene være lagret i en ekstra liste, slik at vi kan lett skille mellom hvilke Delaunay nabopunkter er funnet fra fase to, og hvilke som er funnet fra fase tre. Her vil fase tre kun lese denne informasjonen, og ikke skrive til den. Dette fører til at dette er data vi ikke endrer, og trådene kan lese dette utover sine egne punktsett.

Slik som resten av metodene presentert i 12.2, vil vi heller ikke her trenge noen synkroniseringsmetoder. Dette er blant den tyngste fasen kjøretidsmessig, slik at bruken av synkroniseringsmetoder vil føre til at programmet ville gått merkbart mye tregere.

12.3 Faseoppdelt og parallell Delaunay triangulering

12.3.1 En kode bygget for parallellisering

Mye av det vi har sett på hittil er hvordan delte variabler påvirker parallelle programmer, med hensyn til kjøretider. På en side ønsker vi at programmet skal være bedre enn den sekvensielle versjonen, men samtidig må programmet kunne produsere et riktig resultat hver gang.

I det sekvensielle programmets oppbygning har delte variabler, og en fremtidig parallell versjon, vært i fokus. Ved å se på hvordan vi håndterer de delte variablene i løpet av hele programmet, har vi klart å oppnå en forbedring ved å bruke det parallelle programmet framfor det sekvensielle, med hensyn til kjøretider. Oppdelingen av arbeidet til Delaunay triangulering, slik at vi har mindre håndterbare faser, har ført til at vi har oppnådd et parallellitetsvennlig program. Ved å dele opp i disse fasene kunne vi lettere finne hvordan disse delte variablene ble brukt og hvordan disse burde håndteres i et parallelt program. I hele programmet har det ikke vært nødvendig å lage en eneste *synchronized* metode.

12.3.2 Data fra en fase er kun leselige i videre faser

En måte vi kan være sikker på at en variabel er sikker for overskrivninger fra flere tråder samtidig, er å gjøre variabelen kun leselig, slik at ingen skriver på den. På grunn av at det er såpass klare skiller mellom fasene i denne Delaunay trianguleringen er implementasjonen av programmet slik at vi lar en fase ha kun leserettigheter til resultatene til tidligere faser. For eksempel, så ble det nevnt i 12.2.3 at Delaunay kantene funnet fra fase to er markert i en annen liste, slik at fase tre kan skille mellom alle Delaunay kantene, og de som kun er funnet fra fase to. Siden fase tre ikke kan overskrive denne listen, kan alle trådene lese alle punkters informasjon om sine Delaunay kanter funnet i fase to.

13. Statistikk angående å finne Delaunay kanter blant de k nærmeste nabopunktene

13.1 Utnytte den andre fasen best mulig

13.1.1 Mange måter å bruke fase to

Delaunay trianguleringsprogrammet går ikke direkte til tradisjonelle former for å løse Delaunay trianguleringer på, men går igjennom en ekstra fase, nemlig det å finne de k nærmeste Delaunay nabopunktene, for en gitt verdi k. Det kan derfor være nyttig å se hvordan vi kan bruke dette best mulig. Dette inngår hvor mange nabopunkter er best å se på i forhold til kjøretid. Vi ønsker også å finne ut hvor mange Delaunay kanter og DT vi finner i denne fasen.

Resultatene vi får fra dette kapittelet gir også et innblikk i hva Delaunay trianguleringsprogrammet baserer seg på. For eksempel, vil vi her se hvilken verdi k fikk og hvorfor.

13.2 Antall naboer å lete igjennom

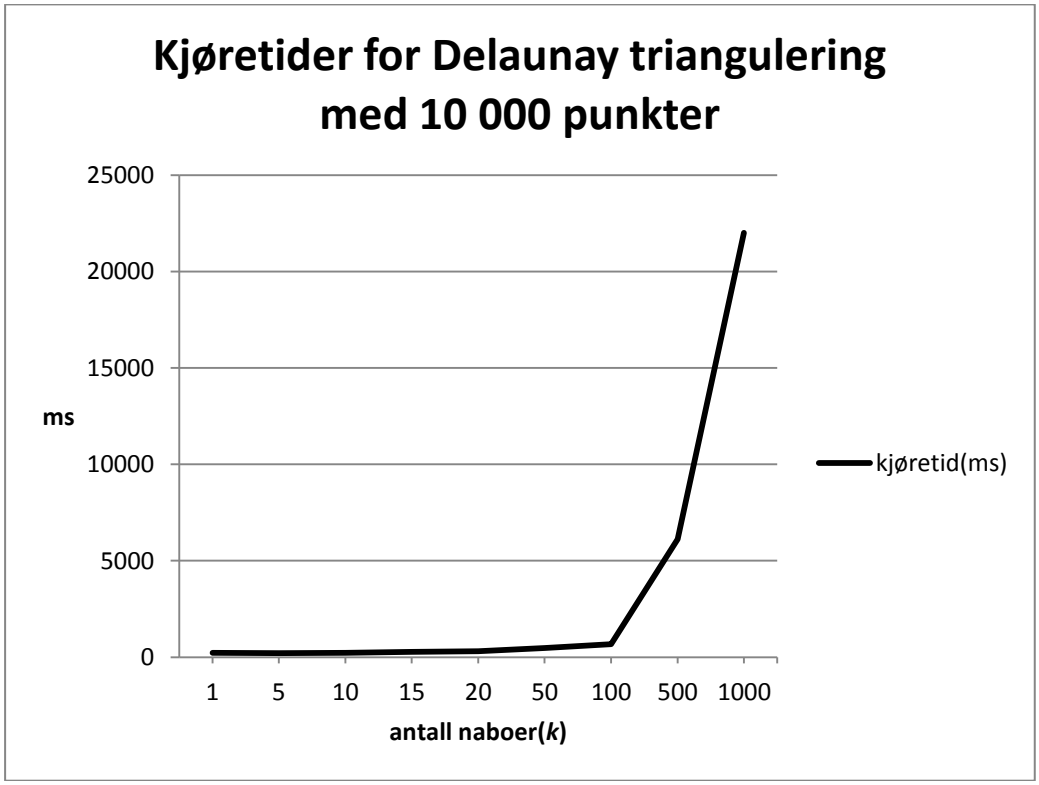
13.2.1 Hvor mange k nærmeste nabopunkter

Det kan være nyttig å finne ut hvor mange naboer vi skal lete etter, for å få en bedre kjøretid. Man kan se på dette på to måter. Ved å lete etter få naboer, vil vi naturligvis finne færre gyldige Delaunay kanter i fase to, men samtidig vil vi ikke bruke så lang tid på å lete etter nye naboer. På den andre siden kan vi lete etter mange naboer, der vi finner flere Delaunay naboer, men vi bruker lengre tid.

For 10 000 punkter totalt	
antall naboer	kjøretid(ms)
1	221
5	214
10	220
15	277
20	316
50	479
100	668
500	6116
1000	22015

Tabell 16: *Viser hvor lang tid det tar programmet å løse Delaunay trianguleringen for 10 000 punkter, for forskjellige antall gyldige Delaunay nabopunkter vi leter etter i fase to. Dataene baserer seg på en Intel core I7 870 prosessor med en klokkefrekvens på 2,93 GHz, med 8 kjerner (4 fysiske med hyperthreading).*

Vi ser at det tar lengre tid når vi leter etter flere naboer. Dette stemmer fordi det tar lengre tid å søke lengre ut i koordinatsystemet, samtidig som det blir vanskeligere å finne gyldige Delaunay nabopunkter etter som vi finner flere punkter. For et gitt punkt A og et nabopunkt B kan alle punkter som er nærmere A enn B mulig ødelegge for at A og B er gyldige Delaunay nabopunkter. Derfor vil det være vanskeligere å finne slike naboer ettersom vi finner flere punkter, da det er flere punkter mellom A og B.



Graf 7: *Baserer seg på tallene i Tabell 16. Viser en stor økning i kjøretider når vi øker verdien k. Dataene baserer seg på en Intel core I7 870 prosessor med en klokkefrekvens på 2,93 GHz, med 8 kjerner (4 fysiske med hyperthreading).*

13.3 antall Delaunay kanter og DT funnet

13.3.1 Statistikk angående antall kanter og trekanter

Det er mer enn kun kjøretider, som er interessant å se på, når vi ønsker å se hvordan fase to fungerer i Delaunay trianguleringsprogrammet vårt. Ettersom vi er ute etter en triangulering, som et resultat, kan det være nyttig å se hvor mye fase to bidrar med, i forhold til Delaunay kanter og fullstendige DT.

Videre vil vi se på forholdet mellom hvor mange gyldige Delaunay kanter fase to finner og det totale antallet når trianguleringen er fullført. Grunnlaget for testene er ganske lik det vi gjorde i 11.14. For hvert antall kjører vi programmet 10 ganger og finner et gjennomsnitt av antall unike Delaunay kanter etter fase to, og et gjennomsnitt av det totale antallet når trianguleringen er ferdig. Vi kjører denne testen to ganger, en gang hvor fase to leter blant de fem nærmeste nabopunktene og en gang hvor den leter blant de ti nærmeste. Vi gjentar dette for DT, altså hvor mange unike DT fase to oppdager i forhold til det totale antallet DT i trianguleringen.

Resultatet er at vi finner ut av hva fase to finner i forhold til det komplette antallet ved fullført triangulering, både for Delaunay kanter og DT. For statistikken med Delaunay kanter er det også inkludert hvor mange gyldige Delaunay kanter fase to finner per punkt i gjennomsnitt.

13.3.2 Antall Delaunay kanter funnet

Vi vil her finne ut hvor mange kanter vi finner sammenlignet med det totale antallet. Vi leter her etter antall unike kanter, slik at hvis punkt A finner punkt B som et nabopunkt, og omvendt, vil det telle som én kant. Vi beregner det totale antallet kanter slik[9]:

$$Antall\ kanter = 2n + 3m - 3$$

Her er *n* antall punkter i den konvekse innhyllingen, mens *m* er antall punkter innenfor den konvekse innhyllingen. Det første vi sjekker er hvor mange Delaunay kanter vi finner i fase to dersom vi leter etter de fem nærmeste nabopunktene per punkt.

Antall punkter	Antall kanter		antall Delaunay nabopunkter, per punkt, funnet i fase 2 (leter blant de 5 nærmeste nabopunktene)	% av antall Delaunay kanter funnet i fase 2 i forhold til det totale antall Delaunay kanter
	Etter fase 2 (leter blant de 5 nærmeste nabopunktene)	Totalt i den komplette trianguleringen		
10	12	21	2,28	57,49 %
50	74	137	2,72	54,12 %
100	156	285	2,85	54,74 %
500	816	1 480	2,99	55,16 %
1K	1 639	2 979	2,98	55,03 %
5K	8 290	14 974	3,03	55,36 %
10K	16 586	29 972	3,03	55,34 %
50K	83 278	149 969	3,04	55,53 %
100K	166 783	299 968	3,05	55,60 %
500K	834 598	1 499 963	3,05	55,64 %
1M	1 669 384	2 999 958	3,05	55,65 %

Tabell 17: *Her ser vi hvor mange kanter i Delaunay trianguleringen fase to finner i forhold til det totale antallet etter fase tre. Fase to legger opp til at vi leter etter 5 gyldige Delaunay nabopunkt per punkt.*

Når fase to leter blant de fem nærmeste nabopunktene ser vi at prosenten for antall kanter funnet holder seg ganske konsekvent i overkant av 55 % for mengder over 100 punkter. Dersom vi øker antall Delaunay naboer vi leter etter til ti får vi følgende data.

Antall punkter	Antall kanter		antall Delaunay nabopunkter, per punkt, funnet i fase 2 (leter blant de 10 nærmeste nabopunktene)	% av antall Delaunay kanter funnet i fase 2 i forhold til det totale antall Delaunay kanter
	Etter fase 2 (leter blant de 10 nærmeste nabopunktene)	Totalt i den komplette trianguleringen		
10	12	20	2,42	61,42 %
50	82	135	3,19	60,27 %
100	171	285	3,36	60,03 %
500	925	1 480	3,61	62,53 %
1K	1 869	2 979	3,65	62,74 %
5K	9 535	14 973	3,72	63,68 %
10K	19 159	29 971	3,74	63,93 %
50K	96 360	149 968	3,76	64,25 %
100K	192 871	299 966	3,76	64,30 %
500K	966 160	1 499 962	3,77	64,41 %
1M	1 933 438	2 999 957	3,77	64,45 %

Tabell 18: *Her ser vi hvor mange kanter i Delaunay trianguleringen fase to finner i forhold til det totale antallet etter fase tre. Fase to legger opp til at vi leter etter 10 gyldige Delaunay nabopunkter per punkt.*

Når vi øker antall naboer vi leter etter til ti, holder vi oss ganske stabilt rundt 64 % for over 5000 punkter. Vi finner nesten 10 % mer Delaunay kanter i fase to dersom vi leter blant de ti nærmeste nabopunktene istedenfor fem.

13.3.3 Antall gyldige DT funnet

Videre fra antall Delaunay kanter, kan vi se hvor mange fullstendige DT vi finner i fase to. Vi baserer oss på at vi finner disse DT slik det ble beskrevet i 11.9.3. Det betyr at når vi leter etter en slik DT begynner søket med en kant, som kan enten være fra fase to eller tre, slik at en slik DT ikke er kun fra frase to. Her vil vi også lete etter unike trekanter, slik at trekanten ABC og BCA teller som kun én DT. For å beregne det totale antallet trekanter bruker vi følgende formel[9]:

$$antall\ trekanter = n + 2m - 2$$

Slik som i 13.3.2, er *n* antall punkter i den konvekse innhyllingen, mens *m* er antall punkter innenfor den konvekse innhyllingen.

Antall punkter	Antall Delaunay trekanter		% av antall gyldige DT funnet i fase to i forhold til det totale antall gyldige trekanter
	Etter fase 2 (leter blant de 5 nærmeste nabopunktene)	Totalt i den komplette trianguleringen	
10	6	12	53,78 %
50	41	87	47,00 %
100	73	186	39,29 %
500	389	982	39,63 %
1K	785	1 979	39,67 %
5K	3 928	9 975	39,38 %
10K	7 959	19 974	39,847 %
50K	39 877	99 969	39,889 %
100K	79 880	199 968	39,946 %
500K	399 444	999 964	39,946 %
1M	799 233	1 999 960	39,962 %

Tabell 19: *Viser hvor mange DT i Delaunay trianguleringen fase to finner i forhold til det totale antallet etter fase tre. Disse trekantene kan bygge fra en kant funnet i fase tre. Fase to legger opp til at vi leter etter 5 gyldige Delaunay nabopunkter per punkt.*

Tabell 19 viser oss at det er en del gyldige DT funnet fra denne fasen, der vi tar utgangspunkt i en kant enten fra fase to eller tre. I denne tabellen leter fase to blant de fem nærmeste nabopunkter. Den andre fasen er ikke laget for å finne trekanter, men kun enkelte kanter. Det at vi finner slike trekanter, er en sideeffekt av dette. Allikevel ser vi et mønster i forhold til antall trekanter holder seg forholdsvis stabilt i underkant av 40 % nesten uavhengig av antall punkter.

Antall punkter	Antall Delaunay trekanter		% av antall gyldige DT funnet i fase to i forhold til det totale antall gyldige trekanter
	Etter fase 2 (leter blant de 10 nærmeste nabopunktene)	Totalt i den komplette trianguleringen	
10	7	12	63,48 %
50	45	88	51,02 %
100	96	186	51,56 %
500	528	982	53,74 %
1K	1 064	1 981	53,72 %
5K	5 474	9 977	54,87 %
10K	10 925	19 974	54,697 %
50K	55 207	99 970	55,224 %
100K	110 637	199 967	55,327 %
500K	554 779	999 962	55,480 %
1M	1 109 780	1 999 960	55,490 %

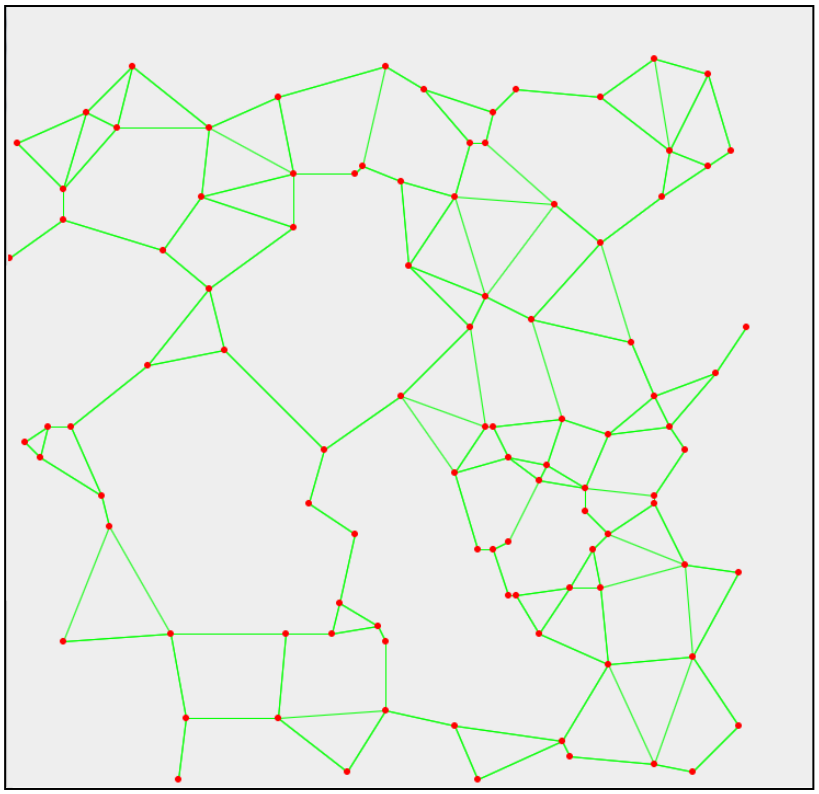
Tabell 20: *Viser hvor mange DT i Delaunay trianguleringen fase to finner i forhold til det totale antallet etter fase tre. Disse trekantene kan bygge fra en kant funnet i fase tre. Fase to legger opp til at vi leter etter 10 gyldige Delaunay nabopunkter per punkt.*

Om vi leter Delaunay kanter blant de ti nærmeste nabopunktene finner vi omtrent 55 % av alle DT for antall punkter over 5000. Når vi øker fra fem til ti nærmeste naboer, finner vi rundt 15 % mer trekanter fra fase to.

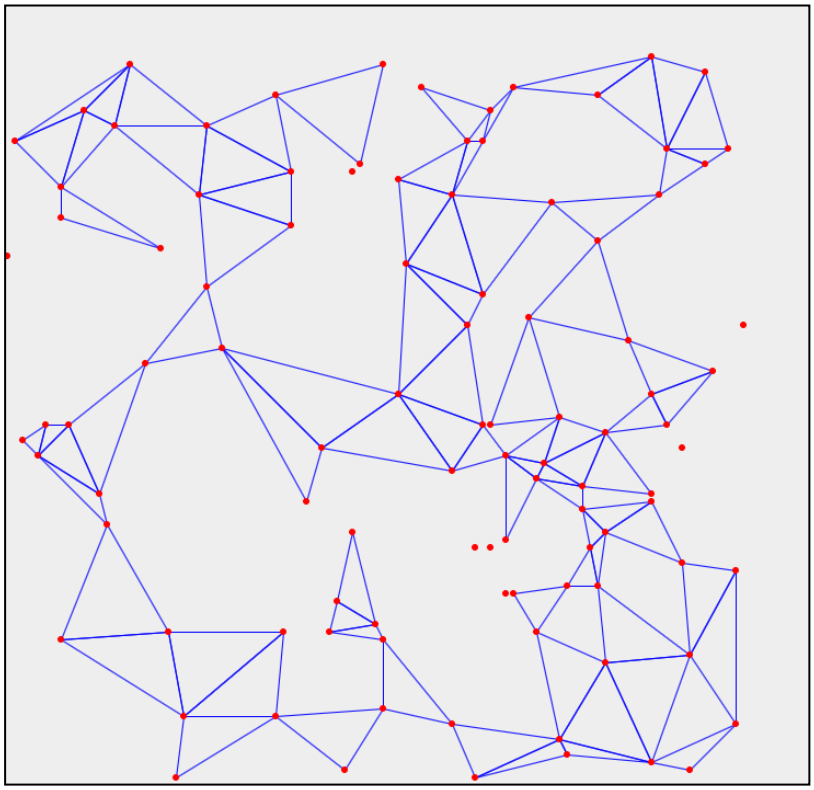
13.4 Figureksempler

13.4.1 Eksempel på en fase to kanter, trekantene og fullført Delaunay triangulering

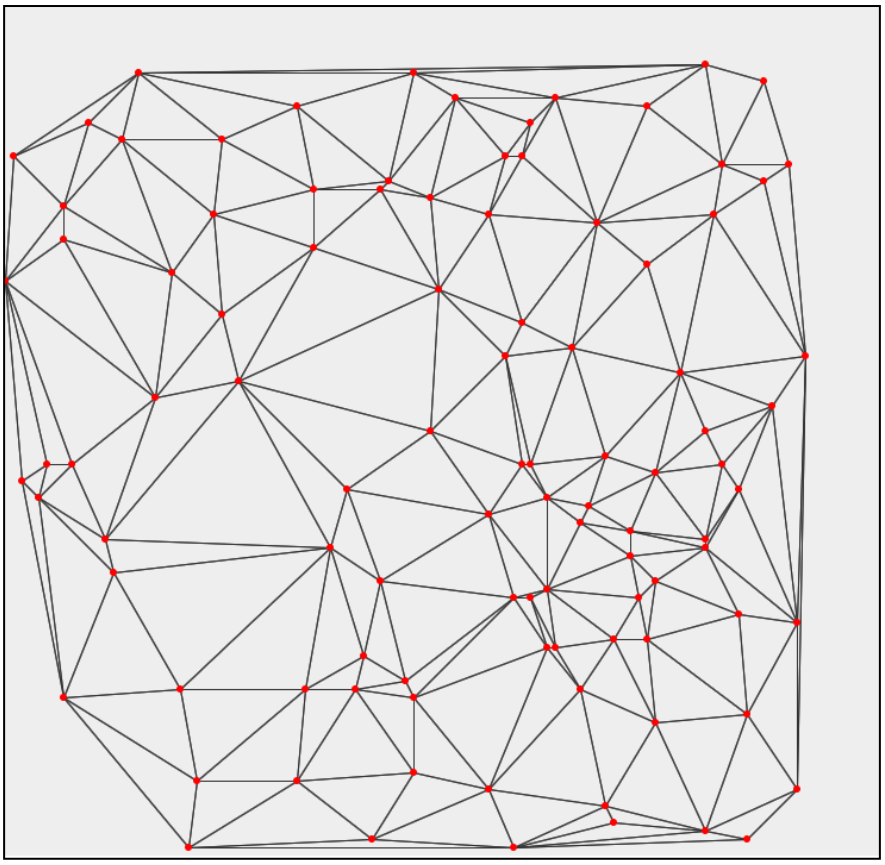
For å vise hvordan statistikken over Delaunay kanter og DT virkelig ser ut, vil de neste figurer vise tre bilder av en løst Delaunay triangulering med 100 punkter. Figur 34 viser alle punktenes gyldige Delaunay naboer etter at fase to har lett blant de 10 nærmeste naboene sine. Til sammen er det funnet 160 Delaunay kanter i denne fasen, mens det er totalt funnet 284 i den komplette trianguleringen. I Figur 35 ser vi hvilke DT vi kunne finne gitt informasjonen fra fase to, nemlig kantene fra Figur 34, samt noen kanter fra fase tre. Det vil si at man leter etter trekanter i fase tre, basert på en kant, som enten er funnet i fase to eller tre, og man sjekker om resten av trekanten finnes med kanter kun fra fase to. Vi finner her 79 DT, mens det totale antallet er 185. Figur 36 er den fullførte Delaunay trianguleringen. Ut i fra figurene ser vi at på mindre steder med en konsentrert mengde punkter, vil vi naturligvis finne flere DT basert på Delaunay kantene fra fase to.



Figur 34: *Bildet viser Delaunay kantene, som er funnet fra fase to, altså Delaunay kantene blant de k nærmeste naboene. K er i dette tilfellet 10. Det er 100 punkter totalt.*



Figur 35: *DT vi ser her er basert på ideen fra 11.9.3. Det vil si at hver DT er funnet i fase tre, og bygger på at vi har en kant, funnet enten fra fase to eller tre, og sjekker om punktene, som utgjør denne kanten, deler et punkt funnet fra fase to. Hver av disse DT har minimum to av sine kanter funnet fra de k nærmeste naboene. Det er de samme punktene, som Figur 34. K er 10, og det er 100 punkter totalt.*



Figur 36: *Den fullførte Delaunay trianguleringen basert på de samme punktene som er i Figur 34 og Figur 35. Det er 100 punkter totalt.*

13.5 Kjøretidstester

13.5.1 Å lete etter gyldige DT fra fase to

Til tross for at vi har funnet en del DT fra fase to, er det ikke sikkert at det lønner seg å lete etter disse. For å finne ut dette sammenligner jeg kjøretidene mellom to Delaunay trianguleringsprogram, begge med en k lik ti i fase to:

- Et program hvor jeg ignorerer DT, basert på kanter fra fase to og en kant fra fase to eller tre, slik at vi leter etter alle DT i fase tre.
- Et annet program der jeg oppretter nødvendig datastruktur og søk, slik at vi ikke trenger å lete etter en DT, som vi allerede har funnet. Dette gjør at fase tre vil begynne på neste DT med en gang.

Programmene vil bli kjørt i en løkke på 11, og det blir tatt en median av disse kjøretidene. Forbedringen av kjøretidene er slik:

$$Forbedring = \frac{kjøretid\ til\ sekvensiell\ Delaunay\ triangulering\ uten\ DT\ fra\ fase\ to}{kjøretid\ til\ sekvensiell\ Delaunay\ triangulering\ med\ DT\ fra\ fase\ to}$$

Antall punkter	Kjøretider(ms) for		Forbedring
	Sekvensiell Delaunay triangulering u/ DT fra fase 2	Sekvensiell Delaunay triangulering m/ DT fra fase 2	
10	2	2	1,23
50	6	5	1,10
100	7	8	0,85
500	17	16	1,09
1K	30	27	1,11
5K	129	118	1,09
10K	251	215	1,16
50K	1 376	1 136	1,21
100K	2 640	2 280	1,16
500K	15 417	14 084	1,09
1M	30 578	30 515	1,00

Tabell 21: *Her ser vi kjøretider for to sekvensielle versjoner av Delaunay trianguleringsprogrammet. Den første (til venstre i tabellen) viser versjonen, som ignorerer trekantene funnet fra fase to, mens det andre(midterst kolonnen i tabellen) er for versjonen som har datastrukturen som trengs for å finne disse trekantene. Dataene baserer seg på en Intel core I7 870 prosessor med en klokkefrekvens på 2,93 GHz, med 8 kjerner (4 fysiske med hyperthreading).*

I Tabell 21 ser vi at det lønner seg å gå videre med disse DT, som baserer seg på kantene funnet i fase to. Til tross for at vi må opprette en egen datastruktur og egne sjekk for hvert punkt for å finne disse DT ser vi at vi får bedre kjøretider om vi inkluderer dette i programmet. Ut i fra statistikken vi fikk fra Tabell 20 ser vi at det er ganske mange DT vi kan finne slik at dette oppveier for det ekstra vi må legge til.

13.6 Konklusjon

13.6.1 Resultatene fra statistikken angående fase to

Delaunay triangulering var et godt eksempel på et faseoppdelt program, da vi oppnådde gode resultater for kjøretidene til den parallelle versjonen laget av Java PRP. Det viste seg også at vi kunne arbeide videre på å forbedre den sekvensielle Delaunay triangulering, spesielt i den andre fasen.

Noe vi ikke har tatt med er hvordan programmet hadde fungert uten fase to i det hele tatt. Dette kommer av at den mer tradisjonelle trianguleringen i fase tre er avhengig av at alle punkter kjenner til minst én gyldig Delaunay nabo. Derimot viser statistikken mye om hvordan vi kan bruke fase to bedre, samtidig som den viser oss hvor effektiv den er i forhold til hvor mye av trianguleringen den finner.

Vi kom fram til at å velge 10 nærmeste naboer å sjekke for Delaunay kanter i denne fasen. Dette kom av at vi naturligvis fant flere Delaunay kanter og DT enn mindre verdier, og fordi at det begynte å gå ta lengre tid å sjekke flere av de nærmeste naboer. Dersom vi skulle sjekket flere av de nærmeste naboene måtte vi utvidet søket etter disse naboene oftere, som gjør at kjøretidene ble dårligere.

Det var også interessant å se hvor mange Delaunay kanter vi fant i fase to, og spesielt hvordan vi kunne bygge disse kantene til å finne gyldige DT, slik det ble beskrevet i 11.9.3. Basert på dette kunne vi finne rundt 55 % av det totale antallet DT. Kjøretidene vi fikk fra Tabell 21 viste oss at vi fikk en liten speedup ved å lete etter disse DT, enn om vi hadde brukt standard søk etter DT i alle tilfeller.

14. Konklusjon og videre arbeid

14.1 Java PRP

14.1.1 Generere parallelle programmer

Målet med Java PRP var å skape et program, som tok et sekvensielt program og genererte en parallell versjon. Fokuset var at den parallelle versjonen skulle være raskere enn den sekvensielle, altså at kjøretidene skulle være bedre. For å komme frem til dette ble det fokusert på delte variabler. For å finne ut om Java PRP klarte å generere vellykkede parallelle programmer, ble det gjort tester med tre forskjellige programmer. Alle disse testene er basert på en Intel core I7 870 prosessor med en klokkefrekvens på 2,93 GHz, med 8 kjerner, 4 fysiske med hyperthreading.

Det første eksempelet besto av at vi parallelliserte et program, som fant det største tall blant en mengde tall. Dette representerte noen av de enkleste programmene, som Java PRP kan håndtere. Den har kun en enkelt rekursjonsmetode med to i fanout, og er ikke et stort program. Det viste seg at Java PRP klarte ganske bra å parallellisere dette programmet, da den parallelle versjonen var over tre ganger raskere enn den sekvensielle når vi nådde større mengder opptil 50 millioner og oppover.

Det andre eksempelet var Quicksort. Dette økte i kompleksitet fra forrige eksempel, men representerte den samme typen program, nemlig de med kun én rekursjonsmetode. I dette eksempelet fikk vi også bedre kjøretider med det parallelle programmet, generert av Java PRP. For store mengder tall, mellom 5 til 100 millioner, fikk vi over to ganger så raskere kjøringer med den parallelle versjonen sammenlignet med den sekvensielle. For dette resultatet inkluderte vi lastbalanse, slik at vi hadde flere tråder enn kun antall prosessorer tilgjengelig, for å få enda bedre kjøretider. For den parallelle versjonen, generert av Java PRP, nådde vi opp mot 1,9 ganger speedup for 100 millioner tall å sortere. Med lastbalanse klarte vi å oppnå en speedup på 2,4.

I det siste og store eksempelet skulle Java PRP parallellisere et program, som tok for seg Delaunay triangulering. Dette var et mye større program enn de to tidligere, og representerte en gruppe mer avanserte programmer, som Java PRP skal kunne parallellisere altså programmer med flere faser. Til tross for dette klarte vi å oppnå ganske gode kjøretider med den parallelle versjonen. For 500 000 punkter å

triangulere klarte den parallelle versjonen å kjøre 3,3 ganger raskere enn den sekvensielle versjonen.

Basert på eksemplene, kan vi se at parallelliseringen har vært vellykket i forhold til kjøretider. Alle har fått en speedup med den parallelle versjonen i forhold til den sekvensielle. Dette kan komme av at disse programmene er laget slik at de er gode å parallellisere, men også at Java PRP benytter seg av gode taktikker for å parallellisere dem, med fokus på delte variabler.

14.1.2 Takle større og faseoppdelte programmer

Java PRP har vært laget av tidligere masterstudenter, med forskjellige variasjoner. Denne versjonen av Java PRP kan takle større programmer i form av faseoppdelte programmer. Det siste eksempelet, nemlig Delaunay triangulering, representerte en form for parallellisering av faseoppdelte programmer.

Hver fase ligner veldig på de to første, enklere eksemplene. En slik overordnet fase består av en parallell og en sekvensiell fase. I en parallell fase blir rekursjonsmetoden utført av de forskjellige trådene. Når disse er ferdig kan man endre data sekvensielt om man ønsker dette. Dette var for å forhindre bruken av synkroniserte metoder mens man kjører de forskjellige trådene, men at man heller gjorde dette etterpå.

Det som gjorde denne faseoppdeling mulig var ideen om en administratormetode. Denne metoden bestemte den sekvensielle gjennomgangen av programmet, også i den parallelle versjonen. Den satte i gang de forskjellige fasene til riktig tid. Dette hadde flere fordeler ved seg, som for eksempel at brukeren og Java PRP hadde mer kontroll av gjennomgangen av programmet, da det kunne potensielt blitt ganske vanskelig, dersom disse fasene kunne startet fra hvor som helst.

Java PRP vil derimot ikke enkelt kunne dekke andre type metoder enn disse tradisjonelle rekursjonsmetoder. En tidligere versjon av Java PRP[19] har parallellisert metoder, der løkker skaper progresjon i metoden, men min versjon av Java PRP er skapt fra bunnen av, og har gått i en annen retning. Dette fører til at den ikke inneholder det samme, som de tidligere versjonene. De er derimot bygget på mange av de samme prinsippene.

14.1.3 Delte variabler

I eksemplene har det vært stor vekt på å unngå synkroniserte metoder. Dette kommer av at mye bruk av slike metoder legger til mye ekstra kjøretid. Rekursjonsmetodene, i eksemplene, har ikke behøvd slike synkroniseringsmetoder. Dette har bidratt til at vi har fått bedre kjøretider for de parallelle versjonene generert av Java PRP.

14.2 Delaunay Triangulering

14.2.1 Parallell versjon

Delaunay triangulering er et godt eksempel på et faseoppdelt program. Dette kommer av at vi klarte tydelig å etablere flere trinn for å komme frem til resultatet og det ble laget rekursjonsmetoder, som tilfredstilte Java PRP sine krav, til hvert av disse trinnene. Trinnene i programmet er som følgende:

- Det første vi ville finne var den konvekse innhyllingen av punktsettet vi fikk.
- Deretter prøvde vi å finne Delaunay kanter blant de k nærmeste nabopunktene til hvert punkt.
- Avslutningsvis gjorde vi standard Delaunay søk etter DT slik at vi hadde en komplett triangulering.

Med disse fasene, som grunnlag, kunne vi bygge opp et Delaunay trianguleringsprogram, som fulgte mange av grunnprinsippene bak Java PRP. Samtidig måtte vi utvide Java PRP slik at den kunne takle flere faser. Resultatene fortalte oss at parallellisering på denne måten var fordelaktig da vi oppnådde en ganske god speedup med opp til 3,3 for dette eksempelet.

14.2.2 En alternativ måte å finne Delaunay kanter og trekanter på

Dette eksempelet utvidet seg fra å se bare på en parallell versjon, generert av Java PRP, til å se om det var mulig å optimalisere den sekvensielle versjonen også. Dette kom i form av å se nærmere på implementasjonen av å finne gyldige Delaunay kanter blant de k nærmeste nabopunktene til hvert punkt, og om vi kunne finne noen faktiske DT med disse kantene, som grunnlag. Dette bygget seg videre til å føre statistikk over Delaunay kanter og DT funnet, og om det lønte seg å søke etter disse DT i forhold til kjøretider.

Det vi fant ut var at det var en del informasjon å hente fra denne andre fasen. For k lik 10 kunne vi få opptil 64 % av alle Delaunay kantene i trianguleringen fra denne fasen. Vi fant også omtrentlig 55 % DT av det totale antallet basert mye på Delaunay kantene funnet i fase to, pluss noen ekstra kanter fra den tredje fasen. Basert på testene, der vi sammenlignet leting etter disse DT mot ignoreringen av disse trekantene, fant vi en liten, men konsekvent speedup med å lete etter dem.

14.3 Videre arbeid

14.3.1 Forslag til forbedringer og utvidelser

Denne versjonen av Java PRP bygget på mange ideer til tidligere versjoner, men allikevel er det skapt fra grunn av. Det er mange muligheter til å utvide og forbedre Java PRP, eventuelt gå i en annen retning enn hva jeg har gjort i denne oppgaven.

14.3.2 Et nettverk av multikjernede maskiner

Min versjon av Java PRP har tatt for seg parallellisering på én multikjernet maskin. Dette gjør at parallelliseringen er begrenset til antall kjerner en maskinen har. Et annet forslag er at man kombinerte slik parallell programmering på flere multikjernede maskiner, altså distribuert i et nettverk. Dette kan gi bedre kjøretider for beregningstunge oppgaver da vi kan dele opp i enda flere kjerner. For å gjøre dette nyttig, må man ta for seg både problematikken i parallellitet i én maskin og parallellitet på flere maskiner. Parallellitet på flere maskiner vil inkludere former for nettverksbehandling.

14.3.3 La Java PRP håndtere delte variabler direkte

I denne versjonen av Java PRP vil ikke delte variabler bli håndtert direkte. Den delen av Java PRP, som generer faseoppdelte, parallelle programmer, tar for seg delte variabler på en slik måte at den deler hver fase inn en parallell del og en sekvensiell del. Et forslag til å gjøre dette på en annen måte er at brukeren kunne lagt en kommentar over slike variabler, og dermed latt Java PRP tatt seg av disse. På denne måten kunne man tatt for seg programmer, som er mer avhengig av å bruke delte variabler, i selve rekursjonsmetoden.

14.3.4 Inkludere andre former for metoder i faseoppdelingen

Hver fase må være programmert i en vanlig rekursjonsmetode for at Java PRP skal kunne parallellisere programmet. Det kunne vært mulig å utvide dette slik at man kunne laget flere typer for metoder. For eksempel, kunne man skape progresjon i metoder via løkker istedenfor rekursjon. Dette er gjort i tidligere Java PRP versjoner, men ikke som en del av løsningen til faseoppdelte programmer. Dette vil være en utvidelse av Java PRP fra denne masteroppgaven.

14.3.5 La Java PRP dynamisk bestemme antall tråder

I Quicksort eksempelet vårt så vi at ved å øke antall tråder fra *n*, der n er antall kjerner tilgjengelig, til n ganger ti fikk vi bedre speedup grunnet lastbalansen dette ga oss. Et forslag til videre arbeid vil være at Java PRP ikke setter antall tråder statisk, men variere basert på størrelsen av mengdene vi arbeider på, og hvor stort rekursjonstreet er. Dette betyr at antall tråder er annerledes for hvert program. Det vil også variere innen et program, basert på mengdene i hver kjøring.

1. Vedlegg A: Java PRP brukermanual

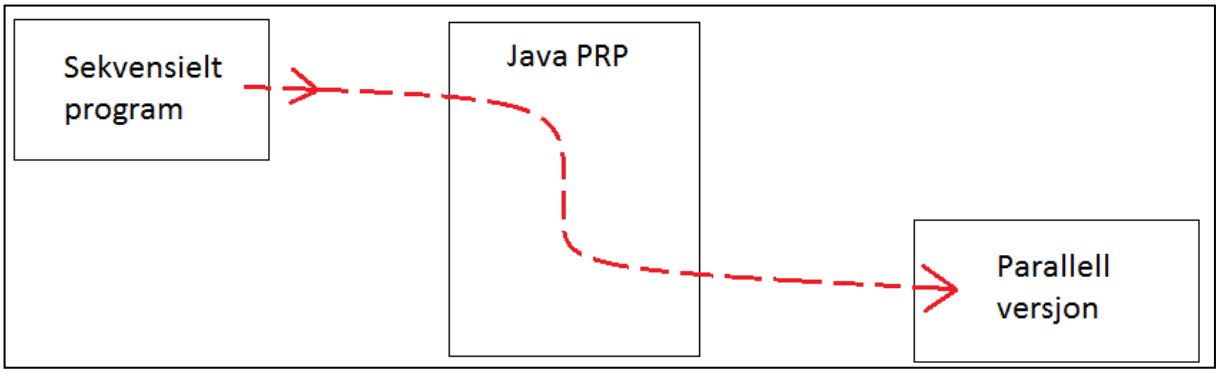
1.1 Introduksjon

1.1.1 Hva er Java PRP

Java PRP (*Parallel Recursive Procedure*) gir oss muligheten til automatisk parallellisering av programmer, som baserer seg en rekursiv løsning av et problem. Å programmere i et parallelt perspektiv kan ofte oppleves som vanskelig og tidkrevende, mye på grunn av at ingen gjennomkjøring er lik den neste. Dette gjør at feilsjekking av programmer tar lengre tid. Derimot kan parallellisering gi oss en fordel i forhold til kjøretid, da vi kan utnytte en flerkjernet datamaskinsarkitektur. Derfor kan Java PRP automatisk ta for seg parallelliseringsdelen, slik at programmereren kan kode med tanke på et sekvensielt perspektiv. Dette kan gjøre at fokuset kan ligge i problemstillingen til programmereren og ikke nødvendigvis på parallelliseringsdelen, samtidig som vi kan muligens oppnå ønskelig forbedring i kjøretid med parallellitet.

1.1.2 Hvordan Java PRP fungerer

Måten Java PRP arbeider på er at den tar imot det sekvensielle programmet og bruker den som en tekstlig fil for å generere en parallell versjon. For brukeren vil den parallelle versjonen være lik den sekvensielle når det gjelder funksjonalitet.



Figur 37: *Fra sekvensielt til parallelt via Java PRP*

1.2 Krav til den originale koden

1.2.1 Nøkkelord

For at Java PRP skal kunne gjøre et program parallelt, må den gjenkjenne noen viktige punkter i programmet. Dette kan den gjøre ved at vi setter noen små kommentarer foran disse punktene, som Java PRP kan kjenne igjen og kan basere sitt arbeid på. Det er to kommentarer som er gjenkjennelig, nemlig:

- /*FUNC*/
  Denne viser hvor rekursjonsmetoden befinner seg, slik at metoden kommer under denne kommentaren.
- /*REC*/
  Her vil vi finne rekursjonskallene innenfor metoden som skal parallelliseres. Kommentaren vil befinne seg over selve kallene, og kan være flere ganger i en metode.

Et eksempel på hvordan en sekvensiell kode, som bruker disse nøkkelordene, er:

```
class SekvensiellKode{
    public static void main(String[] args){
            ....
            ....
            ....
            Eksempel e = new Eksempel();
            <returverdi> svar = e.rekursivMetode(...);
        }
    }
}

class Eksempel{

    /*FUNC*/
    <returverdi> rekursivMetode(...){
        ...
        ...
        /*REC*/
        <returverdi> svar1 = rekursivMetode(...);
        /*REC*/
        <returverdi> svar2 = rekursivMetode(...);
        /*REC*/
        <returverdi> svar3 = rekursivMetode(...);
        ...
        return ...;
    }
}
```

1.2.2 Uavhengige rekursive kall

De forskjellige kallene, som eksisterer inne i den rekursive metoden, kan ikke avhenge av hverandre, og kallene må være etter hverandre uten at det er annen kode i mellom. Dette kommer av at Java PRP skal parallellisere basert på disse kallene, slik at hvis de avhenger av hverandre kan det oppstå situasjoner hvor det ene kallet må være ferdig før de andre. Dette vil føre til at vi får en mer og mer sekvensiell kode, da vi må

uansett la de forskjellige kallene kjøre etter hverandre. Kall på rekursjonsmetoden må også tilordnes til en verdi.

1.2.3 Den originale koden må fungere!

Ettersom Java PRP vil basere sin genererte kode på det originale programmet, vil den ta med seg de samme feilene som finnes der. Java PRP retter ikke opp syntaks eller semantiske feil, den bare parallelliserer. Dermed må den originale koden kunne kompilere og kjøre som et sekvensielt program.

1.2.4 Fanout

Fanout vil si hvor mye vi deler opp rekursjonen vår i, altså hvor mange rekursive kall vi har i rekursjonsmetoden vår. I eksempelet med nøkkelordene er fanout tre, ettersom vi har tre rekursive kall inni rekursjonsmetoden. I Java PRP vil antall parallelliserende elementer hovedsakelig avhenge av hvor stor fanout vi har. Dermed gir det ikke mening om vi kun har ett kall på oss selv. Derfor vil vi at rekursjonsmetoden deler seg opp i to eller flere deler.

1.2.5 Fullføre blokker

I programmering finner vi oss ofte i situasjonen hvor vi har en blokk som kun inneholder én setning. I språk som Java er det tillatt å fjerne krøllparantesene for sånne situasjoner. Derimot bruker Java PRP slike blokker til å generere den parallelle koden, slik at for at den skal kunne fungere optimalt må den originale koden ha fullført alle sine blokker. Java PRP bruker mye av krøllparantesene for å isolere ut eventuelle metoder og klasser. Dette gjør at brukeren av Java PRP må fullføre blokker uansett, for av parallelliseringen skal fungere korrekt.

1.2.6 Første rekursjonskall fra *main*

Det første kallet til rekursjonsmetoden, som skal parallelliseres må komme fra mainmetoden.

1.2.7 Rekursjonsklassen blir endret

Ettersom rekursjonsmetoden, og dens klasse, blir forvandlet fullstendig vil det ikke være mulig å kalle på metoder og variabler, utenfra klassen selv. Derimot kan metoder inne i klassen kalle på andre metoder i denne klassen og i andre klasser, som ikke endres av Java PRP.

1.3 Parallellisere faseoppdelte programmer

1.3.1 Et faseoppdelt program

Java PRP kan parallellisere flere rekursive metoder i et program. Slike program blir delt opp i faser, slik at vi kan betegne programmet som et faseoppdelt program. Hver slik overordnet fase er todelt, for vi har en parallell fase og en synkroniseringsfase tilknyttet rekursjonsmetode.

Den parallelle fase er der selve rekursjonsmetoden blir utført. Det vil her bli satt tråder til å gå igjennom metoden, slik at vi får den parallelle eksekveringen vi ønsker. Disse

fasene er veldig lik det vi har gjort tidligere i Java PRP, der vi har parallellisert kun én rekursiv metode per program.

For at vi skal unngå å måtte synkronisere i selve metoden, legger vi opp til at dette kan gjøres etterpå. Dette gjør at vi unngår å måtte stoppe opp midt i den parallelle eksekveringen. Til tross for navnet, er ikke dette en *synchronized* metode. Denne metoden blir kjørt sekvensielt av maintråden og dermed ikke i parallellitet. Dette er derimot en metode som vi kan bruke til, for eksempel, å spre lokal informasjon, funnet i rekursjonen, til global informasjon.

1.3.2 Administratormetode

Det grunnleggende bak faseoppdelingen ligger i en metode, som fungerer som en administrator for programmet. Denne metoden har som oppgave å starte opp de parallelle fasene og synkroniseringsfasene. Metoden vil avslutningsvis returnere resultatet til hele programmet, som fasene arbeider seg frem til.

Denne metoden blir skrevet av brukeren av Java PRP, og er et krav for at Java PRP skal kunne generere et faseoppdelt og parallelt program. Det følger visse krav til hvordan man kan lage en slik metode. Det første er at metoden må ha en returverdi, som vil være det faktiske resultatet av faseoppdelingen. Den andre går ut på kallene til de forskjellige fasene i programmet. Java PRP prosesserer disse kallene ved å se på to linjer om gangen i denne administratormetoden. Den første vil være kallet på rekursjonsmetoden, og neste vil være kallet på metoden, som starter den sekvensielle koden. Dette gjøres for alle fasene til vi returnerer svaret. Dette gjør at metoden må følge en ganske fast struktur med to og to linjer per fase.

Administratormetoden gir et løfte tilbake til brukeren. Den lover nemlig at denne metoden vil kjøres sekvensielt. Den vil starte opp parallelle faser, men vil ikke fortsette videre før disse fasene er over. Dette gjør at brukeren har større kontroll over programflyten, samtidig som vi kan utnytte effektiviteten vi kan få fra en parallell eksekvering.

Et eksempel på en slik adminstratormetode:

```
/*ADMIN*/
public int minAdminMetode(...){
    int svar = rekursivMetode1(...);
    sekvensiellKode1();
    svar = rekursivMetode2(...);
    sekvensiellKode2(...);
    svar = rekursivMetode3(...);
    sekvensiellKode3(...);
    svar = rekursivMetode4(...);
    sekvensiellKode4(...);
    return svar;
}
```

Eksempelet ovenfor gir en administratormetode, som tar for seg fire faser, der alle fasene inneholder en parallell fase og en synkroniseringsfase.

Av og til hender det at vi ikke trenger en synkroniseringsfase. Vi kan utnytte ideen med at kallet på sekvensiell kode er ubehandlet av Java PRP, siden den kjøres

sekvensielt, og at administratormetoden leser to og to linjer om gangen, ved å la linjen, som representerer den sekvensielle koden, stå tom eventuelt med en kommentar. Ved å endre forrige eksempel, kan vi få en administratormetode, der vi ikke trenger en sekvensiell kode for andre og fjerde fase, til å se slik ut:

```
/*ADMIN*/
public int minAdminMetode(...){
    int svar = rekursivMetode1(...);
    sekvensiellKode1();
    svar = rekursivMetode2(...);
    //ingen sekvensiell kode nødvendig
    svar = rekursivMetode3(...);
    sekvensiellKode3(...);
    svar = rekursivMetode4(...);
    //ingen sekvensiell kode nødvendig
    return svar;
}
```

1.3.3 Nye nøkkelord

For at Java PRP skal kunne starte og generere et faseoppdelt program trenger vi å utvide settet med nøkkelord. De nye vil være:

- /*ADMIN*/
  Denne gjør at det er mulig for Java PRP til å finne hvor administratormetoden vår befinner seg. Den brukes også til å finne ut om dette er et faseoppdelt program eller ikke.

- /*FUNC x*/
  Tidligere har vi hatt nøkkelordet /*FUNC*/ for å finne den rekursive metoden, som brukeren ønsker å ha parallellisert. Dette må nå utvides på grunn av at vi nå har flere rekursive metoder. Derfor må dette nøkkelordet si hvilken fase den er del av. Det vil si at den første rekursive metoden, som skal parallelliseres av Java PRP, vil få nøkkelordet /*FUNC 1*/, og neste /*FUNC 2*/ også videre.

Bortsett fra at endringen av /*FUNC*/ vil de andre nøkkelordene fortsatt bestå slik de har gjort før.

1.3.4 Muligheten til å la rekursjonsmetoder starte fra flere hold

Av og til har vi oppgaver der vi ønsker å starte rekursjonen annerledes enn resten av rekursjonen. For eksempel, dersom vi ønsker å starte opp rekursjonen i flere deler, for å deretter fortsette rekursjonen i færre rekursjonskall. For faseoppdelte programmer kan Java PRP lage oppstartsmetoder for rekursjonsmetodene. Dette gjør at vi ikke trenger mer kompliserte rekursjonsmetoder for å oppnå dette, men heller abstrahere en annerledes oppstart fra resten av rekursjonen. En oppstartsmetode ligner på en veldig enkel versjon av rekursjonsmetode, der vi har en rekke kall på selve rekursjonsmetoden og returnerer det samme tilbake. Mye av kravene for rekursjonsmetodene vil også gjelde for disse oppstartsmetodene, som for eksempel at kallene på rekursjonsmetoden må ha en /*REC*/ kommentar over seg. For å få til dette erstatter man bare kallet på rekursjonsmetode til denne oppstartsmetoden i administratormetoden.

1.3.5 Eksempel på et faseoppdelt program

```
class MittFaseProgram{
    public static void main(String[] args){
        ...
        ...
        ...
        <returverdi> svar = new MittFaseProgram().minAdminMetode(...);
    }

    /*ADMIN*/
    <returverdi> minAdminMetode(...){
        <returverdi> svar1 = new Fase1().rekursivMetode(...);
        sekvensiellKode1(...);
        <returverdi> svar2 = new Fase2().rekursivMetode(...);
        sekvensiellKode2(...);
        return ...;
    }

    void sekvensiellKode1(...){
        ...
        ...
    }

    void sekvensiellKode2(...){
        ...
        ...
    }

}

class Fase1{

    /*FUNC 1*/
    <returverdi> rekursivMetode(...){
        ...
        ...
        /*REC*/
        <returverdi> svar1 = rekursivMetode(...);
        /*REC*/
        <returverdi> svar2 = rekursivMetode(...);
        return ...;
    }

}

class Fase2{

    /*FUNC 2*/
    <returverdi> rekursivMetode(...){
        ...
        ...
        /*REC*/
        <returverdi> svar1 = rekursivMetode(...);
        /*REC*/
        <returverdi> svar2 = rekursivMetode(...);
        /*REC*/
        <returverdi> svar3 = rekursivMetode(...);
        return ...;
    }
}
```

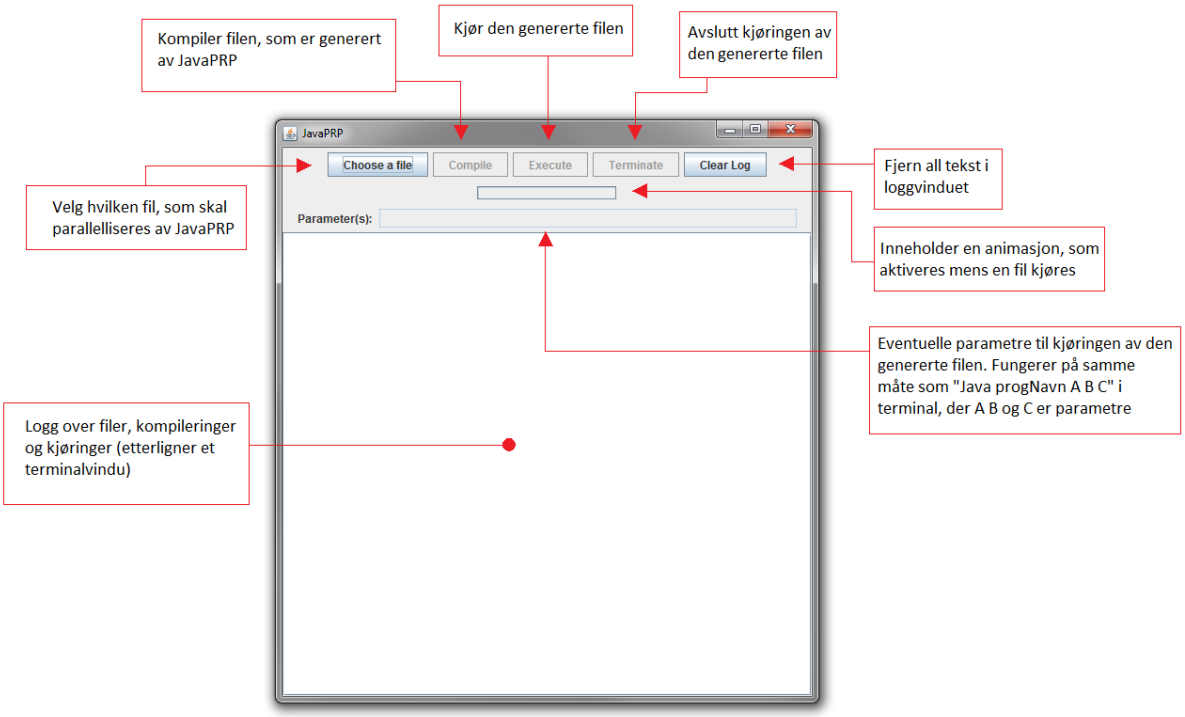
1.4 Bruke Java PRP

1.4.1 Lage sekvensiell kode

Det aller første vi ønsker er at brukeren faktisk lager en program som følger kravene som Java PRP stiller og legger til kommentarene på riktig sted. Dette vil skape basisen for at vi skal kunne bruke Java PRP. Man kan bruke eksempelet i 1.2.1, som en mal på hvordan dette kan gjøres.

1.4.2 Starte Java PRP

Etter at man har skrevet et sekvensielt program kan man starte programmet JavaPRP.java. Dette vil starte et grafisk grensesnitt, som man kan se i Figur 38.



Figur 38: *Dette bildet viser det grafiske grensesnittet til Java PRP programmet ved oppstart. Den inneholder det man trenger for å parallellisere programmer, som oppfyller kravene til Java PRP. Den gjør det også mulig å kompilere og kjøre de parallelle programmene direkte i grensesnittet.*

Dette grensesnittet kommer med flere knapper. Disse knappene fungerer slik:

- *Choose a file*
  Når man trykker denne knappen får man opp et nytt vindu der man kan velge en fil man ønsker å parallellisere. Etter man har valgt en fil, vil Java PRP sette i gang å parallellisere filen med en gang.
- *Compile*
  Denne kompilerer filen, som Java PRP genererte. Altså kompilerer den ikke den sekvensielle filen.
- *Execute*
  Kjører den genererte filen.
- *Terminate*
  Dersom man ønsker å avslutte kjøringen av programmet før den er ferdig, kan man trykke denne knappen.

- *Clear log*
  Fjerner all tekst i loggvinduet.

Som man kan se, inneholder dette grafiske grensesnittet mer enn bare knapper. Andre nevneverdige elementer er:

- Et loggvindu over hva Java PRP har gjort og resultater av kompileringer og kjøringer. Fungerer veldig likt som et terminalvindu.
- Et liten rektangel, som vil kjøre en liten animasjon når kjøringer pågår. Dette gjør at man kan se at programmet holder på med å kjøre et program, slik at man ikke tror at Java PRP stoppet å fungere.
- En tekstboks der man kan legge til eventuelle parametre til programkjøringene. Denne fungerer på samme måte som man ville gjort det i en terminal. Det vil si at dersom man skulle kjørt programmet "Prog.java" med parametrene A B C, ville man skrevet i terminalen "java Prog A B C". I Java PRP skriver man bare "A B C" i tekstboksen, og trykker på execute knappen.
- Grensesnittet vil skrive ut kjøretiden til det parallelle programmet, gitt i millisekunder, i loggen når programmet er ferdig.

Dersom man ikke ønsker å bruke det grensesnittet til å kompilere og kjøre programmer, kan man alltids bruke tradisjonelle måter via terminalvinduet til dette. For å parallellisere må man derimot bruke programmets grafiske grensesnitt.

1.4.3 Tidsmåling av programmet

Ettersom det parallelle programmet skal være raskere enn det sekvensielle, kan det være interessant å se hva kjøretiden er for det Java PRP genererte programmet. Dette er inkludert i Java PRP. Dersom man kjører programmet igjennom det grafiske grensesnittet vil man få en linje, i tekstvinduet, ved slutten av kjøringen. Java PRP måler hvor lang tid det tar å kjøre programmet i nanosekunder for presisjon, og fremviser det i millisekunder for lesbarhet. For et parallelt program, som kjører i 10,1 millisekunder, vil Java PRP skrive ut slik: "*Program execution time: 10.1 ms"*.

1.5 Et eksempel på bruk av Java PRP med én rekursjonsmetode

1.5.1 Lage en sekvensiell kode

Eksempelet vårt baserer seg på å finne det største tallet i et array. For at vi skal kunne utnytte Java PRP må den skrives på en rekursiv måte. Dette vil si at vi deler opp arrayet vårt i to deler per rekursjonskall. Når vi er på mindre mengder å lete igjennom, det vil si basistilfellet vårt, kan vi sjekke hvilket tall som er størst. Til slutt vil rekursjonen gi tilbake sine største verdier oppover i kallene, slik at det første metodekallet vil ende på den det største tallet i hele arrayet.

Programmet tar imot et argument, som er størrelsen på arrayet. Dette kan være nyttig i situasjoner hvor vi vil teste med små og store mengder tall. Før vi starter rekursjonen tar vi og fyller opp ett array bestående av tilfeldige tall mellom 0 og antall tall-1. Deretter starter vi rekursjonen og skriver ut resultatet.

Noe annet vi har med i programmet vårt, som ofte er en naturlig del av å programmere i Java, er import setninger. Dette er noe som Java PRP tar med i det nye programmet, slik at dette skal fungere riktig.

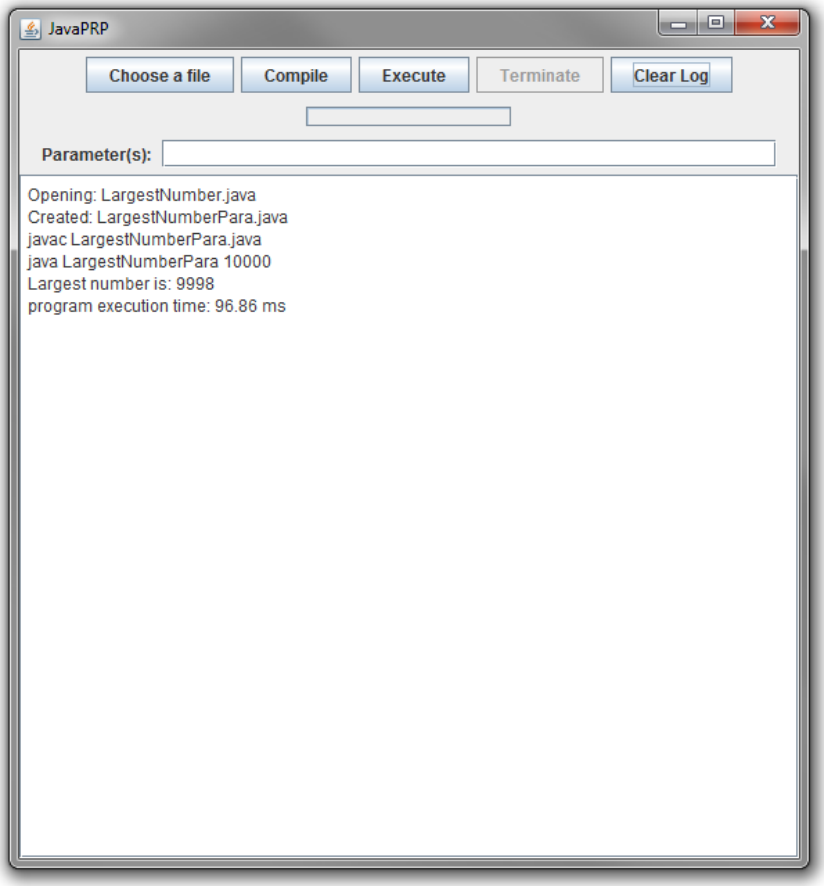
1.5.2 Det sekvensielle programmet

```
import java.util.Random;
import java.io.PrintWriter;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Arrays;

class LargestNumber{
    public static void main(String[] args){
        int len = Integer.parseInt(args[0]);
        int cores = Runtime.getRuntime().availableProcessors();
        int[] arr = new int[len];
        Random r = new Random();
        for(int i = 0; i < arr.length; i++){
            arr[i] = r.nextInt(len-1);
        }
        int k = (new Search()).findLargest(arr,0,arr.length);
        System.out.println("Largest number is " + k);

    }
}

class Search{
    int k = 5;

    /*FUNC*/
    int findLargest(int[] arr, int start, int end){

        if((end-start) < k){
            return largest_baseCase(arr,start,end);
        }
        int half = (end-start) / 2;
        int mid = start + half;
        /*REC*/
        int leftVal = findLargest(arr,start,mid);
        /*REC*/
        int rightVal = findLargest(arr,mid+1,end);
        if(leftVal > rightVal) return leftVal;
        return rightVal;

    }

    int largest_baseCase(int[] arr, int start, int end){
        int largest = 0;
        for(int i = start; i < end; i++){
            if(arr[i] > largest){
                largest = arr[i];
            }
        }
        return largest;
    }
}
```

Dersom vi går igjennom kravene til denne metoden:

- Legg merke til at vi har lagt til */*FUNC*/* over hele metoden og vi har lagt til */*REC*/* foran hvert av de rekursive kallene.
- Kallene kommer rett etter hverandre, og variabelen rightVal avhenger ikke av leftVal, da de skal jobbe med hver sin del av arrayet av tall.
- Vi har en fanout på 2, som betyr at vi deler opp rekursjonen i to for hvert kall som ikke ender i basistilfelle.
- Alle blokker er komplette.
- Programmet er kompilert og kjørt på forhånd, slik at vi vet at den fungerer sekvensielt.

1.5.3 Kjøre med Java PRP



Figur 39: *Dette viser hvordan Java PRP ser ut etter man har valgt en fil, kompilert og kjørt.*

Figur 39 viser hvordan Java PRP programmet ser ut etter vi har kjørt programmet vårt. For å komme til dette punktet ble det gjort følgende:

1. Valgte filen "LargestNumber.java" ved å trykke på *Choose a file*. Dette genererte den parallelle versjonen "LargestNumberPara.java".
2. Kompilerte dette parallelle programmet via knappen *Compile*.
3. Skrev "10000" i parametertekstboksen og kjørte dette programmet ved å trykke *Execute*.

Resultatet er at vi får kjørt det parallelle programmet og vi får ut 9998 som det største tallet. Vi ser også at Java PRP skriver ut kjøretiden, nemlig 96,86 millisekunder.

1.5.4 Den parallelle koden

I den parallelle versjonen vil vi ha 3 klasser.

- *Main klassen*
  Dette er klassen som inneholder navnet ”LargestNumberPara”. Dette er kun en initialiseringsklasse som setter i gang programmet. Samtidig gir det brukeren av Java PRP et velkjent navn til det parallelle programmet.
- *Admin*
  Denne klassen er administratoren for at parallelliseringen går riktig for seg.
- *Worker*
  Denne vil virke mest gjenkjennelig da den inneholder den rekursive metoden som parallelliseres. Metoden *mySolution(...)* vil inneholde den rekursive metoden.

2. Vedlegg B: kildekode

2.1 Sekvensiell "Finn største" annotert for Java PRP

```
import java.util.Random;
import java.io.PrintWriter;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Arrays;

class LargestNumber{
    public static void main(String[] args){
        for(int i = 0; i < 11; i++){
            int len = Integer.parseInt(args[0]);
            int cores = Runtime.getRuntime().availableProcessors();
            int[] arr = new int[len];
            Random r = new Random();
            for(int j = 0; j < arr.length; j++){
                arr[j] = r.nextInt(len-1);
            }
            long start = System.nanoTime();
            int k = (new Search()).findLargest(arr,0,arr.length);
            long timeTakenNS = System.nanoTime() - start;
            System.out.println(timeTakenNS);
        }
    }
}

class Search{
    int k = 5;

    /*FUNC*/
    int findLargest(int[] arr, int start, int end){

        if((end-start) < k){
            return largest_baseCase(arr,start,end);
        }
        int half = (end-start) / 2;
        int mid = start + half;
        /*REC*/
        int leftVal = findLargest(arr,start,mid);
        /*REC*/
        int rightVal = findLargest(arr,mid+1,end);
        if(leftVal > rightVal) return leftVal;
```

```
        return rightVal;

    }

    int largest_baseCase(int[] arr, int start, int end){
        int largest = 0;
        for(int i = start; i < end; i++){
            if(arr[i] > largest){
                largest = arr[i];
            }
        }
        return largest;
    }
}
```

2.2 Parallell "Finn største" av Java PRP

```
import java.util.Random;
import java.io.PrintWriter;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Arrays;
import java.util.*;
import java.util.Stack;
import java.util.NoSuchElementException;
import java.util.concurrent.*;

class LargestNumberPara{
    public static void main(String[] args){
        new Admin(args);
    }
}

class Admin{

    public Admin(String[] args){
        initiateParallel(args);
    }

    void initiateParallel(String[] args){
        for(int i = 0; i < 11; i++){
            int len = Integer.parseInt(args[0]);
            int cores = Runtime.getRuntime().availableProcessors();
            int[] arr = new int[len];
            Random r = new Random();
            for(int j = 0; j < arr.length; j++){
                arr[j] = r.nextInt(len-1);
            }
            long start = System.nanoTime();
            int k  = startThreads(arr,0,arr.length);
            long timeTakenNS = System.nanoTime() - start;
            System.out.println(timeTakenNS);
        }
    }

        int startThreads(int[] arr, int start, int end){

        int coreCount  = Runtime.getRuntime().availableProcessors();
        LinkedList<Worker> allWorkers = new LinkedList<Worker>();
        Stack<Worker> waitingWorkers = new Stack<Worker>();
```

```
        CyclicBarrier b = new CyclicBarrier(coreCount+1);

        Worker root = new Worker(arr,start,end,b);
        allWorkers.add(root);

        boolean sequential = false;
        try{
            while(allWorkers.size() < coreCount){
                Worker w = allWorkers.remove();
                allWorkers.addAll(w.makeChildren());
                waitingWorkers.push(w);
        }
        }catch(NoSuchElementException e){
            sequential = true;
        }

        if(sequential){
            return root.runMySolution();
        }

        ExecutorService es = Executors.newFixedThreadPool(coreCount);
        int threadProgress = 0;
        while(allWorkers.size() > 0 && threadProgress < coreCount){
            Worker w = allWorkers.remove();
            es.execute(w);
            threadProgress++;
        }

        while(allWorkers.size() > 0){
            Worker w = allWorkers.remove();
            w.runMySolution();
        }

        try{
            b.await();
            }catch(InterruptedException ex){
            } catch (BrokenBarrierException ex){
        }

        Worker current = waitingWorkers.pop();
        current.gatherResultInit();
            int answer = current.sol;
        while(!waitingWorkers.empty()){
            current = waitingWorkers.pop();
            current.gatherResultInit();
            answer = current.sol;
        }
        es.shutdown();
        return answer;
    }



}

class Worker implements Runnable{
    int[] arr;
    int start;
    int end;
    CyclicBarrier barrier;
    int sol;
```

```
    LinkedList<Worker> children = new LinkedList<Worker>();

    public Worker(int[] arr, int start, int end, CyclicBarrier barrier){
        this.arr = arr;
        this.start = start;
        this.end = end;
        this.barrier = barrier;
    }

    public void run(){
        sol = mySolution(arr,start,end);
        try{
            barrier.await();
        }catch(InterruptedException ex){
        }catch(BrokenBarrierException ex){
        }
    }

    public LinkedList<Worker> makeChildren(){

        if((end-start) < k){
            return children;
        }
        int half = (end-start) / 2;
        int mid = start + half;
        /*REC*/
        Worker w0 = new Worker(arr,start,mid,barrier);
        children.add(w0);
        /*REC*/
        Worker w1 = new Worker(arr,mid+1,end,barrier);
        children.add(w1);
        return children;
    }


    public void gatherResultInit(){
        sol = gatherResult();
    }

    int gatherResult(){

        if((end-start) < k){
            return largest_baseCase(arr,start,end);
        }
        int half = (end-start) / 2;
        int mid = start + half;
        /*REC*/
        int leftVal = children.get(0).sol;
        /*REC*/
        int rightVal = children.get(1).sol;
        if(leftVal > rightVal) return leftVal;
        return rightVal;

    }

    int runMySolution(){
        sol = mySolution(arr,start,end);
        return sol;
    }

    int mySolution(int[] arr, int start, int end){
```

```
        if((end-start) < k){
            return largest_baseCase(arr,start,end);
        }
        int half = (end-start) / 2;
        int mid = start + half;
        /*REC*/
        int leftVal = mySolution(arr,start,mid);
        /*REC*/
        int rightVal = mySolution(arr,mid+1,end);
        if(leftVal > rightVal) return leftVal;
        return rightVal;

    }
    int k = 5;



    int largest_baseCase(int[] arr, int start, int end){
        int largest = 0;
        for(int i = start; i < end; i++){
            if(arr[i] > largest){
                largest = arr[i];
            }
        }
        return largest;
    }
}
```

2.3 Sekvensiell Quicksort annotert for Java PRP

```
import java.util.Random;

class QuicksortProg{
    public static void main(String[] args){
        int len = Integer.parseInt(args[0]);
        for(int i = 0; i < 11; i++){
            int[] arr = new int[len];
            Random r = new Random();
            for(int j = 0; j < arr.length; j++){
                arr[j] = r.nextInt(len-1);
            }
            long start = System.nanoTime();
            int[] k = new QuicksortCalc().quicksort(arr,0,arr.length-1);
            long timeTakenNS = System.nanoTime() - start;
            System.out.println(timeTakenNS);
        }
    }
}

class QuicksortCalc{
    int INSERT_LIM = 48;

    /*FUNC*/
    int[] quicksort(int[] array, int left, int right){
        if (right-left < INSERT_LIM){
            return insertSort(array,left,right);
        }else{
            int pivotValue = array[(left + right) / 2];
            swap(array, (left + right) / 2, right);
```

```
            int index = left;

            for (int i = left; i < right; i++) {
                if (array[i] <= pivotValue) {
                    swap(array, i, index);
                    index++;
                }
            }
            swap(array, index, right);
            int index2 = index;
            while(index2 > left && array[index2] == pivotValue){
                index2--;
            }

            /*REC*/
            array = quicksort(array, left,  index2);
            /*REC*/
            array = quicksort(array, index + 1, right);
            return array;
        }
    }


    int[] insertSort(int a[],int left, int right){
        if (left < right){
            int i,k,t;

            for (k = left+1; k <= right; k++){
                t = a[k] ;
                i = k;

                while ( a[i-1]  > t ){
                    a[i] = a[i-1];
                    if (--i == left) break;
                }
                a[i] = t;
            }
        }
        return a;
    }


    void swap(int[] array, int left, int right){
        int temp = array[left];
        array[left] = array[right];
        array[right] = temp;
    }
}
```

2.4 Parallell Quicksort av Java PRP

```
import java.util.Random;
import java.util.*;
import java.util.Stack;
import java.util.NoSuchElementException;
import java.util.concurrent.*;

class QuicksortProgPara{
    public static void main(String[] args){
        new Admin(args);
```

```
    }
}

class Admin{

    public Admin(String[] args){
        initiateParallel(args);
    }

    void initiateParallel(String[] args){
        int len = Integer.parseInt(args[0]);
        for(int i = 0; i < 11; i++){
            int[] arr = new int[len];
            Random r = new Random();
            for(int j = 0; j < arr.length; j++){
                arr[j] = r.nextInt(len-1);
            }
            long start = System.nanoTime();
            int[] k  = startThreads(arr,0,arr.length-1);
            long timeTakenNS = System.nanoTime() - start;
            System.out.println(timeTakenNS);
        }
    }

    int[] startThreads(int[] array, int left, int right){

        int coreCount  = Runtime.getRuntime().availableProcessors()*10;
        LinkedList<Worker> allWorkers = new LinkedList<Worker>();
        Stack<Worker> waitingWorkers = new Stack<Worker>();
        CyclicBarrier b = new CyclicBarrier(coreCount+1);

        Worker root = new Worker(array,left,right,b);
        allWorkers.add(root);

        boolean sequential = false;
        try{
            while(allWorkers.size() < coreCount){
                Worker w = allWorkers.remove();
                allWorkers.addAll(w.makeChildren());
                waitingWorkers.push(w);
        }
        }catch(NoSuchElementException e){
            sequential = true;
        }

        if(sequential){
            System.out.println("hoy");
            return root.runMySolution();
        }

        ExecutorService es = Executors.newFixedThreadPool(coreCount);
        while(allWorkers.size() > 0){
            Worker w = allWorkers.remove();
            es.execute(w);
        }

        try{
            b.await();
            }catch(InterruptedException ex){
            } catch (BrokenBarrierException ex){
        }
```

```
        Worker current = waitingWorkers.pop();
        current.gatherResultInit();
        int[] answer = current.sol;
        while(!waitingWorkers.empty()){
            current = waitingWorkers.pop();
            current.gatherResultInit();
            answer = current.sol;
        }
        es.shutdown();
        return answer;
    }



}

class Worker implements Runnable{
    int[] array;
    int left;
    int right;
    CyclicBarrier barrier;
    int[] sol;
    LinkedList<Worker> children = new LinkedList<Worker>();

    public Worker(int[] array, int left, int right, CyclicBarrier barrier){
        this.array = array;
        this.left = left;
        this.right = right;
        this.barrier = barrier;
    }

    public void run(){
        sol = mySolution(array,left,right);
        try{
            barrier.await();
        }catch(InterruptedException ex){
        }catch(BrokenBarrierException ex){
        }
    }

    public LinkedList<Worker> makeChildren(){
        if (right-left < INSERT_LIM){
            return children;
        }else{
            int pivotValue = array[(left + right) / 2];
            swap(array, (left + right) / 2, right);
            int index = left;

            for (int i = left; i < right; i++) {
                if (array[i] <= pivotValue) {
                    swap(array, i, index);
                    index++;
                }
            }
            swap(array, index, right);
            int index2 = index;
            while(index2 > left && array[index2] == pivotValue){
                index2--;
            }
```

```
            /*REC*/
        Worker w0 = new Worker(array, left,  index2,barrier);
        children.add(w0);
            /*REC*/
        Worker w1 = new Worker(array, index + 1, right,barrier);
        children.add(w1);
        return children;
        }
    }


    public void gatherResultInit(){
        sol = gatherResult();
    }

    int[] gatherResult(){
        if (right-left < INSERT_LIM){
            return insertSort(array,left,right);
        }else{
            int pivotValue = array[(left + right) / 2];
            swap(array, (left + right) / 2, right);
            int index = left;

            for (int i = left; i < right; i++) {
                if (array[i] <= pivotValue) {
                    swap(array, i, index);
                    index++;
                }
            }
            swap(array, index, right);
            int index2 = index;
            while(index2 > left && array[index2] == pivotValue){
                index2--;
            }

            /*REC*/
            array = children.get(0).sol;
            /*REC*/
            array = children.get(1).sol;
            return array;
        }
    }

    int[] runMySolution(){
        sol = mySolution(array,left,right);
        return sol;
    }

    int[] mySolution(int[] array, int left, int right){
        if (right-left < INSERT_LIM){
            return insertSort(array,left,right);
        }else{
            int pivotValue = array[(left + right) / 2];
            swap(array, (left + right) / 2, right);
            int index = left;

            for (int i = left; i < right; i++) {
                if (array[i] <= pivotValue) {
                    swap(array, i, index);
                    index++;
                }
```

```
            }
            swap(array, index, right);
            int index2 = index;
            while(index2 > left && array[index2] == pivotValue){
                index2--;
            }

            /*REC*/
            array = mySolution(array, left,  index2);
            /*REC*/
            array = mySolution(array, index + 1, right);
            return array;
        }
    }
    int INSERT_LIM = 48;




    int[] insertSort(int a[],int left, int right){
        if (left < right){
            int i,k,t;

            for (k = left+1; k <= right; k++){
                t = a[k];
                i = k;

                while ( a[i-1]  > t ){
                    a[i] = a[i-1];
                    if (--i == left) break;
                }
                a[i] = t;
            }
        }
        return a;
    }


    void swap(int[] array, int left, int right){
        int temp = array[left];
        array[left] = array[right];
        array[right] = temp;
    }
}
```

3. Referanser

1. Anders Brunland, Knut Hegna, Ole Christian Lingjærde og Arne Maus. (2011). En blandet parallell og rekursiv Kvikksort *Rett på Java* (vol. 3, s. 364-368): Universitetsforlaget.

2. Andrews, Gregory R. (1999). Case Study: MPI *Foundations of Multithreaded, Parallel, and Distributed Programming* (s. 340-341). Addison-Wesley.

3. Andrews, Gregory R. (1999). The Essence of Concurrent Programming *Foundations of Multithreaded, Parallel, and Distributed Programming* (s. 2): Addison-Wesley.

4. Andrews, Gregory R. (1999). Speedup and Efficiency *Foundations of Multithreaded, Parallel, and Distributed Programming* (s. 528-530): Addison-Wesley.

5. Bryant, Randal E., & O'Hallaron, David R. (2010). A Historical Perspective *Computer systems : a programmer's perspective* (2. utgave, s. 190-193): Pearson.

6. Bryant, Randal E., & O'Hallaron, David R. (2010). Thread-Level Concurrency *Computer systems : a programmer's perspective* (2. utgave, s. 56-57): Pearson.

7. Busterud, Petter A. (2012). *Investigating Different Concurrency Mechanisms in Java.* (Master), Universitetet i Oslo.

8. Intel. (2004). Enhanced Intel® SpeedStep® Technology for the Intel® Pentium® M Processor. Hentet, Fra ftp://download.intel.com/design/network/papers/30117401.pdf

9. Krogdahl, Stein (Producer). (2012). Triangulation and convex hull. [Forelesning] Hentet fra http://www.uio.no/studier/emner/matnat/ifi/INF4130/h12/undervisningsmateriale/triangulering+convex-2012-engelsk.pdf

10. Louden, Kenneth C. (1997). Parameter passing mechanisms *Compiler construction Principles and Practice* (1. utgave, s. 381-386): Course Technology Cengage learning.

11. Louden, Kenneth C. (1997). Scope Rules and Block Structures *Compiler construction Principles and Practice* (1. utgave, s. 301-302): Course Technology Cengage learning.

12. Maus, Arne. (1984). Delaunay triangulation and the convex hull of n points in expected linear time. *BIT 24,* 151-163.

13. Maus, Arne (Producer). (2014). INF2440 – Effektiv parallellprogrammering Uke 2, våren2014 - tidtaking. [Forelesning] Hentet fra http://www.uio.no/studier/emner/matnat/ifi/INF2440/v14/powerpoint/uke2.pdf

14. Maus, Arne, & Drange, Jon Moen. (2010). *All closest neighbors are proper Delaunay edges generalized, and its application to parallel algorithms*. Oppgave presentert ved NIK, Høgskolen i Gjøvik.

15. McCabe, David S. Moore & George P. (2006). Measuring center: the mean *Introduction to the practice statistics* (5. utgave, s. 40-42): W.H. Freeman.

16. Oracle. (2013). Passing Information to a Method or a Constructor. Hentet 10.10, 2013, Fra http://docs.oracle.com/javase/tutorial/java/javaOO/arguments.html

17. Oracle. (2014). Processes and Threads. Hentet 19.04, 2014, Fra http://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html

18. Reda, Sherief (Producer). (2008). EN2912C: Future Directions in Computing. Lecture 03: Challenges for Silicon-Based Computing. Hentet fra http://scale.engin.brown.edu/classes/EN2912C/lecture03.pdf

19. Skaret, Kristoffer. (2008). *MPRP: Parallellisering På Multiprosessor.* (Master), Universitetet i Oslo.

20. Weiss, Mark Allen. (2010). storage classes *Data Structures & Problem Solving Using Java* (4. utgave, s. 56): Pearson.

21. Wikipedia, the free encyclopedia. (2013). Transistor count. Hentet 24/01, 2013, Fra http://en.wikipedia.org/wiki/Transistor_count