Java PRP brukermanual

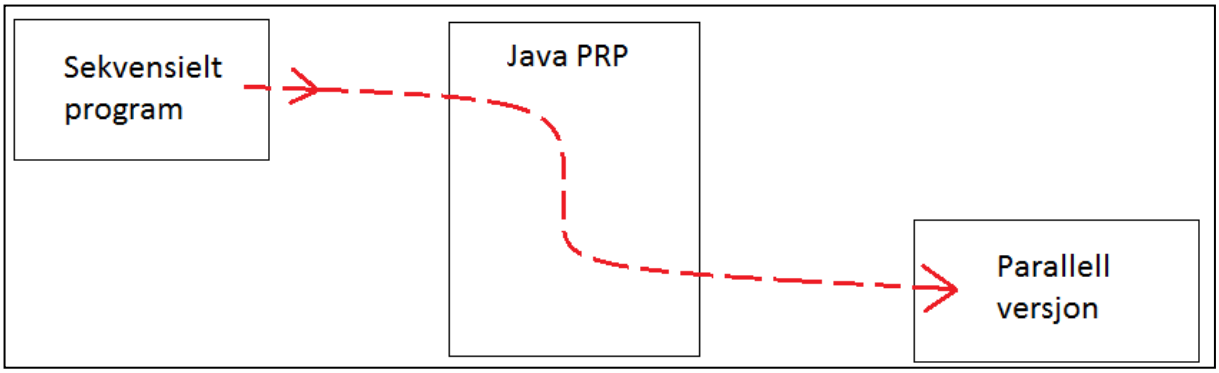
1.1 Introduksjon

1.1.1 Hva er Java PRP

Java PRP (Parallel Recursive Procedure) gir oss muligheten til automatisk parallellisering av programmer, som baserer seg på noen rekursive metoder. 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 1: *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.

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.

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 JavaPRP 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.

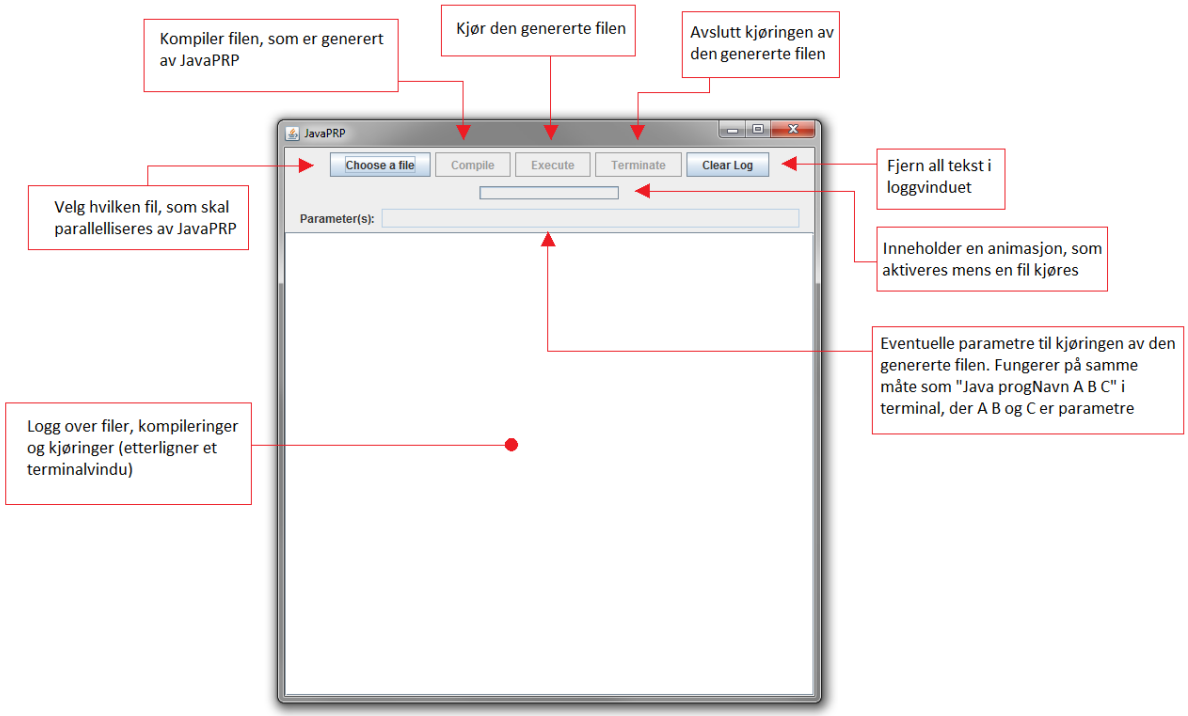
1.4 Bruke Java PRP

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 2.



Figur 2: *Dette bildet viser det grafiske grensesnittet til JavaPRP programmet ved oppstart. Den inneholder det man trenger for å parallellisere programmer, som oppfyller kravene til JavaPRP. 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 JavaPRP sette i gang å parallellisere filen med en gang.
- *Compile*
  Denne kompilerer filen, som JavaPRP 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 JavaPRP 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 JavaPRP 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 JavaPRP genererte programmet. Dette er inkludert i JavaPRP. Dersom man kjører programmet igjennom det grafiske grensesnittet vil man få en linje, i tekstvinduet, ved slutten av kjøringen. JavaPRP 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 JavaPRP skrive ut slik: "*Program execution time: 10.1 ms*".

1.5 Et eksempel på bruk av JavaPRP 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.

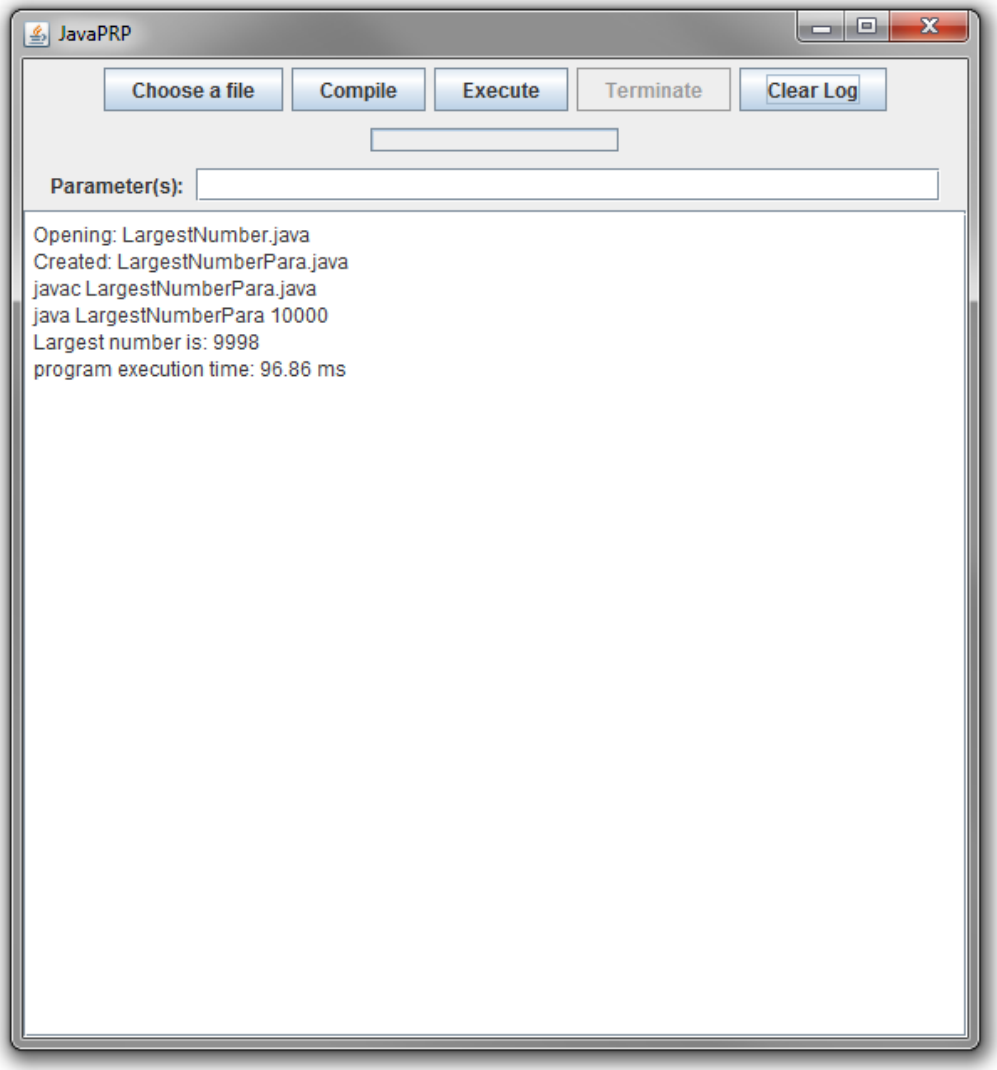
1.5.2 Det sekvensielle programmet

```
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);
        }
        /*CALL*/
        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 3: *Dette viser hvordan JavaPRP ser ut etter man har valgt en fil, kompilert og kjørt.*

Figur 3 viser hvordan JavaPRP 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 JavaPRP 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 JavaPRP 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.