UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Eksamen i:	INF2440— Effektiv parallellprogrammering
Eksamensdag:	7. juni 2016
Tidspunkter:	09.00 – 13.00
Oppgavesettet er på:	3 sider + 1 side vedlegg
Vedlegg:	I) Skisse av Modell2-koden med litt fra oppgave 3,
Tillatte hjelpemidler:	Alle trykte og skrevne notater, utskrifter, bøker ol.

- Kontroller at oppgavesettet er komplett, og les nøye gjennom oppgavene før du løser dem. Poengangivelsen øverst i hver oppgave angir maksimalt antall poeng.
- Du kan legge dine egne forutsetninger til grunn og gjøre rimelige antagelser, så lenge de ikke bryter med oppgavens "ånd". Gjør i så fall rede for disse forutsetningene og antagelsene.
- Til eksamen skal svarene skrives på gjennomslagspapir. Da må du huske å skrive hardt nok til at besvarelsen blir mulig å lese på alle gjennomslagsarkene, og ikke legge andre deler av eksamensoppgaven under når du skriver.
- Til eksamen skal du selv beholde underste arket etter levering av de to øverste til eksamensinspektøren. Nummerér sidene, og husk å skrive kandidatnummeret ditt på besvarelsen.

I vedlegg I) finner du en litt forenklet versjon av Modell2-koden som ble nyttet i kurset (en ytre klasse med main-tråden og en indre klasse hvor objekter av denne blir egne tråder) I den skissen er det markert med store bokstaver ulike områder av denne koden som det blir referert til i oppgavene.

Oppgave 1 (10 poeng)

a) Beskriv meget kortfattet de to viktigste egenskapene ved tråder i et Java-program.

b) Terminerer programmet alltid når maintråden er ferdig? Begrunn svaret.

Oppgave 2 (20 poeng)

a) Tenk deg et program hvor det er en tråd som usynkronisert både leser og øker en variabel 'x' (initielt = 0) med 1 hver gang den er ferdig med en deloppgave, mens de andre trådene går i løkke, leser 'x' og skal terminere når **x == 20**. Kan dette gå galt – vil alle 'lese-trådene' sikkert terminere hvis det løses minst 22 slike deloppgaver? Begrunn svaret (maksimalt 10 linjer).

b) Tenk deg et program hvor det er en tråd som usynkronisert både leser og øker en variabel 'x' (initielt = 0) med 1 hver gang den er ferdig med en deloppgave, mens de andre trådene går i løkke, leser 'x' og skal terminere når **x ≥ 20**. Kan dette gå galt – vil

alle leser-trådene sikkert terminere hvis x økes til minst 22? Begrunn svaret (maksimalt 10 linjer).

Oppgave 3 (25 poeng)

Se på koden i Vedlegg 1, der vi har et lite parallelt program som skriver litt ut i main og i run. Merk den litt uvanlige initiering av CyclicBarrier b i forhold til hvor mange tråder vi har.

a) Hva skriver programmet ut (hvor mange A-er og hvor mange B-er og kommer teksten 'Main terminerer' ut)?
b) Terminerer programmet – begrunn svaret.
c) Hvis du endrer **`num =3`** til **`num=2`**, hva skriver programmet ut da?
d) Hvis **`num=2`** som i pkt. c), terminerer da programmet – begrunn svaret
e) Som du ser står metoden :**`'void sync()`**' deklarert i området B, lokalt i Arbeider-trådene . Tenkt deg at du flyttet denne deklarasjonen opp til området A i den ytre klassen. Beskriv kort hva dette vil ha å si for oppførselen av programmet i pkt. a)-d).

Oppgave 4 (50 poeng)

Vi antar at du har løst Oblig2 med Erathostenes Sil. Vi er interessert i å teste om det **siste desimale siffer** i alle primtall under 1 milliard er jevnt fordelt (dvs. om det er jevnt over like mange 1,3,7 og 9-ere som er siste siffer i primtallene) . Det er opplagt at det er bare to primtall som slutter på 2 og 5, nemlig tallene 2 og 5 selv, og ingen andre primtall slutter på 0,2,4,6 og 8 (fordi slike tall er delbare på 2).

Du lager en Erathosthenes sil slik:

```
ErathosthesSil s = new ErathosthesSil(1000000000);
```

Vi antar at du kaller denne bare en gang og nytter den sekvensielle Silen du laget i Oblig2. i både punktene a) og b) nedenfor.

Erathosthes Sil inneholder en metode:

```
int nextPrime(int i) {
  // returnerer neste primtall >'i'
     return  ..;
}
```

Denne skal du bruke til å undersøke påstanden at siste siffer i primtallene er meget jevnt fordelt (primtallsforskerne undersøker virkelig dette nå).

Oppgave 4.a) Skriv et sekvensielt program som lager en tabell over hvor mange av primtallene > 10 som ender på 1,3,7 eller 9. Bruk da silen du allerede har laget og metoden **`int nextPrime(int i)`** (som du ikke skal skrive, men anta er riktig og bare kan bruke). Beregn også det siste sifferet som færrest primtall slutter på i prosent av det siste sifferet som flest primtall slutter på. Skriv formelen for svaret.

Oppgave 4.b) Skriv et parallelt program som finner samme tabell som i oppgave 4.a. (Du skal altså bare parallellisere det å finne denne tabellen – ikke det å lage Erathosenes Sil eller det å finne den største og minste av 4 tall). Bruk Modell2-koden og forklar bare hvor det du skriver skal plasseres inn der.

Oppgave 5 (40 poeng)

Skriv et sekvensielt og parallelt program som søker etter og finner ut om, og eventuelt da hvor, et tall 's' er i en usortert array: **`int [] a = new int[n].`** Det er mulig at 's', det tallet du leter etter, finnes flere steder i arrayen, og da er det likegyldig hvilke av stedene du svarer.
Du skal bruke Modell2 koden i vedlegget og bare skrive én metode (med nok parametre) som søker i arrayen og som nyttes av både den sekvensielle løsningen og den parallelle løsningen. I tillegg i den parallelle løsningen kan det komme kode run() metoden som sammenligner svaret fra de k trådene du starter, men det er kanskje ikke tilfellet her? Begrunn valget ditt på dette punktet.
Svaret skal foreligge enten som **-1** i en globalt synlig variabel **`int svarIndeks`**; (i området A i vedlegget) som betyr at tallet ikke fantes; eller som et tall **>= 0** i **`svarIndeks`** som da sier en plass i **`a[]`** du finner tallet 's' som du leter etter. Gi også en vurdering av om denne oppgaven følger Amdahl lov eller Gustavsons lov når vi øker n, lengden av a[].

Oppgave 6 (60 poeng)

Du har en foreleser som tror at han har en genial idé til å lage en raskere innstikksortering-metode kalt swapSort. Problemet med innstikksortering, tenkte han, er at det er en del store elementer tidlig (med lave indekser) i arrayen som må skyves langt avgårde mot enden av arrayen, og motsatt mange små verdier alt for langt ut i array-en som sorteres mot begynnelsen. Disse må flyttes (byttes med hverandre) før vi gjør innstikksortering til sist.

a) Hvis vi sammenligner alle par av to elementer (a[i] og a[n/2+i], i = 0,1,…n/2-1) mot hverandre og bytter om de to hvis den som er til venstre er større enn den til høyre, så vil arrayen bli mye raskere å sortere med innstikksortering etterpå.
b) Ved nærmere ettertanke, tenkte foreleseren din, hvis dette er lurt, så kan vi rekursivt gjenta det, først for hel arrayen, så for første halvdel og så for halvdel rekursivt hver for seg, og igjen for disse halvdelenes halvdeler igjen så lenge lengden av det området vi skal bytte om på er > 10. Denne rekursjonen utfører du altså etter at du har swapSortert hele arrayen .
c) Når vi er ferdige med alle disse ombyttingene, gjøres ett kall på innstikksortering av hele arrayen.

Oppgave:
Bruk Modell2-koden og forklar bare hvor det du skriver nedenfor skal plasseres inn der.

1. Programmér denne algoritmen sekvensielt.
2. Lag en parallell versjon av dette sekvensielle programmet (bare parallelliser ombyttingene – ikke det avsluttende kallet på innstikksortering).

Du kan anta at du har gitt en metode:

```
void innstikkSort(int[] a, int left, int right)
```

som sorterer arrayen a[] fra og med element a[left] til og med element a[right].

Til sist : Gi din vurdering om dette er en god idé eller ganske dårlig idé – begrunn svaret.

Appendix I – Modell2 kodeskisse med litt fra oppgave 3:

```
import java.util.concurrent.*;

class Problem {
    //  felles data og metoder A
     static int num = 3;                                  // MERK – oppg.3
     CyclicBarrier b = new CyclicBarrier(num);    // MERK – oppgave3

     public static void main(String [] args) {
          Problem p = new Problem();
          p.utfoer(num+1);                                // MERK – oppg. 3
          System.out.println(" Main-tråden  TERMINERER"); // MERK – oppg. 3
     } // end main

     void utfoer (int antT) {
          Thread [] t = new Thread [antT];
          for (int i =0; i< antT; i++)
              ( t[i] = new Thread(new Arbeider(i))).start();
           try{
             for (int i =0; i< antT; i++) t[i].join();
          } catch(Exception e) {}
     } // end utfoer

     class Arbeider implements Runnable {
          // lokale data og metoder B
          void sync() { try{ b.await(); }  // MERK – oppgave3
                       catch (Exception e) { return;}
           }
          int ind;

          Arbeider (int in) {ind = in;}

           public void run( ) {
                  // kalles når tråden er startet –  MERK oppg.3
                    sync();
                    System.out.println("A");
                    sync();
                    System.out.println("B");
            } // end run
     } // end indre klasse Arbeider
} // end class Problem
```