UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Eksamen i :	INF2440— Praktisk parallell programmering
Eksamensdag :	2. juni 2014
Tidspunkter:	14.30
Oppgavesettet er på :	4 sider
Vedlegg :	Skisse av Model2-koden .
Tillatte hjelpemidler :	Alle trykte og skrevne

- 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. Nummerer sidene, og husk å skrive kandidatnummeret ditt på besvarelsen.

I vedlegget finner du en litt forenklet versjon av Model2-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) Hva er en tråd i Java (svar: Maksimalt 5 linjer)

Svar: En separat programkode som eksekvere som et sekvensielt program som prosesseres i parallell innen rammen av et Java-program(en prosess). Trådene deler samme adresserom (ser samme variable og klassenavn**.**

b) Hvis vi har *k* kjerner i en multikjerne maskin, hvor mange tråder kan vi da ha minimalt og maksimalt i et Java-program?

Svar: Fra 1 til ubestemt mange – si mer enn 100 000 (avhengig av hva operativsystemet greier å understøtte).

Oppgave 2 (10 poeng)

Du skal her se på konsekvenser av Amdahls lov. Anta at du har et program med 25% sekvensiell kode og da de resterende 75% som kan parallelliseres. Anta at du har 8 kjerner og kan parallellisere de 75% perfekt, og at det er ikke noen overhead med å starte og stoppe tråder. Hvilken speedup kan du da forvente deg av ditt 'perfekt' parallellisert program – utled svaret (ikke bare svar et tall)?

Svar: 100/(25+75/8) = 2,91 – siden kandidatene ikke hadde kalkulator (og ikke lenger kan dividere for hånd) er svar som ca. 3 helt OK

Oppgave 3 (15 poeng)

Anta at du i felt A i vedlegget deklarerer følgende variabel, og at du **har startet 15 tråder**:

```
CyclickBarrier grind = new CyclickBarrier(14);
```

Trådene eksekverer kode i sin run()-metode, som før eller siden alle utfører :

```
try{ grind.await();
catch (Exception e) {return;}
```

a) Beskriv situasjonen når den første tråden utfører denne koden.
   Svar: Den venter
b) Beskriv situasjonen når den 13de tråden utfører denne koden .
   Svar:Den første + alle de andre 12 venter.
c) Beskriv situasjonen når den 14de tråden utfører denne koden.
   Svar: Alle de 14 trådene fortsetter (slippes fri) – og den sykliske barrieren resettes til 14.
d) Beskriv situasjonen når den 15de tråden utfører denne koden.
   Svar: Den 15de tråden venter.

Oppgave 4 (20 poeng)

Anta at du har to tråder som aksesserer en felles variabel, deklarert i felt A slik: `int i =0;`

Tråd0 utfører følgende kode: `i++; i++;` før den terminerer, mens
Tråd1 utfører: `i--; i--;` før den terminerer. Det er ingen synkronisering.

Hvilke mulige verdier er det at `i` har etter at begge trådene har terminert?
Tegn diagrammer med tidsakse som viser ***hvordan*** hver av de verdiene du hevder kan bli verdien av `i` til sist, kan fremkomme (når leser og skriver T0 og T1 hvilke verdier?):

Svar: Alle verdiene : -2.-1,0,1,2 er mulige, og det er mange mulige måte å komme frem til hver av disse verdiene:

I = 2

T0 R:0 W:1 R:1 W:2

T1 R:0 W:-1 R:-1 W: -2

I= 1

T0 R:-1 W:0 R:0 W:1		T0:
T1 R:0 W:-1 R:-1 W: -2		
I= 0		
T0 R:0 W:1 R:1 W:2		
T1 R:0 W:-1 R:1 W: 0		
I = -1		
T0 R:0 :1 R:-1 W:0		
T1 R:0 W:-1 R:0 W: -1		
I = -2		
T0 R:-1 W:0 R:0 W:1		
T1 R:0 W:-1 R:-1 W: -2		

Oppgave 5 (40 poeng)

Skriv først en sekvensiell og så en parallell metode for å beregne største kolonne-summen (= summen av elementene i en kolonne) i en matrise: **`int a[][] = new int[n][n];`** La til slutt main-tråden skrive ut svaret (indeksen til kolonnen med størst sum+ selve summen).

a) Før du skriver den parallelle koden skal du beskrive minst to, og helst tre måter å dele opp matrisen for de ulike trådene. Vurdere de oppdelingene du beskriver om hvilken av dem du mener vil gi raskest parallell kode, og begrunn kortfattet svaret.

b) Skriv parallell kode for den oppdelinga du har valgt.

N.B. Du skal **ikke** skrive hele programmet, men bare den sekvensielle metoden, de datastrukturer du trenger og den/de metodene du trenger i det parallelle tilfellet som kalles fra run() - metoden. For begge deler, skriv med kommentar i koden hvilke områder (A eller B) i vedlegget du tenker disse plassert.

Det er virkelig mange måter å dele opp en nxn-matrise a[][] for å beregne største kolonnesum. Noen av dem trenger ekstra data (burde kanskje vært nevnt i oppgaven?). Her er de 4 første jeg har skrevet kode for og testet speedup på. Speedup ble altid målt mot standard sekvensiell kode hvor man enkelt summerte alle kolonnene og fant største sum(Java lagrer data radvis, slik at det å aksessere data kolonnevis går på tvers av cache-linjene og tar laaaang tid).
Anta at vi har k tråder og k kjerner. Hver tråd vet sin indeks 'ind' (0,1,...,k-1) og har i B-området variable left = (n/k) *ind og: right = (n/k)*(ind+1) , og hvis ind == k-1 er right = n;

1) Læreboksdefinisjonen – del opp data slik at tråd ind summerer kolonnesummer i kolonnene:a[][left..right-1],
   Og legger hver sine maxima i en array maxV[] og tilhørende kolonne-indeks i kolM[numThreads]
   Speedup : 2.85 (n=20000), 0.99 (n = 10000),....., 0.16(n= 312),..., 0.0036 (n=39)
2) Kolsum radvis, a[0..n-1][left..right-1]. Data summeres kolSums[i]
   Speedup : 71 (n=20000), 51 (n = 10000),....., 0.77(n= 312),... 0.0017 (n=39)
   Sekvensielt etterpå finnes max kolsum og tilhørende kolonneindeks i kolSums[]
3) Kolsum radvis, a[left..right][0..n-1]. Data summeres kolSums3[k][0..n]
   Speedup : 45 (n=20000), 43 (n = 10000),....., 0.40(n= 312),... 0.006 (n=39)
   Sekvensielt etterpå finnes max kolsum og tilhørende kolonneindeks i kolSums3[][]

4) Tråd ind summerer vektor ind, ind +k, ind +2k,..., inntil indeks >= n og legger største verdi i maxV[k] og tilhørende kolonneindeks i kolM[k]
Speedup : 2.7 (n=20000), 0.99 (n = 10000),....., 0.32(n= 312),... 0.0027 (n=39)
Sekvensiell etterbehandling hvor største verdi finnes i maxV[] og tilhørende indeks i kolM[].

Konklusjon: Det er to lure måter å summere kolonne i en matrise 2) og 3) som fulgte cache-linjene radvis, som la fra seg data i ekstra-arrayer, som så tilsist ble sekvensielt prosessert. Metode 1) og 4) gikk på tvers av cache-linjene og fikk nesten ikke speedup selv for store n.
Transponering først er en dålig ide fordi atall operasjoner ved det er like stort som summeringen.

Program finnKolonneSum.java vedlagt.

Et fullgodt svar er da at man nevner 2 eller 3 av disse oppdelingene og implementerer helst metode 2) eller3).

Sekvensiell data & metode i A:

```
int maxVsum=0, kolMax=0;

 void finnKolSumSeq(int [][] a) {
          int  max;
          for (int j=0; j < n; j++) {
                   max = 0;
                   for(int i = 0; i < n; i++){
                     max += a[i][j]; // for hver kolonne 'j' summer alle elementer 'i'
            }
            if (max > maxVsum) {maxVsum= max; kolMax =j;}
    }
 } // end finnKolSumSeq
```

Data for 2) i område A:

```
int []  kolSums ;
int max=0,maxKol;

void etterProsesseringSekvensiell2(){
  // Kolsum rowvise, left..right. Data kolSums[n]
  max =0; maxKol=0;
  for (int i = 0; i < n; i++){
          if (max < kolSums[i]){
                   max = kolSums[i];
                   maxKol= i;
          }
  }
```

Data for 2) i område B:

```
int left, right,ind;

 public  void paraKolSum2(int [][] a) {
```

```
        for (int i=0; i< n; i++) {
          for (int j = left; j < right; j++){
                kolSums[j] += a[i][j];
          }
        }
    } // end paraKolSum2
```

Oppgave 6 (20 poeng)

Anta at at du **har startet 5 tråder og at** du i felt A i vedlegget deklarerer følgende to variable:

```
int teller1 = 0;
Semaphore abc = new Semaphore(1);
```

Trådene eksekverer kode i sin run()-metode, som før eller siden alle prøver å utføre:

```
try{ abc.acquire();
     teller1++;
     abc.release();
}catch(Exception e) {};
```

a) Hva er verdien av **`teller1`** etter at første tråden prøver å utføre denne koden.
Svar: 1

b) Hva er verdien av **`teller1`** etter at andre tråden prøver å utføre denne koden.
Svar: 2

c) Hva er verdien av **`teller1`** etter at tredje tråden prøver å utføre denne koden.
Svar: 3

d) Hva er verdien av **`teller1`** etter at fjerde tråden prøver å utføre denne koden.

e) Svar: 4

f) Må noen av trådene vente ved utførelsen av punktene a)-d). Begrunn svaret kort.
Svar: Ja, hvis flere tråder samtidig prøver å utføre koden, teller1++ er beskyttet og da en atomisk operasjon, og da utfør en av dem koden og de andre venter i en kø.

Oppgave 7 (70 poeng)

Du har en stor mengde små heltall i byte-arrayen **`a[]`** (f.eks flere hundre millioner), og en søkestring i en byte-array **`s[]`** (f.eks 4-10 tall) og er interessert å finne ut følgene:

a) Hvor mange ganger forekommer søkestringen **`s[]`** **`i a[]`** i samme rekkefølge – svaret er en ArrayList med startindeksen i **`a[]`** hvor det er en slik eksakt likhet.

b) Hvor er det en sekvens av tall i **`a[]`** som forekommer s[] , men hvor det er eksakt en feil. Dvs. at det finnes en sekvens av fortløpende tall i **`a[]`** som er lik søketallene i **`s[],`** men hvor vi har at eksakt en av tallene i **`a[]`** ikke er lik motsvarende tall i **`s[]`**.– svaret er her

også en ArrayList med startindeksen i **a[]** hvor det er en slik type likhet.

Husk at hvis vi søker etter **s[] = 123**123 og a[] inneholder ….**0012312312342399999** …., så vil vi finne **123123** to ganger med null feil og en gang med 1 feil i dette området av **a[].** Vi aksepterer altså at to slike funn overlapper hverandre, men det er jo helt avhengig av søkenøkkelen om det er aktuelt.

(dette tilsvarer på en måte søking etter et gen i en DNA-streng, og søking etter et gen med en feil i en DNA-streng).

Hvis du deler opp data i **a[]** på en eller annen måte i parallellisering av a) og b) må du ta hensyn til at en slik likhet som vi leter etter kan krysse en slik delelinje.
Skriv et sekvensiell og et parallelt program av disse to relativt like søkeprogrammene.

N.B. Også her skal du **ikke** skrive hele programmet (du kan referere til koden i vedlegget)), men bare den sekvensielle metoden **flette**, de datastrukturer du trenger og den/de metodene du trenger i det parallelle tilfellet som kalles fra run() - metoden. For begge deler, skriv med kommentar i koden hvilke områder (A eller B) i vedlegget du tenker disse plassert.

Svar:

Appendix – Model2 kodeskisse:

```
import java.util.concurrent.*;
class Problem {
    //  felles data og metoder A
     public static void main(String [] args) {
          Problem p = new Problem();
          p.utfoer(12);
     }
     void utfoer (int antT) {
          Thread [] t = new Thread [antT];
          for (int i =0; i< antT; i++)
              ( t[i] = new Thread(new Arbeider(i))).start();

           for (int i =0; i< antT; i++) t[i].join();
     }

     class Arbeider implements Runnable {
        int ind; // lokale data og metoder B

           Arbeider (int in) {ind = in;)
          public void run( ) {
```

```
                    // kalles når tråden er startet
            } // end run
        } // end indre klasse Arbeider
} // end class Problem
```