

UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Eksamen i: INF2440— Effektiv parallellprogrammering
Eksamensdag: 2. juni 2015
Tidspunkter: 09.00 – 13.00
Oppgavesettet er på: 3 sider + 2 sider vedlegg
Vedlegg: I) Skisse av Modell2-koden, II) Den sekvensielle radix-sorteringa fra Oblig4 og III) Kvikksort
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. Nummerer 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. I vedlegg II) er den sekvensielle koden for radix-sortering fra oblig4 (som du kanskje trenger i oppg. 7), og i vedlegg III er Kvikksort fra forelesningen Uke9 som du kanskje trenger i oppgave 4.

Oppgave 1 (20 poeng)

- a) Hvilke feil kan vi få ved at det ikke synkroniseres mellom to tråder som har felles variable i Java som det skrives på. (svar: Maksimalt 20 linjer)
- b) Definer vranglås i et parallelt, trådbasert program. Gi et eksempel på en mulig vranglås med bruk av tre Semaphorer mellom tre tråder. (svar: Maksimalt $\frac{3}{4}$ side).

Oppgave 2 (15 poeng)

Anta at du i felt A i vedlegg I deklarerer følgende variabler, og at du **har startet 3 tråder**:

```
CyclicBarrier ab = new CyclicBarrier(2),
              bc = new CyclicBarrier(2),
              ca = new CyclicBarrier(2);
```

Tråd **A** eksekverer før eller siden i sin run()-metode:

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

Tråd **B** eksekverer før eller siden i sin run()-metode:

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

Tråd **C** eksekverer før eller siden i sin kode run()-metode:

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

Spørsmål: Går dette bra; beskriv tre ulike situasjoner av rekkefølger mellom disse synkroniseringene og de ulike ventesituasjoner som da oppstår.

Oppgave 3 (15 poeng)

Anta at du har to tråder som aksesserer en felles variabel, deklart i felt A slik: `int i = 0`, og at hver av trådene har deklartert i sitt lokale område B i hver tråd:

```
ReentrantLock lock = new ReentrantLock();
```

Tråd0 utfører følgende kode: `lock.lock();`
`try { i = i+2;`
`} finally { lock.unlock();}`

før den terminerer, mens

Tråd1 utfører følgende kode: `lock.lock();`
`try { i = i-3;`
 `i = i+1;`
`} finally { lock.unlock();}`

før den terminerer,

Hvilke mulige verdier kan `i` ha etter at begge trådene har terminert? Regn med at Tråd1 leser `i` fra hovedlageret for hver av sine to setninger og skriver verdien ned etter utførelse.

Tegn diagrammer med tidsakse som viser *hvordan* minst tre av de verdiene du hevder kan bli verdien av `i` til sist, kan fremkomme (når leser og skriver Tråd0 og Tråd1 hvilke verdier?).

Oppgave 4 (40 poeng)

Skriv først en sekvensiell og så en parallell metode for å sortere radene i en positiv (dvs. alle elementene er ≥ 0) todimensjonal array `int a[][] = new int[n][n]`; slik at den raden med størst sum er nederst (radnummer: n-1), den med nest største sum er nest nederste rad (radnummer: n-2),....., osv.

Når du sorterer radene skal du flytte om på pekerne til radene framfor å flytte selve elementene. Du kan anta at $n \leq 20\,000$, slik at innstikksortering ikke blir aktuelt, men f.eks kvikksort som du her skriver koden til. Bruk som utgangspunkt den Kvikksort-koden du finner i Vedlegg III og skriv ned hvordan du vil modifisere den for denne oppgaven.

N.B. Du skal **ikke** skrive hele programmet, men bare den sekvensielle metoden og de datastrukturer og den/de metodene (inkludert sorteringa) du trenger; og tilsvarende det parallele tilfellet skrive de datastrukturer og den/de metodene (inkludert sorteringa) du trenger og som kalles fra `run()` – metoden (også sorteringsmetoden). For data-deklarasjonene, skriv med kommentar i koden i hvilke områder (A eller B) i vedlegg I) du tenker dem plassert.

Oppgave 5 (30 poeng)

Anta at du har deklart en array: `double tallene[] = new double [n]`; og du kan anta at elementene i `tallene[]` er sortert stigende.

Du skal nå skrive først en sekvensiell og så en parallell metode (anta at du har k kjerner) som snur innholdet av `tallene[]` slik at tallene blir sortert synkende. Skriv heller ikke her hele programmet, men bare de metodene og evt. data som du trenger i A og B området i vedlegget + hva innholdet av `run()`-metoden i trådene er.

Oppgave 6 (30 poeng)

Du skal her se på konsekvenser av Gustafsons lov. Anta at du har et program som med $n = 1000$ først bruker 50% av kjøretiden i en sekvensiell kode som har kompleksitet $O(n)$, og så 50 % av kjøretiden i en perfekt parallellisert kode med $k = 8$ kjerner hvor koden har kompleksitet $O(n^2)$ og hvor data er delt i 8 like store deler. Regn ut den speedup du får med $n = 1000$ og $n = 2000$ hvor du i begge tilfeller bare har parallellisert den siste delen av koden. Utled svaret, ikke bare kom med et tall svar!

Oppgave 7 (60 poeng)

Det er to typer av ‘den midterste verdien’ av n elementer: a_0, a_1, \dots, a_{n-1} :

- Det geometriske gjennomsnittet $G = (\sum_{k=0}^{n-1} a_k) / n$.
- Medianverdien M, som er slik at halvparten av a_i -ene er $\leq M$ og halvparten er $\geq M$.

(for enkelhets skyld antar vi her at n er et oddetall og muligens et stort tall.)

Oppgave:

- a) (5 poeng) Lag en skisse av hvordan du lett kan parallellisere det å finne det geometriske gjennomsnittet G.
- b) (20 poeng) Forklar hvorfor det er klart mulig, men ikke like raskt å forsøke å finne medianverdien M med parallell kode. (Hint: Hva må vi først gjøre før vi kan finne M?)
- c) (Vanskelig: 35 poeng) Her skal du kan ta utgangspunkt i løsningen din på Oblig4. Da kan du bla. droppe nesten all flyttingen av elementene for siffer 2 fra b til a, men likevel finne medianverdien M både i den sekvensielle og parallelle koden. Lag skisser.

Appendix I – Modell2 kodeskisse:

```
import java.util.concurrent.*;
class Problem {
    // felles data og metoder A
    public static void main(String [] args) {
        Problem p = new Problem();
        p.utfoer(8);
    }
    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) {}
    }

    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
```

Appendix II – Sekvensiell Radix-sortering fra Oblig 4:

```
class SekvensiellRadix{

    static void radix (int [] a) {
        // 2 digit radixSort: a[]
        int max = a[0], numBit = 2, n =a.length;
        // a) finn max verdi i a[]
        for (int i = 1 ; i < n ; i++)
            if (a[i] > max) max = a[i];
```

```

while (max >= (1L<<numBit) )numBit++; // antall siffer i max

// bestem antall bit i siffer1 og siffer2
int bit1 = numBit/2,
    bit2 = numBit-bit1;
int[] b = new int [n];
radixSort( a,b, bit1, 0); // første siffer fra a[] til b[]
radixSort( b,a, bit2, bit1); // andre siffer, tilbake fra b[] til a[]
} // end radix

/** Sort a[] on one digit ; number of bits = maskLen, shiftet up 'shift' bits */
static void radixSort ( int [] a, int [] b, int maskLen, int shift){
    int acumVal = 0, j, n = a.length;
    int mask = (1<<maskLen) -1;
    int [] count = new int [mask+1];

    // b) count=the frequency of each radix value in a
    for (int i = 0; i < n; i++) {
        count[(a[i]>> shift) & mask]++;
    }

    // c) Add up in 'count' - accumulated values
    for (int i = 0; i <= mask; i++) {
        j = count[i];
        count[i] = acumVal;
        acumVal += j;
    }

    // d) move numbers in sorted order a to b
    for (int i = 0; i < n; i++) {
        b[count[(a[i]>>shift) & mask]++] = a[i];
    }
} // end radixSort

} // end class SekvensiellRadix

```

Appendix III - Kvikksortering (uten innstikkSort) fra forelesningen Uke 9:

```

// sekvensiell Kvikksort uten insertSort
void quicksortSek(int[] a, int left, int right) {
    int piv = partition (a, left, right); // del i to
    int piv2 = piv-1, pivotVal = a[piv];
    while (piv2 > left && a[piv2] == pivotVal) {
        piv2--; // skip like elementer i midten
    }
    if ( piv2-left > 0) quicksortSek(a, left, piv2);
    if ( right-piv > 0) quicksortSek(a, piv + 1, right);
} // end quicksortSek

```

```

// del opp a[] i to: smaa og større
int partition (int [] a, int left, int right) {
    int pivVal = a[(left + right) / 2];
    int index = left;
    // plasser pivot-element helt til høyre
    swap(a, (left + right) / 2, right);
    for (int i = left; i < right; i++) {
        if (a[i] <= pivVal) {
            swap(a, i, index);
            index++;
        }
    }
    swap(a, index, right); // sett pivot tilbake
    return index;
} // end partition

void swap(int [] a, int left, int right) {
    int temp = a[left];
    a[left] = a[right];
    a[right] = temp;
} // end swap

```