am. 18. mars, 2016

Oblig 3 i INF2440 – v2016. Parallell Radix med variabelt antall sifre – frist 15.april.

Radix-sortering (eller mer presist: HøyreRadix- sortering) er en vel kjent algoritme som sorterer en array med heltall ved å flytte elementene mellom to arrayer a[] og b[] av samme lengde n. Ved hver flytting blir tallene sortert på ett siffer, og algoritmen starter med å sortere tallene på det siste, minst signifikante sifferet (det lengst til høyre i tallene). Ved neste sortering sorteres det på det nest minst signifikante sifferet,.. osv. helt til alle sifrene i tallene er sortert.

Et siffer er et bestemt antall bit, alt fra 1 til 30, og vi velger selv hvor stort siffer vi sorterer med. Når vi sorterer med flere sifre, behøver ikke alle sifrene være like lange. Er et siffer numBit langt, er de mulige sifferverdiene **: 0..2^{numbit}** -1 (eks. er sifferet 8 bit langt, er sifferverdiene: 0,1,..,255). Effektivitetsmessig lønner det seg at det er fra 8-11 bit i et siffer. Den sekvensielle koden for Radix-sortering er gjengitt i appendix A og i også finnes i oblig-mappa på hjemmesida. De foilene som forklarer sekvensiell Radix er i forelesningene Uke8.

Denne versjonen av Radix-sortering som dere skal parallellisere i Oblig3, MultiRadix, velger selv hvor mange sifre den vil sortere på avhengig av hvor mange bit det er i Max-verdien i arrayen a [] som skal sorteres. Husk å parallelliser denne varianten og ikke den versjonen med to sifre som ble nyttet i 2014 og 2015.

Innlevering

Som du ser inneholder de to algoritmene til sammen 4 steg: a), b), c) og d) som du skal parallellisere. Steg a) har vi parallellisert før og løsningen for steg b) blir skissert i ukeoppgavene i uke 8/9 (se appendix B). Du skal parallellisere de alle de resterende stegene du ikke har kode for og lage en sammenhengende parallell algoritme som sorterer i parallell og som har speedup > 1 for 'tilstrekkelig stor n'.

Det du skal levere er programkoden og en rapport som først viser kjøretider, og speedup for n= 2000, 20 000, 200 000, 2 mill, 20 mill og 200 mill. Løsningen skal også inneholde en enkel test på om arrayene er sortert (a[i-1]≤ a[i], i=1,2,..,a.length-1) og den skal gi en feilmelding med for hvilken indeks det først evt. var feil i den sorterte arrayen. Denne testen kjøres (utenfor tidtakingen) både etter den sekvensielle og parallelle sorteringa.
Beskriv deretter med egne ord i rapporten hvordan steg c) ble parallellisert. Tips: Det kan hende at trådene bør ta en ekstra kopi av count[] som bestemmer hvor den skal flytte sine: 0-er, 1-ere,…..,

Tallene som skal sorteres skal trekkes uniformt mellom 0..n-1 (og denne trekkingen holdes utenfor tidtakingen). Bruk helst Model2-kode for implementasjonen din, men model3-kode kan også nyttes. I alle fall skal tidene du levere beregnes som medianen av minst 3 kjøringer for hver verdi av n.

Obliger i INF2440 innleveres i Devilry. Husk at det sammen selve koden på begge punktene skal ligge en rapport med tabeller og forklaringer over kjøretidene som beskrevet ovenfor. Oblig 3 leveres individuelt og senest innen fredag 15.april kl. 23.59.

Appendix A

Kildekode for MultiRadix-sortering med variabelt antall sifre (tallet 1 i max-beregningen måtte gjøres til en long, dvs. 1L, for å også kunne sortere alle tall mellom 2^{31} og 2^{32})

```
/** N.B. Sorterer a[] stigende – antar at: 0 ≤ a[i]) < 2^32   */

 // viktig konstant
  final static int NUM_BIT =7; // eller 6,8,9,10..

  int []  radixMulti(int [] a) {
                        long tt = System.nanoTime();
                        // 1-5 digit radixSort of : a[]
                        int max = a[0], numBit = 2, numDigits, n =a.length;
                        int [] bit ;

                       // 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 numBits sifre
                        numDigits = Math.max(1, numBit/NUM_BIT);
                        bit = new int[numDigits];
                        int rest = numBit%NUM_BIT, sum =0;

                        // fordel bitene vi skal sortere paa jevnt
                        for (int i = 0; i < bit.length; i++){
                                  bit[i] = numBit/numDigits;
                           if ( rest-- >0) bit[i]++;
                        }

                        int[] t=a, b = new int [n];

                        for (int i =0; i < bit.length; i++) {
                                  radixSort( a,b,bit[i],sum );    // i-te siffer fra a[] til b[]
                                  sum += bit[i];
                                  // swap arrays (pointers only)
                                  t = a;
                                  a = b;
                                  b = t;
                        }
                        if (bit.length&1 != 0 ) {
                                  // et odde antall sifre, kopier innhold tilbake til original a[] (nå b)
                                  System.arraycopy (a,0,b,0,a.length);
                        }

                        double tid = (System.nanoTime() -tt)/1000000.0;
                        System.out.println("\nSorterte "+n+" tall paa:" + tid + "millisek.");
                        testSort(a);
                        return a;
           } // end radix2



          /** Sort a[] on one digit ; number of bits = maskLen, shiftet up 'shift' bits */
           void radixSort ( int [] a, int [] b, int maskLen, int shift){
```

```
//  System.out.println(" radixSort maskLen:"+maskLen+", shift :"+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, i.e pointers
            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
```

Appendix B, Ukeoppgave i uke 8/9:

Denne uka skal vi også se på det å parallellisere steg b) i Radix, det første steget inne i radixSort-metoden. Husk at vi allerede har parallellisert steg a), det å finne største verdien i **`a[].`**

I steg b) ønsker vi å finne hvor mange det er av de ulike sifferverdiene i **`a[]`** – hvor mange 0-ere, 1-ere, .. osv. det er i **`a[]`** på det sifferet vi undersøker. Siden vi vet hvor stort sifferet er (for eksempel 10 bit) så vet vi også at de mulige sifferverdiene da er mellom 0 og 1023 (fordi 2^{10} = 1024). Den sekvensielle koden er en enkel for-løkke:

```
for (int i = 0; i < n; i++){
   count[(a[i]>> shift) & mask]++;
}
```

Vi skal altså ha en array **`count[]`** hvor vi teller opp hvor mange det er av hver mulige verdi på dette sifferet. Det uttrykket **`(a[i]>> shift) & mask`** som finner sifferverdien i **`a[i]`** skal vi heldigvis ikke gjøre noe med i parallelliseringen. Det er gjennomgått i Uke3 på forelesninga, og nå skal vi bare akseptere at det virker som beskrevet.

Vi skal nå beskrive en metode som gjør at alle trådene jobber hele tiden og at vi bare gjør to synkroniseringer per tråd på en **`CyclicBarrier synk`**. Problemet her er at **`count[]`**er en felles variabel, og at vi bryter en av de tre reglene for skriving og lesing på felles variable hvis vi lar to eller flere av trådene skrive samtidig på samme variabel eller samme array-element eller en annen tråd lese hvis en annen tråd skriver..

Du skal følge følgende algoritme for parallellisere dette steget (anta at vi har k tråder, og sorterer på et siffer som har **`numSif`** mulige sifferverdier på det sifferet vi sorterer på):

1) Opprett en to-dimensjonal **`int[][] allCount = new int[antTraader][]`** som fellesdata. I tillegg deklareres også **`int[]sumCount = new int[numSif]`** som fellesdata.
2) Du deler så ***først*** opp **`a[]`** slik at tråd$_0$ får de n/k første elementene i **`a[],`** tråd$_1$ får de neste n/k elementene,…, og tråd$_{antTråder-1}$ de siste elementene i **`a[].`**
3) Hver tråd har en egen **`int[]count = new int[numSif].`** Vi teller så i alle trådene opp hvor mange det er av hver mulig sifferverdi i den delen av **`a[]`** som vi har, og noterer det i vår lokale **`count[].`**
4) Når tråd$_i$ er ferdig med tellinga, henger den sin **`count[]`** opp i den doble int-arrayen som da vil inneholde alle opptellingene fra alle trådene, slik: **`allCount[i] = count`**;
5) Alle trådene synkroniserer på den sykliske barrieren '**`synk`**'.
6) Nå skal vi dele opp arrayen **`allCount[][]`** etter verdier i **`a[]`**, slik at tråd$_0$ får de n/k første elementene i **`sumCount[]`** og de n/k første kolonnene i **`allCount[][]`**, tråd$_1$ får de neste n/k elementene i **`sumCount[]`** og kolonnene i **`allCount[][]`** ,…, osv.
7) Hver tråd$_i$ summerer så tallene i alle sine kolonner 'j' fra **`allCount[0..antTråder-1][j]`** til **`sumCount[j].`**
8) Alle trådene synkroniserer på nytt på den sykliske barrieren '**`synk`**'.

Etter pkt. 8 inneholder **`sumCount[]`** nå det samme som **`count[]`** i den sekvensielle algoritmen (de tre linjene ovenfor), og alle trådene har hele tiden lest og skrevet på ulike array-elementer.