

Oblig 4 i INF2440 – v2015-ver2.

Parallell Radix med 2 sifre – frist 30.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 minste 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: $0..2^{\text{numbit}} - 1$ (eks. er sifferet 8 bit langt, er sifferverdiene: 0,1,...,255). Effektivitetsmessig lønner det seg at det er fra 8-13 bit i et siffer. Den sekvensielle koden for Radix-sortering er gjengitt i appendix A (rettet i ver2 for å kunne sortere tall mellom 2^{31} og 2^{32}) og i også finnes i oblig-mappa på hjemmesida. De foilene som forklarer sekvensiell Radix er samlet i en egen fil: radix.pdf som du også finner på hjemmesida til INF2440.

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) ble skissert i ukeoppgavene uke5 (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 $\text{speedup} > 1$ for 'tilstrekkelig stor n '.

Det du skal levere er programkoden og en rapport som først viser kjøretider, og speedup for $n= 1000, 10\ 000, 100\ 000, 1\ \text{mill}, 10\ \text{mill}$ og $100\ \text{mill}$. Løsningen skal også inneholde en enkel test på om arrayene er sortert ($a[i-1] \leq a[i], i=1,2,\dots,a.\text{length}-1$) 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 $\text{count}[]$ 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. 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 30.april kl. 23.59.

Appendix A

Kildekode for Radix-sortering med 2 sifre (retting i ver2 er i setning a2) : tallet 1 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:  $0 \leq a[i] < 2^{32}$  */
static void radix2(int [] a) {
    // 2 digit radixSort: a[]
    int max = a[0], numBit = 2;

    // a1) finn max verdi i a[]
    for (int i = 1 ; i < a.length ; i++)
        if (a[i] > max) max = a[i];

    // a2) bestem antall bit det må sorters paa
    while (max >= (1L<<numBit) )numBit++;

    // bestem antall bit i siffer1 og siffer2
    int bit1 = numBit/2,
        bit2 = numBit-bit1;

    int[] b = new int [a.length];
    radixSort( a,b, bit1, 0); // første siffer fra a[] til b[]
    radixSort( b,a, bit2, bit1); // andre siffer, tilbake fra b[] til a[]
} // end

/** 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
```

Appendix B, Ukeoppgave i uke 5:

Denne uka skal vi også se på det å parallellisere steg 2 i Radix, det første steget inne i RadixSort-metoden. Husk at vi allerede har parallellisert steg 1, det å finne største verdien i `a[]`.

I steg 2) 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 deklarerer også `int[] sumCount = new int[numSif]` som fellesdata.
- 2) Du deler så *først* opp `a[]` slik at tråd₀ får de `n/k` første elementene i `a[]`, tråd₁ 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 tråd₀ får de `n/k` første elementene i `sumCount[]` og de `n/k` første kolonnene i `allCount[][]`, tråd₁ 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.