Ukeoppgaver i uke 5, INF2440 – v2014

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 **CyclickBarrier 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[antTraader]** 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 to linjene ovenfor), og alle trådene har hele tiden lest og skrevet på ulike array-elementer.

Oppgave: Implementer den sekvensielle og parallelle algoritmen i hver sin metode, og finn eksekveringstider for n=1000,100 000, 1 mill. og 10 mill. Lage en tabell over kjøretidene og speedup for de ulike verdiene av n.

For å få kjørt og testet de to algoritmene må du deklarerer **`int[]a = new int[n]`** og initiere den for eksempel med tilfeldige tall mellom 0: n-1, og vi kan si at vi sorterer på siste siffer som er 10 bit stor, og da er shift = 0 og mask= 2^{10}-1= 1023. Disse fire variablene: a, shift, mask og antTråder kan være parametre til de to metodene.