23. jan. 2020

Oblig 1 i INF3030 – v2020
Finn de *k* største tallene i en stor array

Et problem for internettsøkeprogrammer som Google, Bing og DuckDuckGo, er at et søk kan generere millioner av svar (søker du på ''Trump' i Google (22/1-2020) får du 1,81 milliarder svar!)– mer eller mindre relevante for den som søkte. Vedkommende søker har aldri muligheter for å se på alt. Hun/han vil se på det mest relevante. Hver side (av de mange millioner treff) kan vi anta har en relevans-score, som er et heltall som er slik at jo større dette tallet er, desto mer relevant er vedkommende side.

Vi skal hjelpe DuckDuckGo til å parallellisere løsningen på dette problemet. Anta at du har n svar og at relevans-scoren ligger lagret i heltallsarrayen a[0..n-1]. En triviell sekvensiell løsning A1 i Java er å bruke Javas innebygde sorterings algoritme (Arrays.sort(int [] a)), sortere hele arrayen og så plukke ut de *k* største tallene. Men dette tar alt for lang tid. En klart raskere algoritme A2 er følgende:

1. Vi antar først at de *k* tallene er i a[0..k-1] er de største, og innstikksorterer det området i a[] i synkende rekkefølge (du må trivielt skrive om vanlig innstikksortering til å sortere i synkende rekkefølge – dvs. med de største først).
2. Da vet vi at det minste tallet av de *k* første tallene i a[] da ligger i a[*k-1*]. Dette tallet sammenligner vi etter tur med hvert element i a[*k*..n-1]. Finner vi element a[j] > a[*k-1*], så :
    a. Bytt a[*k-1*] med a[j]
    b. Innstikk-sorter det nye elementet inn i a[0.. *k-2*] i synkende rekkefølge (husk at koden for å innsortere bare ett element er enklere enn full instikk-sortering).
3. Når pkt. 2 er ferdig, ligger de *k* største tallene i a[0.. *k-1*] og ingen av øvrige tallene er overskrevet eller ødelagt.

Oppgave 1 – Sekvensiell Løsning:

Implementer den sekvensielle algoritmen A2 ovenfor. Test den for disse ulike verdiene av n= 1000, 10 000, …,100 mill ved å lage en array med tilfeldige tall (java.util.random) og for hver av disse verdiene, tester du for to verdier av *k* =20 og *k*=100. Test at du får riktig svar ved også å sorterer de samme tallene med Arryas.sort(int [] a), og sammenlign dine svar (synkende rekkefølge) med de siste *k* plassene i a[] etter at du har nyttet Arrays.sort() for å sjekke at det er riktig.

(N.B. for å få samme 'tilfeldige' tall med å trekke tilfeldige tall med klassen Random hvis du vil lage samme array flere ganger, må konstruktøren til klassen Random få et starttall – f.eks `Random r = new Random(7361);` Da vil vi få samme tallsekvens når vi sier: `r.nextInt(n)` i løkke for å få neste tall mellom 0 og n-1`).` Tallene n og *k* tar du med som parametre når du starter programmet ditt (> java <n> <k>)

Skriv to tabeller, en for *k* =20, en for 100, som viser hvilke tid Arrays.sort og Innstikk-metoden A2 ovenfor bruker for ulike verdier av n (n= 1000, 10 000, …, 100 million). Fremstiller du resultatene som kurver, kan du bare ha to kurver, en for k=20 og en for k=100 i samme diagrammet. Tidene som rapporteres skal være medianen av 7 kall på begge metodene slik det ble vist på forelesningen Uke2. Innlever din kode til A2 og tabellen med dine kommentarer.

Oppgave 2 – Parallell Algoritme.

Du skal nå parallellisere A2 så godt du greier med de p kjernene du har på din maskin. Ta f.eks utgangspunkt i parallelliseringen av FinnMax – problemet. Det kan hende at du da står igjen med en liten sekvensiell fase etter at mesteparten av beregningene er gjort (som i FinnMax).

Innlevering

Skriv en rapport med enten to tabeller, en for k = 20 og en for k =100, som viser hvilke tider Arrays.sort(), sekvensiell og parallell Innstikk-metoden A2 ovenfor bruker for ulike verdier av n (n= 1000,10 000, …,100 mill). I tillegg helst også et diagram med to kurver, en for k = 20 og en for k =100, som viser speedup som funksjon av n. Tidene som rapporteres her skal også være medianen av 7 kall på begge metodene slik det ble vist på forelesningen Uke2. Innlever din kode til både den sekvensielle og parallelle løsningen + rapporten hvor det også fremkommer hvilken type CPU (navn og hastighet i GHz + antall kjerner) du brukte ; samt tilslutt en kommentar om for hvilken verdi av n du observerer at den parallelle koden gir speedup >1 eller hvorfor ikke den ikke gir speedup for denne verdien av k. Kommentér også om hvordan kjøretidene endres for de to valgene av k.

Obliger i INF3030 innleveres i Devilry som nå er klargjordt for innleveringer dette semesteret.
Oblig 1 leveres individuelt og senest innen **tirsdag 4. feb. 2019 kl 23.59.00** (ikke 23.59.59).

Tips:

1) Får du en feilmelding når du prøver å kjøre programmet ditt for n = 100 mill at du har for lite hukommelse, kan du starte programmet slik. Da ber du om 6 GB til programmet:

```
> java –Xmx6000m Oblig1 100000000 <+evt andre parametere>
```

(hvis du ikke har 64-bit Java virker ikke dette, maks er vel da 1000m du kan be om. Vennligst last ned 64-bit Java 8 til din maskin fra:

http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html)

Vedlegg: Kode til innstikksortering (a) hele a[0.. *k-1*];

a) (N.B. Denne sorterer i s**tigende** rekkefølge – du må selv skrive den om og la den sortere i **synkende** rekkefølge):

```
/** Denne sorterer a[v..h] i stigende rekkefølge  med innstikk-algoritmen*/
  void insertSort (int [] a, int v, int h) {
            int i , t ;

            for ( int k = v ; k < h ; k++) {
                  // invariant: a[v..k] er nå sortert stigende (minste først)
                    t = a [k+1] ;
                    i = k ;
                    while ( i >= v && a [i] > t ) {
                              a [i+1] = a [i];
                              i--;
                    }
                    a[i+1] = t;
           } // end for k
 } // end insertSort
```

b) N.B. for å sortere inn et nytt element på plass a[*k-1*] trenger du en enklere versjon av koden over fordi de første k -1 elementene a[0.. *k-2*] står jo allerede i synkende, sortert rekkefølge.