Kompilering og Kjøring

Kommando for å kompilere programmet:

```
javac Oblig3.java
```

Kjør programmet slik:

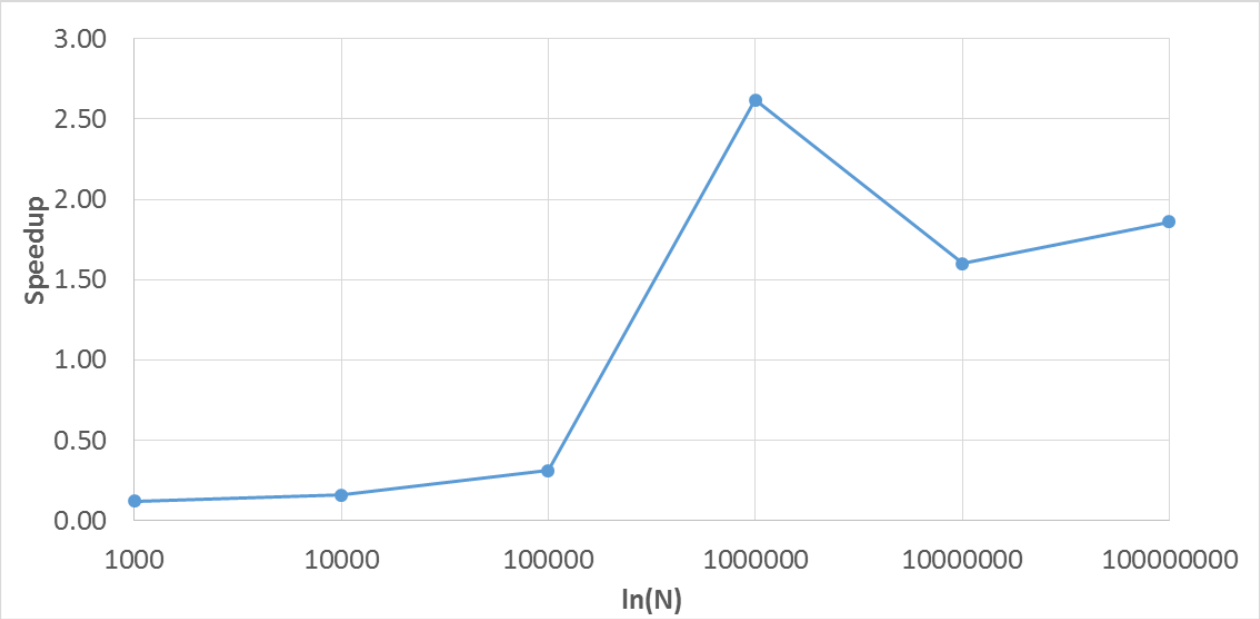
```
java Oblig2 <N> <#TESTS>
```

hvor `<N>` er antall elementer du ønsker å sortere, og `<#TESTS>` (som burde være et positivt oddetall) er antall ganger testen får kjøre. Du trenger ikke å specifisere noen for `<#TESTS>` hvis du ønsker at antall kjøringer blir valgt automatisk.

Resultat

Jeg har testet programmet mitt på både datamaskinen min (som har bare 4 tråder) og på IFIs server (safir.ifi.uio.no, som har 64 tråder). Selv om jeg har fått en veldig bra speedup på IFIs server (speedup ligger mellom 3 og 9), men det som er rart er at både den sekvensielle og parallelle versjonen kjør mange ganger fortere på min egen datamaskin enn på IFIs server. Dette skjønner jeg ikke helt hvorfor. I hvert fall, skal jeg bruke resultat fra datamaskinen min i denne rapporten.

N	ln(N)	Median Tid (ms)		Speedup
		Seq	Par	
100000000	18.42	1334.51	718.81	1.86
10000000	16.12	128.36	80.24	1.60
1000000	13.82	19.69	7.52	2.62
100000	11.51	1.85	6.03	0.31
10000	9.21	0.34	2.09	0.16
1000	6.91	0.11	0.91	0.12



Parallelisering av Radix Sort

Sekvensielle algoritmen for å radix sortere arrayen `a[]` er slik:

- **Steg A**: Finn det største tallet i `a[]`
- **Steg B**: Tell hvor mange det er av hvert sifferverdi og skriv dette i `count[]`
- **Steg C**: Akkumulerer verdiene i `count[]` slik at vi får "pekere" til `b[]`
- **Steg D**: Flytt tallene fra `a[]` til `b[]` basert på de pekere vi fikk fra steg C
- Gjenta steg B-D for alle sifrene

Mitt første forsøk (som ikke fungerte)

For mitt første forsøk, har jeg prøvd å parallellisere den sekvensielle algoritmen steg for steg slik at hele programmet ser ut som en "pipeline". Det vil si at et steg skal parallelliseres slik at det gir et felles output på slutten, og dette output skal da sendes videre til neste steg. Parallelliseringen ser ut slik:

- **Steg A**: La hver tråd finne `localMax` i sin del av `a[]`, og derreter oppdatere `globalMax` synkronisert.
- **Steg B**: La hver tråd telle (i sin del av `a[]`) hvor mange det er av hvert sifferverdi og skrive denne informasjonen inn i en global variabel `allCount[][]`. Når alle trådene er ferdig med dette, skal vi dele antall kolonner i `allCount[][]` jevnt mellom trådene, og la hver tråd (i sin del av `allCount[][]`) summére opp verdiene i hver kollone. Output fra dette steget er en `sumCount[]` array, og denne arrayen skal sendes videre til steg C.
- **Steg C**: Hver tråd skal akkumulere verdiene i sin del av `sumCount[]`, hvor den siste akkumulerte verdien blir offset til neste tråd. Nå la bare den første tråd akkumulere offset verdiene. Og derretter skal alle trådene gå over sin del av `sumCount[]` en gang til for å legge til offset til hver verdi i `sumCount[]`. Output fra dette steget er den oppdatert `sumCount[]` arrayen med riktige akkumulerte verdiene, og denne arrayen skal sendes videre til steg D.
- **Steg D**: ???????

Jeg fant et problem når jeg prøvde å parallellisere det siste steget. Sekvensielle algoritmen for steg D sier at vi må gå gjennom `a[]` helt fra venstre til høyre slik at et tall T_1 (som står før et annet tall T_2 i `a[]`) blir flyttet først, innen vi kan flytte T_2. Ellers skal vi få galt resultat. Men hvis vi parallelliserer steg D slik at hver tråd eier en del av a[], da har vi ingen garanti at T_1 vil bli flyttet først (spesiellt når T_1 og T_2 tilhører to forskjellige tråder).

Mitt andre forsøk (som fungerer)

Jeg synes at problemet med mitt første forsøk er at når jeg sendte den akkumulerte `sumCount[]` fra steg C til steg D, har jeg mistet noen nyttig informasjon. La oss si at den første tråden har funnet 5 tall i *sin del* av `a[]` som har sifferverdien *S*. Men vi har ikke tilgang til denne informasjonen i steg D, fordi `sumCount[]` lagrer bare antall tall som har sifferverdien *S* i *hele* `a[]` arrayen. Uten denne informasjonen synes jeg at det er umulig å parallelisere steg D.

Dette leder meg til å ta bort `sumCount[]` fra koden, og bruke `allCount[][]` direkte istedet. Tanken er at etter steg C, vil `allCount[][]` inneholde riktige akkumulerte verdiene som representerer "pekere" til `b[]`. Og da blir `allCount[][]` sendt videre til steg D. Denne paralleliseringen fungerer veldig fint. Her er hvordan algoritmen ser ut:

- **Steg A** → Finn max verdi: La hver tråd finne en `localMax` i sin del av `a[]`, og derreter bruke en synkronisert metode `updateMax()` til å oppdatere en felles max. Til slutt, synk med de andre trådene ved bruk av en cyclic barrier.

- *Et mellomsteg*: Alle trådene skal utføre en samme algoritme for å dele opp bittene jevnt mellom sifrene, basert på hvor mange bits den `max` verdien har, samt med hvor mange sifre vi skal sortere på.

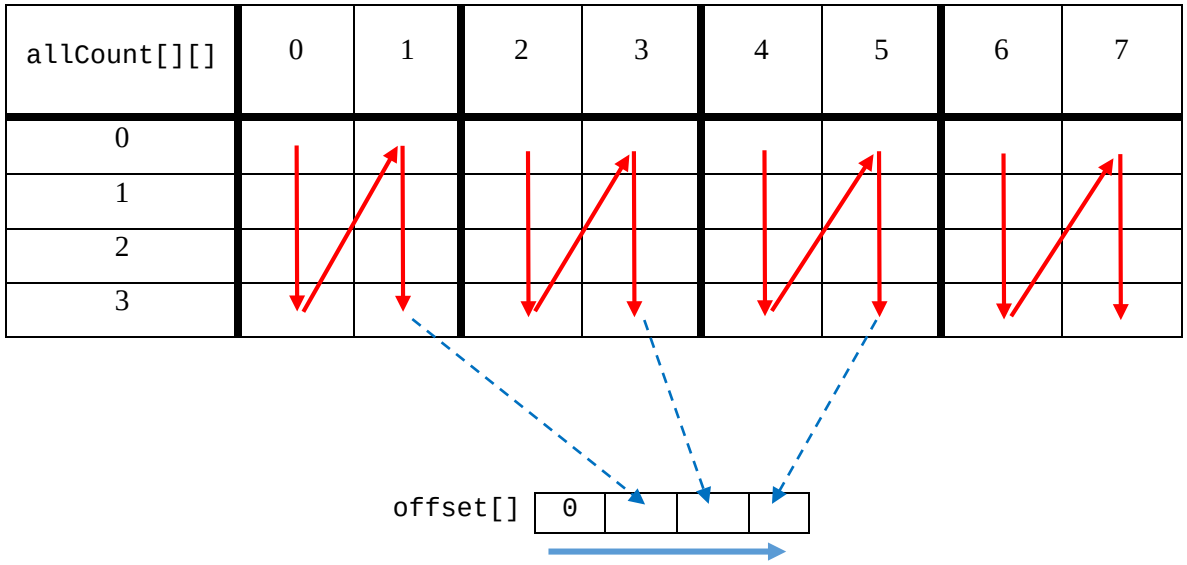
- **Steg B** → La hver tråd telle opp hvor mange det er av hvert sifferverdi i sin del av `a[]`:
  - Hver tråd skal lagre resultatene i en lokal `count[]`, og deretter la den globale tabellen `allCount[]` peke til `count[]` slik at andre trådene kan også se på resultatene.
  - Synk ved bruk av cyclic barrier.

- **Steg C** → Oppdatere `allCount[][]` slik at denne tabellen skal få riktige akkumulerte verdiene på slutten:
  - **C1**: La hver tråd akkumulere `allCount[][]` i sin del slik at den siste akkumulerte verdien blir offset til den neste tråden. Når dette er ferdig, synk ved bruk av cyclic barrier.
  - **C2**: La bare den første tråden akkumulere verdiene i `offset[]` arrayen, mens andre trådene går direkte til neste synkronisering barrier. Dette trenger vi ikke å parallellisere fordi `offset[]` er bare en liten array (med `nthreads` plasser), og da går det fortere sekvensielt.
  - **C3**: La hver tråd legge til offset til hver verdi i sin del av `allCount[][]`. Synk med andre trådene når ferdig.

- **Steg D** → Hver tråd skal få *én og bare én* rad i tabellen `allCount[][]` (tråd-0 får rad-0, tråd-1 får rad-1, osv…). Hver tråd skal da gå gjennom sin del av `a[]` fra venstre til høyre, og flytte tallene fra `a[]` til `b[]` basert på innholdet i raden som tråden har fått. Synk med andre trådene når alt er ferdig.

➔ Antall synkroniseringer (ved bruk av en cyclic barrier) vi må gjøre i steg A er bare 1; mens for steg B-D, trenger vi å synkronisere alle trådene 5 ganger. I tillegg må vi gjenta steg B-D for alle sifrene. For eksempel, hvis vi har n_s sifre, så er antall synkroniseringer lik $1 + 5n_s$. Dette betyr at en mindre n_s gir et mindre tidsbruk siden vi har færre synkroniseringer for å gjøre. Men vi må også huske at en mindre n_s leder til mer bits per siffer, og dette kan senke effektiviteten. Da er det best å kunne regne ut optimale verdier for n_s og antall bits per siffer slik at vi får en bra trade-off. Men dette er veldig vanskelig å si fordi det avhenger av hvilken array `a[]` vi får.

Jeg har tegnet en liten illustrasjon over hvordan steg C ser ut. Her antar jeg at det er 3 bits i siffret (som da gir $2^3 = 8$ mulige sifferverdier), og at det er 4 tråder i datamaskinen. Med andre ord, `allCount[][]` skal være en tabell med 8 kolonner og 4 rader, hvor hver tråd tar 2 av de kolonnene.



Akkumulering av `allCount[][]` (første delen av steg C1)

Siste akkumulerte verdien blir offset til neste tråd (andre delen av steg C1)

Akkumulering av offset[] (steg C2)