Kompilering og Kjøring

Kommando for å kompilere programmet:

```
javac Oblig2.java
```

Kjør programmet slik:

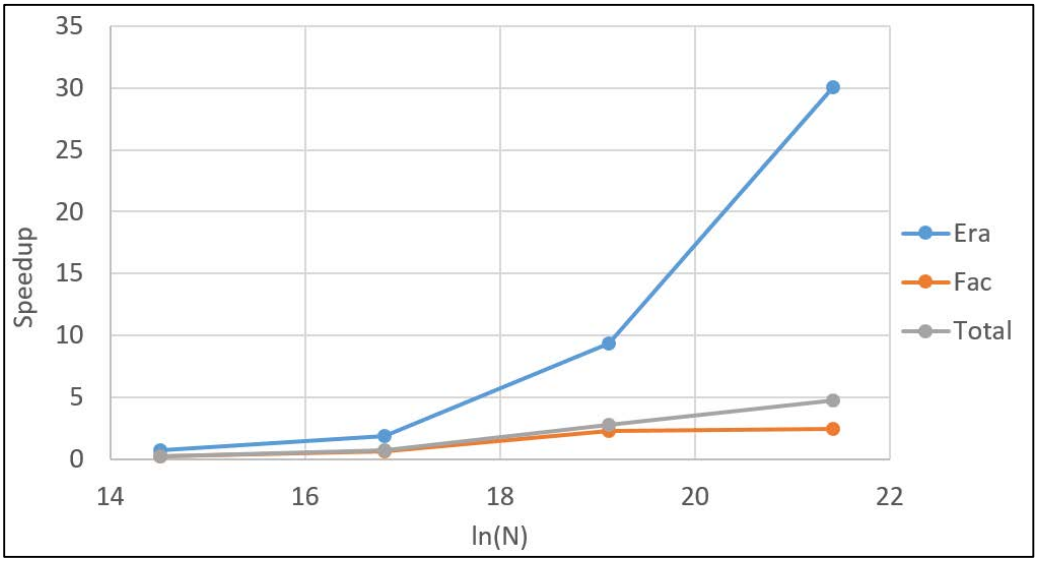
```
java Oblig2 <N> <#threads>
```

hvor `#threads` må være enten 0 eller et positivt tall. Hvis `#threads==0`, så er antall tråder lik antall kjerner som er tilgjengelige i datamaskinen.

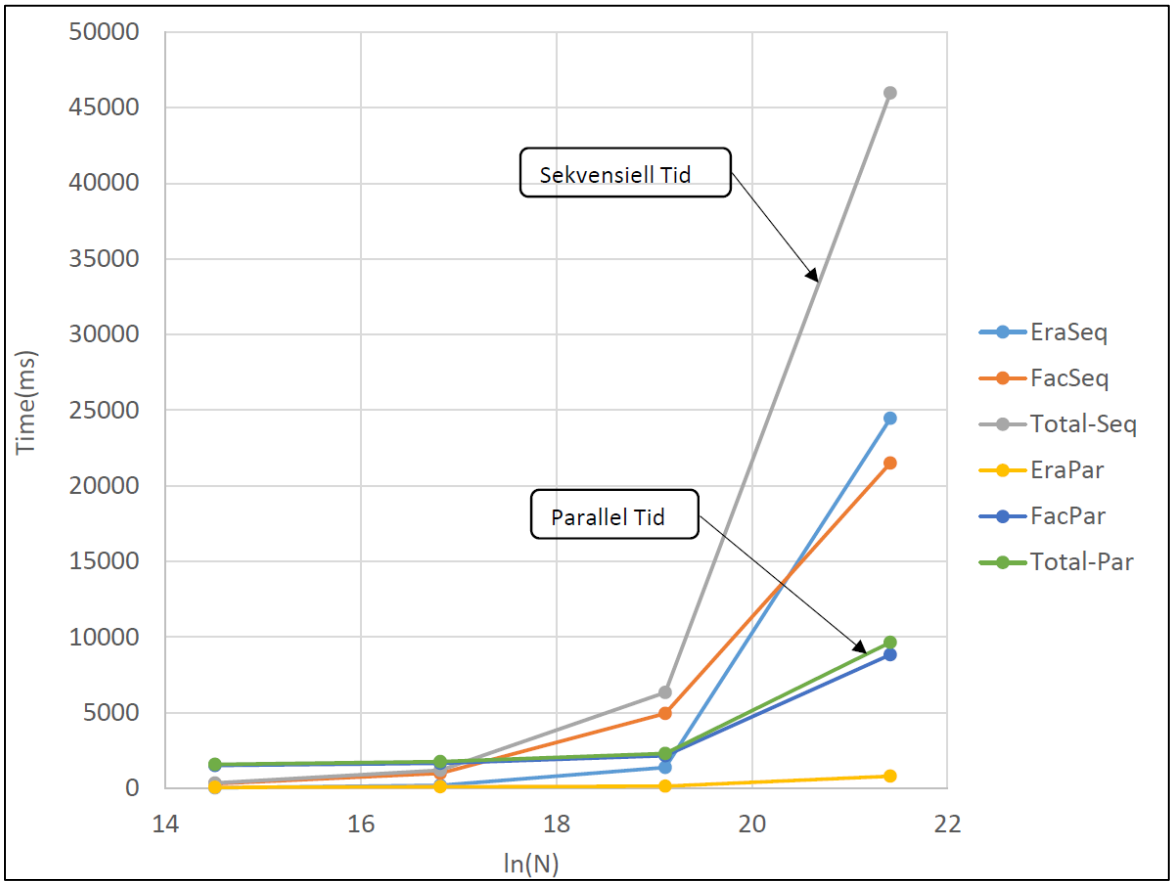
Resultat

Jeg har testet programmet mitt på IFIs server (safir.ifi.uio.no), og denne serveren har 64 tråder tilgjengelig.

N	ln(N)	Algorithm	Seq (ms)	Para (ms)	Speedup
2000000	14.51	ERA	47.99	70.38	0.68
		FAC	316.20	1522.93	0.21
		Total	364.19	1593.30	0.23
20000000	16.81	ERA	205.81	110.16	1.87
		FAC	1005.28	1662.13	0.60
		Total	1211.08	1772.29	0.68
200000000	19.11	ERA	1384.11	147.93	9.36
		FAC	4958.00	2175.86	2.28
		Total	6342.11	2323.79	2.73
2000000000	21.42	ERA	24460.36	813.64	30.06
		FAC	21510.77	8844.14	2.43
		Total	45971.13	9657.78	4.76



Graf 1: Speedup vs. $\ln(N)$



Graf 2: Tid vs. $\ln(N)$

Fra *Graf* 2, kan vi se at tidene for å utføre den sekvensielle og den parallelle løsningen er omtrent den samme for små N. Men når N blir stor så er det en stor forskjell mellom tidene. Dette vises også i *Graf* 1, hvor speedup øker fort når N blir større og større. Selv om *Graf* 2 ser ut som en eksponentiell graf, legg merke til at x-aksen beskriver $\ln(N)$, og ikke N. Det betyr at tidene vokser nesten linært med N.

Fra *Graf* 1 kan vi se at speedup for Eratosthenes Sil (Era) vokser mange ganger fortere enn speedup for faktorisering (Fac) algoritmen. Dette vil si at parallelliseringen av Era algoritmen er mer effektivt enn parallelliseringen av Fac algoritmen. En mulig forklaring til dette er fordi den sekvensielle algoritmen for faktorisering (dvs. `facSeq()` metode) har allerede blitt optimalisert på en god måte, slik at parallelliseringen av denne algoritmen ikke spiller en stor rolle på kjøretidene.

Fra tabellen ovenfor, merker vi at total speedup er > 1 når N er på størrelseorden 10^8. Men hvis vi ønsker å utføre bare avkryssing algoritme (dvs. bare EraSil, uten noen faktorisering), så er speedup enda bedre fordi vi får speedup > 1 fra Era algoritmen når N er bare på størrelseorden 10^7

Parallelisering av Era og Fac algoritmer

Måten som jeg har valgt til å parallellisere Era algoritmen er å lage en liten tabell for alle primtall $\leq \sqrt{N}$. Dette kan gjøres sekvensiellt fordi det ikke tar så mye tid (hvis $N = 2 \times 10^9$, så er $\sqrt{N} = 44721$, og dette ikke er et stort tall). Nå kan vi dele opp resten av odde-tallinjen (dvs. alle odde tall mellom $\sqrt{N}$ og N) jevnt mellom trådene, og la dem avkrysse tall i hver sin del (ved hjelp av primtallene i den lille tabellen). En viktig ting å tenke på når vi synkroniserer trådene er at enhver byte i byteArr kan bare krysses av en og bare en tråd, dvs. vi må dele opp odde-tallinjen slik at 2 tråder ikke kan avkrysse tall i samme byte.

Faktorisering algoritmen kan parallellisere ved å dele opp antall primtall $\leq N$ jevnt mellom trådene. Dette er en bra måte å gjøre parallelliseringen fordi det tar hensyn til last-balansering. Problemet er at det er veldig vanskelig å implementere dette fordi vi trenger da en liste over all primtallene, men byte-arrayen vår inneholder informasjon for alle odde tallene, ikke bare de som er primtall. Vi må da lage en `int`-array over alle primtall $\leq N$ først, før vi kan faktisk gjøre noen faktorisering. Dette ikke bare øker kjøretidene, men krever også mye minne fra datamaskinen.

En bedre og enklere måte å parallellisere faktorisering er å dele opp odde-talllinje jevnt mellom trådene, og la hver tråd bruke bare de primtallene i trådens del. For eksempel, hvis vi har 64 tråder, da kan tråd-0 ta alle primtall $\leq N/64$, og tråd-1 ta de neste primtallene p (hvor $N/64 < p \leq 2 \times N/64$), og så videre. Merk at på denne måten, skal trådene ikke få samme arbeidsmengde fordi "de primtallene $\leq N$ er ganske ujevnt fordelt, med ca. 10% i den laveste delen og $< 4\%$ i den øverste delen" (foilene uke 7, side 15[1]). Men selv om denne måten ikke tar hensyn til last-balansering, får vi forsatt en bra speedup. I tillegg er denne oppdelingen enkel å implementere og ikke krever mye minne fra datamaskinen.

[1] http://www.uio.no/studier/emner/matnat/ifi/INF2440/v16/forelesningene/uke7-2016.pdf