Oblig2ver2, INF2440 v2016, frist: 11. mars

I forelesningene Uke5 og Uke6 samt i ukeoppgavene blir oppgaven med å lage en sekvensiell versjon av det først det å lage og lagre primtall, og så kunne faktorisere store tall (long-variable) beskrevet og løst.

I Oblig 2 skal du så greie å lage sekvensiell kode for, og så parallellisere disse to algoritmene, det å generere alle primtall < N med en teknikk som er hentet fra jernalderen, fra en gresk matematiker som het Eratosthenes (ca. 200 f.k.) og så faktorisere alle tall M < N*N med disse primtallene. Metodene for å lagre og lage store primtall blir presentert i forelesningene i Uke5 og faktorisering i uke 6 og 7 (se på lysarkene fra forelesningene når de kommer), og du kan også lese om den på Wikipedia.no (se: http://no.wikipedia.org/wiki/Eratosthenes'_sil) hvor metoden også er visualisert. Eratosthenes sil nyttes fordi den faktisk er den raskeste. Det eneste avviket vi gjør fra slik den er beskrevet i Wikipedia er flg:

1. Vi representerer ikke partallene på den tallinja som det krysses av på fordi vi vet at 2 er et primtall (det første) og at alle andre partall er ikke-primtall.
2. Har vi funnet et nytt primtall p, for eksempel. 5, starter slik det nå står på Wikipedia (oppdatert av meg feb. 2015) vi avkryssingen for dette primtallet, først for tallet p*p (i eksempelet: 25), men etter det krysses det av for p*p+2p, p*p+4p,.. (i eksempelet 35,45,55,…osv.). Grunnen til at vi kan starte på p*p er at alle andre tall < p*p som det krysses av i for eksempel Wikipedia-artikkelen har allerede blitt krysset av andre primtall < p. Det betyr at for å krysse av og finne alle primtall < N , behøver vi bare og krysse av på denne måten for alle primtall p ≤ sqrt(N). Dette sparer *svært mye* tid.
3. Avkryssingen skal gjøres blant bit-ene i en byte-array hvor hvert element inneholder 7 eller 8 bit som gjennomgått på forelesningen Uke 6.

Sammen med ukeoppgavene ligger det en .java-fil: EratosthenesSil.java, som inneholder skjelettet til en klasse som du kan nytte til å implementere en sekvensiell versjon av Eratosthenes sil. Selvsagt står du helt fritt til å implementere den på en annen måte hvis du vil det, men husk da at du skal ha plass til å ha representert alle primtall < 2 milliarder i den, og at ca 5-10% av alle tall er primtall (mer eksakt : det er omlag $\frac{N}{\ln N - 1}$ primtall < N) .

Oblig2: Du skal lage de to algoritmene : Eratosthenes Sil og Faktorisering i en sekvensiell og en parallell versjon. Programmet ditt skal ta en fritt valgt N (>16) som inndata, og så generere først Eratosthenes Sil av alle primtall <N og så faktorisere og skrive ut de 100 siste tallene t < N*N sekvensielt og ta tidene på det. Deretter skal du 'fjerne' den sekvensielt genererte Eratosthenes Sil, og så generere den om igjen parallelt og så parallelt faktorisere og så igjen skrive ut de 100 siste tallene t < N*N. Når du parallelliserer faktoringen av de 100 tallene er da **IKKE** lov å bare dele disse 100 tallene blant de k trådene og så ta å faktorisere hvert tall sekvensielt i hver tråd. Grunnen til dette er at da har faktoriseringa av ett tall ikke blitt parallellisert, bare summen for tiden av de 100. Faktoriseringa av ett tall vil da ikke gå fortere.

Det dere skal gjøre er å parallellisere faktorisering av hvert tall og så bruke denne parallelliseringa etter tur på de 100 tallene.

Du skal teste både den sekvensielle og parallelle versjonen din av Eratosthenes Sil med å generere alle primtall < 100, og manuelt sjekke at det gikk bra og skrive om det i rapporten. I rapporten din skal du også omtale etter hvilke prinsipper/oppdelinger du brukte da du parallelliserte de to sekvensielle algoritmene.

Uryddig kode uten f.eks innrykk for koden i hver metode vil ikke bli godkjent, og det programmet du lager skal i én kjøring produsere alle tallene det spørres om i oppgaven. Bruk gjerne Modell3 koden som et utgangspunkt for løsningen din.

Du skal rapporten skrive separat speedup for de to algoritmene har laget og hvor du også samlet beregner speedup for den sekvensielle løsningen mot den parallelle. Du skal bruke N=2mill., 20 mill., 200 mill. og 2 milliarder i de kjøringene du leverer inn. Kommentér i rapporten også spesielt om hvordan en 10-doblig av problemets størrelse gir utslag kjøretidene – mer eller mindre enn 10-dobling i kjøretidene? Finner du et mønster? Til innleveringen skal du bare ta med de 5 første og de 5 siste av de 100 faktoriseringene du gjør for hver N sekvensielt og parallelt (men tidene du beregner skal selvsagt være snittet for 100 faktoriseringer for hver gang).

Du kan som sagt godt nytte modell3 eller modell2 -koden som et utgangspunkt for testkjøringene dine, men siden det tar såpass lang tid å generere alle primtall < 2 milliarder (se fasit) og faktorisere 100 stk. 19-sifrete tall, behøver du ikke å lage median av mer enn ett gjennomløp.

Som fasit for faktorieseringen kan du se hva forelesers sekvensielle løsning genererte.

Krav til innleveringen

Oppgaven leveres i Devilry innen kl. 23.59 den11. mars. Det skal innleveres en rapport på .txt eller .pdf – format og en enkelt java fil: Oblig2.java som inneholder alle klassene for både den sekvensielle og parallelle løsningen. Helt i starten av rapporten skal være to linjer om hvordan programmet kompileres og kjøres i et terminalvindu.

Fasit

For å sjekke om du har skrevet et riktig program burde utskriften din se omlag slik ut for N= 2 milliarder (alle tider for den sekvensielle løsningen inntil 2-3 ganger dette er klart akseptabelt – men du må gjerne også lage raskere kode enn foreleseren).

Som du ser er det bare de to første og de 11 siste av de 100 siste tallene t <N*N som er med her. Dette programmet regnet også ut antall primtall (= antall 1-er-bit i bit-arrayen), men det gjør du bare dersom du har tid (ikke krevet).

```
M:>java Oblig2 2000000000
 max primtall m:2000000000
Genererte alle primtall <= 2000000000  paa:     14902.45 millisek
 med Eratosthenes sil ( 0.00015172 millisek/primtall)
 antall primtall < 2000000000 er: 98222287, dvs:   4.91% ,

3999999999999999900 = 2*2*3*5*5*89*1447*1553*66666667
3999999999999999901 = 19*2897*72670457642207
……

3999999999999999989 = 7*89*503*12764504465981
3999999999999999990 = 2*3*5*170809*780598992637
3999999999999999991 = 17*53*211*233*2393*37735849
3999999999999999992 = 2*2*2*223*208513*10753058401
3999999999999999993 = 3*139*4024357*2383567397
3999999999999999994 = 2*112957699*17705743103
3999999999999999995 = 5*159059*303539*16569799
```

```
3999999999999999996 = 2*2*3*3*3*3*7*11*13*19*37*52579*333667
3999999999999999997 = 421*9501187648456057
3999999999999999998 = 2*432809599*4620969601
3999999999999999999 = 3*31*64516129*666666667

100 faktoriseringer med utskrift beregnet paa:   26965.5125ms
 dvs:    269.6551ms. per faktorisering
```