IN3030 Uke 8, v2021

Eric Jul

PT

Inst. for informatikk

Hvad har vi set på i uke 7

- Om primtall – Eratosthenes Sil (ES)
- Hvordan representere (ES) effektivt i maskinen?
- Faktorisering av tall
- Java Measurement Harness
- Oblig3: Primtall, Erastosthenes Sil, faktorisering

Hva skal vi se på i uke 8

- Oblig frister
- Review Primtall
- Hvilken orden O() har Eratosthenes Sil ?
- Alternativer for å parallellisere: Eratosthenes Sil
- Generelt om last-balansering ved parallellisering med eksempel.
- Hvordan parallellisere rekursive algoritmer
- Gå ikke i 'direkte oversettelses-fella'
  - eksemplifisert ved Kvikk-sort, 3 ulike løsninger
- Hvor lang tid tar de ulike mekanismene vi har i Java 6 og 8?

2 måter å lage primtall

Ønsker at finne alle primtal $p_i < N$

- Lage en tabell over alle de primtallene vi trenger
  - Eratosthene sil

- Dividere alle tall < N med alle oddetall <= $\sqrt{N}$?
  - Divisjonsmetoden
  - (Hvorfor ikke oddetall opp til N?)

Sitat fra denne boka (fra 2005) om hvor lang tid det tar å finne ut om et 19-sifret tall er primtall ved divisjon.
Denne boka hevder 1 døgn, vi skriver program som gjør det på ca. 1-3 sek – eller : 24*60*60 = 86 400 x fortere

Og selv om boka deler med all oddetall $<\sqrt{N}$ og ikke bare primtallene $<\sqrt{N}$, skulle det bare gå ca. 10x langsommere (fordi ca 5-10% av alle oddetall $<\sqrt{N}$, er primtall).

Prime Numbers

A Computational Perspective

Second Edition

Richard Crandall
Carl Pomerance

3.1.3 Practical considerations

It is perfectly reasonable to use trial division as a primality test when n is not too large. Of course, "too large" is a subjective quality; such judgment depends on the speed of the computing equipment and how much time you are willing to allow a computer to run. It also makes a difference whether there is just the occasional number you are interested in, as opposed to the possibility of calling trial division repeatedly as a subroutine in another algorithm. On a modern workstation, and very roughly speaking, numbers that can be proved prime via trial division in one minute do not exceed 13 decimal digits. In one day of current workstation time, perhaps a 19-digit number can be resolved. (Although these sorts of rules of thumb scale, naturally, according to machine performance in any given era.)

Minner om at vi kan greie å faktorisere 19-sifrete tall på < 0.3 sek. sekvensielt !
IKKE på en dag !

AntTraader:16, AntKjerner:8 , tider i millisec for 100 elementer

	n	sekv	para	Speedup	sekv/elm	para/elm
EratosthesSil	2000000000	6626.54	4618.58	1.43		
Faktorisering	2000000000	18906.73	10973.91	1.72	189.0673	109.7391
Total tid	2000000000	**25533.31**	15568.38	1.64		
EratosthesSil	200000000	419.42	186.86	2.24		
Faktorisering	200000000	4621.13	2572.63	1.80	46.2113	25.7263
Total tid	200000000	5077.76	2745.52	1.85		
EratosthesSil	20000000	22.90	17.78	1.29		
Faktorisering	20000000	496.74	335.88	1.48	4.9674	3.3588
Total tid	20000000	519.54	353.66	1.47		
EratosthesSil	2000000	1.97	2.01	0.98		
Faktorisering	2000000	77.35	70.68	1.09	0.7735	0.7068
Total tid	2000000	79.37	72.54	1.09		
EratosthesSil	200000	0.23	0.21	1.09		
Faktorisering	200000	8.32	18.93	0.44	0.0832	0.1893
Total tid	200000	8.54	19.14	0.45		

Vi finner også store primtall

```
para: 3999999999999999972 = 2*2*3*199*1675041876046 9
```

Vise at vi trenger bare primtallene <10 for å finne alle primtall < 100, avkryssing for 3 (3*3, 9+2*3,9+4*3, ...)

1	**3**	**5**	**7**	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75	77	79
81	83	85	87	89
91	93	95	97	99

1	**3**	**5**	**7**	**9**
11	13	**15**	17	19
21	23	25	**27**	29
31	**33**	35	37	**39**
41	43	**45**	47	49
51	53	55	**57**	59
61	**63**	65	67	**69**
71	73	**75**	77	79
81	83	85	**87**	89
91	**93**	95	97	**99**

Avkryssing for 5 (starter med 25, så 25+2*5, 25+4,5,..):

1	**3**	**5**	**7**	**9**
11	13	**15**	17	19
21	23	25	**27**	29
31	**33**	35	37	**39**
41	43	**45**	47	49
51	53	55	**57**	59
61	**63**	65	67	**69**
71	73	**75**	77	79
81	83	85	**87**	89
91	**93**	95	97	**99**

1	**3**	**5**	**7**	**9**
11	13	**15**	17	19
21	23	**25**	**27**	29
31	**33**	**35**	37	**39**
41	43	**45 45**	47	49
51	53	**55**	**57**	59
61	**63**	**65**	67	**69**
71	73	**75 75**	77	79
81	83	**85**	**87**	89
91	**93**	**95**	97	**99**

Avkryssing for 7 (starter med 49, så 49+2*7,49+4*7,.):

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45 45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75 75	77	79
81	83	85	87	89
91	93	95	97	99

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45 45	47	49
51	53	55	57	59
61	63 63	65	67	69
71	73	75 75	77	79
81	83	85	87	89
91	93	95	97	99

Er nå ferdig fordi neste primtall vi finner: 11, så er 11*11=121 utenfor tabellen

Hvilken orden har Eratosthnes Sil – algoritmen (N)

- Vi krysser av for alle primtall < $\sqrt{N}$ - hvor mange er det?
  - svar: omlag: $\frac{\sqrt{N}}{\log(\sqrt{N})}$
- Hvor mange kryss settes av p_i mellom p_i^2 og N ?
  - svar: omlag $\frac{N-pi^2}{2}$ / (2* p_i)< $\frac{N}{4}$
- Et øvre estimat for (antall primtall)* (antall kryss per primtall)=
  - O(N) = $\frac{\sqrt{N}}{\log(\sqrt{N})} * \frac{N}{4} = \frac{N\sqrt{N}}{\log(\sqrt{N})} = \frac{2*N\sqrt{N}}{\log(N)} = \frac{\boldsymbol{N\sqrt{N}}}{\boldsymbol{log}(\boldsymbol{N})}$

Om å parallelliser Sil og Faktorisering, **Riktig** og **Raskere**

- Eratosthenes sil - parallellisering
  - Vi krysser av i en bit-tabell.
  - Hvordan gjøre dette i parallell
    - Hva det er vi deler opp, hva gjør hver tråd?
    - Skal vi kopiere noen data til hver tråd?
    - Hva er felles data og hva er lokale data i hver tråd.
- Faktorisering av M - parallellisering
  - Vi dividerer M med alle primtall i E-Silen fra 2:$\sqrt{M}$
    - Parallelliseringen av faktorisering: Hvordan dele opp :
      - Primtallene?
      - Tallene fra 0:M?
    - Skal vi kopiere noen data til hver tråd?
    - Hva er felles data og hva er lokale data i hver tråd.

Riktig og Raskere – Eratosthenes Sil

- Mulig problem
  - E-sil : Hvis to tråder setter av kryss samtidig i en byte, kan det da gå galt?
    - byte[i] = byte[i] & xxxx; // problem ?
    - Hvordan blir dette utført i kjernen
    - Må vel da synkronisere ?

- Raskere:
  - Antall synkroniseringer må ikke bli for stort
    - Antall tråder ganger (evt * 2,3,4) : Helt OK
    - Log (N) ganger, evt log(N)*ant tråder: OK,
    - $\sqrt{N}$ tja ? (hvor stor %-andel er $\sqrt{N}$ av N, N=100, N=10 000)
    - **ikke** :N = $\sqrt{M}$ eller M ganger

Alternativ 1 for Eratosthenes Sil

- Dele opp primtallene mellom trådene så hver tråd tar noen hver og krysser av i hele Sila:
  - Ulempe: 3 og 5 genererer langt de fleste kryssene (lastbalansering)
    - Kunne la $tråd_0$ ta 3, $tråd_1$ ta 5 , osv
  - Problem: to ulike primtall krysser av ulike steder i ***samme*** byte – kan da vel miste ett av kryssene (eks: 3 kryssser 51, 7 krysser 49) Avkryssing er 3 opersjoner: Last opp i register , &-operasjon, lagre tilbake
  - To primtall krysser av for det samme tallet (eks. 7 og 5 krysser av 105) – da taper vi vel (?) ingen ting om ett av kryssene går tapt.
- Hvis alle trådene har sin kopi av Sila, skulle dette gå OK.
  - Må da tilslutt 'slå sammen' disse k stk byte-arrayene .
  - Tiden det tar å slå sammen disse vil kanskje bli 'lang' ??

Alternativ 2 for Eratosthenes Sil:

- Dele opp tallene i Sila : tallene 0:N mellom trådene,
  - Kan ikke dele helt likt – skillet må gå mellom to byter
    - Hvorfor (?)
- Hver tråd krysser av med alle aktuelle primtall på sin del av Sila.
- Problem 1: Hvor starter avkryssingen av primtall p i hver tråds (første) byte.
- Problem 2: Hvordan vet trådene som har området: *low..high* hvilke primtall den skal krysse av for??
- Hvilke er det ? Svar : alle primtall < $\sqrt{N}$ eller $\sqrt{high}$
- Hvordan lage og lagre dem først?
  - Tre alternativer:
    - $Tråd_0$ lager dem først (sekvensiell del – jfr. Amdahls lov)
    - Alle trådene lager de samme primtallene < $\sqrt{N}$ i en lokal kopi
    - Se på dette rekursivt som et problem som kan parallelliseres (dvs . Sekvensiell fase : Først alle primtall < $\sqrt{\sqrt{N}}$ som så lager primtall < $\sqrt{N}$)

Grep 2 – vi lager noen ekstra data

- I FinnMax-problemet hadde vi først:
  - en lokal variabel: lokalMax som hver tråd lokalt oppdaterte til de var ferdige; og så kalte hver tråd en global synkronisert metode som evt. satte ny verdi i globalMax
- Kan vi prøve noe tilsvarende her. I løsning a) trenger alle trådene primtallene < $\sqrt{N}$.
- Skisse - enten:

  a1) Tråd-0 lager disse primtallene først i tabellen mens de andre trådene venter, eller
  a2) Alle trådene lager hver sin lokale tabell over disse primtallene

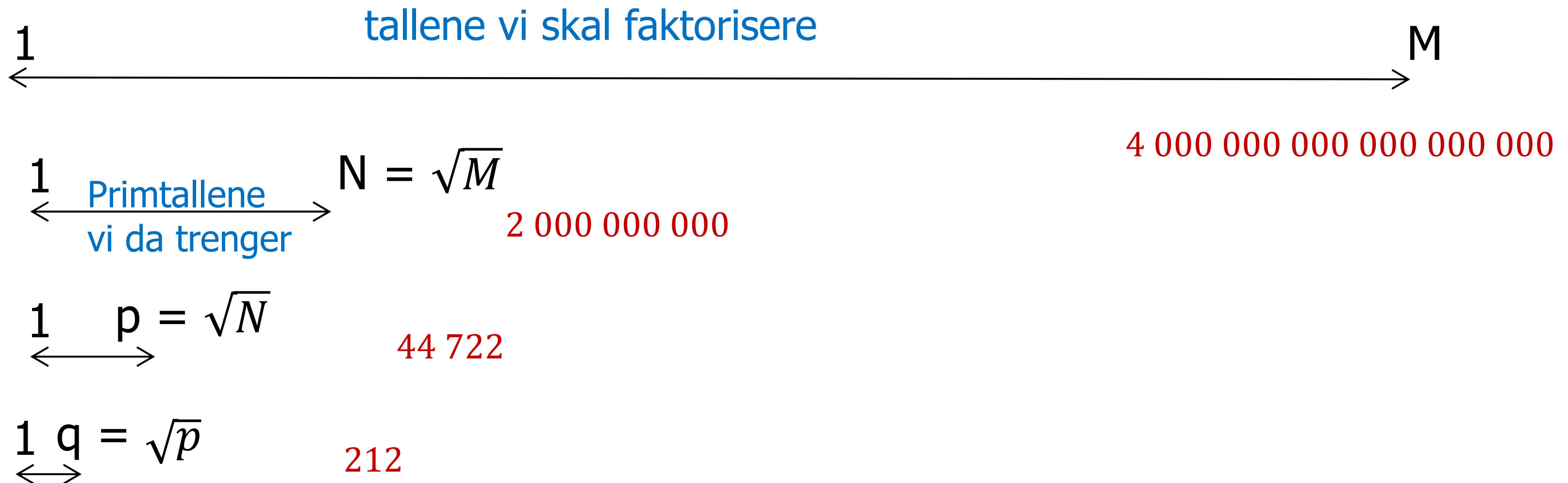
  a3) Enten a1) eller a2), så krysser hver tråd deretter av i sin del av bit-arrayen for alle disse primtallene.

  N.B. Hvis a1) skal tråd-0 bare bruke disse 'små' primtallene til først å krysse av i området opp til $\sqrt{N}$, ikke i hele sitt område.

Billig å lage en slik liten start-tabell for trådene i alt. a)

- Eks: skal vi parallelt krysse av for primtall < 2 milliarder, trenger denne tabellen plass til $\sqrt{2\ mrd} =$ 44722 bit/16= 2796 byter for å finne de primtallene p < 44722.

(og i den lille tabellen trenger vi bare krysse av med primtall q < $\sqrt{44722}$ = 212 for å finne dem. Billig start på å parallellisere det å finne alle primtall < 2000 000 000).



Esil: Hvordan oppbevarer du den lille tabellen

- a) Som en bit-array i en byte-array
  = lengde = 44 722/16 = 2 796 byte
- b) Som en int array = ca. 44 722*4*$\frac{10}{100}$ = 17 890 byte
  (med ca. 10 % primtall < 44 722)

  - Ingen av plassbehovene er store, kan godt kopieres inn lokalt i hver tråd om nødvendig

Parallellisering av Faktoriseringa -

Vi må huske at vi (i Oblig3) ikke skal faktorisere ***alle*** tall < M, ‘bare 100 stk, og faktoriseringen av hvert tall skal parallelliseres

1. Det store tallområdet 0:M deles likt mellom trådenere
2. Primtallene i 0:N deles likt mellom trådene
3. Felles data ?

Eks. 2) Hvordan dele opp faktoriseringa av store tall M ≤ N*N

Vi har nå en Erothostenes Sil over primtallene ≤ N = 2 mrd.

Hvordan da faktorisere i parallell de 100 største tallene:

- ? Del opp de 100 tallene slik at tråd-0 tar de 100/k første ,tråd-1 de 100/k neste,..osv. og så sekvensiell faktorisering av hvert tall. **IKKE** tillatt her! (egentlig mulig, men det senker ikke tiden det tar å faktorisere ett tall, bare det å faktorisere 100 tall)
- Del opp primtall-linja likt slik at tråd- 0 deler på alle primtall ≤ 2mrd/k, tråd-1 de neste primtallene p: 2mrd/k <p ≤ 2* 2mrd/k
  - Blir noen interessante problemer når en av trådene har funnet en faktor – synkronisering trenges bare hvis to tråder samtidig vil levere en ny funnet faktor f.eks i en ArrayList.
  - Hver av trådene bruker da sekvensiell faktorisering på sitt område

Eks. 2: Faktorisering: Parallellisering, oppdeling

a) Mer om : Dele opp primtallslinja i k like deler og la hver tråd dele tallet som skal faktoriseres med de primtallene som er der.
   - De n primtallene < N er ganske ujevnt fordelt, med ca 10 % i den laveste delen og < 4 % i den øverste delen.
     En idé er da å la de k-1 første trådene få (N/log N)/k primtall, neste tråd de neste (N/log N)/k av primtallene, .., og tråd k-1 (den siste tråden) få resten av primtallene.

b) Det er i utgangspunktet ingen mulig oppdeling av ett tall M vi skal faktorisere, men:
   - Hett tips: Når vi har funnet en primtallsfaktor p i M (da er M = p*G), så er det ikke M vi lenger faktoriserer, men resten G – og det gjelder for alle trådene.

Faktorisering alt 1 : Det store tallområdet 0:M deles likt mellom trådenere

Vi er gitt tallet M_i vi skal faktorisere

- ? - ingen effektiv løsning ?

Faktorisering alt 2 : Primtallene 0:N deles likt mellom trådenere

Vi er gitt tallet M_i vi skal faktorisere

- Hver tråd bruker da alle primtallene på sitt område og prøver om noen av disse primtallene er faktorer i M_i
  - Bør områdene deles 'likt'?
  - Hver tråd finner evt. faktorer i M_i legges inn som faktor via en locked metode i en felles ArrayList.
- Hva skal vi effektivisere
  - Et tall med mange faktorer ?
  - Et ekte primtall ?

Faktorisering alt 3:
Primtallene 0:N deles likt ut en etter en etter tur

- Alle trådene tar hver k'te primtall
- Anta at k=4
  - $Tråd_0$ tar: 3, 13,...
  - $Tråd_1$ tar: 5, 17,...
  - $Tråd_2$ tar: 7, 19,...
  - $Tråd_3$ tar: 11, 23,...
- Fordeler
  - Med de andre alternativene: Hvis vi finner f.,eks at 13 er en faktor, så har tråd 1,2 og 3 ikke mer å gjøre (og $tråd_0$ gjør resten av arbeidet)
- Ulempe
  - Hvor raskt er det stadig for en tråd å finne neste k'te primtall i tabellen
  - Kan vi organisere primtallene slik at det går raskere ?

Oblig frister

Oblig IN3030/IN4330 2021:

Nr	Ut	Frist	
1	28/1	10/2	
2	11/2	24/2	
3	25/2	17/3	
4	18/3	14/4	(påske)
5	15/4	5/5	

Last-balansering ved parallellisering

- Load balancing

Hvordan ikke gå i den rekursive fella!

- Vi skal nå gå gjennom hvordan vi behandler rekursjon og parallellisering av programmer med rekursjon
- Først en 'teoretisk' PRAM-lignende parallell løsning
  - som går ca. 1000x langsommere enn en sekvensiell versjon.
- Så skal vi se på to forbedringer som gjør at den uhyre treige løsningen vår tilslutt har en speedup på ca. 4
- Hele problemet bunner i de tallene vi presenterer i dag :
  - Å starte en ny tråd med vent på terminering tar: **92**μs (snitt mange ganger)
  - Å gjøre et metodekall tar: **0.016** μs (snitt mange ganger)

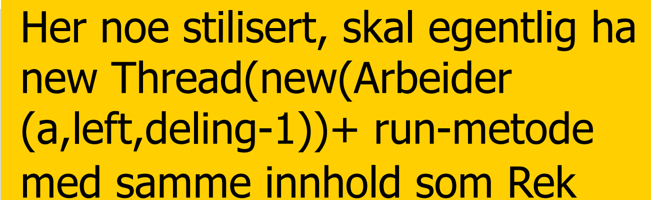
Om rekursiv oppdeling av et problem

- Svært mange problemer kan gis en (sekvensiell og parallell) rekursiv løsning:
  - De fleste søkeproblemer
    - Del søkebunken rekursivt opp i disjunkte deler og søk (i parallell) i hver bunke.
  - Mange sorterings-algoritmer som QuickSort, Flettesortering, og venstreRadix-sortering er definert rekursivt
  - Oblig4 - den konvekse innhyllinga
- Skal nå bruke en ny formulering av QuickSort som eksempel og gi den 3 ulike løsninger:
  A. Ren oversettelse av rekursjonen til tråder
  B. Med to tråder for hvert nivå inntil vi bruker InnstikkSortering
  C. Med en ny tråd for hver nivå og avslutting av tråder når lengden < LIMIT (si 50 000) – deretter vanlig rekursjon

Generelt om rekursiv oppdeling av a[] i to deler

```
void Rek (int [] a, int left, int right) {
        <del opp omradet a[left..right] >
        int deling = partition (a, left,right);

        if (deling - left  > LIMIT ) Rek (a,left,deling-1);
        else <enkel løsning>;
        if (right - deling > LIMIT) Rek (a,deling, right);
        else  <enkel løsning>
}
```

```
void Rek(int [] a, int left, int right) {
        <del opp omradet a[left..right]>
        int deling = partition (a, left,right);
        Thread t1 = null, t2= null;

        if (deling - left > LIMIT )  t1 = new Thread (a,left,deling-1);
        else <enkel løsning>;
        if (right - deling > LIMIT) t2 = new Thread (a,deling, right);
        else  <enkel løsning>
        try{ if (t1!=null)t1.join();
             if (t2!=null)t2.join();} catch(Exception e){};
}
```



```
void Rek (int [] a, int left, int right) {
        <del opp omradet a[left..right] >
        int deling = partition (a, left,right);

        if (deling - left  > LIMIT ) Rek (a,left, deling -1);
        else <enkel løsning>;
        if (right - deling > LIMIT) Rek (a, deling , right);
        else  <enkel løsning>
}
```

A

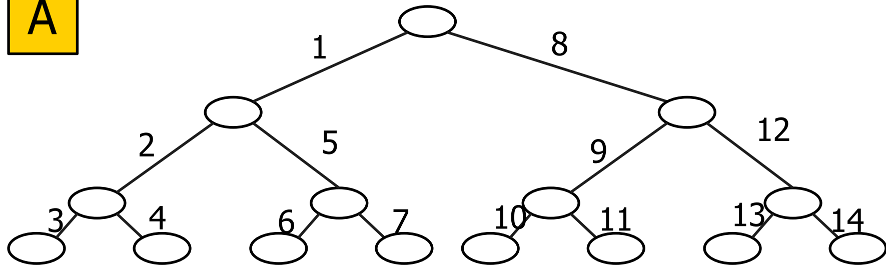
```
void Rek(int [] a, int left, int right) {
        <del opp omradet a[left..right]>
        int deling = partition (a, left,right);
        Thread t1 = null, t2=null;

        if (deling- left > LIMIT )
            (t1 = new Thread (a,left, deling -1)).start();
        else <enkel løsning>;
        if (right - deling> LIMIT)
            (t2 = new Thread (a, deling , right)).start();
        else  <enkel løsning>
        try{ if (t1!=null) t1.join();
             if (t2!=null) t2.join();}
       catch(Exception e){return;};
}
```

B

Oppdeling med **to** tråder per nivå i treet:

- Når ventes det i den rekursive løsningen
  - Har det betydning for rekkefølgen av venting ?
- Når ventes det i den parallelle løsningen A?
  - Har rekkefølgen på venting på t1 og t2 betydning?
- Antar at kall på Rek tar T millisek.
- Hvor lang tid tar A og B
- Hvilken er raskest ?

A

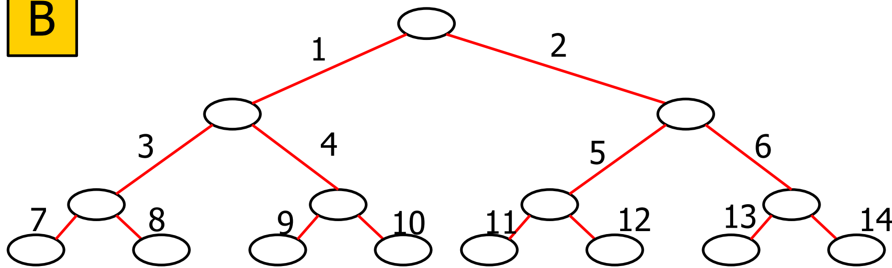


Rekursjon

Tråd

Dybde først - sekvensiell

B



Rekursjon

Tråd

Bredde først - parallell

Oppdeling med **en tråd** per nivå i treet:

- Hvorfor virker dette ?
- To alternativ løsning med 1 tråd
  - Har det betydning for rekkefølgen av venting ?
- Når ventes det i C-løsningen?
  - Har rekkefølgen på venting på t1 betydning?
- Når ventes det i D-løsningen?
- Antar at kall på Rek tar T millisek.
- Hvor lang tid tar C
- Hvor lang tid tar D

Hvilken er klart raskest:
C eller D?

D – er raskest fordi både høyre og venstre gren startes før man venter.

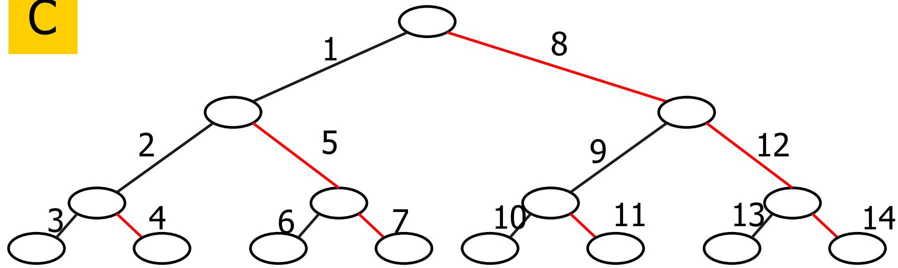
C

```
void Rek(int [] a, int left, int right) {
        <del opp omradet a[left..right]>
        int deling = partition (a, left,right);
        Thread t1;

        if (deling - left > LIMIT )
                Rek (a,left,deling-1);
        else <enkel løsning>;
        if (right - deling > LIMIT)
                t1 = new Thread (a,right,deling-1);
        else  <enkel løsning>
         try{t1.join();} catch(Exception e){};}
```

D

```
void Rek(int [] a, int left, int right) {
        <del opp omradet a[left..right]>
        int deling = partition (a, left,right);
        Thread t1;

        if (deling - left > LIMIT )
                t1 = new Thread (a,left,deling-1);
        else <enkel løsning>;
        if (right - deling > LIMIT)
                Rek (a,deling, right);
        else  <enkel løsning>
        try{t1.join();} catch(Exception e){};}
```

C

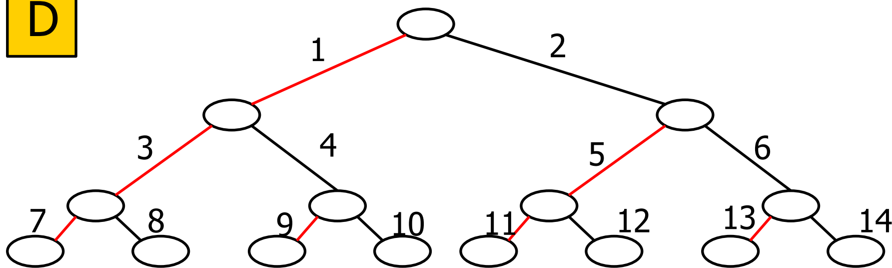


Rekursjon

Tråd

Dybde først - sekvensiell

D



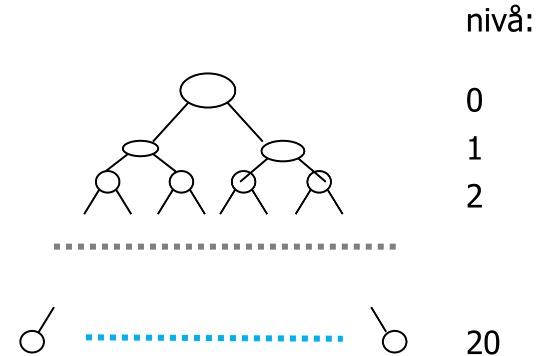
Rekursjon

Tråd

Bredde først - parallell

Hvor mange kall gjør vi i en rekursiv løsning?

- Anta Quicksort av $n = 2^k$ tall
  ($k= 10 \Rightarrow n = 1000$, $k= 20 \Rightarrow n= 1$ mill)
- Kalltreet vil på første nivå ha 2 lengder av 2^{19}, på neste: $4 = 2^2$ hver med 2^{18} og helt ned til nivå 20, hvor vi vil ha $2^{20+1}-1$ kall hver med $1 = 2^0$ element.
- I hele kalltreet gjør vi altså 2 millioner -1 kall for å sortere 1 mill tall !
- Bruker vi innstikksortering for $n < 32 = 2^5$ så får vi 'bare' $2^{20-5+1} = 2^{15+1} -1=$ **65 535** kall.
- Metodekall tar først: **2 us** men så **0.02** μs og kan også optimaliseres bort (og gis speedup >1)
- Å lage en tråd og starte den opp tar først : ca.**3000** μs, men så ca. **62** μs for de neste trådene (med start() og join())

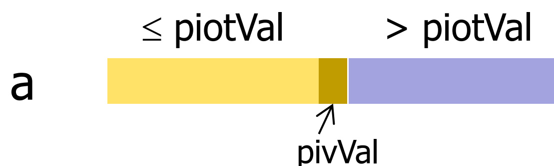


Vi kan IKKE bare erstatte rekursive kall med nye tråder i en rekursiv løsning !

A) Sekvensiell kvikksort – ny og enklere kode

```
// sekvensiell Kvikksort
void quicksortSek(int[] a, int left, int right) {
        int piv = partition (a, left, right); // del i to
        int piv2 = piv-1, pivotVal = a[piv];
        while (piv2 > left  && a[piv2] == pivotVal) {
             piv2--; // skip like elementer i midten
         }

        if ( piv2-left > 16) quicksortSek(a, left,  piv2);
           else insertSort(a,left, piv2);
        if ( right-piv >16) quicksortSek(a, piv + 1, right);
          else  insertSort(a, piv+1, right);
} // end quicksort
```

Etter: partition(a,left,right)




```
// del opp a[] i to: smaa og storre
int partition (int [] a, int left, int right) {
     int pivVal = a[(left + right) / 2];
     int index = left;
    // plasser pivot-element helt til høyre
    swap(a, (left + right) / 2, right);

     for (int i = left; i < right; i++) {
          if (a[i] <= pivVal) {
                swap(a, i, index);
                index++;
          }
      }
      swap(a, index, right); // sett pivot tilbake
      return index;
  } // end partition

 void swap(int [] a, int left, int right) {
       int temp = a[left];
       a[left] = a[right];
       a[right] = temp;
 } // end swap
```

B) En parallell kode (modellert etter A)

```
void Rek(int [] a, int left, int right) {
        <del opp omradet a[left..right]>
        int deling = partition (a, left,right);
        Thread t1 = null, t2=null;

        if (deling- left > LIMIT )
            (t1 = new Thread (a,left, deling -1)).start();
        else insertSort(a,left, deling -1);
        if (right - deling> LIMIT)
            (t2 = new Thread (a, deling , right)).start();
        else  insertSort(a, deling , right);
        try{ if (t1!=null)t1.join();
              if (t2!=null)t2.join();} catch(Exception e){};
}
```

B) Ren kopi av rekursiv løsning: Katastrofe

```
M:>java QuickSort 100 10 100000 1 uke9.txt
Test av  TEST AV QuickSort
 med 8 kjerner ,   Median av:1 iterasjoner, LIMIT:2
```

n	sekv.tid(ms)	para.tid(ms)	Speedup
100 000	34.813	41310.276	0.0008
10 000	0.772	735.838	0.0010
1 000	0.078	66.007	0.0012
100	0.009	3.491	0.0026

Konklusjon:
- For store n speeddown på > 1000
- Kunne ikke kjøre for n > 100 000 pga. trådene tok for stor plass

Hva med en passe LIMIT = 32 ?

```
>java QuickSort 100 10 1000000 1 uke9.

Test av  TEST AV QuickSort
med 8 kjerner ,   Median av:1 iterasjoner, LIMIT:32


      n        sekv.tid(ms)    para.tid(ms)     Speedup
  1000000        89.181        41789.076       0.0021
   100000         8.118          823.432       0.0099
    10000         3.021           55.845       0.0541
     1000         0.060            3.463       0.0173
      100         0.006            0.302       0.0185
```

Konklusjon:

- Mye bedre, men fortsatt 100 x langsommere enn sekvensiell(N = 100 000)
- Greide nå n= 1 mill (fordi færre tråder)
- Fortsatt håpløst dårlig pga. for mange tråder
- Trenger ny idé : Bruk sekvensiell løsning når n < 50 000 ? BIG_LIMIT

Skisse av ny løsning

```
void Rek(int [] a, int left, int right) {
    if ( right – left  < BIG_LIMIT) quicksort (a, left,right);
    else {
            <del opp omradet a[left..right]>
            int deling = partition (a, left,right);
            Thread t1 = null, t2=null;

            //if (deling- left > LIMIT )
                (t1 = new Thread (a,left, deling -1)).start();
            //else insertSort(a,left, deling -1);
            //if (right - deling> LIMIT)
                (t2 = new Thread (a, deling , right)).start();
            //else  insertSort(a, deling , right);
            try{ if (t1!=null)t1.join();
                if (t2!=null)t2.join();} catch(Exception e){};
    }
}
```

Generere nye tråder bare i toppen av rekusjonstreet

- Kan da også stryke kode om LIMIT (vil ikke bli utført) i parallell kode
- Bruken av insertSort gjøres i (sekv) quicksort(..)

Kjøreeksempel med BIG_LIMIT og LIMIT

```
>java QuickSort 100 10 100000000 1 uke9.txt
Test av  TEST AV QuickSort med BIG_LIMIT
 med 8 kjerner ,   Median av:1 iterasjoner,
                   LIMIT:32, BIG_LIMIT:50000


     n      sekv.tid(ms)   para.tid(ms)    Speedup
 100000000   12042.708      3128.675       3.8491
  10000000    1090.252       277.264       3.9322
   1000000      92.958        32.640       2.8480
    100000       7.682         5.198       1.4777
     10000       0.616         0.737       0.8356
      1000       0.051         0.117       0.4354
       100       0.013         0.015       0.8636
```

BIG_LIMIT = 100 000

```
>java QuickSort 100 10 100000000 1 uke9.txt
Test av  TEST AV QuickSort med BIG_LIMIT
 med 8 kjerner ,   Median av:1 iterasjoner, LIMIT:32,
BIG_LIMIT:100000


      n       sekv.tid(ms)   para.tid(ms)     Speedup
 100000000    12110.344       2967.764        4.0806
  10000000     1084.587        277.454        3.9091
   1000000       93.428         32.078        2.9125
    100000        7.894          7.828        1.0085
     10000        0.815          0.617        1.3224
      1000        0.072          0.101        0.7153
       100        0.013          0.015        0.8410
```

N= 200 mill – 20 , BIG_LIMIT = 100 000

```
>java -Xmx6000m QuickSort 20 10 200000000 3 uke9.txt
Test av  TEST AV QuickSort med BIG_LIMIT
 med 8 kjerner ,   Median av:3 iterasjoner, LIMIT:32,
BIG_LIMIT:100000


      n        sekv.tid(ms)    para.tid(ms)      Speedup
 200000000    26091.361        5685.715        4.5889
  20000000     2352.854         598.071        3.9341
   2000000      197.353          61.362        3.2162
    200000       17.003           9.944        1.7099
     20000        1.328           1.332        0.9966
      2000        0.111           0.114        0.9754
       200        0.016           0.016        1.0217
        20        0.000           0.000        1.0000
```

Konklusjon om å parallellisere rekursjon

- Antall tråder må begrenses !
- I toppen av treet brukes tråder (til vi ikke har flere og kanskje litt mer)
- I resten av treet bruker vi sekvensiell løsning i hver tråd!
- Viktig også å kutte av nedre del av treet (her med insertSort) som å redusere treets størrelse drastisk (i antall noder)
- Vi har for n = 100 00 gått fra:

n	sekv.tid(ms)	para.tid(ms)	Speedup	
100000	34.813	41310.276	0.0008	Ren trådbasert
100000	8.118	823.432	0.0099	Med insertSort
100000	7.682	5.198	1.4777	+ Avkutting i toppen

- Speedup > 1 og ca. 10 000x fortere enn ren oversettelse.

Hva så vi på i uke 8

- Oblig frister
- Review Primtall
- Hvilken orden O() har Eratosthenes Sil ?
- Alternativer for å parallellisere: Eratosthenes Sil
- Generelt om last-balansering ved parallellisering med eksempel.
- Hvordan parallellisere rekursive algoritmer
- Gå ikke i 'direkte oversettelses-fella'
  - eksemplifisert ved Kvikk-sort, 3 ulike løsninger