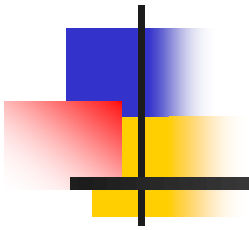


INF2440 Uke 14, v2014

Arne Maus
OMS,
Inst. for informatikk

Resten av IN F2440

- Denne forelesningen
  - Mer om hvordan parallellisere ulike problemer
- 2.mai – forelesning
  - Oppsummering av pensum, presisering av krav til eksamen
- 26. mai – prøveeksamen
  - Husk at de 'beste' oppgavene spares til eksamen
- 2.juni kl. 14.30
  - Tillatt å ha med all skriftlig materiale
  - Ha med utskrift av alle forelesningene (definerer pensum) og Obligene med dine løsninger
  - Intet elektronisk – ingen maskineksamen

Hva så vi på i Uke13

I) Om vindus- (GUI) programmering i Java
- Tråder ! Uten at vi vet om det – event-tråden + main-tråden!

II) Hvordan pakke inn en parallell algoritme slik at andre kan bruke den.
- Hvordan gi din Oblig4 til en venn eller en ny jobb som ikke har peiling på parallellitet og si at her er en metode som sorterer 3-10x fortere enn Arrays.sort() ?
- Brukervennlig innpakking (static eller objektmetode) !
- Nødvendige endringer til algoritmene, effektivitet !
- Fordelinger av tall vi skal sortere.
- Fornuftig avslutning av programmet ditt (hvordan avslutte trådene)

III) Litt om løsning på prog1 og prog2 – finn én sum for alle n < 2 mrd. og alle (antall) Goldbach-summer for alle partall < 1 . mill. sekvensielt og parallelt.

Hva skal vi se på i Uke14

I) Mer om to store programmer, og hvordan disse kan parallelliseres.

II) Sjekke Goldbachs hypotese for n > $4*10^{18}$.

- Jeg skisserte en meget dårlig (ineffektiv) algoritme for dere.
- Hvordan ser en mer effektiv algoritme ut ?
- Her noen 'verdensrekorder' på Goldbach-summer > $4*10^{18.}$
- Parallellisering av denne

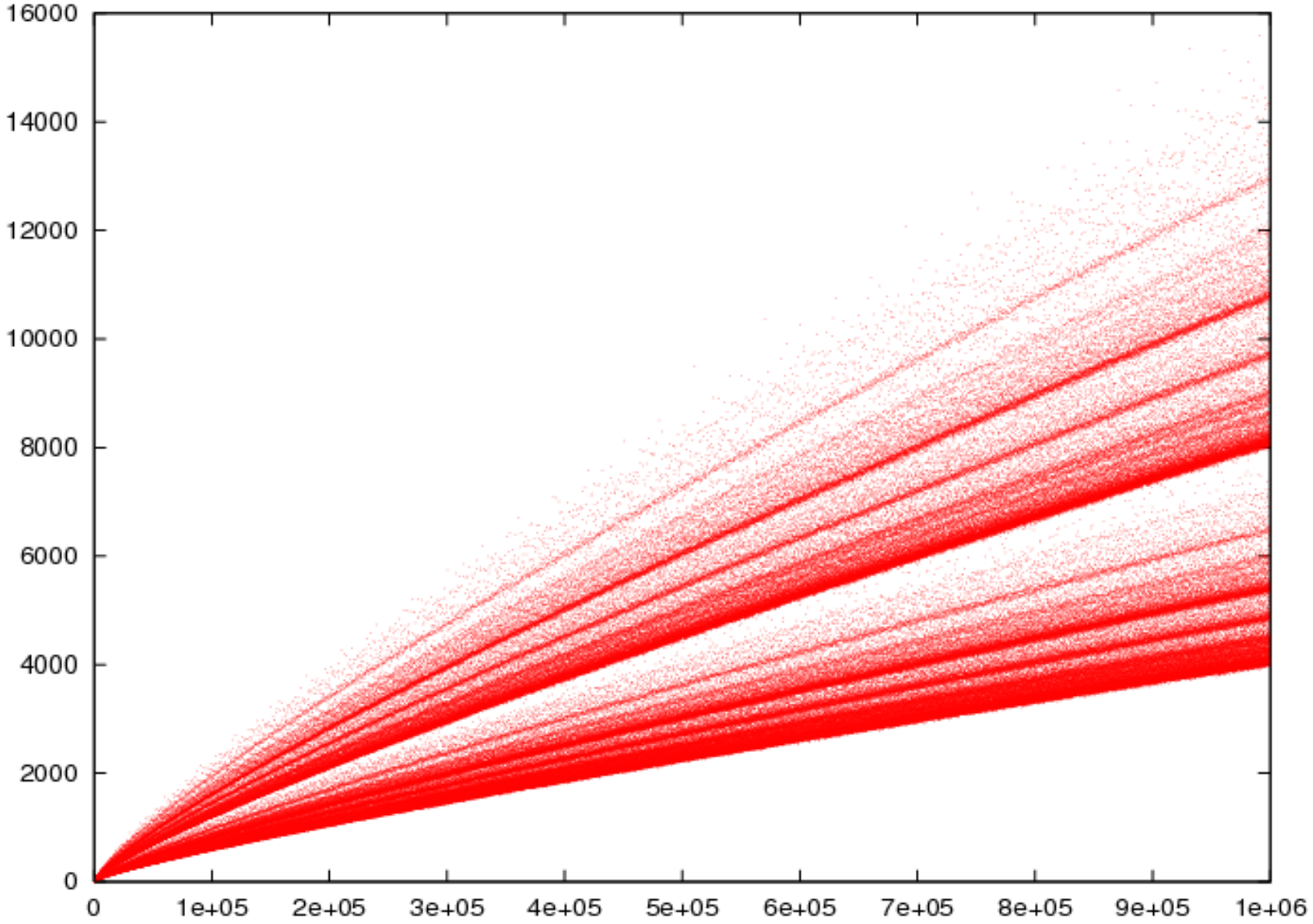
III) Delaunay triangulering – de beste trekantene!

- Brukes ved kartlegging, oljeleting, bølgekraftverk,..
- Spill-grafikk: ved å gi tekstur, farge og glatte overflater på gjenstander, personer, våpen osv.
- Er egentlig flere algoritmer etter hverandre. Skissering av hvordan disse kan parallelliseres.

II) Christian Goldbachs påstand i 1742:

- Alle partall m =n+n > 4 kan skrives som en sum av to primtall som er oddetall.
  - m = p1+p2 (p1 $\leq$ p2)
  - Eks: 6 = 3+3, 14 = 7+7 og 14 = 11+3,….
- Antall slike ulike summer av to primtall for en gitt m kaller vi G(m).
- Bevis at G(m) > 0 $\forall$ m.

G(n) for alle tall < 1 mill – varierer mye, men har en skarp nedre grense som vi ikke greier å vise er > 0 !



Skisse av Prog3 : Finn minst én slik sum m=p1+p2, sekvensielt program $\forall$ m, $4*10^{18} < m < 4*10^{18}$ + antall, m partall, $p1 \leq p2$;

```
<Les inn: antall>
<Lag e= Eratosthens Sil(2147480000)> // nær øvre grense for int

for (long m = 4*10^18 ; m < 4*10^18+antall; m+=2) {
    // for neste tall m, prøv alle primtall p1 ≤ m/2
     for (int p1 = 3, p1 <= m/2; e.nextPrime(p1){
         if ( e.faktorisering (m-p1).size() == 1 )) {
             //  Funnet Goldbach sum
             break;
         }
     } // end p1
      if (p1 > m/2) System.out.println( " BINGO: Funnet $1. mill:"
                        + m + " kan ikke skrives som summen av to primtall ");
} // end m
```

Hvorfor er denne en riktig, men **dårlig** algoritme ??

Betraktninger før skissen til Prog3 – finne minst en GoldbachSum m = p1+p2 , for m> $4*10^{18}$

- Det største antall bit vi kan ha i en bit-array = maksimalstørrelsen av en int (– litt for overflyt)
- max int = 2 147 483 647
- Prøver å lage en Erotosthanes Sil med n= 214748000
  - Da kan vi faktoriser tall < 4 611 670 350 400 000 002
  - Dette gir oss plass for å prøve ut 611 670 350 400 000 000 tall > $4*10^{18}$
- Egentlig skal vi jo prøve ut alle p1 < m/2, men forskningen viser: (One record from this search: 3 325 581 707 333 960 528 needs minimal prime 9781 (= den største p1 : n =p1+p2))
  - Dvs, satser på at: Er det en Goldbach sum for 19-sifrete tall, så er p1 < 2 147 483 647 (som er mye større enn 9781)

Skisse av Prog3 :

Skissen var svært lite effektiv fordi en av operasjonene er tar mye lenger tid enn alle andre og gjentas veldig mange ganger med samme argument

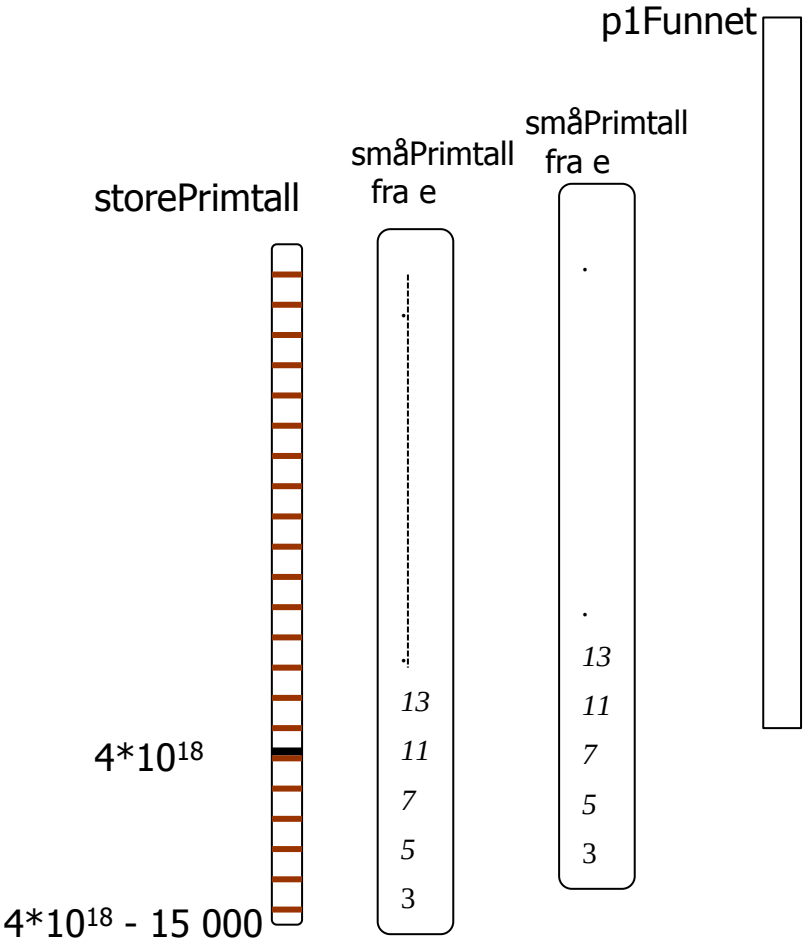
```
<Les inn: antall>
<Lag e= Eratosthens Sil (2147480000)> // nær øvre grense for int

for (long m = 4*10^18 ; m < 4*10^18+antall; m+=2) {
    // for neste tall m, prøv alle primtall p1 ≤ m/2
     for (int p1 = 3, p1 <= 15 000; e.nextPrime(p1){
         if ( e.faktorisering (m-p1).size() == 1 )) {
             println(«OK:»+ m + « = « +(m-p1) + « + » +p1); break;
         }
     } // end p1
     if (p1 > 15 000) System.out.println(m +", kanskje ikke sum av to primtall");
} // end m
```

e.faktorisering (m-p1).size() tar hver ca 1/6 sek i parallell, men vi må teste ca 40 tall før vi finner et primtall (og da tar det 1. sek), og bare hvert 10 primtall passer (men i verste tilfellet er det hvert 1000 primtall).
Dvs. Å finne en GSum på denne måten tar ca. 100 sek. hver gang vi får OK, eller ca. 115 døgn på å lage 100 000 slike nye summer.
Det **må** finnes en bedre algoritme som bla. ikke beregner de samme store primtallene svært mange ganger.

Idé: Hvis det er en sum m = p1+p2, så er p1 < 15 000

Lag en int [] p1Funnet, si 100 000 lang, og en ArrayList
storePrimtall med de første 10 000 store primtall > $4*10^{18}$ - 15 000.



- Legg sammen alle små primtall p1 fra e (3,5,..) med først den minste p2 fra storePrimtall, og lagrer p1 i p1Funnet for alle slike summer som er ≥ $4*10^{18}$
- Gå neste store primtall p2 og gjenta summeringene med de samme små primtallene p1 fra e. … osv
- Når alle slike summer er utført, håper vi at det i p1Funnet står en p1 i hver plass som representerer tallene $4*10^{18}$, $4*10^{18}+2$, $4*10^{18}+4$, …, $4*10^{18}+100\ 000$
- Står det ikke en p1 i en av disse plassene, er det 'kanskje' et tall som ikke har en GoldbachSum ($1mill.?)

Hvorfor begynne adderingen på $4*10^{18}$ - 15 000, hvorfor ikke på $4*10^{18}$?

Skisse1 av ny algoritme for prog3

<Les inn: antall>

a) Lag e= Eratosthens Sil(2147480000)> // nær øvre grense for int

b) Lag en int[] p1Funnet som er ca. 200 000 lang>

c) Lag alle store primtall vi trenger og lagre dem i en ArrayList storePrimtall>

d) for hvert element p2 i storePrimtall, adder de små primtallene p1 fra e slik at summen er < $4*10^{18}$ + 200 000 og lagre disse p1-ene i p1Funnet>

e) gå så gjennom p1Funet og skriv ut de 100 000 summene du forhåpentligvis kan finne der>

hvis p1Funnet[i] != 0 så representerer det summen:

m = p1 + p2

$i+ 4*10^{18}$ = p1Funnet[i] + ($i+ 4*10^{18}$ - p1Funnet[i])

< legg til evt optimalisering av pkt. d) >

Programskisse av ny Prog3 :

Den gamle skissen var svært lite effektiv fordi å finne store primtall tar mye lenger tid enn alle andre og gjentas veldig mange ganger med samme argument

```
<Les inn: antall>
<Lag e= Eratosthens Sil(2147480000)> // nær øvre grense for int
long start = 4*10^18 – 15 000, m;
ArrayList  storePrimtall = new ArrayList(); // plass til 10 000 store primtall > start
int [] p1Found = new int [100 000] ; // p1 i summen: m = p1+p2

// lag først alle store primtall
for ( m = start ; m < 4*10^18+antall; m+=2) {
     if (e.faktorisering(m).size==1) storePrimtall[(m-start)/2] = m;
}
int i =0;
for (m = storePrimtall.get(i); i < storePrimtall.size(); i++) {
    // for neste tall m, legg til alle primtall p1 ≤
     for (int p1 = 3, p1 <= 200 000 ; e.nextPrime(p1)
           p1Found[(m-start+p1)/2] = p1;
}
m = 4*10^18; i =0;
while ( p1Found[i] != 0  && i++ < 100 0000 )
  println(m + i *2) + « =« p1Found[i] «+» + (m+i*2 - p1Funnet[i] )
);
```

Noen resultater fra sekvensiell kjøring , store primtall og summer > $4*10^{18}$

Totaltid: ca. 27 min. for 8000 store primtall + en Gsum for alle: $4*10^{18} < m < 4*10^{18}+12\,000$

```
3999999999999996017
3999999999999996029
3999999999999996067
3999999999999996103
3999999999999996119
3999999999999996151
3999999999999996247
3999999999999996257
3999999999999996313
3999999999999996323
.....
4000000000000003669
4000000000000003691
4000000000000003693
4000000000000003717
4000000000000003787
4000000000000003819
4000000000000003841
4000000000000003849
4000000000000003871
4000000000000003903
4000000000000003921
4000000000000003933
4000000000000003973
4000000000000003981
```

```
4000000000000000000 = 3999999999999996119 + 3881
4000000000000000002 = 3999999999999996151 + 3851
4000000000000000004 = 3999999999999996151 + 3853
4000000000000000006 = 3999999999999996017 + 3989
4000000000000000008 = 3999999999999996119 + 3889
4000000000000000010 = 3999999999999996067 + 3943
4000000000000000012 = 3999999999999996791 + 3221
4000000000000000014 = 3999999999999996067 + 3947
4000000000000000016 = 3999999999999996247 + 3769
4000000000000000018 = 3999999999999996017 + 4001
4000000000000000020 = 3999999999999996017 + 4003
 ..........................................................
4000000000000011976 = 3999999999999996017 + 15959
4000000000000011978 = 3999999999999996119 + 15859
4000000000000011980 = 3999999999999996067 + 15913
4000000000000011982 = 3999999999999996719 + 15263
4000000000000011984 = 3999999999999996103 + 15881
4000000000000011986 = 3999999999999996067 + 15919
4000000000000011988 = 3999999999999996017 + 15971
4000000000000011990 = 3999999999999996017 + 15973
4000000000000011992 = 3999999999999996103 + 15889
4000000000000011994 = 3999999999999996257 + 15737
4000000000000011996 = 3999999999999996119 + 15877
4000000000000011998 = 3999999999999996331 + 15667
4000000000000012000 = 3999999999999996029 + 15971
```

Parallellisering av ny algoritme for prog3

<Les inn: antall>

a) Lag e= Eratosthens Sil(2147480000)> // Har vi parallellisert tidligere

b) Lag en int[] p1Funnet 200 000 lang> // IKKE parallellisering (hvorfor ?)

c) Lag alle store primtall vi trenger og lagre dem i en ArrayList storePrimtall>
// faktorisering har vi parallellisert før

d) for hvert element p2 i storePrimtall, adder de små primtallene p1 fra e slik at summen er < $4*10^{18}$ + 200 000 og lagre disse p1-ene i p1Funnet>
// Ny parallellisering. Hvordan dele opp – dele opp hva?

e) gå så gjennom p1Funet og skriv ut de 100 000 summene du forhåpentligvis kan finne der> // triviell – IKKE parallellisering

< legg til evt optimalisering av pkt. d) >

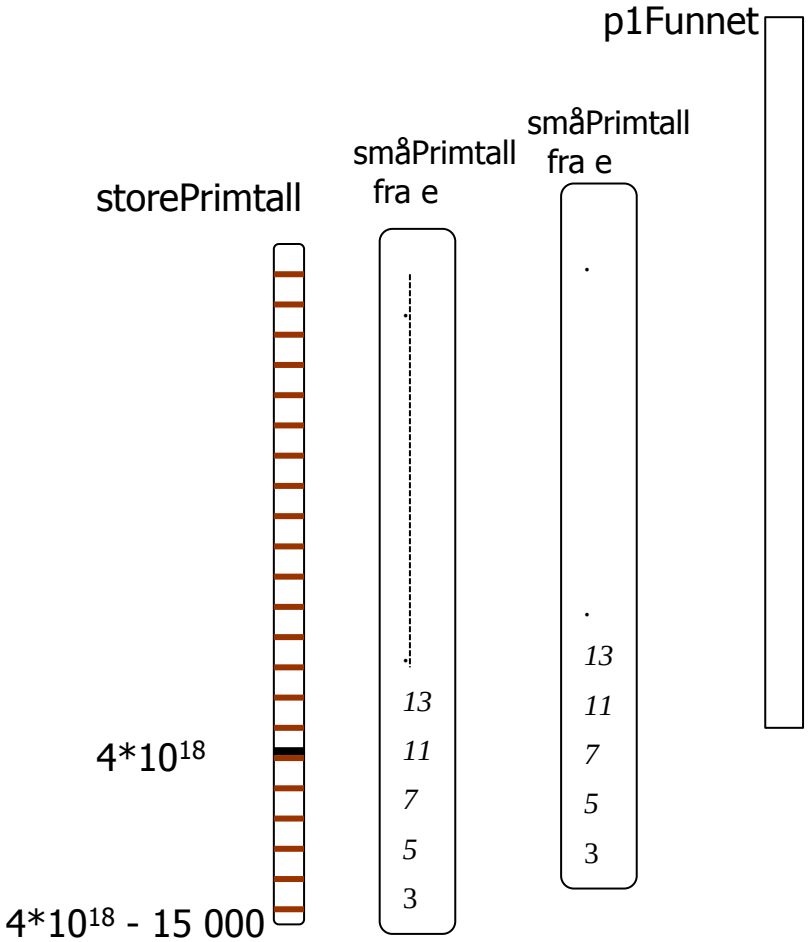
Parallellisering av pkt d) – finne p1 (og p2) for en m

Hva er delte data her som det skrives i ?

Svar: Heltall i p1Funnet.

Spørsmål: Trenger vi her synkronisering ?

Svar: For å være helt ærlig: Nei - uansett hvilken p1 vi får ned i en plass i p1Funnet[i] er den OK, gyldig – vi bryter regel 1!! Men bør vi det?



- Legg sammen alle små primtall p1 fra e (3,5,..) med først den minste p2 fra storePrimtall, og lagrer p1 i p1Funnet for alle slike summer som er > $4*10^{18}$
- Gå neste store primtall p2 og gjenta summeringene med de samme små primtallene p1 fra e. … osv
- Når alle slike summer er utført, håper vi at det i p1Funnet står en p1 i hver plass som representerer tallene $4*10^{18}$, $4*10^{18}+2$, $4*10^{18}+4$, …, $4*10^{18}+100\ 000$
- Står det ikke en p1 i en av disse plassene, er det 'kanskje' et tall som ikke har en GoldbachSum ($1mill.?)

Parallellisering av pkt d) – finne p1 (og p2) for en m =p1+p2

Hvor mange operasjoner snakker vi om. Anta 10 000 store primtall og 100 000 små primtall.

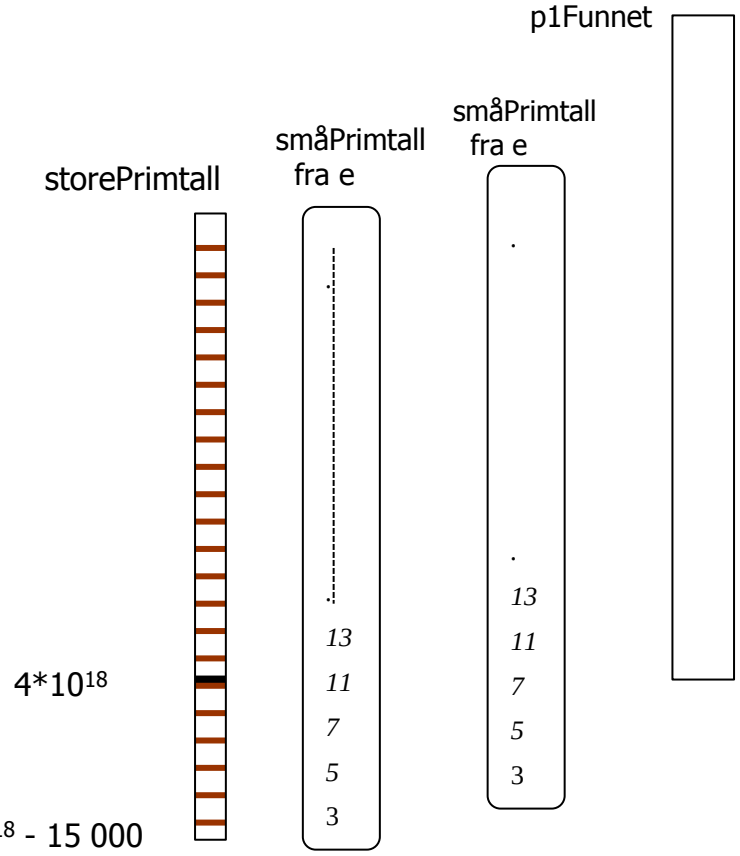
Svar: ca. 1 milliard (10^9) addisjoner - verdt å parallellisere !

Kan parallellisering gjøre dette raskere?

Svar: Ja, deler opp storePrimtall i k deler (en for hver tråd)

Spørsmål: Hvilken speedup kan vi forvente?

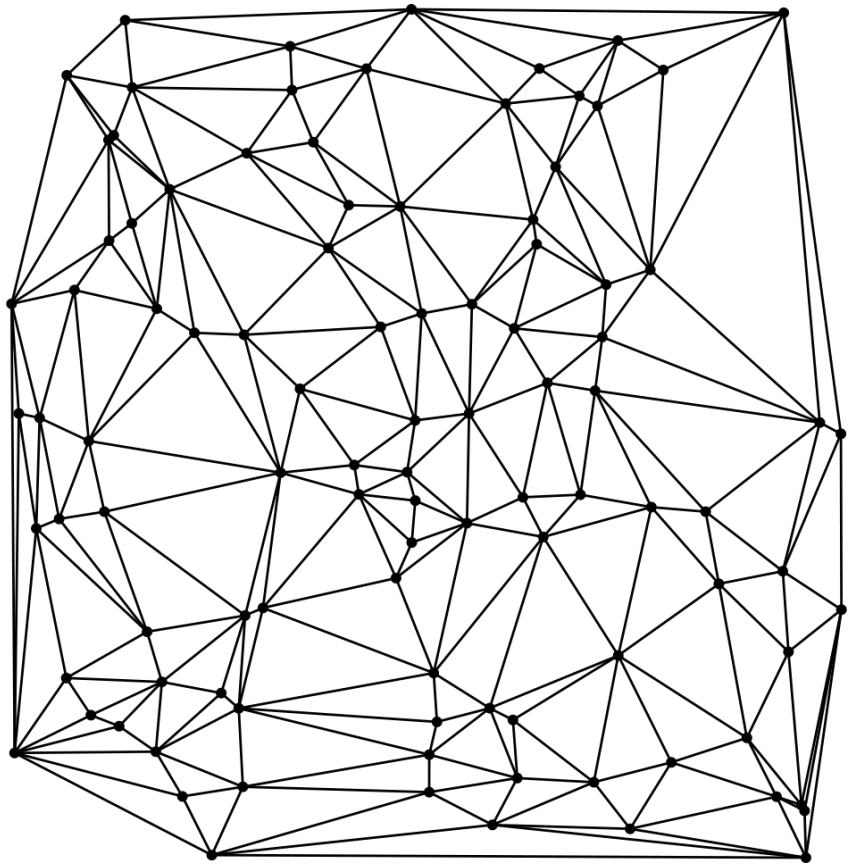
Svar: Om lag k.



- Legg sammen alle små primtall p1 fra e (3,5,..) med først den minste p2 fra storePrimtall, og lagrer p1 i p1Funnet for alle slike summer som er > $4*10^{18}$
- Gå neste store primtall p2 og gjenta summeringene med de samme små primtallene p1 fra e. … osv
- Når alle slike summer er utført, håper vi at det i p1Funnet står en p1 i hver plass som representerer tallene $4*10^{18}$, $4*10^{18}+2$, $4*10^{18}+4$, …, $4*10^{18}+100\ 000$
- Står det ikke en p1 i en av disse plassene, er det 'kanskje' et tall som ikke har en GoldbachSum ($1mill.?)

III) Triangulering – å lage en flate ut fra noen målinger

- Av og til vil vi representere noen målinger i 'naturen' og lage en kunstig, kontinuerlig flate:
  - Oljeleting – topp/bunn av oljeførende lag
  - Kart – fjell og daler
  - Grafiske figurer:
    - Personer, våpen, hus,..
- (x,y) er posisjonen, mens z er høyden
- Vi kan velge mellom :
  - Firkanter – vanskelige flater i en firkant (vridde)
  - Trekanter – best, definerer et rett plan
- Rette plan kan lettest glattes for å få jevne overganger til naboflater.



Mye av dette arbeidet hviler på:

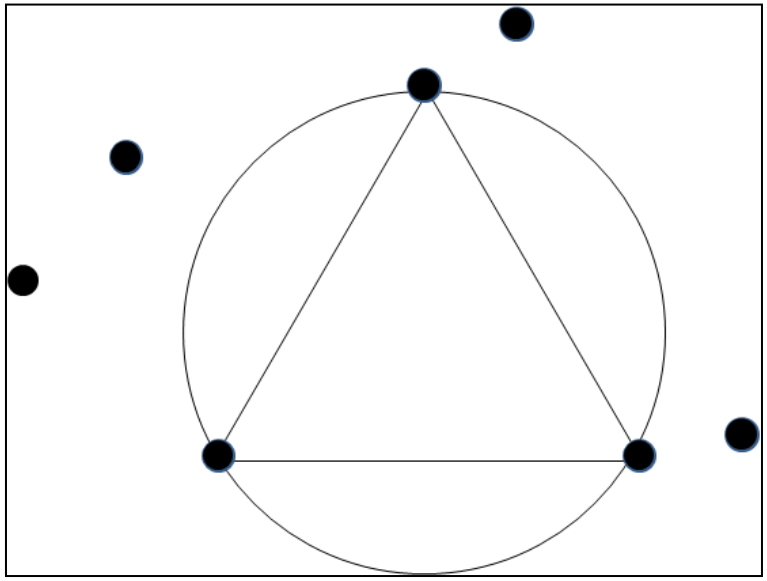
- En Delaunay flatemodell jeg laget i 1980-84 på NR (Kartografi). Simula og Fortran 77. Solgt bla. til et bølgekraftverkprosjekt (Sintef) og Oljedirektoratet.
  - Maus, Arne. (1984). Delaunay triangulation and the convex hull of n points in expected linear time. *BIT 24,* 151-163.
- Masteroppgave: Jon Moen Drange: « Parallell Delaunay-triangulering i Java og CUDA.» Ifi, UiO, mai 2010
  - A. Maus og J.M. Drange: «All closest neighbours are proper Delaunay edges generalized, and its application to parallel algorithms», NIK 2010, Gjøvik, Tapir, 2010
- Masteroppgave: Peter Ludvik Eidsvik: «PRP-2014: Parallelle, faseoppdelte rekursive algoritmer» Ifi, UiO, mai 2014

Delaunay triangulering (1934)

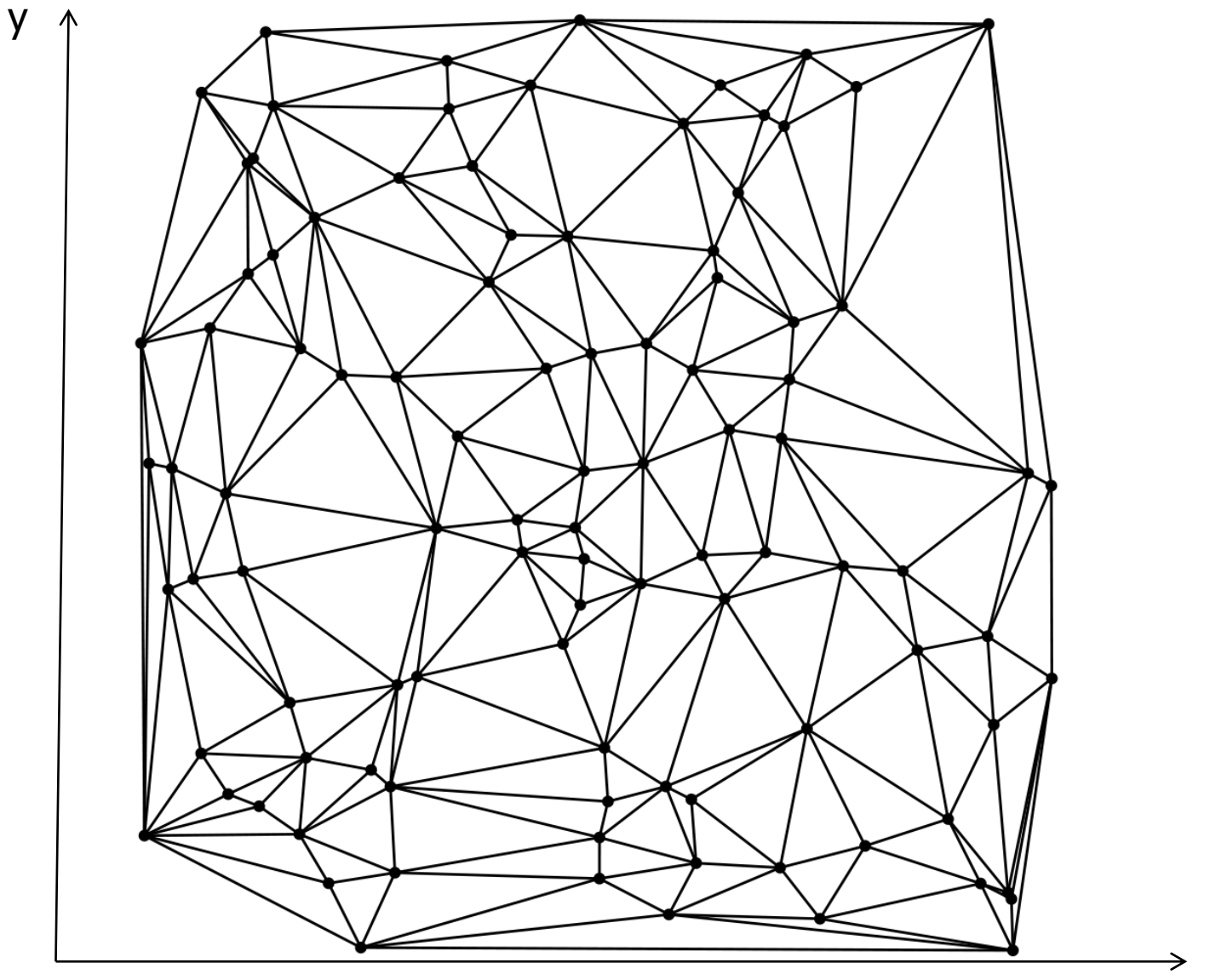
Boris Nikolaevich Delaunay 1890 – 1980, russisk fjellklatrer og matematiker
(etterkommer etter en fransk offiser som ble tatt til fange under Napoleons invasjon av Russland, 1812)

Vi har n punkter i planet
Forbind disse punktene med hverandre med et trekantnett slik at:

- Ingen linjer (trekantsider) krysser hverandre
- Man lager de 'beste' trekantene (maksimerer den minste vinkelen, dvs. færrest lange tynne trekanter)
- Den omskrevne sirkelen for tre de hjørnene i enhver trekant inneholder ingen av de (andre) punktene i sitt indre

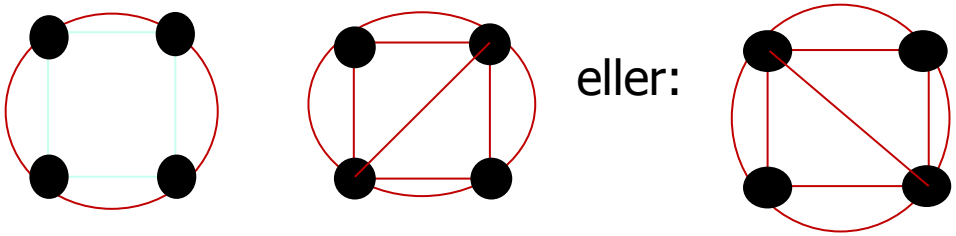


Delaunay triangulering av 100 tilfeldige punkter

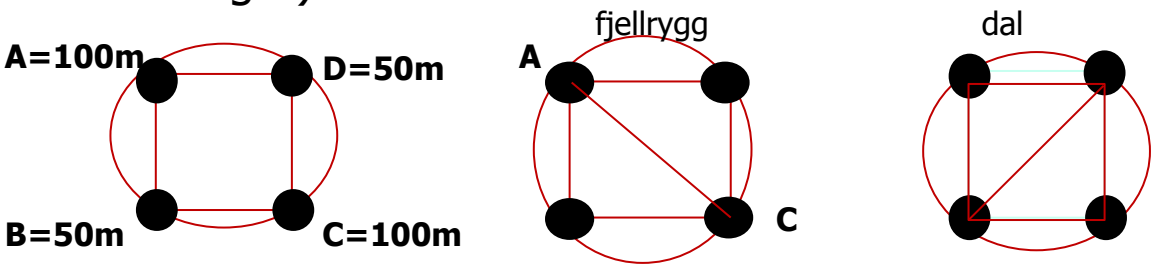


Noen egenskaper ved Delaunay triangulering (DT)

- Hvis ikke alle punktene ligger på en linje, er en DT entydig med følgende spesialtilfelle:
  - Hvis 4 punkter ligger som hjørnene i et kvadrat, må vi finne en regel om hvilke av to trekanter vi skal velge (kosirkularitet):



  - Anta at hjørnene i er målinger av høyder i terrenget. Går det en dal fra B til D eller en fjellrygg A til C ? (ikke avgjørbart uten å se på terrenget).



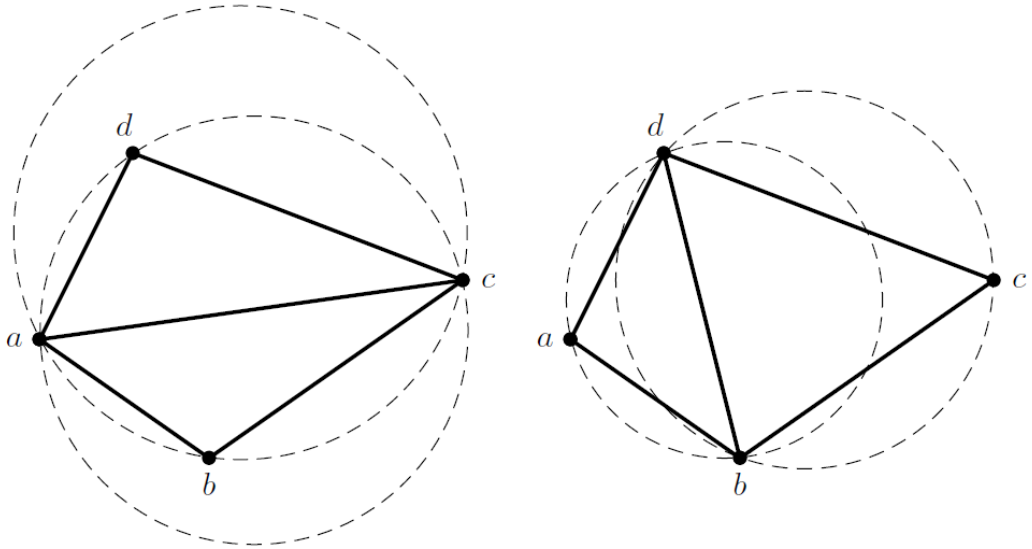
  - Vi velger bare én konsekvent – f.eks siden fra (størst y, minst x) til (minst y, størst x) – dvs. linjen A-C

Noen flere egenskaper ved Delaunay triangulering

- I snitt har et indre punkt P (et punkt som ikke er på den konvekse innhyllninga) seks naboer, men antall naboer kan variere mellom 3 og n-1
  - Et punkt P har f.eks n-1 naboer, hvis n-1 av punktene ligger på en sirkel og P er et indre punkt i sirkelen.
  - Et indre punkt P har tre naboer hvis f.eks vi har 4 punkter, og P ligger inne i en trekant av de tre andre punktene.
- Alle punktene må følgelig ha en fleksibel liste av punkter som er dets naboer (dvs. de punktene den danner trekanter med)
- For at vi kan vite hvilke trekanter et punkt deltar i, må denne nabolista sorteres (mot klokka).
  - Grunnen til dette er at naboene finnes i en 'tilfeldig' rekkefølge
- Har vi k punkter n1,N2,..Nk i en sortert naboliste for punktet P vil PNiN(i+1) være en trekant og PNkN1 også være en av de k trekantene P deltar i.
- Enhver indre trekantside deltar i to trekanter.

Delaunay algoritmer; mange & få gode

- De aller første for å lage en DT (Delaunay Trekant) ABC:
  - Velg et punkt A, prøv alle mulige (n-1) av B_i, og igjen for hver av B_i -ene – alle mulige (n-2) valg av C_j . Test så om A B_i C_j tilfredstiller sirkel-kriteriet.
  - Å finne én DT tar da $O(n^2)$ tid og finne alle DT tar **$O(n^3)$** tid !
- I kurset INF4130 undervises en flippingsalgoritme som i verste tilfellet er $O(n^2)$.



Delaunay – algoritmer her:

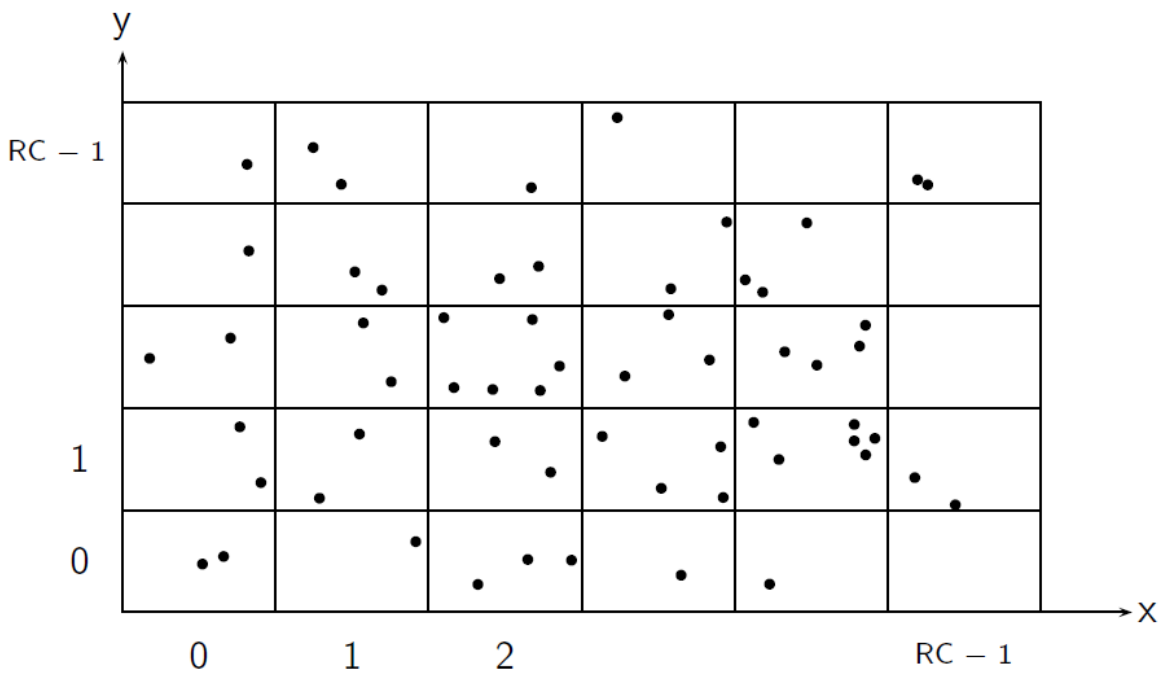
- Her skal vi konsentrere oss om to algoritmer som er O(nlogn):
  - a) Konveks innhylling + sirkelutvidelser fra kjent linje AB i en DT
  - b) Konveks innhylling + Nærmeste naboer + sirkelutvidelser fra kjent linje AB i en DT for resten
- Først beskrive a) fordi den også brukes i b)
- Rask kode som løser a) og/eller b) er ca. 2000 LOC. Vil her bare skissere algoritmene.

Stegene i en Delaunay triangulering

1. Sorter punktene i et rutenett (bokser).
   - For lettere å finne punkter nær et annet punkt
2. Finn den konvekse innhyllinga
   - Dette avgrenser søk og er kjente trekantersider i en DT
3. Foreta selve trekanttrekkinga:
   1. Finn neste C ut fra kjent trekantside AB
   2. Sorter naboer for et punkt rundt et punkt (retning mot klokka)
4. Spesielt problem: hvordan måle en vinkel nøyaktig og raskt ?
5. Bør vi bruke heltall eller flyttall som x,y i posisjonen til punktene?
   1. Svar: Heltall – ellers blir vinkelberegning umulig på små vinkler + at de fleste oppmålinger (som GPS) er i heltall

Delaunay – triangulering har følgende delalgoritmer

1) Sorter datapunktene i bokser (ca. 4-10 punkter i hver boks):

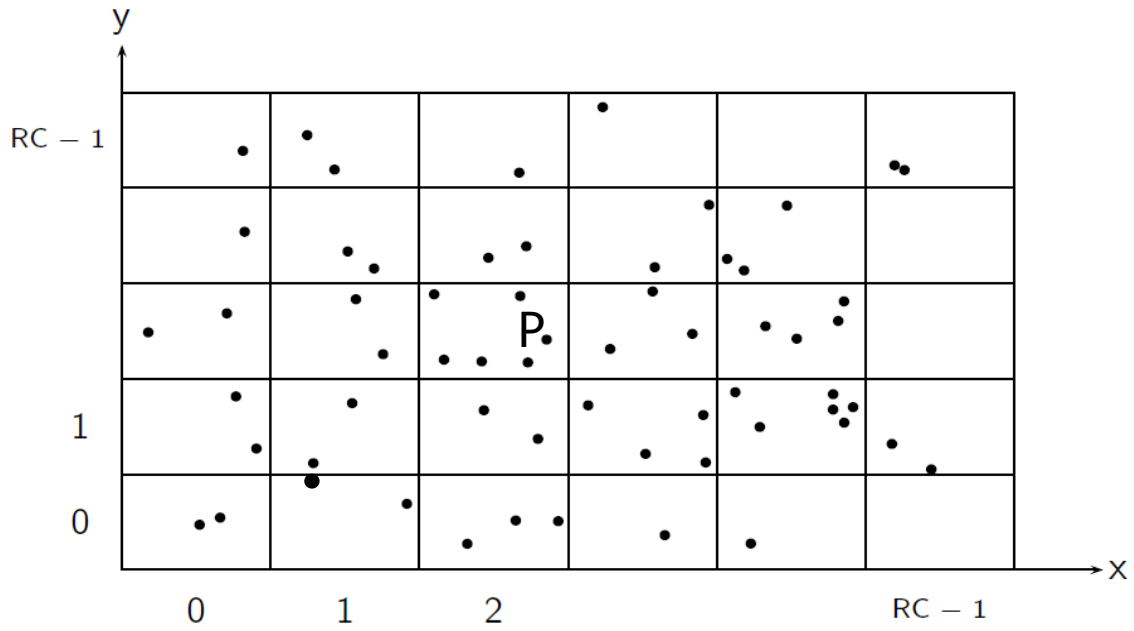


Med 1 000 000 punkter blir dette en 316x316 rutenett med 10 punkter i hver boks. Datastruktur forteller hvilke punkter det er i en boks.

Bruk f.eks Radix-sortering og senere parallell Radix .

Hvordan/hvorfor bruke en boks-oppdeling

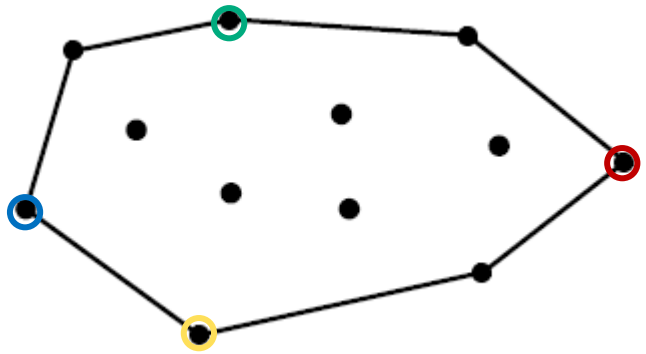
- Si vi skal finne interessante punkter i nærheten av et punkt P:



- Man slår først opp i boksen som P er i, men hvis man der finner punkter som ligger i en avstand hvor 'nærmere' punkter kan ligge i en annen boks, ser man på alle punkter i en 3x3 boksene rundt P (evt. 5x5, osv)

Finn den konvekse innhyllinga

- Dette avgrenser søk og er kjente trekantsider i en DT



- Vi vet at fire (i spesielle tilfeller :tre eller to) punkter er med på den konvekse innhyllinga:
  - max x
  - min x
  - max y
  - **min y**

Hvordan finne den konvekse innhylninga?

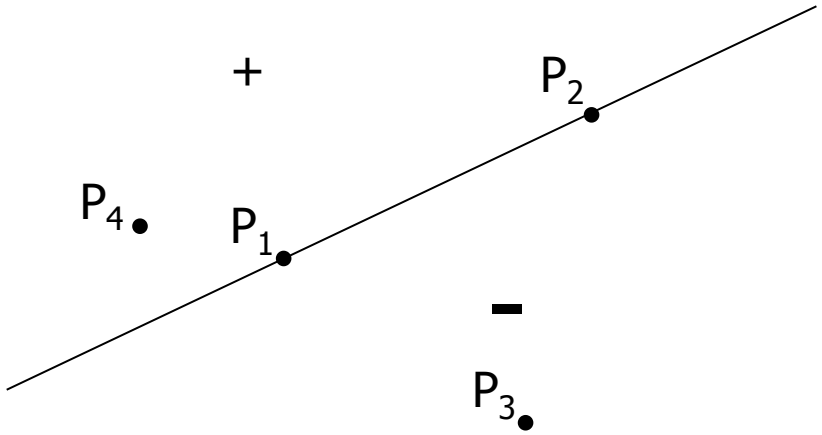
For en rett linje *fra* $P_1(x_1,y_1)$ *til* $P_2(x_2,y_2)$ har vi :

$$y - y_1 = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$$

Ut fra dette kan vi finne a,b og c i den generelle linjeligninga:

$$(*) \quad 0 = ax + by + c$$

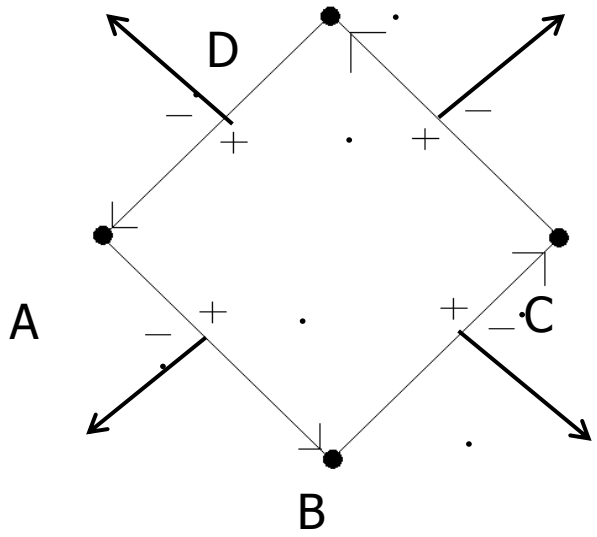
Setter vi et punkt på linjen inn i (*) får vi 0. Setter vi inn et punkt P_3 til høyre for linja inn i (*)i får vi et negativt tall. Setter vi inn et punkt P_4 til venstre for linja inn i (*) får vi et positivt tall. Dette bruker vi til å finne innhyllinga og senere sortering av naboer..

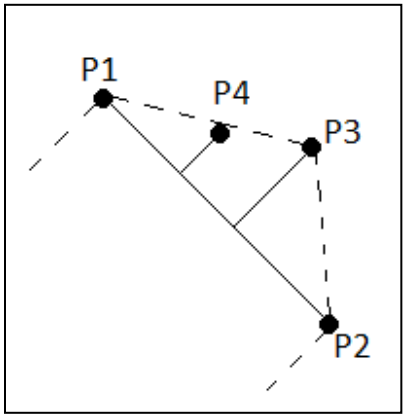


Avstanden fra et punkt P til en linje er bare tallverdien av: punktet P satt inn i linjeligninga.

Sekvensiell, rekursiv løsning av konveks innhylling

- Gitt min/max x og y, gå rekursivt ut fra hver linje AB, BC, CD, DA.





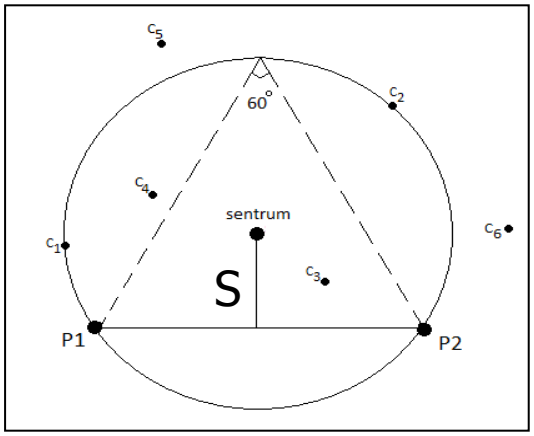
- Finn de punkter som ligger til høre for linja AB, BC,.. (negativ verdi i linjeligninga).
- Finn det punkt P i denne mengden som ligger lengst fra linjen. Det er nytt punkt på den konvekse innhylninga.
- Gå så rekursivt ut fra de to nye linjene som er funnet: AP og PB , osv for alle sidene, mens du fortsatt jobber med den mengden du tidligere fant for de på utsiden av linja

Parallell løsning av konveks innhylning

- Vi har definert en sekvensiell løsning, som starter med :
  - Finn min og maks x og y-verdier i en mengde
    - Hvis n er stor bør dette parallelliseres, og det har vi om lag gjort før (finnMax)
  - Etter dette kommer 4 rekursive kall på hver sin linje, som så deler seg i 8 rekursive kall, 16 kall….
    - Rekursive løsningen har vi løst mange ganger før.
    - Start med 4 tråder – evt. 8 tråder, men etter det bør man gå over til vanlig sekvensiell rekursjon
    - stopp der med flere tråder fordi det er ikke mange punkter på innhyllings – om lag $\sqrt{n}$ punkter på innhyllinga.

Foreta selve trekanttrekkinga -1:

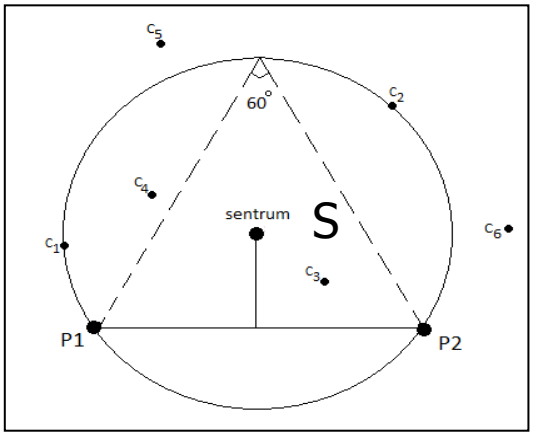
- Gitt DK (Kant i Delaunay trekant) P1-P2. Skal finne punkt C slik at P1p2C er en DT.



- Lager så midtnormalen på P_1P_2, og finner så sentrum S slik at alle perfeirivinklene på sirkelen med sentrum S har en vinkel på 60°.
- Åpner alle bokser rundt S slik at vi finner alle punkter C som er inne i sirkelen og over P_1P_2.
- Finner den C av disse med størst vinkel P_1CP_2.
- Når man finner at ABC er en trekant, noteres dert i A at C er nabo, i B at (A og) C er nabo(er) til B, og i C at A og B er to nye naboer.

Foreta selve trekanttrekkinga - 2:

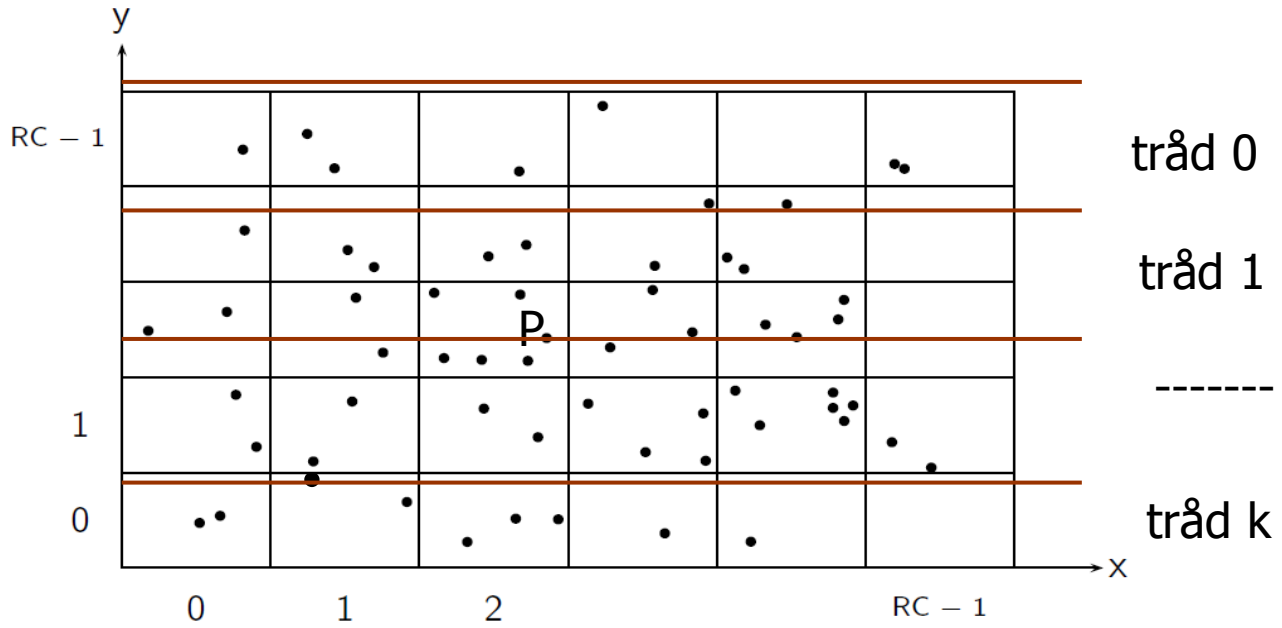
- Var det ingen punkter inne i sirkelen, må sentrum S skyves til vi har laget en f.eks ca. dobbelt så stor sirkel.



- Åpner alle bokser rundt ny S slik at vi finner alle punkter C som er inne i sirkelen og over P1P2.
- Finner den C av disse med størst vinkel P1CP2.
- Hvis det fortsatt ikke er punkter inne i **ny større sirkel**, gjør vi den enda større,.., åpner nye bokser til vi får minst ett punkt C inne i sirkelen.

Parallellisere trekanttrekkinga - 1

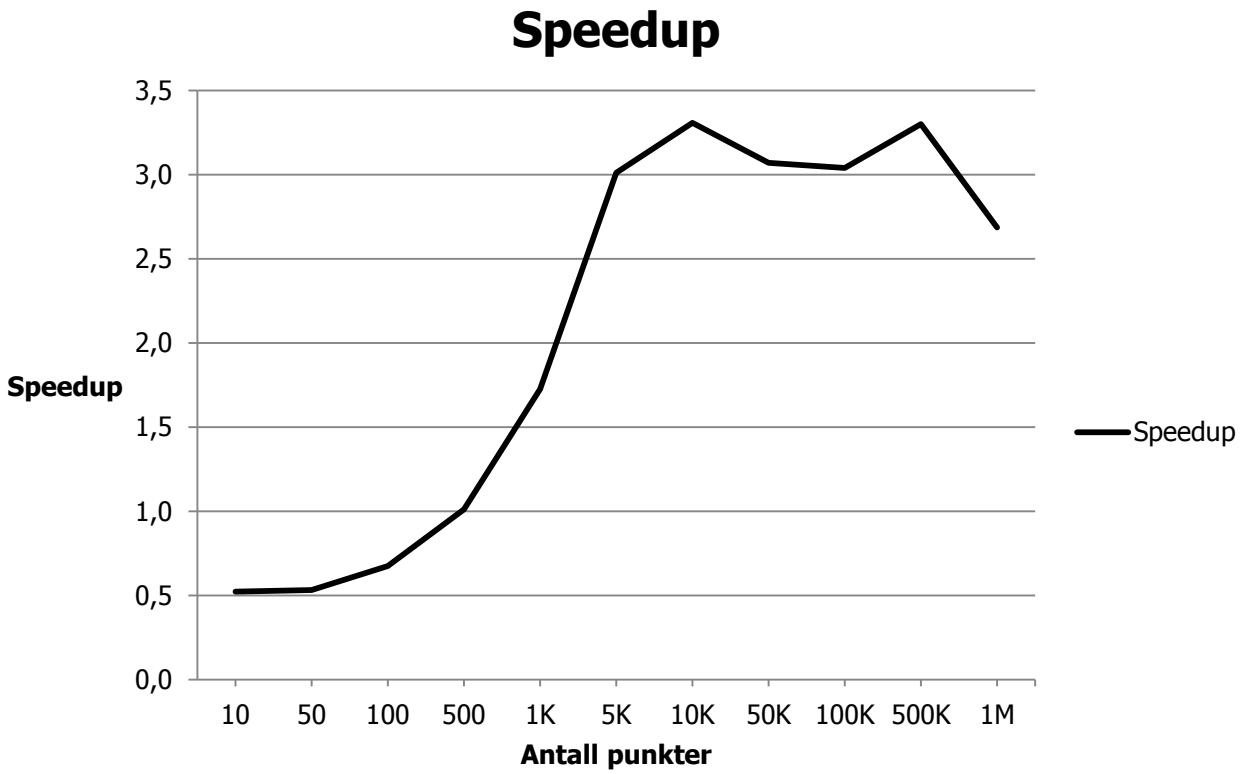
- Hvilke delte data har vi – dvs. data som to tråder muligens kan skrive samtidig i?
  - Svar: Lista over den konveks innhyllinga og særlig nabolista for hvert punkt.
- Deler punktmengden radvis per tråd (se for deg langt flere bokser):



Parallellisering av sirkel-metoden – 2.

- Hvis en tråd 'eier' både A,B og C i en ny trekant , kan alle nye naboer noterers has A, B og C.
- Antar at en tråd_i 'eier' A, men enten ikke både B eller C:
  - Da behøver A bare noterer nye naboer i de punktene den selv eier fordi en DT er entydig, og den samme trekanten ABC blir også funnet av de trådene som eier B og C, og da notert av de trådene.
  - Alternativt, fordi det første involverer ekstraarbeid (finne samme trekant flere ganger) kan hver tråd lage en liste av nye naborelasjoner som blir oppdatert i en senere sekvensiell fase.

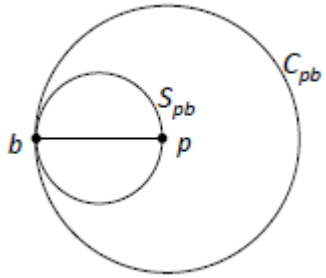
Speedup med bruk av JavaPRP på DT-problemet



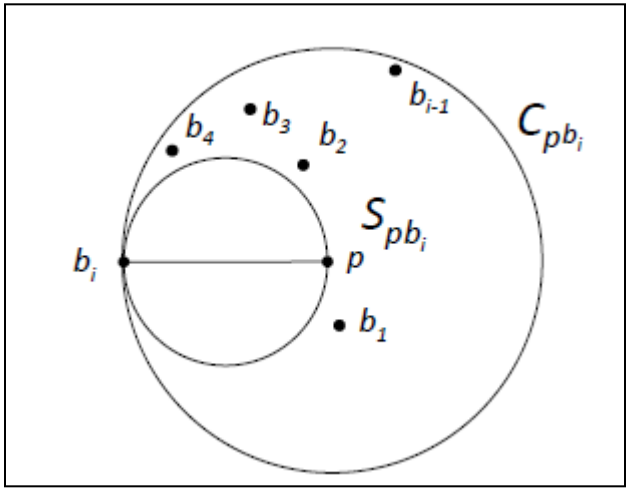
Graf 6: *Viser speedup vi får ved å beregne Delaunay triangulering i parallell, generert av Java PRP. Dataene er hentet fra Tabell 14 og baserer seg på en Intel core I7 870 prosessor med en klokkefrekvens på 2,93 GHz, med 8 kjerner (4 fysiske med hyperthreading).*

Alternativ metode for å finne trekantnaboer

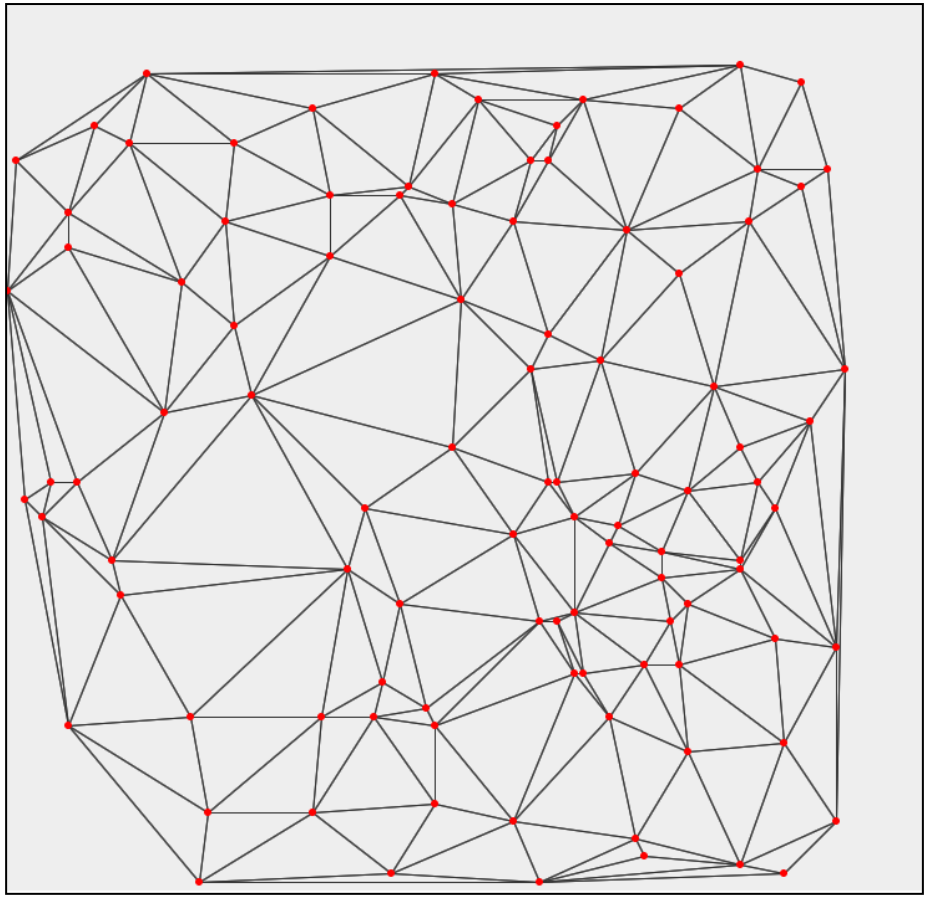
- Linjen fra ethvert punkt p til dens nærmeste nabo b er en trekantside i DT.

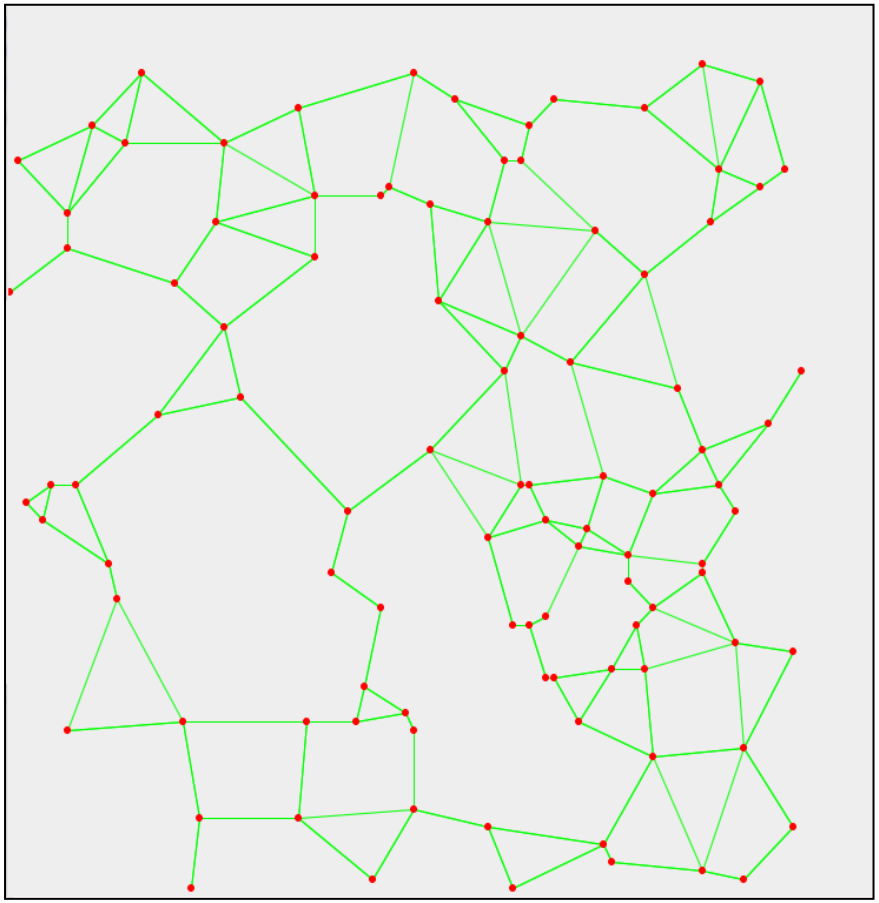


- Også de m nærmeste naboene til p er D-trekantsider hvis sirkelen p – 'nær nabo' b_i ikke inneholder noen av de andre punkter som ligger nærmere p – dvs. b_j hvor j <i

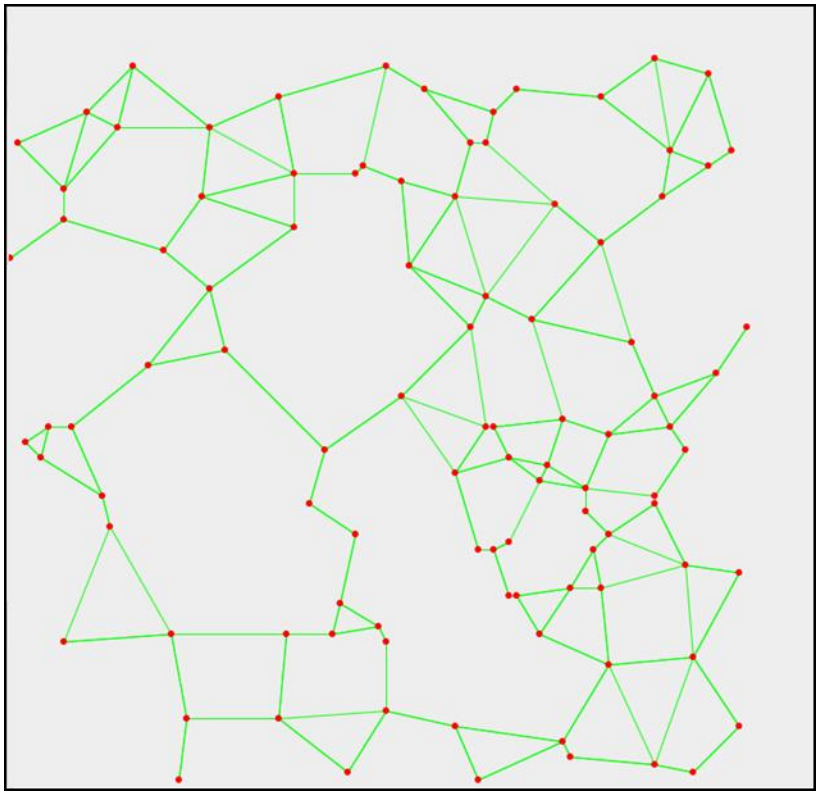


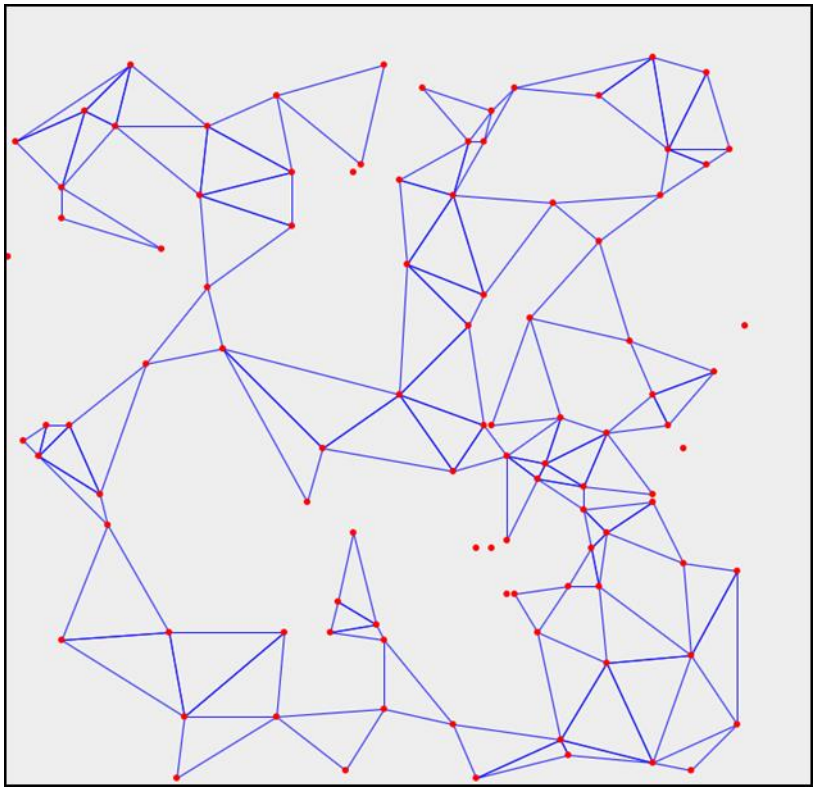
Fullstendig DT og de trekantsidene man finner med 10 nærmeste naboer





Trekantsider og trekanten funnet ned 10 nærmeste naboer-metoden (resten fylles ut med Sirkel-metoden)





Noe speedup for nærmeste nabo-metoden, sekvensielt

Antall punkter	Kjøretider(ms) for		Forbedring
	Sekvensiell Delaunay triangulering u/ DT fra fase 2	Sekvensiell Delaunay triangulering m/ DT fra fase 2	
10	2	2	1,23
50	6	5	1,10
100	7	8	0,85
500	17	16	1,09
1K	30	27	1,11
5K	129	118	1,09
10K	251	215	1,16
50K	1 376	1 136	1,21
100K	2 640	2 280	1,16
500K	15 417	14 084	1,09
1M	30 578	30 515	1,00

Oppsummering om parallellisering av Delaunay triangulering

- Mange metoder og alternativer
- Hvert 'stort' delproblem kan 'lett' gis en parallell løsning
- En god sekvensiell (og da parallell) løsning krever god innsikt i selve problemet.
- Parallelliseringa krever innsikt i særlig hva er felles data og hvordan IKKE synkronisere for mye på disse, men i hovedsak la parallelliteten gå i usynkroniserte parallelle faser etterfulgt av litt sekvensiell kode.

Hva så vi på i Uke14

I) Mer om to store programmer, og hvordan disse kan parallelliseres.

II) Sjekke Goldbachs hypotese for n > $4*10^{18}$.

- Jeg skisserte en meget dårlig (ineffektiv) algoritme for dere.
- Hvordan ser en mer effektiv algoritme ut ?
- Her noen 'verdensrekorder' på Goldbach-summer > $4*10^{18.}$
- Parallellisering av denne skissert

III) Delaunay triangulering – de beste trekantene!

- Brukes ved kartlegging, oljeleting, bølgekraftverk,..
- Spill-grafikk: ved å gi tekstur, farge og glatte overflater på gjenstander, personer, våpen osv.
- Er egentlig flere algoritmer etter hverandre. Skissering av hvordan disse kan parallelliseres.