Kompilering og Kjøring

Kommando for å kompilere programmet:

```
javac –d bin Oblig4.java
```

Kjør programmet slik:

```
java –d Oblig4 <#NTHREADS> <#TESTS>
```

hvor `<NTHREADS>` er antall tråder (potens av 2), og `<#TESTS>` (et positivt oddetall) er antall ganger testen får kjøre. Du trenger ikke å spesifisere noen for de to feltene hvis du ønsker at de skal bli valgt automatisk. I tillegg, for å gjøre det enklere å teste, har jeg laget en makefile som kan kjøre programmet med 16 tråder og 3 tester.

Paralleliseringen

Beskrivelse av algoritmen (relevant for begge Sekv og Para versjoner):

```
/* ++ Step 1: Find minx and maxx.
 * ++ Step 2: Divide all the points into 2 lists (those on the left, and those
 *    on the right of line maxx->minx). At the same time, find 2 points that
 *    are furthest away from both sides of the line.
 * ++ Step 3: Recursion (which consists of 3 sub-steps):
 *    -- Step 3A: Check for the base case
 *    -- Step 3B: With p3 being the furthest point to the right of line p1->p2,
 *       this step will put all the points to the right of the line p1->p3 and
 *       the line p3->p2 into 2 separate lists. At the same time, find 2 points
 *       (on the right side) that are furthest away from the 2 lines above.
 *    -- Step 3C: Next recursive calls */
```

Parallellisering av Steg 2 og 3:

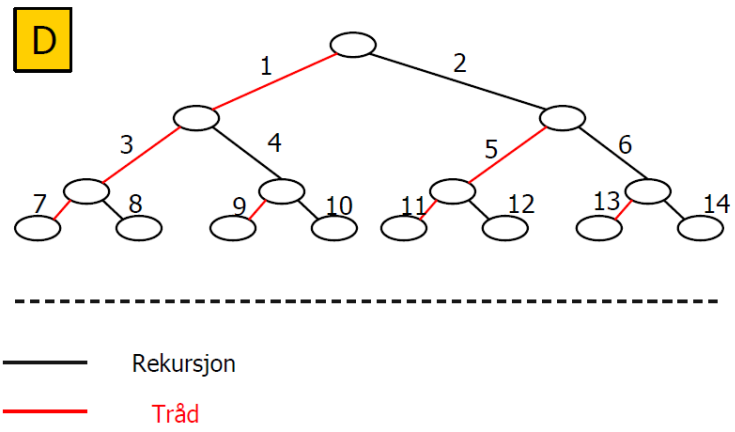
- For steg 2, deler jeg hele punktermengden slik at hver tråd kan jobbe med n/k punkter uavhengige fra hverandre. Hver tråd skal dele punkter i sin del inn i 2 *lokale* mengder (en som inneholder punkter på den venstre side av `maxx->minx` linje, og en som inneholder punkter på den høyre side). Alle tråder skal bruke en samme monitor som har `left` og `right` som globale variabler. Begge to er en `InnList` array som har plass til å lagre alle trådenes lokale mengder. Det betyr at vi kan asynkront oppdatere de to `InnList` arrayer når trådene fra steg 2 er ferdig.
- Siden output som vi får fra steg 2 er `left` og `right` arrayer, må vi da endre signaturen til `parRec()` metoden i steg 3, slik at den kan håndtere en InnList array istedenfor bare et vanlig InnList objekt. Signaturen til `parRec()` ble endret fra:

  ```
  public void parRec(int level, int p1, int p2, int p3, IntList ptSets, IntList koHyll);
  ```

  til:

  ```
  public void parRec(int level, int p1, int p2, int p3, IntList[] ptSets, IntList koHyll);
  ```

- Parallellisering av rekursjonen i steg 3 er veldig enkel. Jeg har valgt å parallellisere dette steget på akkurat samme måte som i kvikksort: Hver node i rekursjonstreet representerer en tråd. Og hver tråd skal starte en ny tråd (som skal jobbe på venstre gren), mens denne tråden kan jobbe på høyre gren. Resultatene (dvs. `koHyll` listene) vil blir slått sammen etterpå ved å bruke metoden `merge()` i `InnList` klassen.



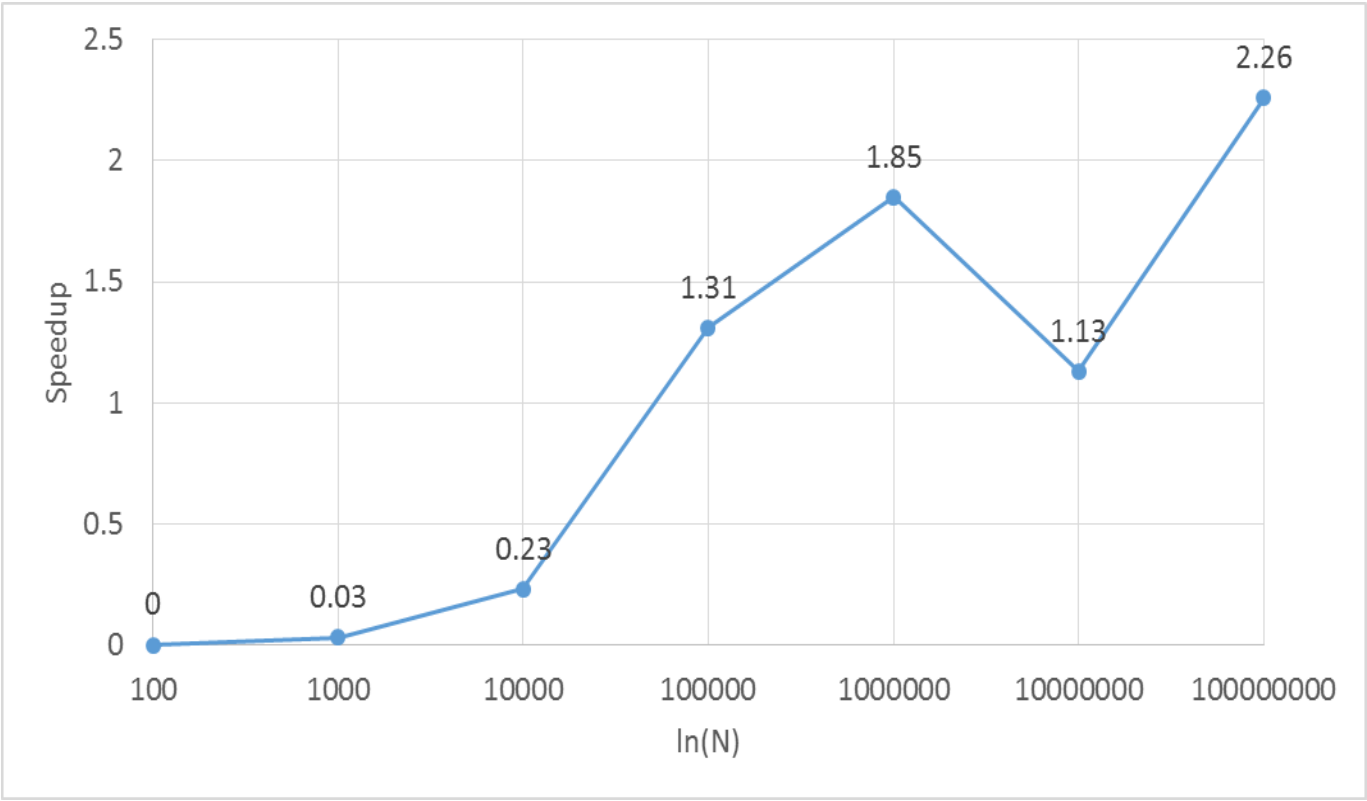
Resultat og Speedup

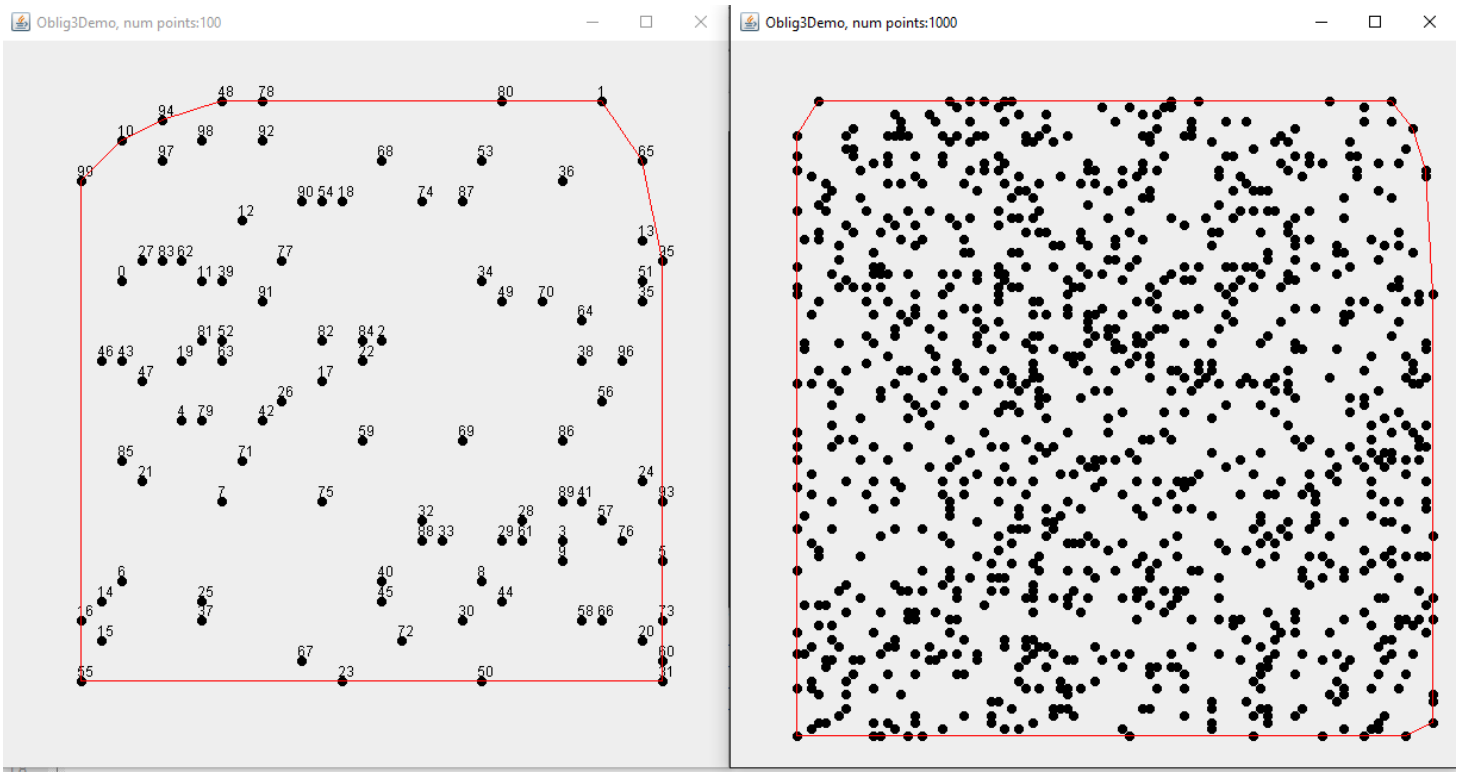
For denne obligen, brukte jeg ikke min egen datamaskin fordi den har bare 4 tråder. Med 4 tråder har vi bare 2 nivåer i rekursjonstreet, som ikke er nok for å få en stor speedup. Jeg trengte da en annen datamaskin med flere en 4 tråder. Derfor har jeg testet programmet mitt på IFIs server (rubin.ifi.uio.no) istedet (se printout.txt). Et annet ting er at algoritmen fungerer OK for 16 tråder, mens speedup blir veldig dårlig når jeg kjører med 64 tråder (for $n = 10^8$). Dette er kanskje fordi n ikke er stor nok for oss til å bruke 64 tråder. Merk at `level` av rekursjonen er avhengig av bare hvor mange tråder vi bruker, ikke på hvor stor n er. Det betyr at hvis vi har 64 tråder istendenfor 16, så får vi et dypere rekursjonstre som kanskje er unødvendig for $n \leq 10^8$.

Jeg har prøvd å teste programmet flere ganger på rubin.ifi.uio.no, men fikk aldri speedup større enn 2.5. For $n = 10^8$, varierer speedup veldig mye mellom $1.75 \rightarrow 2.5$. Noen ganger fikk jeg speedup 2.5 og 2.26 (som vist i grafen nedenfor), men for de fleste av testene fikk jeg bare speedup 1.75, selv om jeg har brukt full parallellisering for denne obligen (dvs. jeg har parallellisert alle de tre stegene fra algoritmen). Dette her betyr at parallelliseringen min ikke er bra nok.

N	Median Tid (ms)		Speedup
	Seq	Par	
100000000	7655.44	3383.61	2.26
10000000	688.85	611.93	1.13
1000000	123.13	66.42	1.85
100000	11.47	8.74	1.31
10000	1.28	5.45	0.23
1000	0.13	4.99	0.03
100	0.02	5.70	0.00

Resultat fra rubin.ifi.uio.no, kjøres med 16 tråder og 3 tester.


2.5
2
1.5
1
0.5
0
Speedup
0
0.03
0.23
1.31
1.85
1.13
2.26
100
1000
10000
100000
1000000
10000000
100000000
ln(N)


Oblig3Demo, num points:100
Oblig3Demo, num points:1000

Jeg synes at problemet kanskje ligger på hvordan jeg parallelliserer steg 3 av algoritmen. For steg 3, har jeg et rekursjonstre med flere nivåer. Den første nivåen har 2 tråder (én som jobbe med `left`, og den andre som jobber med `right`). Neste nivå har 4 tråder, og neste har 8 tråder, og så videre. Men problemet er at vi har mest arbeid å jobbe med på toppen av rekursjonstreet (dvs. punktermengden er størst når vi er på toppen). Dette her betyr at jeg har brukt minst antall tråder for den største arbeidsmengden, og dette senker effektivitet av hele algoritmen. En mulig løsning som jeg kan tenke på er å la alle tråder jobbe samtidig i alle nivåer av rekursjonstreet. Det vil si at når vi går til neste nivå, trenger vi ikke å starte en ny tråd, men vi skal opprette en ny monitor i stedet (se på skissen av denne idéen nedenfor). Dessverre, på grunn av tidsbegrensing med eksamer på andre fag, har jeg ikke tid å prøve denne idéen.