Oblig 4 – IN3030
Ruben Eide Saur (rubenes)

Introduksjon
CPU til laptopen jeg kjørte programmet på: Intel Core i7 – 3610QM 2,3 Ghz 4 kjerner, 8 tråder. Kjøres i Ubuntu. Formålet i denne oppgaven var å lage en parallell versjon av MultiRadix sorterings algoritmen. Deretter sammenligne kjøretider for forskjellige verdier av n.

Brukerveiledning
Programmet kjøres med «java Oblig4 n seed k», eller slik hvis det er stor n: «java -Xmx7500M Oblig4 n seed k». Argumentene: n er hvor stort array som skal sorteres på, seed er seed til random number generatoren som genererer arrayet og k er antall tråder man skal kjøre programmet med. Programmet kompileres med «javac Oblig4.java».

Parallell Radix sort
Del A (finne max) av sorteringen gjøres parallellt ved at hver tråd får en del av a[] hvor den finner max-verdien, deretter legger tråden denne max-verdien i et globalt array med trådnummeret sitt som indeks. Etter dette gjør en av trådene den siste «finn-max» biten på dette globale arrayet, og setter max-verdien til en global variabel. Del B var å finne frekvensen av sifferet (dette brukes til å bestemme hvor siffer skal plasseres til slutt). Her teller hver tråd sin del av a[] opp i en lokal count[], og legger count[] til en global dobbel array med trådnummeret sitt som indeks. I del C så regnes det ut «pekere» ut fra count, for å bestemme hvor sifferne skal settes inn. Måten jeg gjorde dette på var at jeg loopet over hele count og summerte verdiene fra alle trådene i det globale dobbeltarrayet. Måtte også legge til summen av verdiene fra trådene med lavere indeks en det som jobber for å offsette, slik at det blir plass siffer for de lavere trådene. Etter dette brukte jeg det lokale count arrayet som jeg laget til å sette tilbake sifferne.

Implementasjon
Jeg startet med å lage den parallelle metoden som finner max verdi, den er ganske triviell og har forklart en del hvordan jeg gjorde dette over. Etter dette så fant jeg hvor mange bits som trengs for å representere det største tallet. Deretter delte jeg antall bits på en konstant (bestemmer ca. hvor mange bit som er per siffer) for å finne ut hvor mange siffer som skal brukes. Deretter distribueres bitene som skal brukes i et array, dette løkkes over(og flyttes med en skift variabel) og brukes i kall på den parallelle radix sorteringen. Før trådene som sorterer startes, regner jeg ut start og stopp indekser som sendes inn til trådene samt indeksen for tråden. Step B av algoritmen er nå, når de teller frekvens av sifferene. Da bruker trådene start og stop verdiene til å løkke over sin del av a[] og setter inn i et lokal count array, etter at en tråd er ferdig med denne løkken vil den sette inn det lokale arrayet i et globalt dobbel count array på sin trådindeks. Etter dette synkroniseres trådene slik at alle trådene er ferdig med å telle når neste steg begynner. Etter dette så er det Step C, som starter med at trådene «nuller» ut det lokale count arrayet som ble laget tidligere ettersom det er her «pekerne» nå skal lagres. Dette gjøres ved at det løkkes over alle indeksene til det lokale count arrayet, også summeres verdiene fra 0 opp til indeksen for alle trådene sin del av globalCount[][] i tillegg summeres verdien på indeksen for alle trådene mindre enn den nåværende tråden sin del av globalCount[][]. Den siste delen er offsetten som jeg nevnte tidligere. Det er ikke noe problem at alle trådene jobber på den samme globalCount[][] ettersom de ikke endrer på det, men bare leser fra det og summerer lokalt. Etter dette så er Step D som bare er å sette inn i arrayet, ved å bruke det lokale count arrayet som nå ble laget. Trodde opprinnelig at jeg behøvde å ha en CyclicBarrier mellom C og D her, men ettersom alle siffer for alle trådene blir mappet til forskjellige plasser så er det bare å sette inn når den har regnet ut hvor den skal sette inn. Etter at metoden som kaller på denne har løkket over alle sifrene, returner den det sorterte arrayet.

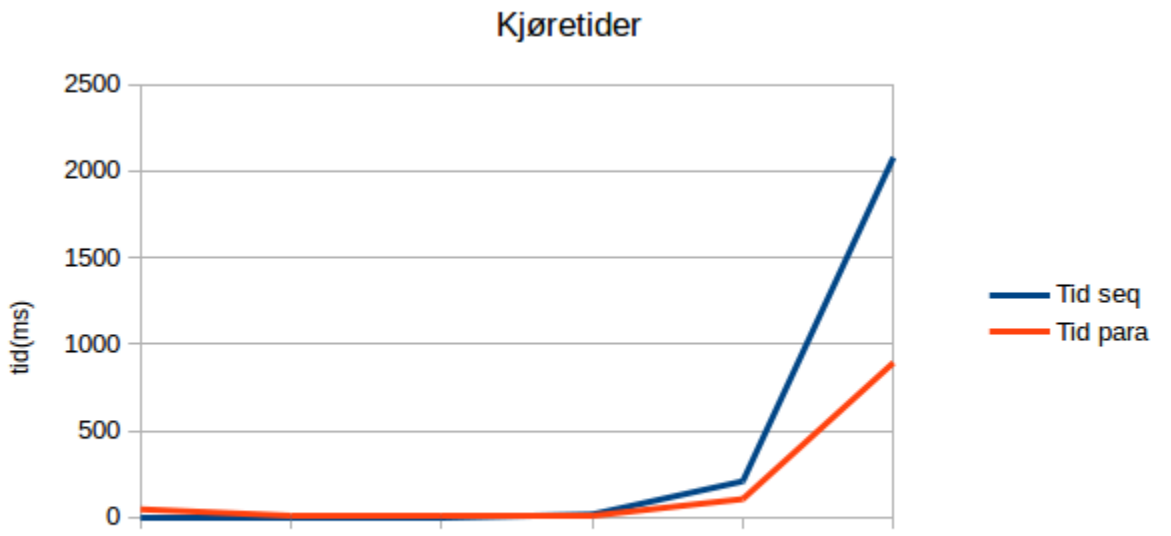
Målinger

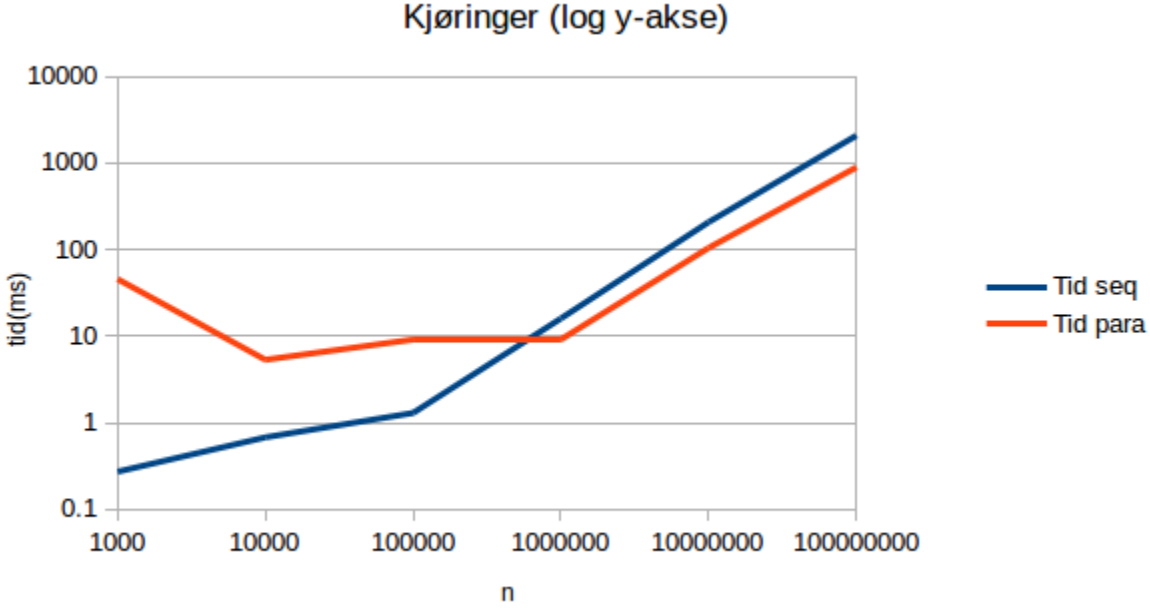
Her er en tabell over kjøretider og speedups, samt screenshot av disse kjøringene:

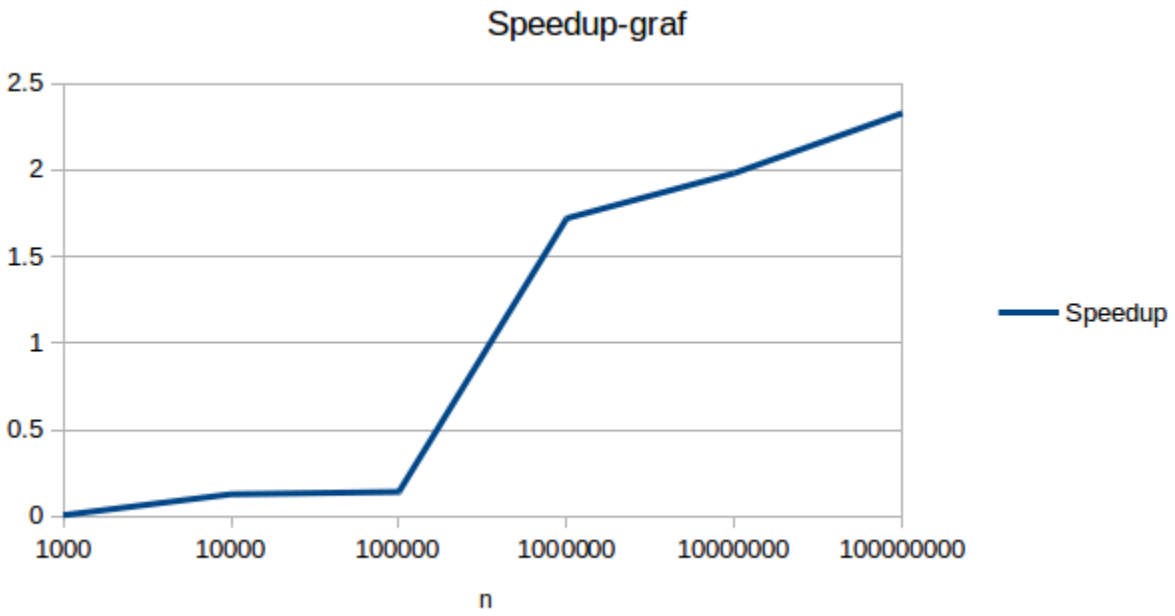
n	Tid seq	Tid para	Speedup
1000	0.271	45.695	0.005930627
10000	0.676	5.331	0.126805477
100000	1.293	9.185	0.140772999
1000000	15.954	9.27	1.721035599
10000000	207.541	104.71	1.9820552
100000000	2078.86	892.768	2.328555683

```
rubenes@rubenes-K55A:~/Dropbox/Sem6/IN3030/Oblig4$ java -Xmx7500M Oblig4 1000 1337 8
n: 1000 seed: 1337 kjerner: 8
Tid seq: 0.271753ms Tid par: 45.695924ms Speedup: 0.01
rubenes@rubenes-K55A:~/Dropbox/Sem6/IN3030/Oblig4$ java -Xmx7500M Oblig4 10000 1337 8
n: 10000 seed: 1337 kjerner: 8
Tid seq: 0.676988ms Tid par: 5.331114ms Speedup: 0.13
rubenes@rubenes-K55A:~/Dropbox/Sem6/IN3030/Oblig4$ java -Xmx7500M Oblig4 100000 1337 8
n: 100000 seed: 1337 kjerner: 8
Tid seq: 1.293202ms Tid par: 9.185405ms Speedup: 0.14
rubenes@rubenes-K55A:~/Dropbox/Sem6/IN3030/Oblig4$ java -Xmx7500M Oblig4 1000000 1337 8
n: 1000000 seed: 1337 kjerner: 8
Tid seq: 15.954663ms Tid par: 9.270052ms Speedup: 1.72
rubenes@rubenes-K55A:~/Dropbox/Sem6/IN3030/Oblig4$ java -Xmx7500M Oblig4 10000000 1337 8
n: 10000000 seed: 1337 kjerner: 8
Tid seq: 207.541024ms Tid par: 104.710512ms Speedup: 1.98
rubenes@rubenes-K55A:~/Dropbox/Sem6/IN3030/Oblig4$ java -Xmx7500M Oblig4 100000000 1337 8
n: 100000000 seed: 1337 kjerner: 8
Tid seq: 2078.860626ms Tid par: 892.768681ms Speedup: 2.33
rubenes@rubenes-K55A:~/Dropbox/Sem6/IN3030/Oblig4$
```

Her er grafer laget ut fra tabellen:







Ut fra grafene kommer det ganske tydelig frem at den parallelle metoden begynner å lønne seg når n=1000000 (1mill). Og speedupen fortsetter å bli noe større ved høyere n. Opp til denne n verdien taper vi altså for mye tid ved å sette opp den parallelle algoritmen, og det lønner seg da heller å kjøre den sekvensielle algoritmen. Jeg kjørte programmet med 8 kjerner, ettersom CPU-en min har 8 tråder.

Konklusjon

Det viste seg at den parallelle sorteringen lønte seg for alle n verdiene større enn 1mill, som forventet så var den tregere for de små verdiene. Det var interessant å se at den hadde ganske bratt utvikling, ved at den var mye tregere opp til 100 000 og for høyere enn dette så ble speedupen bare bedre og bedre.