Problem 1

Her gjør vi noe a la det samme som oppgave 2 forrige uke, men siden oppgavene tar forskjellig tid (i motsetning til forrige uke) så burde vi alltid prioritere de oppgavene som er del av den "kritiske linjen". Vi setter opp tabeller, til venstre er tiden som er gått og til høyre er hvilke oppgaver som blir gjort, hvis det er tid igjen på en oppgave så står tiden som er igjen i parantes. Først for to arbeidere, a)

tid	arbeider 0	arbeider 1
10	3	4
20	1	2
26	5	6 (3)
29		6
37	7	

b)

tid	arbeider 0	arbeider 1
10	1	2
16	5	3 (4)
20	4 (6)	3
26	4	6 (5)
31		6
38	7	

Så for tre arbeidere, a)

tid	arbeider 0	arbeider 1	arbeider 2
10	2	3	4
19	6	1 (1)	
20		1	
26	5		
34	7		

b)

tid	arbeider 0	arbeider 1	arbeider 2
10	1	2	3
16	5	4 (4)	
20	6 (7)	4	
27	6		
34	7		

Så for fire arbeidere, a)

tid	arbeider 0	arbeider 1	arbeider 2	arbeider 3
10	1	2	3	4
16	5	6 (3)		
19		6		
27	7			

b) Vi legger merke til at 34 som vi fikk i forrige oppgave for graf b) er det samme som den kritiske linjelengden for grafen og flere arbeidere vil derfor ikke hjelpe, svaret blir altså 34 for fire arbeidere her.

Problem 2

Definisjonen av begrepene finner du i boka, her er fasiten, en kolonne representerer en graf og en rad representerer en av oppgavene.

	(a)	(b)	(c)	(d)
1.	8	8	8	8
2.	4	4	7	8
3.	$\frac{15}{4}$	$\frac{15}{4}$	2	$\frac{15}{8}$
4.	8	8	3	2
5.	$\frac{15}{8}$, 3, $\frac{15}{4}$	$\frac{15}{8}$, 3, $\frac{15}{4}$	$\frac{7}{4}$, 2, 2	$\frac{15}{8}$, $\frac{15}{8}$, $\frac{15}{8}$

Problem 3

Vi skal bevise at

$$\left\lceil \frac{t}{l} \right\rceil \leq d \leq t - l + 1$$

for en oppgave-avhengighetsgraf hvor t er antall oppgaver, d er største grad av samtidighet og kritisk linjelengde l. Først kikker vi på

$$\left\lceil \frac{t}{l} \right\rceil \leq d$$

vi kjenner fra boka at $\frac{t}{l}$ er gjennomsnittlig grad av samtidighet, nødvendigvis vil den da være mindre eller største grad av samtidighet siden de andre mulige gradene av samtidighet kan bare gjøre gjennomsnittet mindre. Så vi har $\frac{t}{l} \leq d$, i tillegg er d et heltall, så vi kan i de tilfellene hvor $\frac{t}{l}$ ikke er det (da gjelder $\frac{t}{l} < d$) gå opp til nærmeste heltall.

Så kikker vi på

$$d \leq t - l + 1$$

anta for en eller annen $k \in \mathbb{N}$ at vi har en graf med k noder hvor alle nodene enten er del av de nodene vi teller i største grad av uavhengighet eller er del av den kritiske linjen (graf (c) fra forrige oppgave er en slik graf). For en slik graf har vi $k = d + l - 1$, altså at det totale antallet noder er alle nodene vi teller som del av største grad av uavhengighet og alle nodene i den kritiske linjen minus den ene noden som vi teller i begge.

Hvis vi har en **hvilken som helst** graf med k noder og legger til en node slik at den enten kan telles som del av den største graden av samtidighet eller blir del av den kritiske linjen får vi en graf med $k + 1$ noder hvor enten d øker med én, eller en graf med $k + 1$ noder hvor l øker med en, altså er forskjellen mellom det totale antallet noder uforandret i forhold til $d + l$, så får typen grafer vi nevnte tidligere får vi $k + 1 = (d + 1) + l - 1$ eller $k + 1 = d + (l + 1) - 1$, i begge tilfeller er likheten bevart.

Så hvis vi tenker oss en graf med 1 node, så har den $t = 1$, $l = 1$ og $d = 1$, vi ser at $1 = 1 + 1 - 1$ holder i dette tilfellet, altså vil alle grafer vi bygger ut fra denne ene noden oppfylle likheten så lenge vi bare legger til noden på måten vi tidligere nevnte.

Hvis vi derimot tar en graf og legger til en node slik at den hverken kan telles som del av den største graden av samtidighet eller slik at den blir del av den

kritiske linjen så vil det totale tallet av noder t øke, mens d og l blir uforandret, for disse grafene vil altså $t > d + l - 1$.

Så for å oppsummere, for noen spesielle type grafer har vi $t = d+l-1$, dette inkluderer basisgrafen med bare én node, alle transformasjoner vi kan gjøre på denne grafen gjennom å legge til noder vil enten gjøre at likheten blir uforandret eller øke t i forhold til d og l, dette medfører at vi får $t \geq d+l-1$ for alle mulige oppgave-avhengigshetsgrafer, dette er ekvivalent med $d \leq t - l + 1$.

Problem 4

Her har vi i hovedsak to ting å gjøre, først må vi per hus regne ut og lagre lengden fra huset til stasjonen, så må vi for alle lengder finne den minste. Siden vi ikke har noe avhengighet mellom husene når vi skal regne ut lengden til stasjonen per hus så kan vi trivielt parallellisere den delen, det å finne den minste lengden av dem alle kan også gjøres i parallell ved en binær reduksjon med $<$ operatoren.

Så til slutt lager vi pseudokode for å illustrere algoritmen

```
parallel_for (i = 0; i < number_of_houses; i++) {
  dx = houses[i].x - station.x;
  dy = houses[i].y - station.y;
  distances[i] = sqrt(dx*dx + dy*dy);
}

least = parallel_reduction(distances, number_of_houses, minimum);
```

en naiv implementasjon av denne algoritmen med MPI ville ikke vært det mest effektive, når man vet mer om miljøet man kjører i eller gjør antakelser (for eksempel at MPI har flere prosesser som hver har en del av arrayet) er det ting man gjøre bedre, for eksempel en skisse som bruker MPI kan gjøre noe slikt (vi antar at prosess 0 har houses arrayet til å begynne med og vi ønsker til slutt å ende opp med svaret på prosess 0). Først vil prosess 0 regne ut en blokkfordeling som den kan bruke til å sende ut jevnt fordelte blokker av houses arrayet med `MPI_Scatterv`, nå har hver prosess en del av houses arrayet, hvis hver prosess da sparer på den minste distansen mens de regner ut distansene så blir vi kvitt distances arrayet og vi gjør bare en `MPI_Reduce` med én distance variable per prosess.

Problem 5

I problemer relatert til strengsøk er det vanlig å refere til det man søker etter som nåla, og det man leter i som høystakken. Her er basisoppgaven å sammenlikne nåla med en del av høystakken, samt å holde orden på eventuelle treff. Det holder ikke å fordele høystakken med en helt naiv blokkfordeling, vi må ha overlapp mellom blokkene som blir fordelt slik at hele nåla kan testes mot alle mulige posisjoner i høystakken også på enden av blokkene som blir fordelt.

Så hvis vi reintroduserer den vanlige blokkfordelingen for en prosess med

rang k ut av P prosesser og et array med n elementer så har vi

$$\begin{aligned} L(k,P,n) &= \left\lfloor \frac{k \cdot n}{P} \right\rfloor \\ H(k,P,n) &= L(k+1,P,n) \\ S(k,P,n) &= H(k,P,n) - L(k,P,n) \end{aligned}$$

så sier vi at lengden til nåla er l_n og lengden til høystakken er l_h da er det altså $l_h - l_n + 1$ forskjellige startposisjoner nåla kan ha når vi skal søke. Fordelingen vi da ønsker er at prosess k ut av P prosesser skal få elementer fra høystakken som begynner å telle fra indeks $L(k,P,l_h-l_n+1)$ med lengde $S(k,P,l_h-l_n+1)+l_n-1$. Da har vi ordnet overlapp og blokkene som blir fordelt er også så like som mulig i størrelse.

Algoritmen vi da har er å fordele med overlapp, la hver prosess søke i sin lokale del og telle antall treff, for å så til slutt summere treffene de fikk. En svakhet her er at overlappet som fører med hver prosess gjør at algoritmen blir mindre effektiv med tanke på minne når antallet prosesser økes, I verste tilfelle så fordeler vi hvert indre element i høystakken opp til l_n ganger. En annen ting som er verdt å nevne her er at det finnes bedre enn naive algoritmer for strengsøk i den lokale høystakken, det finnes blant annet også parallelle versjoner av boyer-moore algoritmen.

Problem 6

Hvis vi bare ønsker å finne den aller første instansen av nåla i høystakken så burde vi endre oppdelingen av høystakken fra en blokkfordeling til en syklisk fordeling slik at alle prosessene først bare jobber i begynnelsen av høystakken, vi må også i hver iterasjon sjekke om en annen prosess har funnet nåla slik at alle prosessene blir enige om å terminere algoritmen.

Problemer med dette er at hvert felt som blir fordelt igjen trenger en overlapp, så de mulige problemene vi hadde med minne blir forverret av en syklisk fordeling. Alle til alle kommunikasjon er også potensielt dyrt hvis det skjer ofte, spesielt går dette utover parallelliteten da prosessene blir synkronisert globalt.

Vi kan lindre disse problemene ved å bestemme oss for noen parametre, vi kan lindre minneproblemene ved å fordele større felt i den sykliske distribusjonen, vi bytter da sjansen til å finne den første instansen av nåla litt fortere mot å bruke mindre minne. Vi kan lindre synkroniseringskostnaden i hver iterasjon med å bestemme oss for å bare sjekke om en annen prosess har funnet nåla etter et gitt antall iterasjoner, vi bytter da sjansen til å avslutte søket fortere globalt mot mindre kommunikasjon mellom prosessene og derfor potensielt raskere søk lokalt.

Alle disse parameterene har altså fordeler og ulemper og å bestemme dem nøyaktig er vanskelig, de vil ha forskjellig verdi per høystakk og nål og en kan bare få gode utgangspunkt ved å profilere programmet med en mengde realistiske verdier.