UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

Parallelle beregninger med MPI i delt og distribuert minne

En sammenligning av fire problemer implementert på multikjerne og maskinklynge

Lars-Erik Opdal

Masteroppgave våren 2016

Parallelle beregninger med MPI i delt og distribuert minne

Lars-Erik Opdal

13 Mai, 2016

Sammendrag

Parallel Programming has been thrown upon us by the industry which can't deliver what we really want - ever faster CPUs. Instead they give us more computing cores at a steadily decreasing cost. Programmers are now forced to write parallel programs if we want to solve bigger problems. The purpose of this master thesis is to compare the two most popular programming platforms (cluster and multicore/shared memory) among the four available platforms (Cluster, multicore, GPU and cloud/internet). The comparison if not done solely on speedup/performance, but also on development time, ease of use, debugging time and size and type of problem solved. The master thesis will program a small selection of parallel algorithms on the four problems: Sorting, Matrix Multiplication, Delaunay Triangulation and the Mandelbrot set.

Forord

Jeg ønsker å takke mine veiledere Arne Maus og Viktoria Stray for den uvurderlige hjelp jeg har fått gjennom gode kommentarer og viktige bemerkninger underveis, samt for all kunnskap og erfaring de har delt i arbeidet med min masteroppgave. En spesiell takk rettes også til Oddleif Wille som var min klasseforstander på ungdomskolen. Jeg er i tillegg svært takknemlig for alle venner og familiemedlemmer som har bidratt til og støttet meg i arbeidet med denne oppgaven.

Innhold

Sammendrag 2
1. **Introduksjon** 8
1.1. Semiparallellitet – Ressursfordeling i OS med tråder og prosesser .. 10
1.2. Parallellisering 11
1.3. Sammenligning av to metoder for parallelle beregninger 15
1.4. Brukergrensesnitt for å sende og motta meldinger 17
1.5. Programmeringsparadigmer 17
1.6.Ressursbruk 18
1.7. Behov for mer datakraft 18
1.8. En ny type datamaskiner kommer 19
1.9. Oppsummering 20
2. **Bakgrunn** 21
2.1. Abelklyngen ved UiO 21
2.1.1. Systemspesifikasjoner 21
2.1.2. Abelklyngens topologi 22
2.1.3. Maskinvare på Abelklyngen 23
2.2. Sammenligning av to metoder for å gjøre parallelle beregninger 23
2.2.1. Multikjerne med felles minne 24
2.2.2. OpenMP 25
2.2.3. Maskinklynge med distribuert minne 25
2.3. Problemtyper 26
2.3.1. P1 - Sortering 26
2.3.2. P2 - Matrisemultiplikasjon 27
2.3.3. P3 - Delaunay triangulering 30
2.3.4. P4 - Fraktaler 37
3. **Metode** 42
3.1. Bruk av Abelklyngen 43
3.1.1. Kjøring av job-skript på Abelklyngen 44
3.1.2. SLURM - køsystemet på Abelklyngen 44
3.1.3. Eksempel på hvordan et SLURM job-skript kan være 45
3.1.4. Array jobs med *arrayrun* 45
3.1.5. Relevante problemstillinger fra brukere på Abelklyngen 46
3.1.6. Strømbruk - Abelklyngen 48

3.2. P0 – Kommunikasjonstest 50
3.2.1. Ping-testen 50
3.2.2. Java på Abelklyngen 50
3.3. P1 – Sortering 51
3.3.1. Mergesort med C og MPI 51
3.3.2. Oppbygging av struktur for mergesort 52
3.3.3. Sortering i Java 53
3.4. P2 – Matrisemultiplikasjon 53
3.4.1. Design av parallellisering 53
3.5. P3 – Delaunay triangulering 54
3.5.1. Design – sekvensiell versjon 54
3.5.2. Implementasjon - sekvensiell versjon 54
3.5.3. Oversetting av DT fra Java til C 54
3.5.4. Design av parallellisering 63
3.5.5. Oppdeling av punkter mellom prosessene 64
3.5.6. Reduksjon - samle data og skrive ut til fil 68
3.5.7. Innsamling av data og skriving til fil 71
3.5.8. Løsning med en serie av filer 72
3.5.9. Arbeidsflyt for testing og verifisering av resultat 73
3.6. P4 – Fraktaler 74
3.6.1. Algoritmen for Mandelbrotmengden (MS) 74
3.6.2. Parallellisering av MS 74
3.6.3. Kjøring av algoritmen (MS) 76
4. **Resultater** 79
4.1. P0 – Kommunikasjonstest 79
4.1.1. C-MPI med ping-test 79
4.1.2. Java-MPI med ping og ping-pong 79
4.1.3. Ping-test med J-MPI 79
4.1.4. Ping-pong-test med J-MPI 79
4.1.5. Ping-pong-test med C-MPI på Abelklyngen 80
4.1.6. Kostnadsanalyse for send/recv og ping-pong 80
4.2. P1 – Sortering 80
4.2.1. Java-MPI med mergesort på multikjerne 80
4.2.2. Java-MPI med mergesort på Abelklyngen 80

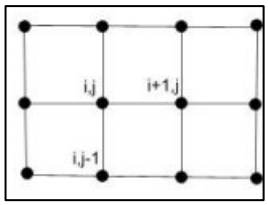
4.2.3. Dårlig resultat – stopper her 81
4.2.4. Kjøring av parallell mergesort med C-MPI 81
4.3. P2 – Matrisemultiplikasjon 82
4.3.1. Matrisemultiplikasjon på multikjerne 82
4.3.2. Matrisemultiplikasjon på Abelklyngen 82
4.4. P3 – Delaunay triangulering 83
4.4.1. Delaunay triangulering i delt minne 83
4.4.2. Delaunay Triangulering på Abelklyngen 84
4.5. P4 – Mandelbrotmengden (MS) 89
4.5.1. Drøfting av resultat 91
5. **Diskusjon** 93
5.1. D1 – Drøfting av brukervennlighet 93
5.2. D2 - Drøfting av ytelse 96
6.**Konklusjon** 101
6.1. Brukervennlighet 101
6.2. Ytelse 101
7. **Videre arbeid** 102
Vedlegg 1 103
Vedlegg 2 104
Ordliste 104
Ordforklaring 105
SI grunnenheter i oppgaven 105
SI prefikser i oppgaven 106
Figurer 107
Tabeller 108
Referanser 109

1. Introduksjon

Fordi noen typer problemer krever mye datakraft trenger man parallellprogrammering. Moores lov vil ikke være gjeldende i fremtiden[1]. Derfor trenger vi parallellprogrammering for å utnytte maskinvare på en effektiv måte, siden økning i prosessorkraft ikke vil eskalere slik den har gjort i perioden 1950-2000. Noen typer problemer er så store at det er behov for mer datakraft enn én maskin har. For å løse problemer som værmelding, kryptografi og simulering av molekylær dynamikk må man sette sammen flere maskiner i nettverk. Parallellprogrammering er et konsept som har blitt introdusert av IT-industrien. Det er et biprodukt av at denne industrien ikke klarer å mette det virkelige behovet, som er mer datakraft og raskere maskiner. Produsenter av maskinvare lover hele tiden nye multikjerne prosessorer og at ytelsen stadig vekk dobler seg[2]. Det forbrukerne får er flere kjerner eller beregningsenheter til en stadig lavere pris. Dette fører til at utviklerne tvinges til parallellprogrammering for å løse større problemer. Hensikten med denne masteroppgaven er å sammenligne de to mest brukte plattformene for parallellprogrammering, som er delt minne og distribuert minne. Det vil bli laget en beskrivelse av design, og implementasjon av noen parallelle algoritmer vil bli utført. Jeg skal undersøke ytelse og brukervennlighet i de to paradigmene.

Oppgaven kommer til å fokusere på beregninger som blir utført i en klynge, sammenlignet med beregninger som blir utført på en multikjerneprosessor. Oppgaven vil også sammenligne to forskjellige programmeringsmiljøer. Det første programmeringsmiljøet som blir beskrevet er *Message Passing Interface* (MPI). Dette er en meldingsbasert programmeringsteknikk som brukes på Abelklyngen i et Unix-miljø hovedsakelig med språket C. Det andre miljøet vil være en datamaskin med multikjerneprosessor. Det vil bli utført en teoretisk og praktisk sammenligning av noen kjente algoritmer, med fokus på å sammenligne eksekveringstid og *speedup* for datasett av forskjellig størrelse. Tidsbruk for feilsøking vil også bli utført og evaluert.

Det synes rimelig å skille mellom fire typer av datamaskiner for parallelle beregninger. Denne oppgaven fokuserer på forskjellene mellom dem og hva de egner seg til. Fokuset ligger på to av disse maskinarkitekturene, og de vil bli sammenlignet med bruk av programmeringseksempler. Den første måten er å sette sammen datamaskiner i en klynge. Det er flere måter å gjøre dette på: Tradisjonelt sett har det mest vanlige vært å sette opp maskinene i et rutenett, det vil si at hver datamaskin er koblet til *fire* andre, med unntak av den som er i hjørnet som bare får *to* tilkoblinger og de som ligger langs kanten som får *tre* naboer, se *Figur 1*. Det finnes også stjernenett, der datamaskiner står koblet sammen med *åtte*, *tre* og *fem* naboer.



Figur 1. 2D-mesh Topologi. Et eksempel på hvordan maskiner kan settes sammen til en klynge.

Dersom det i tillegg er *wraparound*-struktur får alle datamaskiner i rutenettet *åtte* naboer. Dette kan implementeres med *switcher* som bruker den fysiske adressen (MAC-adressen) til mottakeren for å avgjøre hvor data skal sendes. Det er en *switch* i hver *hylleseksjon* med noder og en serie med *switcher* i en egen hylleseksjon som er koblet til de andre hylleseksjonene med nodene. Det er denne teknikken som er utnyttet i Abelkyngen, se *Figur 6*. Den andre måten innebærer bruk av flere tråder som kan jobbe parallelt med samme oppgave på en multikjerneprosessor. I en multikjerneprosessor har kjernene felles hukommelse. Det gjør at kommunikasjon mellom dem blir raskere og at det blir mindre forsinkelser enn i en maskinklynge. Den tredje måten handler om å bruke beregningsenheten på en annen komponent i datamaskinen, dette blir ofte betegnet som en akselerator. Et eksempel på dette er et grafikkort med en *Graphical Prosessing Unit* (GPU).

En GPU er velegnet til å gjøre samme instruksjon samtidig på mange tusen dataelementer. For eksempel matrisemultiplikasjon. Den eneste instruksjonen som skal utføres er å multiplisere og addere rad- og kolonneelement til $n \times n$-matrisen. Dette skal gjøres $n \times n$ antall ganger. En GPU egner seg fint til løsning av likningssett med *Gauss-eliminasjon* der n er stor, og til andre typer numeriske beregninger som krever lite kommunikasjon under beregning[3]. En GPU er klassifisert som *Single Instruction Multiple Data* (SIMD) jfr. Flynn's Taxonomy, se *Figur 2*.

Andre formål kan være algoritmer som bruker *rå kraft*. Det er en algoritmeteknikk som utforsker alle muligheter med «prøve og feile-metoden» for å finne en løsning. Men fordi en GPU ikke er en vanlig datamaskin er det visse områder den er svak på. Det er vanskelig å designe algoritmer fordi man må følge mange flere regler, og beregningsenheten kan bare gjøre én instruksjon av gangen. Derfor blir for eksempel en «if else»-betingelse en stor flaskehals fordi programmet må vente på evaluering av *if* før *else*. I tillegg har en GPU generelt tregt minne. Den fjerde måten er å gjøre beregninger på nett-skyen. Det finnes flere portaler på Internett som stiller datakraft til disposisjon. Bioportal er et web-grensesnitt som blir brukt til å gi blant annet CERN og andre aktører tilgang til Abelklyngen[4]. CERN produserer store mengder data som må beregnes, hvis ikke er det ikke hensiktsmessig å spare på dem[5]. Beregning på skyen er en fin måte å få tilgang til beregningskraft og den er velegnet for problemer med mye beregninger og lite I/O-aktivitet. Planet Lab er et annet slikt verktøy[6]. På nettstedet Amazon er det mulig å leie maskiner til å gjøre beregninger. Det kan også virke den andre veien ved at det er mulig å stille sin private

datamaskin til disposisjon slik at den kan kjøre beregning i bakgrunnen, mens man gjør andre ting. Seti[7] er et slikt prosjekt. Det går ut på å lete etter en intelligent form for kommunikasjonssignaler fra verdensrommet. De bruker opptak av elektromagnetisk stråling for å undersøke dette. I denne oppgaven skal jeg prøve å besvare to sentrale spørsmål innen høyytelses-beregning (HPC), nemlig om brukervennlighet og ytelsesforbedring. Jeg skal undersøke brukervennligheten til verktøyene som er tilgjengelig for å gjøre parallelle beregninger og beskrive hvilke ytelsesforbedringer vi får ved å benytte dem.

	Single instruction stream	Multiple instruction streams
Single data stream	SISD	MISD
Multiple data streams	SIMD	MIMD

Figur 2. Flynn's Taxonomy er en beskrivelse av de forskjellige modellene av maskintyper for parallellberegninger med unntak av SISD som naturligvis ikke en parallell maskin.

Problemstilling 1 – Brukervennlighet
Hvor vanskelig er det å ta i bruk HPC med delt og distribuert minne, og hvilken av de to typene er mest tidkrevende å lære seg? Det vil si tid for utvikling og tid for eksekvering sammenlagt. Hva kreves for at de skal fungere effektivt? Hvilke programmeringsparadigmer bør man velge for å oppnå ønsket resultat i denne sammenhengen? Hvorfor er det eventuelt enklere å utvikle programmer for delt minne sammenlignet med distribuert minne og hva er mest tidkrevende å få til å virke? Er objekt-orientering en bedre programmeringsteknikk enn den imperative for å gjøre utvikling av parallelle programmer enklere? Dersom det er mulig å lage prototyper i høynivå programmeringsspråk, kan det kanskje gjøre denne prosessen mindre tidkrevende.

Problemstilling 2 – Vurdering av ytelse
For å sammenligne ytelsen hos de to metodene ser jeg på fire problemer (sending av data, sortering, matriseoperasjoner og beregning av datapunkter for grafikk). Hvilke problemer egner de to typene seg best for? Hvor oppnår vi eventuell speedup og god effektivitet når vi parallelliserer en algoritme? Hva kan simuleres med de to forskjellige typene (med andre ord: Hva kan de brukes til)? Kan beregninger i distribuert minne gi oss mer høyoppløselige og nøyaktige resultater på tidligere løste problemer? Hvilke problemstørrelser kan de to forskjellige typene behandle?

1.1. Semiparallellitet – Ressursfordeling i OS med tråder og prosesser

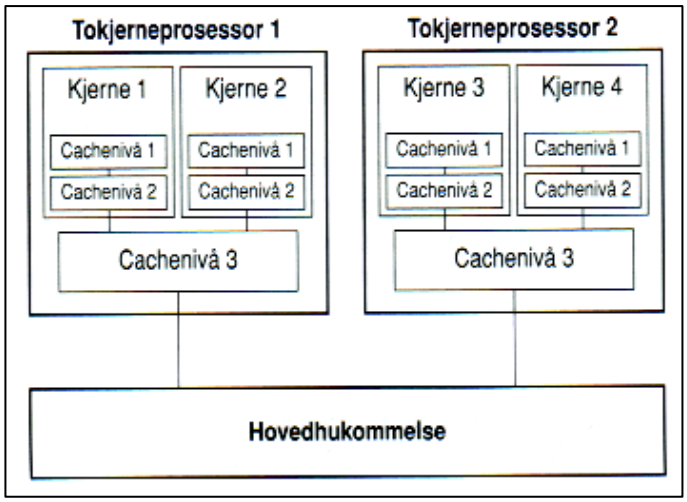
For at ressurser på datamaskinen skal utnyttes best mulig, er det mange prosesser som overlapper hverandre. Dette fører til at maskinvare blir bedre

utnyttet fordi oppgaver som er mer ressurskrevende kan kjøre i bakgrunnen, mens mindre ressurskrevende oppgaver kan kjøres samtidig. En annen fordel er at noen systemer blir lettere å utvikle. Eksempler på dette kan være systemer der mange brukere skal utføre en handling samtidig. Et eksempel kan være et bestillingssystem. Der kan det være lurt å starte en ny tråd for hver bruker slik at det ikke blir kø på bruk av maskinvare. Og fordi disse trådene har tilgang på et felles dataområde, vil det bli mulig å sikre at *to* brukere ikke kjøper samme plass. Dette kalles *concurrency* og i eksempelet over er det kun behov for en beregningsenhet, selv om det kan være flere. Det må ikke forveksles med parallellitet og parallelle beregninger, der man bruker flere beregningsenheter enn *én* for å utføre flere instruksjoner i parallell.

1.2. Parallellisering

Parallellisering går ut på å dele opp et problem i mindre biter slik at flere av oppgavene kan bli utført samtidig. Parallellisering kan få noen typer problemer til å bli løst raskere, mer nøyaktig og med høyere oppløsning. Et eksempel på en slik beregning kan være et værvarsel for hele jordkloden. Med et overflateareal på $5.10072 \times 10^9 \times km^2$ og en valgt romlig oppløsning på $1 \times 1\ km^2$ for en sektor i et geografisk område blir antall sektorer som må beregnes 5.1×10^8. Vi må også velge antall tids-steg for beregningen. Det vil si om resultatet skal beregnes time for time, minutt for minutt eller i sekunder. Dersom vi forenkler hva som skal beregnes i hver sektor og sier at det er 100 flyttalls operasjoner og 5.1×10^8 antall sektorer vil dette kreve $100 \times 5.1 \times 10^8 \times 3600 = 1.836 \times 10^{15}$ FLOPS = 1836 teraFLOPS hvis vi skal ha en tidsoppløsning på 1 sekund over et intervall på 1 time. En vanlig datamaskin i dag med en 2.6 GHz prosessor har en teoretisk kapasitet på omtrent 10 gigaFLOPS. Det er 1 million ganger mindre enn hva behovet er for dette problemet. I følge Moores lov[8] er det slik at antall transistorer på en prosessor dobler seg hver 18. måned. Etter ca. år 2005 hadde klokkehastigheten kommet opp mot ca. 3 GHz. Det førte til at kjøleteknologien ikke lenger klarte å holde prosessoren stabil. Da vokste det frem en ny type arkitektur, nemlig Dual Core (2 kjerner), senere Quad Core (4 kjerner) og til slutt Octa Core (8 kjerner) under fellesbetegnelsen multikjerne. Denne utviklingen førte til at beregninger utført av en datamaskin nå forekommer mer i parallell for å utnytte den tilgjengelige datakraften slik at flere oppgaver kan gjøres samtidig.

Et dataprogram bruker minne for å gjøre beregninger og for å flytte eller lagre data. Datamaskinens minne er delt opp i *seks* nivåer (mens cache har *tre* nivåer), se *Figur 3*. Register og cache brukes hovedsakelig til å utføre instruksjoner som beregner data, for eksempel logiske og matematiske operasjoner. Hovedminne blir brukt som en bro mellom CPU og harddisk for å kunne ta ut resultater fra en beregning og lagre dem på disk.



Figur 3. Minnemodell for en Quad Core-prosessor.[9].

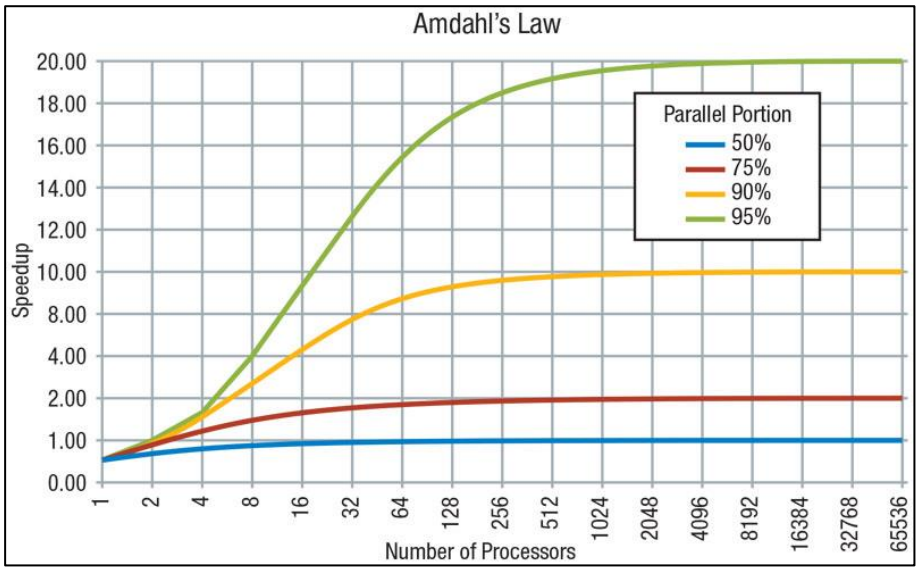
Når et program eller en bruker jobber med å beregne felles data, kan det oppstå problemer som *vranglås* og *datakappløp*. Det vil si at et program ikke får tilgang på data slik som forventet. For å løse eventuelle problemer med dette, må data synkroniseres på en hensiktsmessig måte. Det er viktig at dataene blir innrapportert synkront, slik at data ikke kan bli overskrevet eller feilsortert.

Hastighetsforbedring ved parallellisering er ofte begrenset i forhold til Amdahls lov. Amdahls lov stammer fra norsk-amerikaneren Gene Amdahl [10] som laget store vektormaskiner. Det vil si mange kjerner på en rekke med delt minne, slik som en GPU. Dette tilsvarte en SIMD-maskin. Disse maskinene var meget godt egnet til matrisemultiplikasjon. Loven beskriver hvor stor *speedup* man kan oppnå ved å parallellisere et program, se *Likning* 1.

I 1967 publiserte Amdahl en artikkel der han samlet data fra flere simuleringer av naturlover og fysiske fenomener. Likningen blir aldri nevnt i artikkelen, men kan utledes fra grafen[11]. Amdahls resultat viser en graf for prosentandel det er mulig å kjøre i parallell på en datamaskin. Den har senere blitt omtalt som Amdahls lov. I en algoritme er det alltid noen punkter som må utføres sekvensielt på grunn av en datamaskins arkitektur og oppbygging.

$$S(N) = \frac{1}{(1-\mathrm{p}) + \frac{\mathrm{p}}{\mathrm{N}}} \tag{1}$$

I Amdahls lov er *P* er andelen instruksjoner som kan skje i parallell, og *N* er antall beregningsenheter. *Speedup* kan generelt defineres som $\mathrm{S(n)} = \frac{\mathrm{T_1}}{\mathrm{T_n}}$, hvor $\mathrm{T_1}$ er sekvensiell tid for en algoritme og $\mathrm{T_n}$ er tiden det tar med n antall prosessorer. *Figur 4* viser en graf for speedup med Amdahls lov.

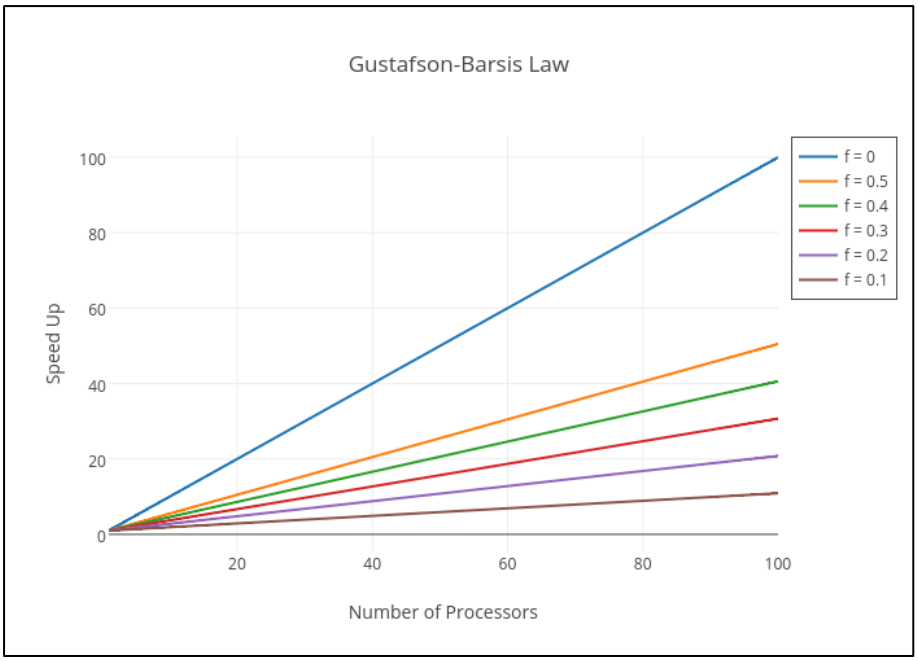


Figur 4. Speedup som funksjon av antall kjerner og andel parallell kode i programmet[12].

Gustavsons lov[13] sier at beregninger som har store datasett med tilfeldig innhold kan parallelliseres mer enn det Amdahls lov sier. De to lovene står dermed i kontrast til hverandre, siden Amdahls lov sier at effektiviteten av parallellisering begrenses til andelen som må utføres i sekvens. Gustavsons lov påpeker begrensningen ved Amdahls lov siden denne ikke utnytter fordelen av ekstra datakraft som blir tilgjengelig når flere maskiner kobles sammen og når problemet blir større. Siden Amdahls lov ikke tar hensyn til økning i problemstørrelsen *N* når vi legger til *P* antall prosessorer, så foreslår Gustavson en likning for skalert speedup, se *Linking* 2.

$$S(P) = P - \alpha \times (P - 1) \qquad (2)$$

S er hastighetsøkning, P er antall prosessorer, og α er den delen som ikke kan parallelliseres. Dersom det gjøres et forsøk på å optimalisere en beregning, vil noen følge Amdahls lov, mens andre følger Gustavsons. Dette er avhengig av størrelsen på datasettet og antall kjerner og særlig om kjøretiden for den sekvensielle delen av programmet har en annen orden og lavere, enn den parallell delen.Hvis det siste er tilfellet, vil Gustavsons lov gjelde. *Figur 5* viser en graf for *speedup med Gustavsons lov*.



Figur 5. Speedup-funksjon av antall kjerner[14].

En tredje måte å måle forventet ytelse av parallellisering er Karp-Flatt metrikk. Den beskriver i hvilken grad koden er parallelliserbar. Gitt at vi har *speedup* S med P prosessorer, der P er større enn *én*. Da kan den sekvensielle delen defineres slik av e. Se likning 3.

$$e = \frac{\frac{1}{S} - \frac{1}{P}}{1 - \frac{1}{P}} \tag{3}$$

Lave verdier av e gir bedre parallellisering. Den sekvensielle delen e blir som regel ikke beregnet når vi skal definere *speedup* eller effektivitet. Selv om det er et ledd for sekvensiell del i Amdahls lov, blir den ikke beregnet, bare brukt som en konstant. Derfor tar for eksempel ikke Amdahls lov hensyn til lastbalanse og annet overhead i kommunikasjon. Dersom vi har et problem med gitt størrelse som skal kjøres i parallell er det naturlig å forvente seg at effektiviteten synker når antall prosessorer som brukes går opp. Når e er beregnet kan vi avgjøre hvorfor algoritmen har en synkende effektivitet. Da er det mulig å avgjøre om det skyldes graden av parallellitet som kan oppnås med koden, eller forsinkelse som følger av mer kommunikasjon på grunn av flere prosessorer som blir brukt. Dette kan være en fordel å undersøke før man setter i gang med implementering og kjøring[15].

1.3. Sammenligning av to metoder for parallelle beregninger

Den første metoden med maskinklynger brukes dersom et stort problem skal løses og det er en fordel at det ikke er et stort krav om kommunikasjon. Et eksempel på dette kan være statistiske beregninger som kreves for å lage en værmelding. Den andre metoden med *delt minne* er best egnet til å løse mange små problemer samtidig, der det kreves mye kommunikasjon under beregning av data, for eksempel sortering.

Når man jobber med parallellberegninger er hastigheten på transaksjonene veldig interessant fordi det hovedsakelig er dette som er kostbart på en klyngemaskin. Dette kan undersøkes nærmere med *ping-pong*-testen som funksjon av pakkestørrelsen og antall pakker. Denne oppgaven vil fokusere på de to raskeste. I et multikjernemiljø kan det utføres en *ping-pong*-test på følgende måte: *To* prosesser startes opp fra main til et C-program, den ene sier *send*, og *recv* når den mottar, mens den andre sier *recv* og så *send* når den er klar. Når prosessene er ferdig med å kommunisere med hverandre kan dette skrives ut på skjermen som *send-recv-send-recv* fra den første prosessen, fordi meldingen med data da har kommet frem og tilbake mellom dem. Tiden dette tar blir registrert og dette kan gjøres med forskjellige størrelser på dataene for å få en modell av tidsaspektet det gir. *Ping-pong*-testen kan også utføres på en multikjernemaskin og i en maskinklynge. Testen kan gi oss tiden det tar å sende data fra node til node. En klynge har ofte en høyhastighets optisk tilkobling mellom nodene. På Abelklyngen heter det *Infiniband.* Et viktig poeng her er at *Infiniband* er raskt, men det tar tid å laste data inn i hukommelsen. Abelklyngen utfører denne type test på 0,95 mikrosekunder innenfor hver *hylleseksjon*, det vil si at maskinene står ved siden av hverandre. Alle prosessorene er like raske, ca. 2.6 GHz, så det er kostnadene på transaksjonen som er avgjørende, det vil si tiden som blir brukt. Forøvrig merker vi oss at kommunikasjon mellom *to* nabo-noder i en hylleseksjon tar omtrent samme tid som 2000 instruksjoner. Det har kommet fram gjennom egne målinger jeg har gjort på systemet. Disse resultatene blir presentert og drøftet senere i oppgaven, under resultat-kapittelet.

Fordelen med klynge er at det er større tilgang til beregningskraft, det vil si antall kjerner og mye minne. Dersom det er lite kommunikasjon av data, kan det argumenteres for at en klynge er det beste valget. Men dette trenger ikke å være tilfellet dersom klyngen har *Infiniband* mellom nodene, det vil si en rask optisk kommunikasjonsprotokoll som sørger for at forsinkelsen blir minimal, men likevel langt større enn i en maskin med delt minne. Dersom det er mye kommunikasjon kan det argumenteres for at en maskin med multikjerne er mest effektiv fordi det ikke er behov for å sende data til andre enheter som er kostbart. Men i det store og hele blir det størrelsen på problemet, og ikke hvilken type maskin det er, som har noe å si. Med andre ord: Et stort problem krever en stor maskin, det vil si en klynge, og et mindre problem blir beregnet mest effektivt dersom det er kort vei mellom

maskinvaren som i en multikjerne. Det er mengden kommunikasjon som blir en avgjørende faktor.

1.4. Brukergrensesnitt for å sende og motta meldinger

MPI er den gjeldende standarden for parallell programmering i distribuert minne[16]. Det er en standard for dataoverføring og et meldingsbasert system designet for kommunikasjon mellom prosesser. Systemet definerer syntaks og semantikken til et bibliotek for å skrive meldingsbaserte programmer i Fortran- og C-språkene. Fortran er et programmeringsspråk som brukes til numerisk analyse og vitenskapelige beregninger. Språket ble utviklet på 50-tallet av IBM for vitenskapelige applikasjoner samt ingeniørberegninger, og har kommet i flere forbedrede versjoner siden da[17].
Fortran blir fortsatt brukt til beregningsintensive områder som værmelding, fluid-dynamikk, teoretisk fysikk og kjemi. Fortran er det mest brukte språket innen høyytelsesberegning og brukes til å lage klassifisering av verdens raskeste klyngemaskiner.

Det finnes mange effektive implementasjoner av MPI som er godt testet og offentlig tilgjengelig. De brukes som en motivasjon for at utviklere skal få interesse for fagfeltet og slik at nye metoder og teknikker blir oppdaget. Brukergrensesnittet til MPI gir en visuell topologi og kommunikasjons-funksjonalitet mellom prosesser som kjører samtidig på flere maskiner eller noder. Dette blir utført på en spesiell måte for hvert språk. Det vil si at det gjøres på en spesifikk måte for det språket som blir brukt i forhold til syntaks.

1.5. Programmeringsparadigmer

Det finnes hovedsakelig fire måter å programmere en datamaskin på:

1. Imperativt, det vil si å beskrive hva man skal og hvordan man skal gjøre det trinn for trinn. Eksempler på denne type språk er C og Pascal.

2. Funksjonsorientert, der oppgaven som skal utføres er en evaluering av en matematisk funksjon. Det betyr at en funksjon kan ta en funksjon som parameter. Det finnes ingen globale tilstander etter kompilering og det er ikke lov å modifisere objekter etter de er konstruert. Eksempler på denne type språk er Lisp og Haskell.

3. Objektorientert programmering gjør det mulig å lage en modell av virkeligheten der metoder kan representere et objekts egenskap og der objekter med felles egenskaper kan samles i samme klasse. Eksempler på denne type språk er Java, C#, C++ og Simula.

4. Deklarativ programmering, også kalt logikkbasert programmering, er satt sammen av logiske slutninger. Det består både av et logisk språk og tilleggsfunksjoner som gjør programmeringsspråket allsidig. Eksempel på denne type språk er Prolog.

Programmeringsspråket C ble utviklet på slutten av 70-tallet som et portabelt lavnivå språk for UNIX-operativsystemet. Det er hovedsakelig et «portabelt assemblerspråk», et lag mellom høynivå og lavnivå. C er også et såkalt imperativt språk, men er i all hovedsak et funksjonsbasert språk der maskinkoden ikke kan overføres til andre arkitekturer. Det er hovedsakelig dette språket jeg kommer til å bruke i de fleste eksemplene som blir vist i denne oppgaven.

1.6. Ressursbruk

Kompleksitetsteori handler om ressursbruk i forhold til å løse et teoretisk problem. Disse ressursene er som regel tiden det tar å løse problemet, antall handlinger eller instruksjoner som må utføres og den digitale hukommelsen, det vil si lagringsplass som må være tilgjengelig. Dette skaper en fysisk begrensning for beregning av data. Hovedproblemet er dagens beregningsmaskiner og vår måte å klassifisere problemer på. Det fører til at noen problemer i praksis ikke kan løses selv om *n* er liten, for eksempel 10. Et annet eksempel på dette er NP-problemet *Travelling salesman problem* (TSP) med $O(N!)$ der N er antall byer, som bruker *rå kraft* for å finne beste løsning. Den sjekker alle veier og finner den raskeste. NP-problemer som tar eksponentiell tid å løse, er praktisk sett uløselige dersom det er snakk om N = 20 og mer. Dersom det er snakk om å besøke 20 byer på en mest mulig effektiv måte, får vi en størrelsesorden på 20! som tilsvarer omtrent 2.4×10^{18}. Dersom vi skulle besøkt 100 byer, hadde antall mulige utfall blitt 9.3×10^{157}. På grunn av dette faktum hjelper det ikke om alle atomer i universet hadde vært en kjerne i en datamaskin. Da hadde vi fortsatt ikke hatt nok kraft eller kapasitet til å løse problemet i parallell. Antall atomer i universet er estimert til 10^{80} Hydrogenatomer. Det betyr at universet i seg selv har en begrensning for hva som kan løses av problemer der ute, dersom det hadde vært slik. Med dagens teknologi er eneste mulighet for optimalisering å utvikle nye algoritmer som utnytter datakraften mer effektivt og i parallell.

1.7. Behov for mer datakraft

Stadig flere elektroniske spor legges daglig igjen av mennesker. Siden 80-tallet har trenden vokst eksponentielt, og vi har aldri produsert så mye digital data før i historien[18]. Denne enorme mengden kan hjelpe forskere til å identifisere markedstrender, kvalitetssikre forsking, forhindre sykdom, bekjempe kriminalitet og lage sanntids trafikkmeldinger. Dersom man skal være i stand til å bruke denne store mengden data må den behandles. En metode for dette går ut på at man ser på et mønster i et system, og ut fra observasjoner er det mulig å lage en modell som kan forutse hendelser ved bruk av statistikk. For å ta et praktisk eksempel kan vi se på kriminalitet i et geografisk område: Ut fra eksisterende data om hvor og når en kriminell handling forekommer, er det mulig å forutse sannsynligheten av at det vil forekomme i fremtiden, i tillegg til hvor og når. En algoritme går gjennom all relevant data og kommer med antagelser som politiet kan bruke til å ha

en bedre fordeling av konstabler på vakt. Dette kan føre til en nedgang i kriminalitet i det aktuelle området. Å utføre en algoritme som denne vil kreve stor beregningskraft dersom det skal være mulig å gjøre dette over større områder. Men det samme gjelder egentlig for alle situasjoner og hendelser man ønsker å forutse, for eksempel en værmelding eller naturkatastrofe[19].

1.8. En ny type datamaskiner kommer

En kvantedatamaskin er en maskin som bruker prinsipper for kvantemekanikk for å behandle data.
Den første kommersielle kvantemaskinen kom i 2011 med navnet *D-Wave One*[20]. Den hadde 128 Qbit. Forskjellen mellom en bit og en Qbit er følgende: En bit kan ha verdi *1* eller *0* mens en Qbit kan ha en superposisjon, det vil i praksis si *1* og *0* samtidig, altså kan den gjøre to ting samtidig. I praksis betyr det at det Qbit har eksponentielt større kapasitet enn en bit, det vil si 2^n større. Når man har 8 Qbit tilgjengelig betyr det at beregningsenheten er $2^8 = 512$ ganger raskere enn 8 bits. Det er ikke en vanlig datamaskin fordi den ikke er såkalt *Turing komplett*, den er mer som en akselerator, slik som GPU. Dens oppbygging følger ikke *logic gate*-modellen. Den følger modellen for *Adiabatic quantum computation* (AQC), og den kan defineres som en *Quantum Annealing Processor*. Den kan brukes til diskrete optimaliseringsproblemer, der det er mange variabler og en optimal løsning, slik som TSP. Det ble utført en test der en vanlig datamaskin bruke 30 minutter på å løse en TSP, og tilsvarende TSP ble løst på under et halvt sekund på *D-Wave Two* [21]. Det er en speedup på minst 3500 ganger. Maskinens oppbygging medfører at den må programmeres på en spesifikk måte for å bli tatt i bruk. Og den kan bare løse én type problemer, nemlig kombinatoriske optimaliseringsproblemer. Typisk dem som tar eksponentiell tid å løse med vanlige datamaskiner.

Selve utviklingen av et program som kjører på en kvantemaskin skjer derimot på en vanlig datamaskin som kommuniserer med kvantemaskinen. Beregningsenheten må ha en temperatur på 20 mK – 80 mK, det vil si nesten absolutt nullpunkt. Grunnen til dette er at det er atomer som brukes som bits. De må holdes i ro ved veldig lav temperatur for å kunne brukes til avlesning. Den er ikke i stand til å gjøre annet enn å *knuse tall*, derfor er det behov for en maskin som har et operativsystem der logikken kan programmeres fra. Det er imidlertid mye usikkerhet rundt denne nye maskinen, blant annet problemer med å måle ytelse på en fornuftig måte. NASA, Google og *Universities Space Research Association* (USRA) har kjøpt en *D-Wave Two* med 512 Qbit som skal brukes til forskning innen maskinlæring og kunstig intelligens[22]. I tillegg har Lockheed Martin en flerårig kontrakt med D-Wave og har kjøpt sin egen. De påstår at det var nødvendig for å kunne utvikle deres nyeste jagerfly[23].

Andre kritikere har påpekt usikkerhet i forhold til om det foregår kvantesammenfiltring inni kjernen til prosessoren. Kvantesammenfiltring

betyr at to eller flere kvanteobjekter må beskrives som é*n* selv om de er *to* forskjellige steder samtidig. Det er nettopp dette som skal gi kvantemaskiner dets fortrinn. Det er lite trolig at kvantemaskinen vil erstatte den generelle datamaskinen. Derimot vil den kanskje kunne brukes som en akselerator i et tungregningsanlegg for å løse enkeltproblemer som er veldig tidkrevende for en generell datamaskin. Det kan være faktorisering av primtall eller andre typer algoritmer som bruker *rå kraft*, for eksempel kryptering.

1.9. Oppsummering

Tungberegning er et stort fagfelt med mange mulige metoder for å oppnå det resultatet man leter etter. Jeg ønsker å undersøke om det er mulig å få en tilnærmet lineær *speedup* med en skalert økning av antall beregningsenheter. Uavhengig av hva resultatet blir her vil det være viktig, slik at videre arbeid kan utføres. Derfor ligger fokuset på to metoder, nemlig MPI i maskinklynge og på en multikjernemaskin. Noen algoritmer blir implementert og testet i de forskjellige paradigmene. Til slutt blir det utført en sammenligning av metodene og hva de best kan brukes til, før resultatene blir oppsummert og diskutert.

2. Bakgrunn

2.1. Abelklyngen ved UiO

Abelklyngen er en klyngemaskin som er oppkalt etter matematikeren Nils Henrik Abel (1802-1829). Den består av over 650 datamaskiner, også kalt noder. Dette tilsvarer over 10.000 kjerner. Abelklyngen har en teoretisk kapasitet på 209.664 *teraFLOPS*. *Floating-point operations per second* (FLOPS) er et mål for ytelsen til en datamaskin. Det betyr at Abelklyngen er i stand til å utføre 209.664×10^{12} flyttalls operasjoner per sekund. Abelklyngen er designet for å være allsidig og for å utføre mange arbeidsoppgaver samtidig, hovedsakelig små parallelle jobber opp til bruk av 1000 kjerner. Matematikk, astrofysikk, geofysikk og kjemi samt økonomi og samfunnsfag er eksempler på fagfelt som kan dra nytte av Abelklyngen, fordi den gir mulighet for å behandle større datasett med en høyere oppløsning.

Datamaskiner før i tiden brukte ofte en såkalt *Time-sharing*-modell, der brukere delte tid på prosessoren mellom seg. Dette er også en metode som brukes i moderne tungregningsanlegg. Et køsystem sørger for å utnytte Abelklyngens kapasitet til det maksimale. Abelklyngen bruker et køsystem som heter *Simple Linux Utility for Resource Management* (SLURM). For å koble seg til Abelklyngen trenger man en datamaskin med nettilgang som vil fungere som en terminal til anlegget.
De fleste studenter ved Institutt for informatikk på UiO har mulighet til å få en prøvekonto med begrensede ressurser på Abelklyngen fordi det er flere kurs som bruker den i undervisningssammenheng. Dersom man skal ha en konto med tilgang på alle ressurser må man søke gjennom Nortur[24]. De tildeler CPU-timer for større offentlige og private aktører.

2.1.1. Systemspesifikasjoner

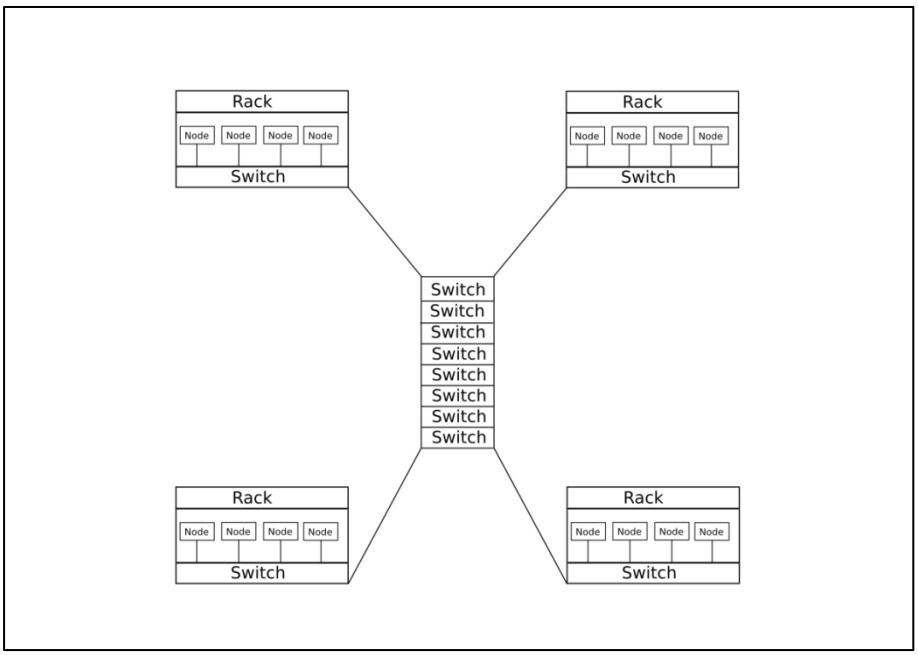
Alle beregningsnodene i klyngen har 64 GiB RAM, med unntak av åtte spesielle noder med 1 TB minne (*hugemem*). En vanlig node har 16 fysiske kjerner med *hyperthreading* som er koblet sammen med Infiniband som har en hastighet på 56 Gbit/sek. Infiniband er en nettverkstilkobling som brukes for å koble maskiner sammen i en klynge. Det gir pålitelighet, høy ytelse og liten forsinkelse på data som overføres. Alle nodene i klyngen, det vil si de maskinene som er koblet sammen, kjører Linux-distribusjonen CentOS i 64 bit-versjon. Abelklyngen er en oppstilling av noder (datamaskiner) med et system rundt som støtter den med brukertilgang, administrering og aksess av data. De 650+ beregningsnodene er alle Dual Intel E5-2670 Xeon Sandy Bridge som kjører på 2.6 GHz med 16 fysiske kjerner med *hyperthreading*, til sammen 32 logiske kjerner. Hver node har 64 GiBytes med DDR3 minne som kjører med en klokkehastighet på 1600 MHz, det gir 4 GiB minne per fysiske kjerne med ca. 58 GiB per sekund sammenlagt båndbredde når alle de fysiske kjernene blir tatt i bruk.

Lagring blir gjort tilgjengelig med to partisjoner som hver har en kapasitet på 6-8 GiB per sekund når sekvensielle I/O operasjoner blir utført. Filsystemet som blir brukt heter Fraunhofer Parallel File System (FhGFS). Det er et filsystem som er optimalisert for *high performance computing* (HPC). Selve lagringsenheten er en 2 TB SAS disk med RAID level 6.

Alle noder har Infiniband med 56 Gbit/s mellom en switch til alle nabo-nodene i en hylleseksjon. Fra en switch i hylleseksjonen går koblingen til hylleseksjon med switcher og videre til neste hylleseksjon, siden det er IP over Infiniband på alle noder (som bidrar til rask TCP/IP kommunikasjon). Abelklyngen er tilkoblet andre datasentre i Norge med en 10 Gbit Ethernet link og én dedikert 10Gbit link til Tier-1 på CERN.

2.1.2. Abelklyngens topologi

Abelklyngen er satt sammen på en måte som gjøre den egnet for små parallelle jobber på opptil 1000 kjerner. Det er en switch i hver hylleseksjon med noder. Derifra går det en tilkobling til en annen hylleseksjon med kun switcher. Deretter går koblingen videre til en annen hylleseksjon med en switch og noder. Hver switch har 36 porter. Det går 28 porter til nodene i hver hylleseksjon og 8 porter til de andre *switchene* i egen hylleseksjon. Da blir det en *overbooking* på antall noder i forhold til switcher på 28/8 = 3.5 mellom *hylleseksjonen*, men ikke internt på én *hylleseksjon*. Dersom man holder seg i én hylleseksjon har man $28 \times 16 = 448$ kjerner. Algoritmen som blir brukt for ruting er *fat-tree.*



Figur 6. En forenkling av Abelklyngens topologi. Rack er det samme som en hylleseksjon. Det er omtrent 24 hylleseksjoner i virkeligheten.

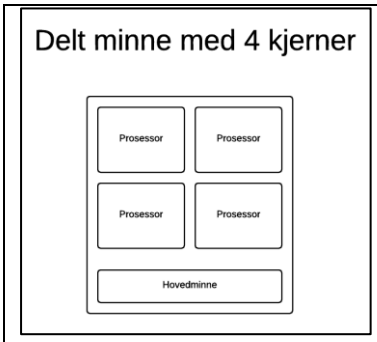
2.1.3. Maskinvare på Abelklyngen

Enhet	Kapasitet/antall
Noder (maskiner)	650 stk. Dual Intel E5-2670 Xeon Sandy Bridge CPU 2.60GHz, 64GiB RAM
Kjerner	10 000+
Maksimal ytelse	258 teraFLOPS
Minne	44 TB
Lagring	440 TB, FhGFS

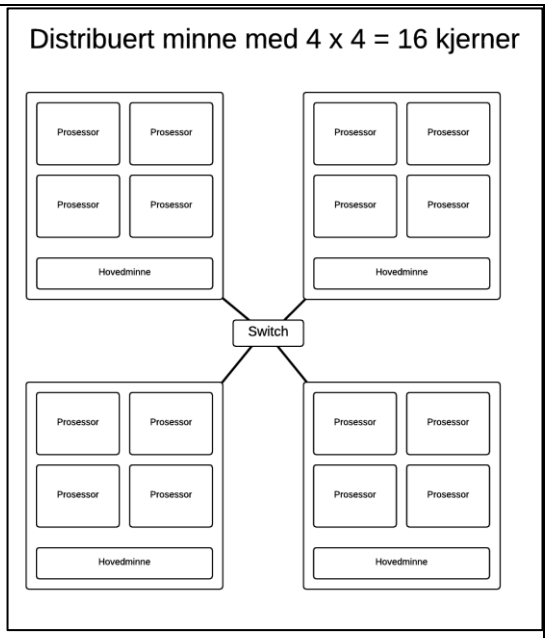
Tabell 1. Abelklyngens kapasitet.

2.2. Sammenligning av to metoder for å gjøre parallelle beregninger

Når det gjelder problemer som har behov for mye kommunikasjon under beregning, er det rimelig å anta at en multikjernearkitektur vil være raskere og mer effektiv fremfor en maskinklynge. Grunnen til dette er at den fysiske avstanden mellom hver kjerne er mikroskopiske. Dette til forskjell fra klyngearkitektur der avstanden mellom to noder kan være flere meter. Først skal vi se på algoritmen som skal brukes for sortering. Videre skal vi også se på matrisemultiplikasjon, der jeg vil sammenligne resultater fra både multikjerne og klynge. Deretter skal vi se på delt minne og klyngearkitektur og hvordan programmer implementeres der. Vi skal se på Delaunay triangulering, der økning av antall kjerner gir en tilnærmet lineær *speedup* i og utenfor det optimale intervallet, og til slutt viser jeg en implementasjon av Mandelbrot-mengden som har relativ god ytelse sammenliknet med sekvensiell implementasjon. Begge typene er klassifisert som MIMD, men det er en vesentlig forskjell i hvordan de er bygget opp og blir brukt. Vi skal se nærmere på de *to* typene parallelle beregninger. *Figur 7* og *Figur 8* viser litt av denne fysiske forskjellen.



Figur 7.Delt minne.



Figur 8. Distribuert minne.

2.2.1. Multikjerne med felles minne

Denne modellen er den mest vanlige MIMD-typen. Omtrent alle datamaskiner per dags dato har felles minne, også kalt delt minne. Det betyr hovedsakelig at kjernene har tilgang på samme hovedhukommelse (RAM). For å utnytte denne maskinen må den programmeres med *tråder*. En multikjerne med 4 logiske beregningsenheter som *Figur 3* viser kan uføre 4 beregninger i parallell, men det er kun dersom vi implementerer løsningen av problemet med tråder. De fleste programmeringsspråk har støtte for dette. I C kan man bruke *POSIX Thread* eller *openMP*. I Java har man pakken *cunncurrent* for parallell programmering, og i Python kan man velge mellom ganske mange programpakker for *Symmetric Multiprocessing*. Alle de nevnte programmeringsspråkene har også mulighet for å bruke MPI lokalt.

Et delt minneområde er tilgjengelig for alle kjerner på prosessoren. Men det er noen begrensninger på ytelsen til minnet fordi det ligger i forskjellige nivåer. Minnet kan være globalt, det vil si at alle prosessorer har tilgang på det, eller det kan være lokalt. Da er det kun én prosessor som har direkte tilgang på data. Dersom det tar like lang tid å aksessere minne uavhengig av om det ligger lokalt eller globalt har vi en modell som kalles *uniform memory access* (UMA). Ellers kalles dette for *non-uniform memory access* (NUMA). Det er viktig å kjenne til dette i forhold til design og parallellisering av en algoritme som skal kjøres på en multikjerne. En annen ting som må nevnes er at det er flere typer minne, hovedminne og cache nivå 1-3 eller 1-6. Men jeg forenkler dette ved å kun snakke om hovedminne og cache. Cache er raskere enn hovedminne, men begrenset i mengde. Globalt minne er forøvrig en av grunnene til at en multikjerne er enklere å programmere enn maskinklynge fordi det har dette såkalte globale

minnet. Det betyr i praksis at alle prosessorer har tilgang på samme data, og at det ikke er behov for å flytte data rundt mellom prosessorer. På grunn av bruk av cache mellom hovedminne og prosessor, vil noe data bli behandlet på samme tidspunkt av flere prosessorer. Det vil si at noe av arbeidet blir utført flere ganger. For å muliggjøre bruk av delt minne er det behov for en måte å oversette minneadresser slik at data kan bli funnet i hele minnet og at man på en måte garanterer at parallelle instruksjoner kan brukes på flere kopier av samme data. Dette kalles også *cache coherence*. Det er som nevnt tidligere et konseptuelt felles minneområde bestående av flere lokale *cacher* som har som formål å synkronisere data i *cachene*. Med unix-programmet *top* kan vi følge med ressurs/arbeidsfordeling av maskinvare/kjerner på maskinen som kjører. Når vi for eksempel starter opp fire kjøringer av et C-program, kan vi se hvordan de forskjellige beregningsenhetene blir belastet. Det samme vil skje dersom vi starter et MPI-program som bruker fire MPI-prosesser[25].

2.2.2. OpenMP

OpenMP er en metode for å bruke tråder til parallelle beregninger i C-programmer. Trådene kjører tilsynelatende parallelt og blir allokert til forskjellige prosessorer av en OS-rutine. De delene av en sekvensiell algoritme som kan parallelliseres må markeres med pre-prosesseringsdirektiver i koden.
Se *Figur 11*. Et trivielt eksempel kan være å summere tall i en vektor. Antall tråder er som regel satt til samme antall som man har prosessorer. Det er hovedsakelig *runtime*-miljøet som bestemmer dette, men man kan selv endre dette med miljø-variabler for OS. Hver tråd har en unik *ID* som kan hentes ut med funksjonen *omp_get_thread_num()* som returnerer et tall mellom 0 og (antall tråder –1) som finnes i headeren *omp.h*. Etter eksekvering av den parallelle koden blir *join* brukt for å samle trådene og sette sammen dataene som er delt opp igjen. OpenMP har blitt brukt til matrisemultiplikasjon i denne oppgaven fordi det er en av metodene for å utføre parallelle beregninger med en multikjerne og språket C, men også fordi det muliggjør en hybridløsning med delt og distribuert minne som kan ha veldig god ytelse[26].

2.2.3. Maskinklynge med distribuert minne

I distribuert minne har alle prosesser sitt eget minneområde. Disse prosessene kan bestå av en eller flere beregningsenheter (kjerner). Det vil si at de kan være en-kjerne- eller multikjerneprosessorer.
I et moderne tungregningsanlegg slik som Abelklyngen kan man forvente at alle maskin-nodene har en multikjerneprosessor. For å kommunisere mellom maskin-nodene må de utveksle meldinger mellom seg. Disse meldingene brukes til å overføre data og for å synkronisere de forskjellige prosessene som jobber samtidig. Dette kalles message passing interface (MPI).
I dette såkalte meldings-paradigmet er det støtte for å kjøre forskjellige deler av et program på hver prosessor. Kommunikasjon blir utført ved å sende og motta meldinger mellom flere enheter som regel avgrenset til å være én og én datamaskin. De to mest grunnleggende operasjonene i dette

paradigmet er *send* (sende melding) og *recv*. Maskiner som er satt sammen må bruke en adresse for å kunne sende og motta en melding. Det vil si at alle maskiner/prosessorer må ha en unik identifikator for at flere prosessorer skal kunne kjøre programmet parallelt. En måte å finne ut hvilket ID-nummer en prosess har, er å bruke UNIX-kommandoen *whoami* som gir navn eller PID til prosessen eller brukeren som kjører denne instansen av programmet. Denne identifikatoren blir som regel kalt *my_rank.* Et annet viktig konsept er *num_procs,* som angir antall prosesser som er med på denne parallelle kjøringen av programmet. Konseptene som er presentert over er nok til å lage et program med MPI som kjører i et distribuert miljø. På den andre siden er det enkelt å simulere MPI på en multikjerne som har felles minne. Siden en multikjerneprosessor har flere kjerner kan den representeres som flere logiske enheter i operativsystemet. Det fungerer omtrent som å kjøre et program på kun *én* node i en maskinklynge. Eller som om alle noder kun har *én* kjerne på sin prosessor. Minne vil da bli reservert for hver kjerne og delt opp slik at det blir som i den distribuerte modellen beskrevet over. Men det må nevnes at denne simuleringen kan bli mer kostbar enn å bruke felles minne slik som beskrevet i forrige kapittel. *Send* og *recv* koster mye mer enn *fork* og *join*, fordi man i de sistnevnte bare gjør oppslag i et minneområde, fremfor å utveksle en melding. Når vi bruker MPI på en maskinklynge blir det egentlig en blandet kjøring med både delt og distribuert minne. Vi velger selv om dette er noe vi vil styre. Dersom vi kun velger å kjøre et program på *én* node, vil MPI simuleres slik som beskrevet over med delt minne. For mange problemer er dette hensiktsmessig. For å bruke MPI på maskinklynge med distribuert minne må det være minimum to noder som er allokert, det er først da vi faktisk bruker distribuert minne på ordentlig og ikke simulert. På en annen side, dersom man velger en tilfeldig allokering av prosessorer og antallet er over 8, kan man nesten alltid forvente at det er distribuert minne som brukes[25]

2.3. Problemtyper

Jeg skal nå presentere fire forskjellige problemtyper som er valgt for å sammenligne delt og distribuert minne. Sortering ble valgt fordi det var behov for et *enkelt* problem til å starte med. Etter det utforsket jeg matrisemultiplikasjon, som er litt mer tidkrevende å implementere i distribuert minne. Etter det implementerte jeg Delaunay triangulering. Det var meget tidkrevende fordi det involverte flere steg fra P0, P1 og P2 satt sammen. Til slutt ble den praktiske delen av oppgaven avsluttet med P4 for å prøve å vise eksempel på bra ytelse. Der skal det produseres en fraktal og vise dens egenskaper ved å produsere noen forskjellige bilder. Det er et problem som er *pinlig parallelliserbart*, der det forventes høy ytelse og effektivitet fordi problemet krever lite eller ingen kommunikasjon for å produsere hver piksel.

2.3.1. P1 - Sortering

Det finnes flere typer sorteringsalgoritmer, men hovedsakelig *to* typer: Sammenligningsbasert og innholdsbasert. De som er sammenlignbare har vanligvis en tidskompleksitet på O(n log(n)). De som er innholdsbaserte kan

komme ned på O(K∗N), der k er antall bit for å representere dataelementet. Vi skal se på en type sammenlignbar sorteringsalgoritme som bruker *splitt og hersk*-teknikken for å dele opp og sortere mindre biter av vektoren rekursivt. Mergesort-algoritmen har hovedsakelig 2 steg:

I.). Del opp en usortert liste i n antall sublister med ett element i hver. En liste med kun ett element er definert som sortert. Nå har vi n antall sublister som alle er sortert nederst i et binærtre.

II.). Ved å sette sammen sublistene fra nederste til øverste nivå i treet som nå har blitt laget, blir det produsert nye sublister på hvert nivå frem til det kun er *én* igjen som er sortert. Sublistene blir satt sammen slik at høyre og venstre barne-nodes verdi blir sammenlignet så de nye sublistene har stigende rekkefølge, dersom målet er å sortere tall i stigende rekkefølge.

Distribuering av data i parallellisering
Partisjoneringen av data i hvert steg kan gi svært varierende størrelse på sublisten. For hver runde blir den største delen beholdt av hovedprosessen og den minste sendt til den mottakende barneprosessen. Grunnen til at dette blir gjort, er at det reduserer mengden med kommunikasjon-overhead på grunn av sending av små lister. En annen god grunn er at det gir mindre *idling* for prosessen som skal sendes fordi alle prosesser som er ledige kan dele data.Dersom en liste skal sorteres med tre prosesser vil p0 sende den minste delen p1. Den gjenværende p2 vil nå bli henvist til å dele med p0. Det betyr at før p1 får tilsendt data, har p0 mulighet til få en annen prosess til å ta imot data. Når en prosess har en vektor som er på mindre enn 50 elementer vil kan det være lurt å bruke innstikksortering istedenfor det rekursive oppdelingssteget.

2.3.2. P2 - Matrisemultiplikasjon

Multiplikasjon mellom to matriser kan utføres dersom antall kolonner i den første matrisen er lik antall rader i den andre matrisen. Dersom A er en $m\ x\ n$ matrise og B er en $n\ x\ p$ matrise, kan vi beregne produktet av matrisen AB med dimensjoner $m\ x\ p$. Matrisemultiplikasjon (MM) er assosiativ. Det betyr at (AB)C=A(BC), (A+B)C = AC+BC og C(A+B) = CA+CB. Men MM er generelt ikke kumulativ fordi det ikke er alle produkter AB som er definert for BA, derfor er ofte AB ulik BA. Eventuelt må vi ha *to* kvadratiske matriser fra starten. Det vil si at vi må ha matrisen $m\ x\ n$ og $n\ x\ p$, der $m == p$ som er symmetrisk med hoved-diagonalen, for å vise at AB == BA.

Sekvensiell algoritme
Dersom vi utfører en multiplikasjon mellom matrise A med dimensjon $m \times l$ og matrise B med dimensjonene $l \times n$, blir resultatet en matrise C med dimensjonen $m \times n$. Det kan vises med denne formelen, som viser utregning av hver enkle celle i resultatmatrisen C:

$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} \times b_{k,j}$$

For å implementere formelen over trenger vi en nestet løkke for å traversere hvert matrise-element, og en indre løkke for å beregne verdiene av *C* i hver celle. En pseudokode for dette står beskrevet under. Se *Figur 9*.

```
1.   procedure MAT_MULT (A, B, C)
2.   begin
3.      for i := 0 to n - 1 do
4.          for j := 0 to n - 1 do
5.              begin
6.                 C[i, j] := 0;
7.                 for k := 0 to n - 1 do
8.                     C[i, j] := C[i, j] + A[i, k] x B[k, j];
9.              endfor;
10.  end MAT_MULT
```

Figur 9. Pseudokode for matrisemultiplikasjon (MM).

Parallell algoritme

Det er fortsatt den sekvensielle algoritmen som blir brukt i den parallelle versjonen. Men forskjellen er intervallet som skal behandles, fordi det er fordelt mellom prosessene. Intervallet hver prosess skal behandle kan defineres som en blokk. En $n \times n$ matrise kan defineres som en $q \; x \; q$ matrise med blokker som ligger i intervallet $A_{i,j} (0 \leq i,j < q)$, da vil en blokk være tilsvarende en $\frac{n}{q} \times \frac{n}{q}$ sub-matrise av A. Pseudokoden under beskriver denne endringen i intervallet. Se *Figur 10*.

```
1.   procedure BLOCK_MAT_MULT (A, B, C)
2.   begin
3.      for i := 0 to q - 1 do
4.          for j := 0 to q - 1 do
5.              begin
6.                 Initialize all elements of C_{i,j} to zero;
7.                 for k := 0 to q - 1 do
8.                     C_{i,j} := C_{i,j} + A_{i,k} x B_{k,j};
9.              endfor;
10.  end BLOCK_MAT_MULT
```

Figur 10. MM for en blokk.

Når matrisemultiplikasjon (MM) skal parallelliseres for distribuert minne, finnes det hovedsakelig to måter å gjøre dette på: (I) radvise 1-dimensjonale blokker, eller (II) 2-dimensjonal blokkinndeling. Metode 1 er å bruke en oppdeling med radvise 1-dimensjonale blokker. Utgangspunktet for denne metoden er at matrise A og B er partisjonert som rader og fortelt radvis mellom prosessene. Antall rader i matrisen trenger ikke å være delelig med antall prosesser fordi dette kan beregnes slik at prosessen får en mest mulig jevn fordeling, men den kan naturligvis ikke bli lik for alle prosesser.

Optimalisering med delt minne og openMP

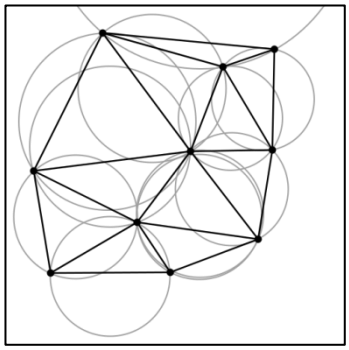
Jeg skal bruke metode 1 for denne implementasjonen. I tillegg skal programmet ta i bruk openMP for å utnytte delt minne på hver prosess i den regneintensive delen, som er innerste løkke i MM-algoritmen (MMA). Det betyr at når distribusjon av A og B er ferdig skal hver prosess bruke sitt delte minne for å gjøre beregning av C mer effektiv. Denne MMA er en hybrid mellom distribuert og delt minne og må kjøres på en maskinklynge for at algoritmen skal kunne utføres mest mulig optimalt. Det er hovedsakelig designet for distribuert minne, men å ta i bruk delt minne er her en triviell optimalisering på to linjer kode[27]. Men det må gjøres noen innstillinger for å få det optimalt. OpenMP kan selv finne beste oppdeling av oppgavene med "auto", men "guided" ser ut til å være mest effektiv for MM. Se *Figur 11*.

```
1  #pragma omp parallel private(i, j, k) num_threads(nthreads)
2  #pragma omp for schedule(guided)
3  for (i = 0; i < m; i++) {
4      for (j = 0; j < n; j++) {
5          C->matrix[i][j]=0.0;
6          for (k = 0; k < l; k++){
7              C->matrix[i][j] = C->matrix[i][j] + A->matrix[i][k] * B->matrix[j][k];
8          }
9      }
10 }
```

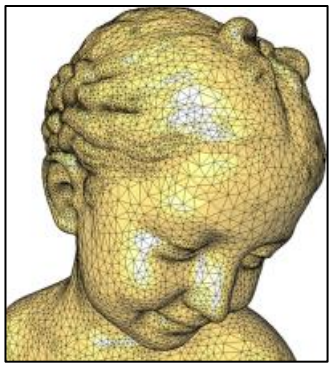
Figur 11. De to øverste linjene er pre-prosesseringsdirektiver for kompilatoren (GCC).

2.3.3. P3 - Delaunay triangulering

Delaunay triangulering (DT) er en type triangulering som har den egenskapen at ingen punkter i trianguleringen ligger inni omkrets-sirkelen til en av de andre trianglene. Se Figur 12. DT har også den egenskapen at den maksimerer den/de minste vinkelen/vinklene, noe som gjør DT veldig nyttig når man skal lage polygoner som brukes i avbildning av terreng og i andre typer datagrafikk. Delaunay triangulering er også en metode der resultatet kan brukes til å finne et *minimum spaning tree* (MST) fra et sett med punkter *P*. Et resultat av dette er at det gir mer nøyaktighet i noen bruksområder som for eksempel generering av kart der kote-trekking blir brukt til å definere topp- og bunnpunkter (daler og fjell). Det gir mer nøyaktig avbildning av et landskap fordi vanlig interpolering ofte kan gi unøyaktighet, da det er basert på et gjennomsnitt av to eller flere punkters plassering i to eller flere dimensjoner[28]. Ved å bruke kuler som omkranser omkrets til de aktuelle punktene, er det mulig å utvide DT til tre dimensjoner eller høyere. Dersom alle punktene ligger på en linje, kan det ikke finnes en DT, fordi det ikke er mulig å definere en trekant. Hvis man har fire eller flere punkter på samme sirkel så kan det konstrueres to mulige trianguleringer som begge oppfyller kravet for DT, det vil si at omkrets-sirkelen til trianglene ikke har punkter i seg[29].



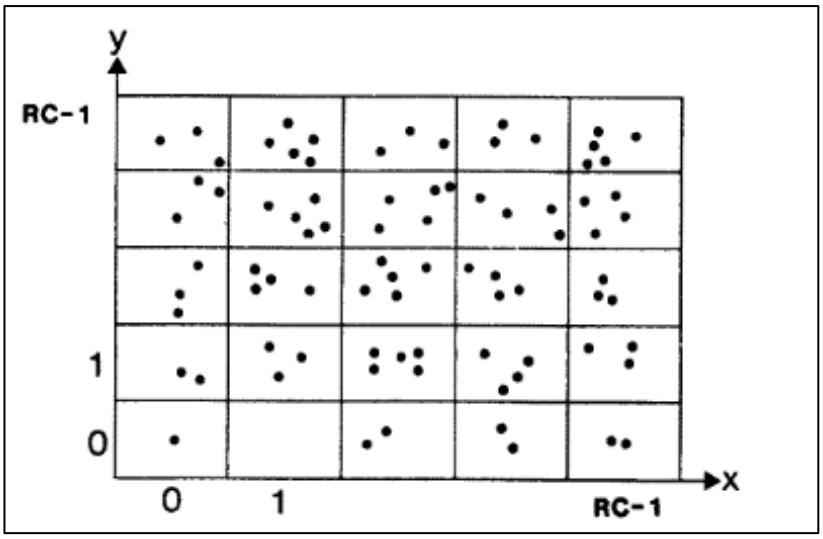
Figur 12. Delaunay triangulering med omkrets-sirkel for hvert triangel.



Figur 13. Her vises hvordan DT blir til datagrafikk. I denne 3D-modelllen er det variasjonen i trekantflates vinkel i forhold til lyskilden og øyepunktets plassering som skaper en illusjon av dybde i bildet[30].

Beskrivelse av stegene i DT algoritmen
Det finnes flere algoritmer for å gjøre en *Delaunay triangulering*. Arbeidet går hovedsakelig ut på å finne det beste punkt *C* innenfor en sirkel med punktene A og B på omkretsen og hvor linjesegmentet AB er en kjent *Delaunay-kant* (DK). Punktene som er ferdig beregnet må være tilgjengelig i en datastruktur med relasjon til alle trianglene og kantene. Algoritmen som er brukt er en variant av splitt og hersk-metoden, og den kjøres i lineær tid. Denne fremstillingen følger i stor grad en gjengivelse [31] og [32].

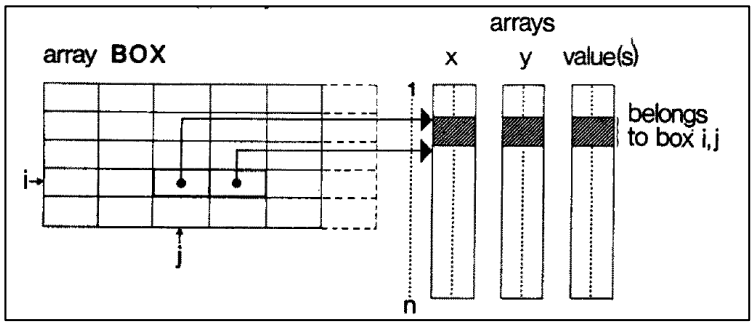
Initiering: (lese inn data, oppdeling, sortere)
Vi starter med et sett med punkter P i planet $RC \times RC$ der alle P_i har et *x*- og *y*-koordinat x_i og y_i. Første steg i DT er å dele opp punktene og legge dem inn i *rektangulære bokser.* Antallet rektangler er proporsjonalt med n antall punkter. Det kan defineres som $\sqrt{\frac{n}{2}}$. Dersom det skal være i snitt *to* punkter i hver boks og vi har 50 punkter, blir det $\sqrt{\frac{50}{2}} = \sqrt{25} = 5$. Det vil si *fem* rader og *fem* kolonner. Se *Figur 14*.



Figur 14. Oppdeling av punktene i rektangulære bokser[31].

For å gi en bedre geografisk distribusjon før vi beregner den konvekse innhyllingen KO og DT, er det hensiktsmessig å sortere punktene først. Det gjør at det blir enklere og raskere å finne KO i neste steg. Det utføres med Radix-sortering og vi sorterer først på y-koordinat og så på x-koordinat. Men dette er ikke vesentlig, og det går helt fint å endre rekkefølgen. For å lagre resultat av oppdelingen, det vil si hvilket punkt som tilhører hvilken boks i planet $RC \times RC$, brukes en matrise med et element for hver boks/rektangel. Det gitte elementet peker til det første punktet i en korresponderende boks med vektor for x- og y-koordinatene og punktets indeks *i* for mengden. Antall punkter i $boks_{i,j}$ ligger i denne boksen og frem til $boks_{i,j+1}$. Alle punkter mellom dem tilhører boks[i][j]. Det er derfor lagt

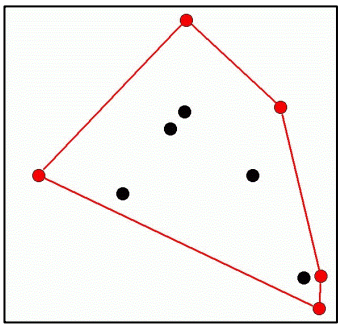
til en ekstra kolonne slik at vi kan gå opp til $j = RC - 1$. Nå har vi en datastruktur som gir oss raske oppslag og er partisjonert slik at punktene kan beregnes på en effektiv måte senere. Se *Figur 15*. Dette steget av algoritmen tar $O(KN)$ fordi vi kun gjør en sortering av x- og y-verdien med Radix-sortering. Det betyr at punktene ikke vil ligge sortert inni boksene sine, men vi må likevel undersøke alle punktene i hver boks for å finne de beste DT-kantene. Da hjelper det ikke om de ligger sortert i hver boks [31].



Figur 15.Datastruktur for punkter [31].

Den konvekse innhyllingen

En konveks innhylling er et sett med punkter *P* i planet der det minste sub-settet *K* omkranser *P*. Dersom vi har en punktmengde i planet, er den konvekse innhyllingen et sett med ekstremalpunkter fra den opprinnelige mengden. Det blir omtrent som å dra en strikk rundt alle de ytterste punktene slik at det blir en figur med en sammenhengende kant rundt hele seg. Se *Figur 16*.



Figur 16. Det er fem punkter på KO og totalt 10 punkter i P.

Neste steg i DT-algoritmen er at vi må definere den konvekse innhyllingen KO for *P*. Den må være på plass for å utføre en effektiv og lovlig korrekt DT av mengden *P* med *n* antall punkter. Vi tar utgangspunkt i oppdeling fra forrige steg og oppretter så en boolsk vektor *V* for å legge inn hvilke punkter som er på KO og ikke. Punktets indeks blir vektoren i sin indeks, altså V_i. Så gjør vi følgende:

1.
Først finner vi det punktet med maksimal x-verdi og setter den lik A, og så

finner vi det med den minimale x-verdien og setter den lik B.

2.1
Vi gjør kall på sub-algoritmen CONVEX med A og B som parameter for å finne alle punkter på KO som er over linjestykke AB

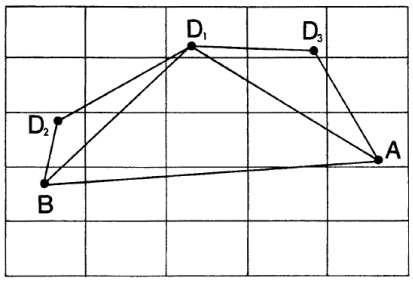
2.2
Vi gjør kall på sub-algoritmen CONVEX med B og A som parameter for å finne alle punkter på KO som er over linjestykke BA, men under AB.

CONVEX(I,J):
Denne funksjonen ser gjennom alle bokser som er over eller har et skjæringspunkt med linjestykket IJ. Av alle disse punktene i boksene finner vi punktet D som har den lengste distansen fra IJ.

To ting kan skje:
(I)
Dersom distansen har negativ verdi, det vil si om punkt D er over IJ, så legges D inn i V mellom I og J. Vi gjør så rekursivt kall på CONVEX(I,D) og CONVEX(D,J)

(II)
Eller dersom distansen har positiv verdi, det vil si at D er under linjestykket IJ, returnerer vi bare uten å kalle på CONVEX rekursivt.

Siden punktene ligger delvis sortert i bokser, kan vi gå fra et punkt i kanten på boksen til linjestykket IJ som skal undersøkes. Det første steget som innebærer å finne ekstremalverdiene for x-koordinatene kan også utføres ved å søke langs kanten på boksen. Da vil noen punkter bare trenge å undersøkes *én* gang, mens andre *to* eller flere ganger. Dette kan også utføres med tiden O(N). Se *Figur 17* og forklaring under.

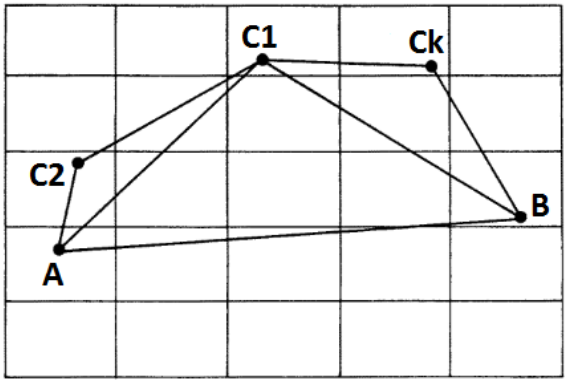


Figur 17. Beregning av den konvekse innhyllingen [31].

I gjennomsnitt må $\frac{n}{2}$ punkter undersøkes for å finne D1 som ligger i midten. Videre ser vi at linjestykke AD1 og D2B ligger under AB. For å finne D2 må vi lete gjennom $\frac{1}{4} \times \frac{n}{2} = \frac{n}{8}$, og det samme gjelder for D3. For å oppsummere finner vi alle punkter på KO over AB ved å undersøke $\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots = n$ antall punkter som tar O(N) [31].

Nærmeste nabo er en Delaunay-kant (DK)
Første finner vi alle punkter A_i, og dets nærmeste nabo B_j Da er A_iB_j en lovlig DK og er utgangspunkt for det senere søket etter alle trekantene $A_iB_jC_k$ [32].

Triangulering
Det tredje steget er selve algoritmen for å beregne DT. Her bruker vi en matrise ED som holder på nabo-relasjonene til punktene i *P*, og en FIFO-liste for Delaunay-kanter som er funnet men ikke behandlet foreløpig. Kanten AB og BA blir lagret *to* ganger siden A er nabo til B, og B er nabo til A. Videre antar vi at KO til *P* er funnet og at alle punktene på KO ligger i ED med sin tilhørende nabo-relasjon. La B og A være *to* naboer på KO slik at resten av P er over linjestykket AB. Se *Figur 18*.



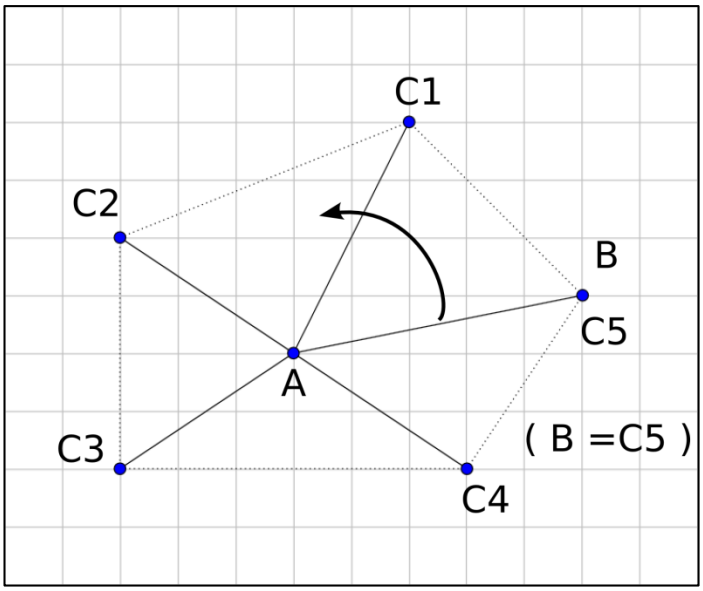
Figur 18.Kandidater til punkt C i trekanten ABC [31].

Vi gjør følgende:
1. Punktet *C* i *P* over DK AB som har den største vinkelen BCA finner vi ved å bruke sub-algoritmen *FinnNesteC* som er beskrevet på neste side.
2. Merk alle kanter BC og AC som ikke er undersøkt. Kanten BC blir lagt inn i FIFO listen dersom den ikke har vært lagret i ED tidligere.

3. Dersom den AC som ble funnet før steg 2 ligger i ED, så tar vi neste kant fra FIFO-lista. Så setter vi A= 'C og B='B, slik at vi får kanten 'B'C og så går vi til steg 1 igjen. Men dersom FIFO-listen er tom er triangulering ferdig.

4. Ellers setter vi B=C og starter fra steg 1 igjen.

Dette er en algoritme som går i retning mot klokka rundt punkt A. Den finner nye triangler; ABC1, AC1C2, AC2C3 osv. til vi møter en tidligere oppdaget kant som stopper traverseringen (steg 3).
Vi har da funnet alle Delaunay-kanter hvor A deltar: AC1, AC2,…ACk. Da har vi også funnet alle de *k* antall Delaunay-trianglene hvor A deltar; AC1C2, AC2C3,… ACk-1Ck. Da får vi Ck=B[31]. Se Figur 19.

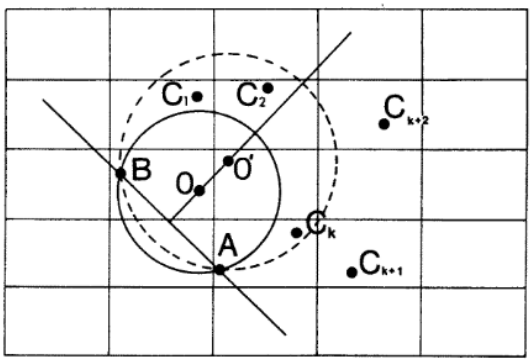


Figur 19. Innsamling av Delaunay triangler runde et punkt A.

Finn neste punkt C

Sub-algoritme *FinnNesteC*, det vil si søkesirkelen fra steg 1, undersøker alle punkter i en boks dersom:

1. Punktene ligger over linjestykket AB.
2. Punktene er i skjæring eller inne i søkesirkelen som ligger over A og B og har sitt senter O på midtnormalen til kanten AB. Se *Figur 20* som viser søk etter beste C til trianglene ABC der grunnlinjen AB er gitt.



Figur 20. Beregner neste punkt C[31].

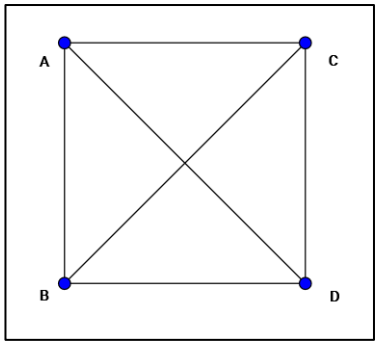
Først velger vi O som kandidat til sirkelsenter slik at vi får en vinkel på 120° for trianglene BOA (BO=60' +OA=60'). Da vil alle punkt D' utenfor søkesirkelen ha mindre vinkler ADB enn alle punktene i søkesirkelen. Når de aktuelle boksene er funnet, blir alle punkter (C1,C2,3,Ck) testet for å finne den C' som gir størst vinkelen for BC'A. Dersom det ikke er noe punkt i de valgte boksene eller den beste kandidaten er utenfor søkesirkelen men i en av de aktuelle boksene, blir en ny søkesirkel opprettet. Søket kan

da repeteres for de nye boksene som nå er med i beregning til et punkt i den nye søkesirkelen som er opprettet.

Hvis vi bare finner *én* kandidat til punkt innenfor den siste søkesirkelen, så øker vi radiusen slik at dette punktet kommer med. Dersom ingen punkt blir funnet blir radiusen for hver nye sirkel økt slik at den hele tiden er *i* ganger større enn det opprinnelige arealet. Denne algoritmen finner Delaunay-triangler ved å bruke definisjonen for DT når den hele tiden søker over linjestykket AB for å finne punktet C som definerer triangelen ABC. Vi kan være sikre på at det ikke er noe punkt i søkesirkelen under AB når vi ser etter neste punkt over AB, fordi dersom denne kandidaten fantes ville den blitt valgt til punktet C tidligere fremfor den som ble valgt og undersøkes nå. [31].

Verifisering av resultat

DT-algoritmen vi beskriver her behandler såkalt ko-sirkularitet. Det vil si at det er fire eller flere punkter på randen av den miste søkesirkelen. For å verifisere resultatet av DT finnes det en metode for å sjekke at alle kantene er lovlige ved å traversere matrisen med løsning for å se om det finnes punkter som går fra for eksempel fra A- til B, C, D og fra B- til A, B, D. Dette blir omtrent som å sjekke løsning for kryssende kanter. Men fordi vi behandler ko-sirkulære punkter, skal dette ikke skje i praksis.



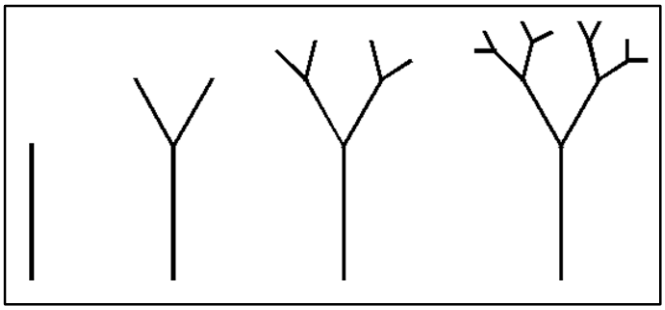
Figur 21. Her ser vi et eksempel på en kryssende kant, som ikke er lovlig i Delaunay triangulering.

Lengden på nabolisten

På grunn av en fornuftig skaleringsmetode og 2D-partisjonering er det mulig å beregne datasett på over 10^6. Med for eksempel 100 millioner (10^8) blir domenet for partisjonering 10000x10000, som betyr at største *x*- eller *y*-verdi er $\sqrt{10^8} = 10000$. Ettersom jeg bruker kjente datasett er det bare å telle opp maksimalt antall naboer i kjøring av DT på $10^6 - 10^8$
Med 1 mill. (10^6) punkter, blir maksimalt antall naboer 13.
Med 10 mill. (10^7) punkter, blir maksimalt antall naboer 14.
I et unikt datasett ble det funnet 17. Dette er høyeste antall som er observert med denne DT-algoritmen. Derfor setter jeg en total lengde på hver streng til 20 heltall.

2.3.4. P4 - Fraktaler

Fraktaler er sett av geometriske objekter med mønstre som er repeterende og synlige på alle skalaer. Fraktal geometri finnes overalt i naturen, som gjentagende mønstre i for eksempel trær, cellestrukturer og elveløp. Slike naturlige former var lenge ansett som umulig å tolke matematisk, men gjennom blant annet Benoit Mandelbrots oppdagelser ble det klart at også slike "røffe" usymmetriske former var mulig å fremvise med matematiske formler[33]. Ofte kjennetegnes fraktaler i naturen og som matematisk genererte objekter ved at strukturen gjentar seg selv i det potensielt uendelige. Et tre har grener som igjen har sine grener med nye grener, som alle er mindre kopier av det originale treet, se *Figur 22*. De skiller seg fra tradisjonelle geometriske objekter gjennom hvordan de oppfører seg ved for eksempel dobling eller halvering av objektets lengder. Ved multiplikasjon med heltall vil ikke en fraktal nødvendigvis gi resultater i heltall. Den øker i kraft av sin *fraktal-dimensjon*, som er mer komplisert enn én dimensjon, men enklere enn to dimensjoner. Det vil si at fraktal-dimensjonens verdi er et tall mellom 1 og 2. Avstanden mellom to punkter som måles i planet av en fraktal er ikke-definerbar fordi objektets form teoretisk sett er uendelig selvrepeterende.



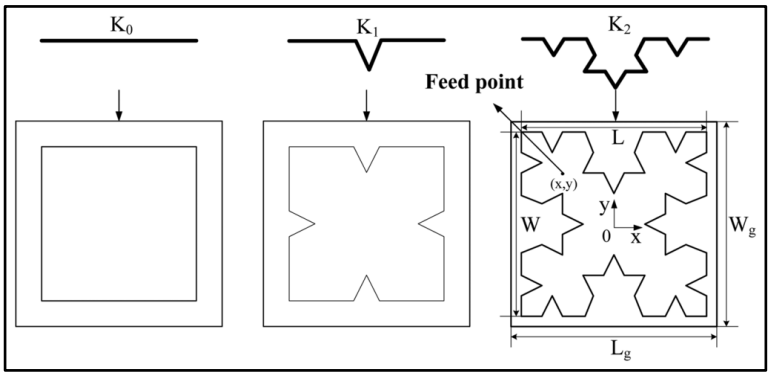
Figur 22. Her ser vi hvordan en linje blir til et tre etter 3 iterasjoner av samme operasjon[34].

Det vi nå kjenner som *fraktaler* har vært gjenstand for undersøkelse i matematikken siden 1800-tallet, men ble først omtalt som *fraktaler* av Benoit Mandelbrot i 1975 i hans bok Fractals: Form, Chance and Dimension. Fraktaler er også kjent som utvidende symmetri eller selv-utviklende symmetri. Under følger noen bruksområder for fraktaler.

Antenner

Nathan Cohen ved Boston University begynte arbeidet med antenne-elementer basert på fraktal geometri i 1988[35]. Ved å forme antennemottakere i fraktale mønstre viste det seg at størrelsen på antennene kunne reduseres til en brøkdel, og samtidig motta signaler like godt som større "tallerken"-antenner, se *Figur 23*. En annen enorm fordel ved fraktal geometri i antenne-elementer er at den enkelte antennen er i stand til å motta signaler i flere frekvenser. Dette har blant annet muliggjort det moderne designet i mobiltelefoner, som ikke behøver én mottaker per frekvens (for eksempel tre antenner dersom telefonen skal motta GSM, WIFI-signaler og Blutetooth-signaler). Ved små, fraktal geometri-baserte antenne-elementer

er det altså mulig å motta mange ulike signaler på en telefon av fysisk liten størrelse. Antennen trenger heller ikke plasseres på utsiden av telefonen for å motta signalene, takket være fraktal geometri. Denne formen for antenne-teknologi brukes også i satellitter.



Figur 23. Her ser vi fraktal geometri i et antennesystem[36].

Datagrafikk og bildebehandling

Computer-generated imagery (CGI) brukes i omtrent alt av videospill, spesialeffekter til film og annen digital grafikk i dag. CGI som vi kjenner det ville ikke vært mulig uten bruk av fraktal geometri[37]. Ved å dele opp geometriske figurer med fraktale formler, kan man (dersom den fysiske datakapasiteten er kraftig nok) på få minutter skape et visuelt detaljnivå som er mer komplekst enn hva et menneske kunne gjort manuelt. Med dette utgangspunktet har også digital bildebehandling og bildekomprimering fått verktøy som er svært nøyaktige og åpner for detaljnivåer som uten fraktal geometri ikke ville vært mulig å oppnå. Topografiske kart utvikles på samme måte, der høydeforskjeller i teorien kan detaljeres ned til mikroskopisk nivå ved hjelp av fraktale algoritmer.

Meteorologi

Skyformasjoner og luftstrømmer er former for multifraktale systemer og kan forstås ved hjelp av fraktal geometri[38]. Verktøy brukt av meteorologer baserer seg derfor blant annet på fraktale algoritmer for å kunne forutse værets utvikling basert på observasjoner. Havstrømmer, stormer, hetebølger og andre værfenomener lar seg lettere og mer forutsigbart forstå gjennom fraktal geometri enn gjennom klassiske matematiske beregninger.

Kardiologi

Innen medisin og kardiologi er det mange som jobber med å utnytte bruksverdien til fraktal geometri[39]. Der for eksempel røntgenbilder og ultralyd ikke kan vise oss tilstander på det mikroskopiske plan i kroppen, kan fraktal geometri avsløre strukturer og mønstre i for eksempel blodårer og hjerterytmer som ikke er mulig å oppdage ellers uten å faktisk åpne opp

kroppen. Dette kan blant annet gi leger og kardiologer bedre verktøy for å kunne påvise kreft på et tidlig stadium eller oppdage forstadiet til for eksempel blodpropp.

Klimaforskning
Klimaforskere har tatt i bruk fraktal geometri for å forstå utvikling av temperatur globalt og i spesifikke geografiske områder over tid, og noen mener også at vi kan forstå fullt ut hvordan klimaet og atmosfæren oppfører seg basert på fraktale algoritmer[38]. Kartlegging av karbonopptakspotensialet til en skog har mye å si for klimaforskere som forsøker å forutse karbonutslipps påvirkning på det globale klimaet. Ved å undersøke den fraktale geometrien til trær og andre planter i for eksempel regnskog har klimaforskere fått et mye mer nøyaktig verktøy for å anslå karbonbindingspotensialet hos trær, skoger og enorme skogområder, som uten fraktale algoritmer ville krevd svært mye arbeid å kartlegge med samme grad av nøyaktighet.

Mandelbrotmengden
Benoit Mandelbrot var matematikeren som for alvor tok i bruk og ga navn til fraktal geometri. Mandelbrotmengden ble oppkalt etter ham, som vil bli forkortet MS (Mandelbrot-settet) videre i oppgaven. Gjennom datidens framvoksende datateknologi viste Mandelbrot verden noen av de vitenskapelige nytteverdier fraktal geometri åpner for. Ved hjelp av 1970-tallets moderne datamaskiner med mikroprosessorer klarte Mandelbrot å gjøre effekten av den banebrytende geometrien synlig for øyet, noe som medførte at fraktal geometri ble populært på mange andre felter enn matematiske og vitenskapelige på grunn av dets estetiske aspekter. Dette åpnet opp nye muligheter for blant annet designere og digitale kunstnere, i tillegg til utallige vitenskapelige bruksområder. MS er en fraktal. Settet har den egenskapen at geometrien endres ved forstørrelse av det, men igjen gjentar sin opprinnelige form på et annet nivå av forstørrelse. Settet er bygget opp av mindre versjoner av hovedstrukturen, slik at den fraktale selv-likheten tilhører hele settet og ikke kun de mindre bestanddelene. MS gjentar seg selv i sitt opprinnelige mønster uten noen ende, slik at uansett hvor langt inn man zoomer vil man se det samme mønsteret. Dette kan repeteres i det uendelige.

Komplekse tall
For å forstå funksjonen for MS trenger vi en liten innføring i komplekse tall. Det er viktig for å kunne forstå hvordan bildet blir produsert av funksjonen, og sammenhengen mellom fargevalg for hvert punkt. Komplekse tall er blant annet nødvendig for å løse differanselikninger og andregradslikninger på formen

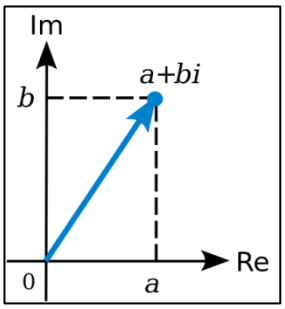
$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

der leddet $b^2 - 4ac$ er negativt. Når man kommer opp i denne situasjonen finnes det en løsning. Vi tenker oss at det finnes et tall i som kan representere $\sqrt{-1}$. Et tall som er opphøyd i andre potens og gir et negativt

tall vil gjøre at vi kan trekke fra røtter på negative tall. Vi kan regne med disse tallene på vanlig måte slik som med reelle tall, men vi må huske at $i^2 = -1$. Et komplekst tall kan beskrives på formen:

$$z = a + ib$$

der *a* er et reelt tall og *ib* er den imaginære delen. Nå skal vi se hvordan vi kan representere et komplekst tall ved hjelp av geometri. Vi definerer et plan der x- og y-aksen representerer den reelle og imaginære delen av tallet. Slik som på *Figur 24*.



Figur 24. Den reelle og imaginære aksen i et komplekst plan med punktet z=a + ib.[40]

Vi kan nå representere et komplekst tall i planet ved å bruke avstanden fra origo til et annet punkt og vinkelen mot den reelle aksen. Da får vi vektoren (a,b) som vi kan måle lengden av, fremfor å bruke koordinater på den reelle og imaginære aksen der $a + ib \neq 0$. Nå er det mulig å konvertere et komplekst tall til vanlig tall og bruke det for å representere et punkt i MS. [40]

Algoritme for å beregne en Mandelbrotmengde (MS)

En MS er en mengde med punkter i det komplekse planet som har omtrent stabile verdier. Det vil si at de kan øke og minke men ikke gå utenfor en gitt grense når funksjonen blir iterert over. Funksjonen som blir brukt er *Likning 4.*

$$z_{k+1} = z_k^2 + c \qquad (4)$$

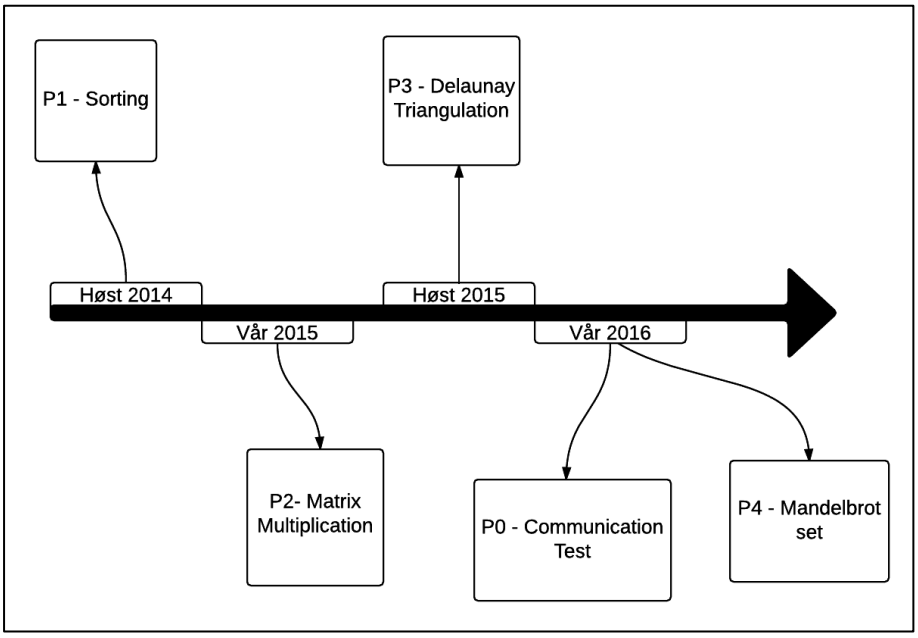
Likning 4. Mandelbrotmengden

Der $z_0 = 0$ og *c* er et komplekst tall som representerer posisjonen til et punkt i det komplekse planet, som er en geometrisk tolking av disse tallene jf. forrige avsnitt. Verdien til indeks $k + 1$ er da verdien man får av *c* pluss element med indeks *k* av *z*. Størrelsen av *z* er dens distanse fra origo. Det vil si at det er lengden av vektoren som blir definert av det komplekse tallets reelle og imaginære del. Dersom størrelsen $z = a + ib$ er tilsvarende $z = \sqrt{a^2 + b^2}$ og kan skrives på denne måten, og verdien av *z* blir større eller det samme som 2, så vil *c* akkumulere utenfor en satt grense. Det betyr at punktet ikke er med i MS. Men dersom vi iterer over 1 gitt *n* antall ganger og finner ut at z_n er mindre enn 2, kan vi konkludere med at punktet er i

settet S. Eksempelvis dersom $z_k < 2$, så skal *c* være med i MS. Hvis $z_k > 2$ skal *c* ikke med i MS. Verdien av *c* blir definert for hvert punkt i planet $h\ x\ w$ på grunnlag av en rekke faktorer som har å gjøre med bildeutsnitt, zoom-faktor, valgt oppløsning og punktets x- og y-koordinat i planet. Fargen for et punkt i planet, blir beregnet på grunnlag av den endelige verdien som z får. Denne verdien blir skalert med en logaritmisk funksjon som beregner lys-intensiteten til et tall mellom 0 og 254. Denne verdien blir *input* til en annen funksjon som beregner en egen 8-bit verdi for rød, grønn og blå piksel ut fra en forhåndsdefinert gradient.

3. Metode

Våren 2014 begynte jeg å undersøke temaet parallelle beregninger. I starten var det en stor jungel å finne frem i. Etter litt over to år med prøving og feiling har jeg gjennomført og kjørt fire prosjekter på Abelklyngen ved UiO. Jeg har kalt disse P0-P4, se Figur 25. Jeg startet med sortering (P1), dette ga ingen overraskende resultater. Etter det valgte jeg å se på matrisemultiplikasjon (P2) som vanligvis skal gi stor ytelsesforbedring med parallellisering. For å prøve å vise hvordan et større program med mange komponenter kan parallelliseres brukte jeg Delaunay triangulering som eksempel. Til slutt avsluttet vi med et enkelt problem som kunne skaleres opp til 512 kjerner med *speedup*, slik at jeg lettere kunne vise ytelsesforbedring fra en sekvensiell algoritme til en parallell algoritme med delt minne og med distribuert minne.



Figur 25. Tidslinje for praktisk arbeid i oppgaven.

For å kunne gjennomføre denne oppgaven var det en enorm mengde praktisk arbeid som var nødt til å utføres. Det er interessant å se hvor mange timer jeg brukte på de fem delprosjektene og årsak til tidsbruk. Timeantall i tabellen under viser antall effektive timer med programutvikling.

	Beskrivelse	Ca. antall timer med programutvikling	Kommentar
P0	Kommunikasjonstest	20 timer	Det tar tid å bruke Abelklyngen uansett
P1	Sortering	150 timer	MPI tar mye tid å lære seg
P2	Matrise multiplikasjon	200timer	Det tar mye tid å skrive C programmer med MPI
P3	Delaunay triangulering	500 timer	Oversettelse av kode mellom forskjellige abstraksjonsnivåer.
P4	Mandelbrotmengden	50 timer	Mange store filer øker ventetid

Tabell 2. Timer brukt i oppgaven på programutvikling.

920 timer = 25.6 arbeidsuker = ca. 6.4 mnd. tilsvarer over halvparten av tiden som er brukt på oppgaven. Rekkefølgen på problemene var ikke konsekvent, da jeg ikke viste hva jeg skulle velge å starte med. Designprosessen for et prosjekt er ikke nødvendigvis konsekvent eller rasjonell[41]. Sett i ettertid ville det kanskje vært mer naturlig å begynne med P0, men det var vanskelig å se nytten av kommunikasjonstesten så tidlig i prosjektet. Jeg synes det ble mer relevant når jeg skulle skrive oppgaven og legge frem resultat og observasjoner fra feltarbeidet.

3.1. Bruk av Abelklyngen

Abelklyngen bruker et Unix operativsystem og har ingen grafisk brukergrensesnitt med unntak av *xserver* som gjør at applikasjoner som bruker vinduer også kan kjøres. Det betyr at anlegget hovedsakelig ikke vil være tilgjengelig for brukere uten kompetanse på dette området. Å bruke en terminal er også mer tidkrevende enn et grafisk brukergrensesnitt for de fleste brukere. Kommandoene man bruker kan fort bli lange, og skrivefeil på ord og utrykk samt spesialtegn er de vanligste feilene å gjøre med mindre man bruker det daglig i en profesjonell kontekst. Da blir det en ekstra utfordring for brukere uten erfaring og det kan fort bli ekstra *kognitiv last*[42] som gjør at læringskurven blir brattere.

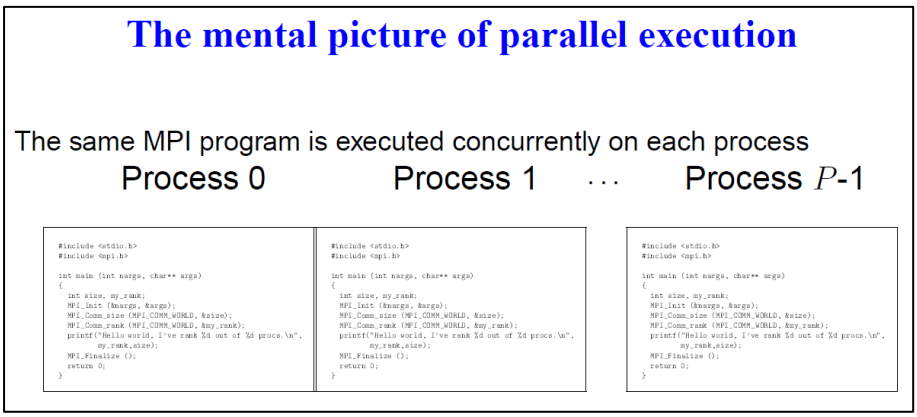
Det finnes derimot grafiske brukergrensesnitt til hjelpemidler som UiOs *LifePortal*[43]. Uten erfaring med parallell programmering og MPI kan det være veldig tidkrevende å sette seg inn i bruk av et tungregneanlegg. Det anbefales å ha noe erfaring med Unix, C og litt *shell scripting* for å ta i bruk anlegget effektivt. USIT (Universitetets senter for informatikk) holder kurs om HPC og bruk av Abelklyngen 2 ganger i året.

3.1.1. Kjøring av job-skript på Abelklyngen

Når programmet har kjørt og eksekveringstiden er notert, er det viktig å se den i sammenheng med andre kjøringer. Det vil si at det må velges ut et representativt resultat basert på flere kjøringer. En triviell måte å gjøre dette på er å regne gjennomsnittet av 10 eller 100 kjøringer. En annen mulighet er å bruke median av datasettet. Dette for å unngå ugyldige resultater som eventuelt kunne fått stor innflytelse på gjennomsnittet. Det kan for eksempel bli høyere tidsmålinger på grunn av Garbage Collection eller andre OS-rutiner som utgjør en feilkilde her[44].

3.1.2. SLURM - køsystemet på Abelklyngen

I terminalen brukes kommando *sbatch*, for å sende et *batch script* til SLURM, det vil si køsystemet for tilgjengelige ressurser på Abelklyngen. Scriptet sendes til SLURM fra kommandolinjen eller fra en fil, der #SBATCH er kodeordet foran en kommando. Etter at jobben er sendt til SLURM får den en jobb-id. Det kan ta litt tid før ressurser blir allokert, derfor må den eventuelt vente i kø til de nødvendige ressursene er tilgjengelige. Fildeskriptorene *standard out (0)* og *standard error (2)* blir sendt til en fil med navn "slurm-%j.out", der %j er jobb-id-nummeret. Når ressursallokeringen er ferdig, kjører SLURM en kopi av batch-scriptet på den første noden som er allokert før en kopi av programmet blir sendt ut til resten av nodene som skal brukes. Det må nevnes at SLURM kan være en flaskehals siden det er en kø. Det kan ofte bli feil i filbane eller annet som har å gjøre med mappestrukturer og input- og output-filer. Dersom man ikke oppdager dette før jobben blir kjørt får man bare ut feilmelding fra programmet og må legge dem inn i køen på nytt og vente til det er ledig for kjøring.



Figur 26. Her vises et konseptuelt bilde av hvordan et program blir kjørt i distribuert minne. Prosess 0 til prosess (P - 1) kjører på hver sine prosessorer. En node i delt minne eller flere distribuert[45].

3.1.3. Eksempel på hvordan et SLURM job-skript kan være

Det skal kjøres et skript for å starte opp et gitt antall jobber som er sendt inn til køsystemet med forskjellige node-konfigurasjoner (noder, prosessorer) = (2,2),(2,4),(2,8) osv. på Abelklyngen. Dette kan gjøres på følgende måte: Først må det gjøres en interaktiv innlogging til Abelklyngen fra en innloggingsnode. Deretter kan *sbatch*-kommandoer brukes fritt. Dette kan være hensiktsmessig siden jeg skal kjøre et variabelt antall node-konfigurasjoner, med varierende antall av både noder og prosessorer. Først kjøres et Python-skript som starter opp et antall SLURM-script per node-konfigurasjon, i SLURM-scriptet ligger kall til *sbatch* som sender inn jobben til Abelklyngen. Resultatene og utskrift fra programmet som kjøres på Abelklyngen blir lagret i filer. Da må resultatfilene behandles slik jeg får ut mediantiden av kjøringen. Jeg bestemmer selv format på resultatet fra programmet, så det blir trivielt å behandle dem. Når alle disse jobbene er ferdige kan medianen for hver serie beregnes og lagres på fil.

Fordi antall noder og prosessorer ikke er konstant, brukes interaktiv innlogging og *sbatch* direkte. Følgende løsning er en kombinasjon av å bruke et SLURM-script og interaktiv innlogging: Programmet har blitt kjørt på Abelklyngen med tilfeldig valgte noder rundt i anlegget. Det betyr at tilordningen av kjernenes plassering også er tilfeldig. Men det gjøres selvsagt forsøk på å tildele mest mulig optimalt. Antall noder som er blitt brukt ligger i følgende distribusjon: 1,2,4,8,16,32 og 64. Hver node har tilgang på 16 fysiske kjerner, men det har ikke blitt kjørt med fulle noder, de er som nevnt blitt distribuert tilfeldig rundt i anlegget der det er ledig. Dette er ikke et optimalt testmiljø på grunn av transaksjonskostnader for å sende data ut til beregningsenheter som ikke er på samme node eller i samme *hylleseksjon*. Derfor ble kjøringen også utført eksplisitt på én node, og i én hylleseksjon. Det er også et kjent faktum at dagens parallelle datamaskiner bruker langt mer tid på å sende et element fra en prosess til en annen, enn det tar å sammenligne to elementer.

3.1.4. Array jobs med *arrayrun*

Dersom man kun skal kjøre en jobb mange ganger for så å lagre resultatene for hver kjøring finnes det et annet verktøy. Det heter *arrayrun* og er nyttig dersom man skal kjøre flere instanser av samme oppgave, og den kan brukes dersom man har mange datasett som man ønsker å utføre samme operasjon på. Men etter at ArrayJob har fullført må man selv samle inn og sette sammen data-settene. *Arrayrun* kan brukes istedenfor egne scripts, og det er mer effektivt for SLURM å gjennomføre det, da det blir ett kall istedenfor kanskje 100 stk. som i et python-script som gjør samme oppgave. Et eksempel på når man skal bruke *arrayjobs* kan være dersom loddtrekning kan løse vårt problem. Dette er også kjent som Monte Carlo-simulering. For eksempel kan MC-simulering gi sannsynlighet for en sekser på terningen. Dersom vi lager et trivielt program som utfører dette, vil resultatet av «trekning» være ikke-deterministisk, det vil si forskjellig resultat for hver gang. For å gi en mer nøyaktig distribusjon av simuleringen må dette utføres flere ganger slik at man kan finne median eller gjennomsnitt av loddtrekningen (MC-simuleringen). Da er det nyttig å bruke *arrayrun.*

3.1.5. Relevante problemstillinger fra brukere på Abelklyngen

Noen problemstillinger er sentrale i forhold til bruk av Abelklyngen og dukker opp flere ganger. Jeg vil her gå gjennom to eksempler for å belyse dette. I det første eksempelet vil brukeren bevisst bruke parallellprogrammering. I det andre er det usikkert hva brukerne velger, eller om de har gjort et bevisst valg. I begge problemstillingene er det viktig å vurdere tidsbruk opp mot nytteverdi. Det er tidkrevende både å bruke Abelklyngen og å tilegne seg de nødvendige forkunnskaper. Brukeren bør først gjøre en solid vurdering på om det er fornuftig bruk av tid.

Eksempel I. Forsøk på ytelsesforbedring

Problemstilling: En forsker kommer med et program som har en effektiv implementasjon for løsning av et kjent problem. Programmet er kanskje skrevet i et høynivå språk (Java eller Python) eller i C. Han ønsker å parallellisere algoritmen for å undersøke hvordan den skalerer og hvor raskt det er mulig å løse det kjente problemet med mer nøyaktighet og høyere oppløsning.

For å gi en rask introduksjon innebærer det å forstå minne- og kommunikasjonsmodellen som brukes i dette paradigmet. Etter det kan man starte med å designe parallelle algoritmer, for så å implementere og teste dem. Det er en læringsprosess som kan kreve mange hundre timer å mestre. Dersom man skal finne ut hvor raskt det er mulig å løse problemet og om problemet eventuelt er skalert til nye høyder, eller det kreves en høy oppløsning, bør man absolutt investere tid i å lære seg å bruke en klynge. For å utnytte ressursene til et regneanlegg kreves det en god del forarbeid. Dette er faktorer man må overveie før man setter i gang.

For å utnytte en maskinklynges distribuerte minnemodell, trenger man et rammeverk som kan styre oppstart av prosesser på forskjellige noder og som har mekanismer for å kommunisere mellom disse prosessene. MPI er et slikt rammeverk. Høynivå språk som Python og Java må implementere disse rammeverkene for å kunne brukes. Det samme gjelder i C. Derfor er det ikke noen begrensning på språket man ønsker å bruke, men på ytelsen det gir. For å utnytte en klynges *hardware* på riktig måte, kan det være veldig krevende å få høynivå språk til å klare å kommunisere med MPI over et nettverk. Man kan argumentere for at det går vesentlig raskere å utvikle et robust program i et høynivå språk fremfor et lavnivå språk der man må gjøre nærmest alle ting selv (initialisering av variabler, allokering av minne, frigjøring av minne (garbage collection)), ettersom det finnes få datastrukturer og brukerfunksjoner som er mer typisk i høynivå språk. Derfor må disse implementeres etter bruksområdet for at de skal fungere på den måten som kreves.

Siden programmet ikke bare skal parallelliseres men også brukes i en distribuert minnemodell, trengs et rammeverk som har hardware-støtte for allokering og kommunikasjon på forskjellige noder i regneanlegget. Derfor

velges MPI som er den ledende programpakken i akademia og industrien[46]. Forøvrig burde valget falt på C som programmeringsspråk fordi det har bedre implementasjon av MPI enn høynivå språk har.

Eksempel 2. Praktisk bruk av HPC-ressurs
En forsker eller student i bioinformatikk (eller annet fagfelt) synes behandling av et kromosom-data tar mye tid. Det kan ta opp til en arbeidsdag å kjøre programmet på en vanlig datamaskin i laben eller på en laptop, og de har 24 filer med data de skal behandle. De blir tipset om å bruke Abelklyngen fordi det vil gå raskere. Ingen av partene har tenkt noe særlig over kompleksiteten det innebærer og bare antar at det er lurt.

Dersom det tar 8 timer å kjøre programmet én gang og dette skal gjøres 24 ganger, vil det totalt ta 96 timer i kjøretid. For å få kjørt programmet på en node trenger man ikke å skrive om programmet, det kan også kjøres på samme måte som på en vanlig maskin bortsett fra at kjørekommandoen må eksekveres fra et skript. Dersom man har tilgang til Abelklyngen kan man logge inn med forespørsel om allokering av antall kjerner, minne, diskplass med mer. Dersom denne innloggingen er vellykket og alle ressurser er allokert kan man sette i gang et jobbskript. For å få til det kan man bruke en mal for oppsett av filen og eventuelt en liten guide etter behov. Man må beregne at det tar 8-16 timer å lære seg å bruke Abelklyngen med én node. Det er det som skal til for å få kjørt et program på Abelklyngen.

Det er mulig å kjøre programmet et ønsket antall ganger med forskjellig input-fil i SLURM-skriptet som legger den i køen. Da kan man velge å bruke alle 16 fysiske kjerner på en node som tilsvarer å kjøre 16 jobber, hvis det er 16 filer å lese. Men dersom dataene kan deles opp, og blir behandlet i 16 deler, vil mulig speedup være opp mot 16. Om det tok 8 timer på en laptop burde det ta 30 min på Abelklyngen for å behandle en fil, dersom man kjører det som 16 jobber. Og så kan man sette i gang alle de 16 kjøringene samtidig. Det vil tilsynelatende bli parallelt. Og man kan i teorien behandle alle 16 filene omtrent samtidig. Eventuelt kan dette også gjøres med de 24 store filene. Send inn 24 jobber til Abelklyngen. Det vil i prinsippet gå 24 ganger raskere enn på brukernes laptop og de slipper at den er opptatt store deler av arbeidsdagen.

Oppsummering
Det er viktig å foreta en analyse av hva man kan tjene av tid fordi det kan være meget omfattende å sette i gang. Når det gjelder (Eksempel 1) så må programmet re-designes og alt som hører med i den prosessen. Når det gjelder (Eksempel 2) så er det ikke behov for re-design, men kjøring blir heller ikke en parallell beregning fordi programmet må parallelliseres slik som beskrevet i forslaget over.

3.1.6. Strømbruk - Abelklyngen

HPC er meget kostbart. Som eksempel har det blitt nevnt at en stor andel av UiOs energiutgifter går til HPC og Abelklyngen. Et essensielt spørsmål og viktig poeng i sammenligning av delt og distribuert minne er strømbruk. En node bruker ca. 250W på CPU (uavhengig av antall kjerner), og 250W på avkjøling av denne. Dersom det er 650 noder og en strømpris på 50 øre pr. kilowattime får jeg følgende utregning på neste side:

	Beregning	**Resultat**
Strømforbruk daglig pr. Node	500w x 24h = 12000	12Kw
Strømforbruk daglig pr. Node i kroner	12Kw x 50 øre = 600 øre	6 NOK
Strømforbruk daglig for Abelklyngen i kroner	6 kr x 650 antall noder	3 900 NOK
Strømforbruk pr. mnd. for Abelklyngen i kroner	3900 kr x 30 dager	117 000 NOK
Strømforbruk pr. år på Abelklyngen i kroner	3900 kr x 365 dager	1.42 mill. NOK

Tabell 3. Strømbruk for Abelklyngen.

I andre land der strøm er mer kostbart kan det fort bli det dobbelte i kWh. Da blir det plutselig 2.8 millioner kroner. Hver gang MPI sprer seg til en ny node øker kostnaden med 500w. Strømutgiften blir da $500\text{w} \times$ antall noder som brukes av MPI i distribuert minne, sammenlignet med en multikjerne som aldri bruker mer enn 300w (vanlig PC).

3.2. P0 – Kommunikasjonstest

3.2.1. Ping-testen

Jeg utfører en ping-test på Abelklyngen for å undersøke påstanden om at det går flere hundre ganger tregere å sende et heltall fra en node til en annen med C-MPI enn det tar å bytte om to heltall på en maskin, det vil si å skifte minneadresse på *to* variabler. Dette ble utskrift fra terminal:

	Kjøretider
Skifte minneplassering på et heltall, 10^6ganger	0,634 ms
Sende et heltall 10^6ganger	243,1 ms

Tabell 4.Ping-test med C-MPI.

Det vil si ca. 373 ganger raskere å skifte verdi på to heltall sammenlignet med å sende et heltall fra en node til en annen. Dette er ikke en ping-pong-test, kun enveiskommunikasjon.

3.2.2. Java på Abelklyngen

Etter at Abelklyngen ble prøvd ut med Mergesort i C, ønsker vi å se hvordan Java kan brukes med MPI på Abelklyngen. Fordi det er tidkrevende å utvikle C-programmer som er robuste og fungerer. Motivasjonen for dette er at det er meget tidkrevende å oversette eller utvikle gode robuste parallellprogrammer i distribuert minne med C. Derfor ønsker vi å se på muligheten for å bruke Java som utviklingsverktøy i dette domenet. Det forventes at ytelsen vil bli noe lavere, men formålet for bruk av Java er hovedsakelig å undersøke *speedup* og effektivitet for parallellisering av sorteringsalgoritmer.

Java-tråder

En tråd er en sekvens av instruksjoner som kan bli behandlet av operativsystemet. Man kan godt kalle dette en instans av programmet som kjører, men alle trådene er i samme adresserom og det vil si i samme prosess. I denne sammenheng er en tråd en mekanisme som gjør det mulig å utføre parallellberegninger på en multikjerneprosessor. Objekter og andre ressurser kan brukes av tråder. All felles data trådene behandler må synkroniseres.

Java MPI bindings

En binding for programmeringsspråk er et API som gir tilgang på funksjoner som er i et bibliotek til et annet programmeringsspråk. Det vil si å overføre en metode fra et språk til et annet. En binding er en slags innpakning som gjør at *to* språk kan bruke samme bibliotek. Som et eksempel er veldig mange OS-brukerprogrammer skrevet i lavnivå språk, Assembler eller C, noen ganger C++. For at høynivå språk som Java, Lisp og Python skal kunne bruke disse OS-rutinene, må det være en binding til det aktuelle biblioteket i det andre språket. For at Java skal kunne bruke MPI må det brukes *bindings* til C-implementasjonen for MPI. Dette må

kompileres ved siden av MPI. Så må man kjøre *java virtuell machine* (JVM) med *mpirun*. For å se hvordan Java kjører på Abelklyngen ble det prøvd ut noen tester for å se hvordan det virker og om det fungerer slik som forventet.

Java og C - høynivå mot lavnivå
Når det kommer til Java og C, er den viktigste forskjellen ytelsen som teknologien kan gi. Den viktigste grunnen til dette er at språkene er designet på to forskjellige måter. Etter kompilering må Java-programmet kjøres på en Java Virtual Machine (JVM). Mens i C kan det kompilerte programmet kjøres direkte på maskinvare, det vil si på en prosessor. I noen få tilfeller vil Java og C kunne konkurrere på ytelsen i for eksempel hastighetstester. Men det er tydelig at det vil bli noe *overhead* med bruk av JVM. For det meste er det C som vinner i testing. Det er grunnen til at C fortsatt blir brukt i applikasjoner som krever høy ytelse og mye minne. Men igjen er det mye mer krevende å utvikle applikasjoner i C, fordi man ikke har noe særlig *støtte-hjul*. For å si det med andre ord: Java kan sammenlignes med en ny stasjonsvogn som har ABS-bremser, airbag og setebelter. Mens C er mer som en formel-1 bil. Med andre ord er Java fint om det gir ønsket ytelse, mens C bør brukes når det er behov for høy ytelse og man har kompetanse til det[47].

3.3. P1 – Sortering

3.3.1. Mergesort med C og MPI
For å undersøke hvilken metrikk som best beskriver et problem som blir parallellisert, trengs det å gjøres noen eksperimenter. I dette første eksperimentet opprettes en vektor med tilfeldige tall. Vektorens lengde vil være det største mulige tall i denne sammenhengen. Dersom antall prosesser er 2, og det er 1000 elementer, blir det 500 elementer for hver prosess å sortere. Dette kan gjøres i parallell med forskjellige algoritmer, for eksempel *Quicksort*, *Radix* eller *Mergesort*. Hver prosess bruker som regel en sekvensiell sorteringsalgoritme når den skal gjøre sitt lokale arbeid. Grunnen til dette er at alle prosesser må gjøre sin del av beregningene og det har ingen hensikt å parallellisere denne delen fordi det ikke vil gå noe raskere. En parallell beregning er hovedsakelig å gjenta en sekvensiell beregning på antall prosesser som brukes. Derfor blir ofte en parallellisering av sortering mer en øvelse i distribusjon og innsamling av data. *Root-tråden*, det vil si den som starter opp alle de andre vil til slutt samle sammen alle de sorterte tabellene og flette dem sammen. Et eksempel på fordeling av data kan være:

$$\frac{antall\ elementer}{antall\ tråder} = \frac{n}{p} = \frac{1000}{2} = 500$$

For å få varierte resultater brukes forskjellige verdier av n og p. Størrelsen på datasettet vil variere mellom *tre* forskjellige lengder av tabell, [10^5-10^8]. Det kan være lurt å starte med to kjerner også på maskinklyngen for å få det til å fungere. Resultatet kan måles med funksjonen *MPI_Wtime*, som gir

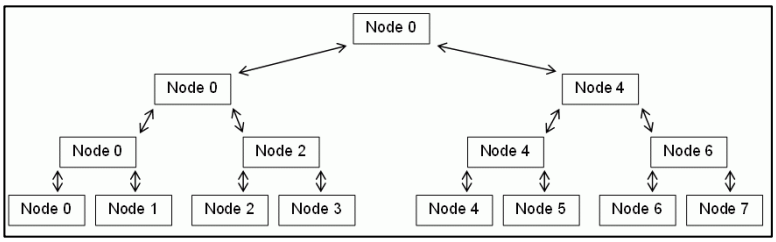
tiden i sekunder med god presisjon.

Etter at tabellen er sortert må det gjøres en test for å verifisere at resultatet er riktig. Da må en funksjon gå gjennom hele tabellen som tar O(N) i tid og gjøres ved å sjekke om $vektor_i \leq vektor_{i+1}$ og eventuelt avslutt dersom det ikke er det.

3.3.2. Oppbygging av struktur for mergesort

For å utnytte maskinklyngen optimalt er det viktig å tenke struktur i forhold til hvordan problemet blir delt opp. Å la en prosess sende ut data til mange prosesser som sorterer en del hver, vil føre til at den første prosessen blir sittende igjen med mye arbeid selv. En løsning her er å implementere et binær-tre med en tabell. Det oppstår følgende relasjon mellom nodene: For et tre med K nivåer blir venstre (barn-node $(2 \times \mathrm{K}) + 1$, høyre barn-node blir $(2 \times \mathrm{K}) + 2$ og $\frac{k-1}{2}$ blir indeks for dens forelder-node. En node representer i denne sammenhengen en MPI-prosess. Ideen her er at forelder-nodene deler sin tabell i to og sender dem til barne-nodene. Barne-nodene gjør sortering og sender det tilbake til forelder-nodene som utfører flettingen av tabellen.

Kommunikasjon mellom nodene er kostbart og burde reduseres så mye som mulig. Det er en fare for at forelder-noder står stille og venter på at data skal innrapporteres. En mulighet til forbedring er å la foreldre sende halvparten av data til barne-noden og sortere halvparten selv, da blir den også en del av neste nivå i treet, se *Figur 27*.



Figur 27. Sendeskjema for MPI-prosessene.

Dette implementeres ved hjelp av en vektor som kan ha pekere fra de forskjellige posisjonene med en spesiell rekkefølge. Da blir det egentlig bare *to* nivåer, og ikke *fire* som tegningen kan gi inntrykk av.
Hvilke barne-noder som hører til sine forelder-noder må være kjent slik at data kan sendes riktig vei oppover i treet. Dette kan beregnes ut fra treets høyde. Node 0 på nivå 3 må kommunisere med node 4. Dette problemet kan løses ved å skru på bit 2 fra venstre. Fordi binærtreet er implementert med en vektor, er det mulig å finne alle relasjoner mellom barn- og forelder-nodene med beregning som bruker bit skifting-teknikken (bit-shift).

Følgende *bit-shift* operasjon kan utføre dette:
Prosess-id || 1<<(2)

På nivå 2 må node 0 kommunisere med node 2, mens node 4 trenger å

kommunisere med node 6. Da holder det å skru på bit 1 fra venstre. På øverste nivå må noder som har partall-id kommunisere med oddetallsnoder. Dette gjøres ved å skru på bit 0.
Generelt kan dette skrives som:
Prosess-id || (1<<(nivå-1),

På denne måten kan en MPI-prosess regne ut hvem som er prosessens forelder og barn på grunnlag av sin egen prosess-id. Og på den måte sende til riktig mottaker i trestrukturen. [48]

3.3.3. Sortering i Java

Jeg har oversatt mergesort fra C til Java. Programmet var skrevet med MPI og jeg kjører det med MPI-Bindings på Abelklyngen.

3.4. P2 – Matrisemultiplikasjon

Det som er viktig her er antall oppgaver og størrelsen på dem. Det avgjør hvilken grad av granularitet dekomponeringen kan få. Det vil si oppdelingen for en parallell beregning. Dersom det blir mange små oppgaver blir dette kalt for finmasket granularitet. Dersom designet for den parallelle algoritmen skal være effektiv for dette problemet er dette noe man må ta hensyn til. Programmet som utfører MM gjør følgende:

1. Den første prosessen (0) leser inn matrise A og B fra en datafil binært.
2. Prosess 0 distribuerer A og B radvis til de andre prosessene.
3. Alle prosesser beregner sitt intervall av $A \times B = C$, parallelt.
4. Prosess 0 samler inn data fra de andre prosessene om C i forskjellige intervaller.
5. Prosess 0 skriver C-matrisen til en fil.

3.4.1. Design av parallellisering

I denne løsningen skal jeg bruke funksjonen *scatterv* for å distribuere matrisene radvis. Etter at alle har beregnet sin del, brukes *send/recv* for få radene tilbake på riktig plass hos root-prosessen. Siden jeg har valgt en radvis inndeling, starter jeg med å allokere minne til alle prosesser. Så starter root-prosessen å lese matrisene fra fil. Jeg transponerer B-matrisen etter innlesning for å gjøre det enklere med distribusjon senere og for å bedre ytelsen. Deretter blir dimensjonene distribuert til de andre prosessene. Deretter beregnes antall rader som skal fordeles mellom prosessene. Root beregner hvordan matrisen skal deles opp og fyller pekerne som skal brukes i *scatterv*. Så blir B-matrisen til alle de andre prosessene også transponert. Det gjøres for at radene skal ligge riktig før beregning. Alle gjør kall på *scatterv* som distribuerer matrise A og B fra root der lastbalanse er tatt hensyn til. De forskjellige prosessene får litt ulikt antall elementer for å oppnå dette. Så fyller root-prosessen opp sine tomme rader med «0» slik at det blir tryggere og enklere å plassere radene som kommer fra andre prosesser. Alle beregner radene sine og sender dem som blokker til root sin C-matrise. Dette gjøres like mange ganger som det er prosesser. Etter dette

blir B-blokka sent til naboprosess som trenger den for å beregne neste C-blokk. B-blokkene til $prosess_i$ og $prosess_{i+1}$ må ha forskjellig størrelse for at de alle skal passe inn i C til slutt. Vi velger å fylle opp C diagonalt fremfor radvis fordi det går raskere. *d_index* definerer hvilken diagonal og iterasjon vi er på, i forhold til antall prosesser. Root-prosessen tar imot alle C-blokker fra de andre i stigende rekkefølge på prosess-id.

3.5. P3 – Delaunay triangulering

3.5.1. Design – sekvensiell versjon

Denne versjonen av Delaunay triangulering kan ta inn et sett med punkter (x,y) i planet. Input til programmet er antall punkter og antall punkter pr. boks. Planet blir delt opp i bokser som er proporsjonal med N. Denne datastrukturen gjør det mulig å dele opp DT i en *splitt og hersk*-tankegang. Deretter sorteres punktene parvis langs x- og y-aksen, for å kunne behandles i stigende rekkefølge. Neste steg er å finne den konvekse innhyllingen. Det gjøres ved at man først finner alle maks. og min. ytterpunkter for x og y. Så legges de inn i en liste, deretter beregnes neste punkt ut fra et av ytterpunktene. Når alle punkter har blitt beregnet, returneres antall punkter som lå på innhyllingen. Nå er det mulig å gjøre en DT. Så beregnes de punktene som er inni den konvekse innhyllingen. Dette gjøres også ved å finne alle de nærmeste naboene til alle punktene. For hver av disse indre punktene, finnes så den nærmeste naboen fordi kanten mellom et punkt og dens nærmeste nabo er en Delaunay-kant (DK)[32].

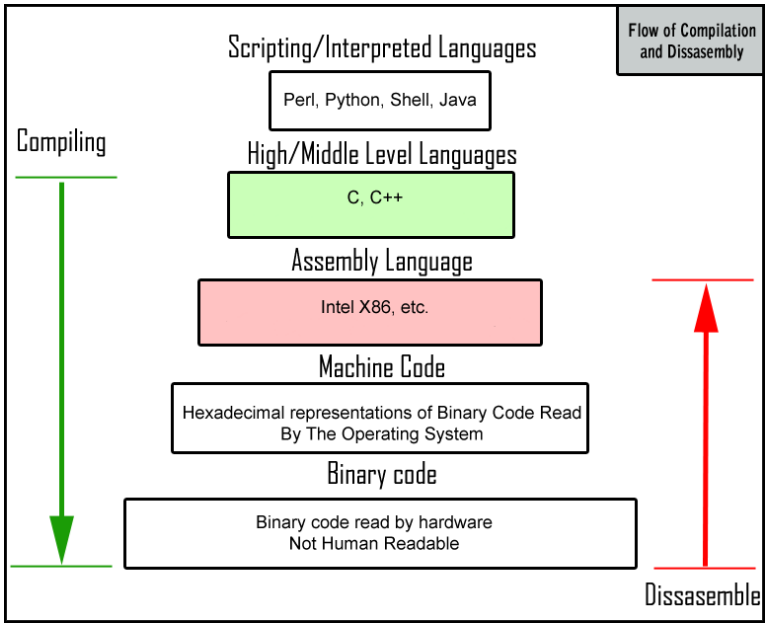
3.5.2. Implementasjon - sekvensiell versjon

Datastrukturene som er nødvendig for å få designet til fungere, innebærer bruk av bokser for partisjoneringen av planet. Det er tilstrekkelig med todimensjonal heltalls-matrise for å få til dette. x- og y-koordinatene til alle punktene legges i vektorer. En boolsk vektor brukes til å merke av hvilke punkter som er behandlet, en FIFO-liste implementert med heltalls-vektor og en heltalls-vektor brukes for å holde orden på alle naboene et punkt har. Videre blir beregning av triangler lagt inn i en todimensjonal heltalls-matrise ved at første indeks peker til det aktuelle punktet, og andre indeks representer punktets naboer. Det er tilstrekkelig for å vise en Delaunay triangulering og kan brukes til å lage en avbildning av planet. For å sortere punktene som kommer inn brukes *Radix*-sortering implementert med *bitshift*. Det gir en tidskompleksitet på $O(kN) = O(N)$, der k er antall bits på hvert tall[31].

3.5.3. Oversetting av DT fra Java til C

Det største feltarbeidet som ble gjennomført i denne oppgaven var å oversette en prototype av DT skrevet i Java til C. Dette var krevende fordi det var så mange som 1000 linjer som skulle oversettes og disse er på forskjellige abstraksjonsnivåer, se figur 29. For å få kjørt den på Abelklyngen måtte den porteres. Til å begynne med måtte det gjøres noen betraktninger for å se om det var gjennomførbart og hensiktsmessig å oversette et program på omtrent 1000 linjer kode. Svaret var ja av mange grunner: For det første var dette en algoritme som var ryddig og mulig å forstå, sammenlignet med mange andre versjoner som ligger på internett. Å

implementere DT fra bunnen av er meget tidkrevende og det er vanskelig å garantere at det fungerer innen rimelig tidsbruk. Min veileder har implementert algoritmen med Java, og det var relativt enkelt å forstå oppbyggingen av programmet selv om det er relativt stort. For det andre er DT ganske kompleks og det er viktig å forstå hvordan programmet og algoritmene fungerer for å kunne gjøre en fornuftig parallellisering. En utfordring i oversettelsen var å bryte opp objekt-orienteringen og dele opp programmet i flere filer med *struct'er* som brukes til å implementere datastrukturene. Dette er en velprøvd versjon av algoritmen som gir meget gode resultater og kjøretider i sekvensiell versjon. Delaunay triangulering er en ganske kompleks prosedyre og egnet seg bra for å teste ut kapasiteten til Abelklyngen.



Figur 28. Forskjellige nivåer av abstraksjon i programmeringsspråk[49]. Oversetting mellom et nivå er tidkrevende.

	C	Java
Funksjon	int max(int a,int b)	int max(int a,int b)
Opprette datastrukturer	typedef struct{ int x; int y; }Point;	Class Point{ int x; int y; }
Peker	int *a;	Finnes ikke
Tilgang på datastruktur	Point *p = (Point *)malloc(sizeof(Point)) p->x = 1;	Point p = new Point (); p.x=1;
Minne allokering	malloc(int antall bytes)	New/ automatisk
Frigjøring av minne	free(*ptr)	Automatisk, (garbage coll.)
Lagring av datastrukturer	Heap, stack eller data_block	Heap
Unntak og kjøretidsfeil	Manuelt med *assert(!(True))*	automatisk
Overloading av funksjoner	nei	ja

Tabell 5. Sammenligning av C og Java syntaks.

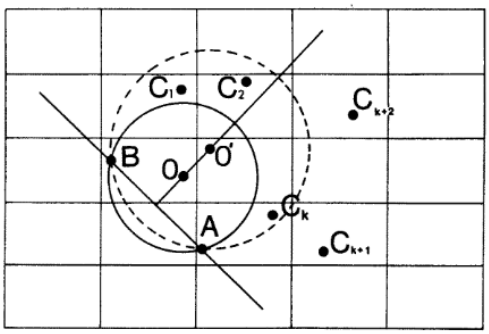
De *tre* viktigste forskjeller som gjør at det er mer krevende å skrive programmer i C er at vi må jobbe med pekere, vi må allokere minne selv og vi får ikke «runtime-error-message» (det vil si melding om feil under kjøring). I C må dette eventuelt gjøres manuelt med *assertion*[50]. De er disse faktorene som hovedsakelig gjør at C programmering er mer tidkrevende å bruke fremfor Java[46]. Ved første øyekast er Java og C ganske like i syntaks, men der stopper likhetene også. Se Tabell 5.
For å oversette DT-algoritmen er det hovedsakelig to muligheter man har:

I.) En-til-en oversettelse. Det vil si å foreta en gjennomgang av hver linje, fra start til slutt.
Ulempe: Det kan bli mye *ubrukt* kode.

II.) Lag en systemspesifikasjon for programmet. Skriv inn ny hovedstruktur med tomme filer og funksjoner. Så oversetter man alle funksjoner en-til-en, i en fornuftig rekkefølge i forhold til hva som er nødvendig for programmet.
Fordel: Koden blir mer ryddig.

Hvilken av angrepsmåtene over man bør bruke, er avhengig av størrelse på koden og programmets strukturelle oppbygging. Noen ganger kan en kombinasjon være lurt. Når oversettelsesteknikken er valgt er det viktig å sette opp et godt testmiljø for å klare å holde oversikt på de 1000 linjene med variabler og funksjoner. Det kan også være gunstig å lage noen egne programpakker med funksjoner man bruker ofte. Jeg måtte eksempelvis skrive flere matematiske funksjoner som ikke har samme oppførsel i C som i Java (Random, Max, Min, Abs). I Java er de innebygget mens i C måtte jeg skrive dem slik at de oppførte seg og kunne brukes på samme måte som i Java-versjonen av DT. Oversettelsen av (DT) programmet fra Java til C ga

også mulighet til å drive med litt testdrevet utvikling[51] som er et kraftig verktøy for å utvikle programmer effektivt mens man tester funksjonalitet. Når programmet skal testes og feilsøkes, bør man bruke samme punktspredning (*randomPoints*) i Java-versjonen av DT-programmet (JDT) og C-versjonen av DT (CDT). Under feilsøkingen var hovedproblemet at jeg ikke fant punktet *c* i trekanten *a-b-c*. Derfor prøvde jeg å løse en punktspredning uten ko-sirkularitet, noe som viste seg å være enklere. Med andre ord, når punkt *c* ikke blir beregnet riktig er det fordi det er ko-sirkularitet. Ko-sirkularitet i denne sammenhengen betyr at vi har *to* eller flere konkurrerende kandidater for punktet *c*, i trekanten *a-b-c* med samme vinkel over AB.



Figur 7. Ko-sirkularitet i beregning av en Delaunay triangel. C1-Ck er de forskjellige kandidatene til c i a-b-c.

Dette ble gjort for å forenkle kjøring av programmet for å se at det virket for datasett uten ko-sirkularitet. Dette gikk heller ikke så bra. For å få DT til å terminere, måtte punkt *c* settes til 1 eller mer, og løkken rundt funksjons-kallet på *findNextNeigbour* måtte bort. Da ble det mulig å få kjørt programmet slik at det terminerte.

Under testing og feilsøking ble sortering implementert feil på grunn av lite dokumentasjon om hvordan punkt-par skal sorteres, og det kom mange udeterministiske resultater under feilsøking. Feilsøking kan fort bli som å finne nåla i høystakken. Det kan være meget vanskelig, men det finnes noen måter å gjøre det på så lenge man går systematisk til verks. Feilsøking i C kan være meget tidkrevende fordi feil i minne-allokering ikke blir rapportert fra kompilatoren. Dette er et eksempel på skrivefeil fra utviklingen av CDT:
"printf("(c->x[%d] + c->x[%d])/2 = %d \n", a,b, (c->x[a] + c->x[b]) / 2 ;".
C-kompilatoren gir *segmentation fault*. Selv om vi her ser at det er skrivefeil med antall parenteser, er det ikke så lett å alltid oppdage det i koden.

Neste steg var å bruke et feilsøkingsverktøy. GDB og Valgrind ble tatt i bruk. De gir linjenummer til adresser for feil i minne-allokering, også kjent som *segmentation fault*. Disse verktøyene er viktig å lære seg dersom man skal lage robuste programmer som skal kunne parallelliseres på en distribuert minneplattform.

Dersom man sender med en tom peker blir det ikke sagt ifra om det fra kompilatoren, det vil si at mangel på en Context-peker i et funksjonskall ikke nødvendigvis gir en feilmelding. Dette er fordi kompilatoren ikke bryr seg om pekeren er tom eller peker på noe, med mindre man selv sier ifra om det. Derfor er det viktig å bruke *assertion* for å oppdage dette. I C må man selv skru *assertion* av og på etter hvor de er plassert rundt i koden. Det er veldig viktig å skru av denne mekanismen når feilsøking er ferdig, fordi de kan gi opphav til nye feil i programmet.

På et tidspunkt kom jeg frem til at funksjon *line* var *overloaded* i JDT. Derfor måtte det lages to forskjellige varianter i CDT, line_int() og line_double(). I JDT var også dist-funksjonen *overloaded*, og dette måtte ryddes opp i for å få den til å virke med *double*.

Et lite gjennomtenkt design fra starten kan gi store problemer under feilsøking. For eksempel ideen om at man bare sender inn en *struct* som parameter når det trengs flere enn to variabler fra den. Designet må være konsekvent og avvik fra det vil bare bli rot. Neste steg var å teste JDT mot CDT i stegvise deler av programmet. Steg 1 er *initate* i JDT mot *newContext* i CDT med sorteringen og beregning av rektangelet. Her sjekket jeg at alle utskrifter var like i begge versjoner, før jeg gikk videre.

For at CDT skal få lik oppførsel som JDT må man huske på å bytte ut *malloc* med *calloc* som er initialisert med verdien *0*, siden en vektor i Java automatisk blir initialisert med *0*. I første forsøk av oversettelsen ble dette ikke tatt hensyn til. Flere funksjoner i JDT tok høyde for at tomme plasser i en vektor er det samme som *0*, men det er ikke slik i C, med mindre man selv sier at det skal være slik.

Feilberegning av allokert minne

Det er viktig å bruke sizeof(int*) og ikke sizeof(int) dersom man skal ha en matrise, siden det er en peker med peker i C. Et tidlig problem i oversettelsen var feil i allokering av en *struct.*

En peker er 8 bytes lang. En struct eller type_def blir selvsagt en sum av alt den har i seg, sizeof(Context)=144 og sizeof(Context*)=8. Og det er 144 som blir riktig. Et annet eksempel er når man skal lage en matrise. Først allokeres antall pekere i en retning:

*matrise = (double **) malloc(lengde*sizeof(int *));*

Så allokeres data som pekes på:

for(i=0;i<lengde;i++) matrise[i]=(int)malloc(bredde *sizeof(int));*

Dersom man glemmer «*» i malloc(lengde*sizeof(int *)) blir det lengde ganger 4 og ikke lengde ganger 8.

DT uten ko-sirkularitet

Det var da mulig å kjøre en triangulering når ko-sekularitet ble skrudd av. Neste steg var å feilsøke CDT slik at det kunne kjøres med ko-sekularitet, deretter å kjøre et eksempel med ko-sekularitet, og til slutt et tilfeldig generert datasett. Etter det måtte data skrives til en fil og så kjøres i JDT for å verifiseres grafisk, og med en testutskrift. Det ble ikke brukt tid på å

oversette den grafiske fremstillingen av løsningen på grunn av at det ikke faller innenfor oppgavens område.

Oppdagelse av feil med funksjon for absolutt verdi
Etter at det ble kjørt datasett under 100 punkter med og uten ko-sirkulære tilfeller dukket det opp feil i CDT på datasett med 100 punkter og større. Mest sannsynlig var det et ko-sirkulært tilfelle som fikk CDT til å lage et ugyldig triangel med kryssende kanter. Det er to variabler *(sr og sr2)* som brukes til å lagre radiusen på sirkelen som blir undersøkt når man har et ko-sirkulært tilfelle. Det ser ut som en *overflow* i *sr2* eller *sr*. De starter å akkumulere når flyttallsvariabelen *sx*, som brukes for å beregne midtnormalen i x-retning, blir negativ. Dette forkommer ikke i JDT.

Med bruk av *assert* ble *overflow* oppdaget. Spørsmålet er hvorfor det blir overflow. Det er åpenbart at hoved-løkken gikk for mange ganger, siden *sr* og *sr2* akkumulerte til den maksimale grensen for flyttall i C. Siden jeg ikke fant ut av dette med det første, lurte jeg på om det var fordi jeg hadde skrevet noe feil i kode. Eneste måte å finne ut dette på var å sjekke med utskrift fra JDT.

Funksjonen *findNextNeighbour* er hovedkomponenten i DT-funksjonen.
Det var denne funksjonen som måtte undersøkes nærmere for å finne feil som ledet opp til et ugyldig punkt for *c*.

Etter å ha sett nøye på verdiene som går inn og ut av denne if-testen:
if (abs(dist - minDistToAB) < context->EPSILON && c != p)
Og ved å bruke utskrift som ble dumpet til fil, var det mulig å oppdage hvor feilen lå.
Dersom man ser litt på funksjonen *findNextNeighbour,* er det opplagt at man må se mye utskrift for å komme til roten av problemet, siden det er *tre* nesta-løkker ikke medregnet de i funksjonskall. Se *Figur 31* som viser et utdrag av utskriftsdump.

```
initiate()
    sort points by (x,y)
    put points in grid structure
end initiate

cohull()
    find_cohull_points()
end cohull

delaunay_triangulation()
    for i = 0 to cohull_length:
        findRestOfNeighbours()
            findNextNeighbour()
                if(coCircular):
                    findCoCirc()
                end if
            end findNextNeighbour
        end findRestOfNeighbours
    end for

    for i = 0 to rest_of_points_not_on_chull_length:
                closestNeigbourTo()
                findRestOfNeighbours()
                    findNextNeighbour()
                        if(coCircular):
                            findCoCirc
                        end if
                    end findNextNeighbour
                end findRestOfNeighbours
    end for
end delaunay_triangulation
```

Figur 29. Pseudokode for DT-algoritmen. Algoritmen for å finne den konvekse innhyllingen er i Figur 30.

```
find_cohull_points()
    for i = 1 to number_of_points:
        if (x[i] < x[minx]) minx = i
        if (x[i] > x[maxx]) maxx = i
        if (y[i] < y[miny]) miny = i
        if (y[i] > y[maxy]) maxy = i
    end

    put minx in theCoHull_list
    put minx and minx in coHullMap_list
    /*
    * cohullRec() used in rest of code finds a possible next point
    * minP between p1-p2 closest to p1 on
    * the convex hull. Recurse on p1-minP
    * and  minP-p2
    */
    if (miny != minx):
            putAfter miny and minx in theCoHull_list
            put miny and miny in coHullMap_list
            cohullRec(minx,miny);

    if (maxx != miny):
            putAfter maxx and miny in theCoHull_list
            put maxx and maxx in coHullMap_list
            cohullRec(miny,maxx)


    if (maxy != maxx && maxy != minx):
            putAfter maxy and maxx in theCoHull_list
            put maxy and maxy in coHullMap_list
            cohullRec(maxx,maxy);

    if (maxy != minx):
        cohullRec(maxy,minx)
    else
        cohullRec(maxx,minx)

    return theCoHull.numElem
end cohull
```

Figur 30. Pseudokode for cohull-algoritmen (den konvekse innhyllingen).

Figur 31. Diff sjekk mellom JDT og CDT ble brukt for å oppdage hvor feil var.

Jeg utførte en kjøring av JDT og en med CDT, og dumpet utskriftene jeg ville se nærmere på til en fil. Så tok jeg en og flere *diff-sjekker* av filene. Til slutt klarte jeg å oppdage en slags avrundingsfeil som dukket opp på grunn av en *typecasting* av input-parameter. Det finnes nemlig ikke så mange IDE og feilsøkings-verktøy for HPC. Eclipse har versjonen PTP som kan brukes for HPC med C eller C++. Men Eclipse er del av et større økosystem med

Java og Java teknologi. Det finnes ingen ordentlig satsning på utviklings og feilsøkings verktøy for HPC kan det virke som[52].

Det ser ut til at JDT bruker flyttalls-varianten av *abs* og CDT bruker heltall-varianten av *abs*. De returnerer ulike verdier, som var årsaken til at if-testen der *abs* blir brukt til å evaluere ble feil og returnerte *true* litt for ofte. Det førte til beregningsfeil i beregning av punktet *c* og løsningen i DT. Derfor skiftet jeg til *fabs* i CDT som en flyttalls-variant, og det fungerte fint. Da var det ikke lenger noen differanse mellom utskriften og oppførselen til JDT og CDT da data ble lest fra fil.
abs i Java funger slik at dersom man sender inn et heltall til funksjonen, så er det heltall-varianten som blir brukt. I klassen Math er abs definert for typen *int*, *float*, *double*, og *long*. Da blir den typen som sendes inn som parameter den typen som *abs* returnerer. I JDT er dette *double*. I *math.h* som C bruker, er det slik at man har *abs, fabs* og *labs*. Dersom man ikke er klar over dette blir det fort avrundingsfeil, ettersom flyttall-inputs naturligvis vil bli *castet* til heltall.

Feil med random generator
Siste hindring for å få kjørt CDT, ble å få *NPartallsPunkter* til å virke. I JDT ble Random.nextInt() brukt. I CDT implementerte jeg en lignende funksjon med bruk av rand() fra stdlib. Det viste seg å ikke fungere helt på samme måte som da jeg brukte *modulo operator* (Det vil si %) med rand() i C for å gjengi oppførelsen til Random.nextInt(). Det var vanskelig å oppdage at det var en feilkilde. Men en god huskeregel her er at man ikke kan forvente samme oppførsel fra innebygde rutiner i et språk og et annet. Derimot kan man forvente samme oppførsel på konsepter som bruk av operatorer, og resten av semantikken, siden det er språkuavhengige konsepter. Derfor ble det implementert en *Pseudorandom number generator* (PRNG) med *xorshift*. Da ble det mulig å garantere lik oppførsel i JDT og CDT. Den gikk raskt å implementere og ga relativt rask ytelse sammenlignet med den innebygde varianten for begge språk.

Det er viktig å se gjennom programmet og passe på at alt minne er frigjort når det skal.
Det må ryddes opp i eventuelle minnelekkasjer, fordi dette kan gjøre utvikling og feilsøking av parallellisering mer kompleks enn nødvendig. *Valgrind* er et program man bør lære seg dersom man skal lage robuste C-programmer og i tillegg parallellisere dem for distribuert minne.

Målt ytelse for Delaunay triangulering i C og Java
Dersom man sammenligner C og Java på ytelse vil C som regel gjøre det best sammenlagt, selv om kanskje noen tester går raskere i Java[47]. Oversettelsen og porteringen av DT har derimot ikke som mål å oppnå bedre ytelse på C-versjonen. Målet er å kjøre DT på Abelklyngen. Det er absolutt mulig å få bedre ytelse med C, men jeg har ikke brukt mye innsats på det her i denne oppgaven. Derfor har testene ganske like tider. En annen grunn er at typiske Java-konstruksjoner som *class* og *object* ikke nødvendigvis lett oversettes til optimal C-kode. Koden kan være triviell for

Java-kompilatoren å optimalisere, men det betyr ikke at en oversettelse til C vil kunne optimaliseres av C sin kompilator med samme resultat.

Resultater fra kjøring på Lenovo bærbar med Intel i7-2620MCPU @ 2.70 GHz (2 kjerner med hyperthreading, (4 logiske beregningsenheter)), 512 GB SSD disk og 8 GB ram. Tidene er inkludert lesing av fil. Dette var nødvendig for å garantere samme datadistribusjon for begge kjøringer av DT siden det er implementert forskjellig i C og Java.

3 kjøringer av JDT og CDT med $n = 10^5$:

	$n = 10^5$	$n = 10^5$	$n = 10^5$
C	381 ms	378 ms	379 ms
Java	701 ms	687 ms	705 ms

Tabell 6. JDT og CDT med $n = 10^5$.

3 kjøringer av JDT og CDT med$n = 10^6$, (1 mill.):

	$n = 10^6$	$n = 10^6$	$n = 10^6$
C	5253 ms	5258 ms	5244 ms
Java	5170 ms	5160 ms	5114 ms

Tabell 7. JDT og CDT med $n = 10^6$.

Oppsummering
Som tidligere nevnt var hensikten ikke å vise at C har bedre ytelse enn Java. Men uten å gjøre noen store optimaliseringsforsøk ble resultatene ganske like. Erfaringen jeg sitter igjen med er at et det ville gått fort å gjøre en oversettelse fra C til Java, med tanke på kompilering og feilsøking, og at ytelsen til CDT er som vist i avsnittet over. Dersom man ønsker å teste en teori eller algoritme for å løse et problem vil man oppnå svaret raskere ved å implementere det i Java. Men dersom selve resultatet skal produseres raskt vil det gå fortere i C. Java er fint til noen typer forskning, C er for industrien når ytelse er viktig[47]. Det jeg ønsker å si her, er at det er mye vanskeligere å få noe til å virke slik det skal med C fremfor Java.

3.5.4. Design av parallellisering
Dette kapittelet beskriver en parallellisering av DT i distribuert minne (MIMD) og stegene:

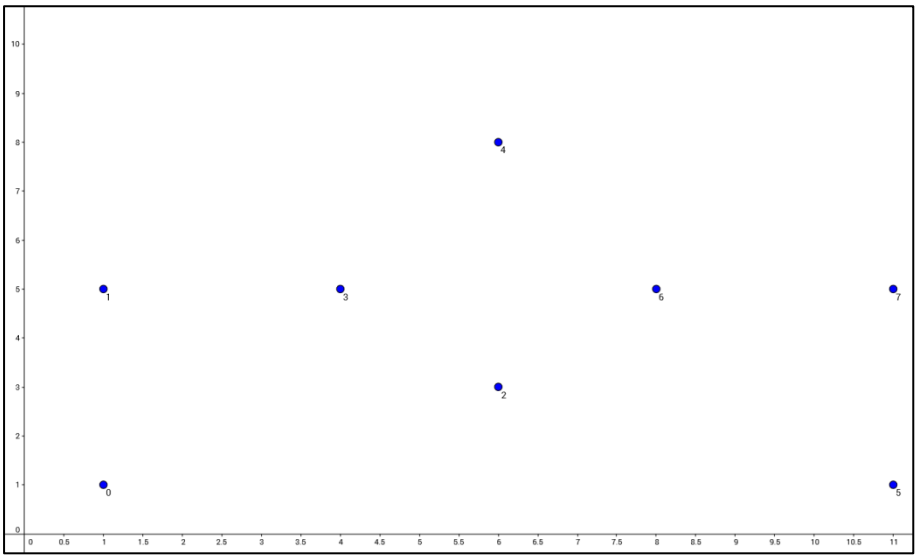
1. Alle prosesser initialiserer variabler og legger data inn i datastrukturer.
2. Alle prosesser sorterer punktene.
3. Alle prosesser finner konveks innhylling for punktspredningen.
4. Alle prosesser kaller på DT funksjonen. Intervall for hver prosess beregnes på følgende måte: Dersom vi har N punkter og P prosesser, skal hver prosess ta $\frac{N}{P} + rest\ punkter$. Eksempel med 10 punkter

og P=2, proc0=(0,1,2,3,4), proc1=(5,6,7,8,9). NB. Dette gir ikke nødvendigvis en rettferdig fordeling på grunn av at det vil avhenge av distribusjonen av punkter og om de ligger på den konvekse innhyllingen eller i den. Alternativt kan dette fordeles rettferdig, men det krever at vi teller opp eksakt hvor mange av punktene som er på innhyllingen og de som ikke er det. Så kan dette fordeles på P antall prosesser.

5. Når prosessene gjør kall på DT blir prosess-id sendt med.
6. I DT-funksjonen vil hver prosess beregne de punktene de har ansvar for. Dette er avhengig av datadistribusjonen og boksinndelingen.
7. Punktene som skal behandles vil være dem som ligger i matrisen *delaunayEdges* og antallet ligger i allNB[prosess_id].
8. Etter alle prosesser er ferdig med DT har alle sitt eget delresultat. Det finnes 3 måter å samle dette inn på:
   I.) Root-prosessen gjør *gather* for å samle *delaunayEdges* og skriver resultat til disk på en fil.
   II.) Alle prosesser skriver data til et delt filområde. Resultatet blir en fil på disk.
   III.) Alle prosesser skriver data til hver sin fil. Resultatet blir flere filer på disk.

3.5.5. Oppdeling av punkter mellom prosessene

Hvordan inndelingen blir, avgjøres til en viss grad av datasettets distribusjon. Den er også avhengig av antall prosesser som blir brukt. For eksempel vil antall punkter på og innenfor den konvekse innhyllingen avgjøre hvordan fordelingen blir mellom prosessene. I listen til den konvekse innhyllingen ligger punktene sortert på koordinater mot klokka. Under følger en bildeserie av hvordan parallellisering utføres. Dette er en forenkling der det kun brukes 2 kjerner.



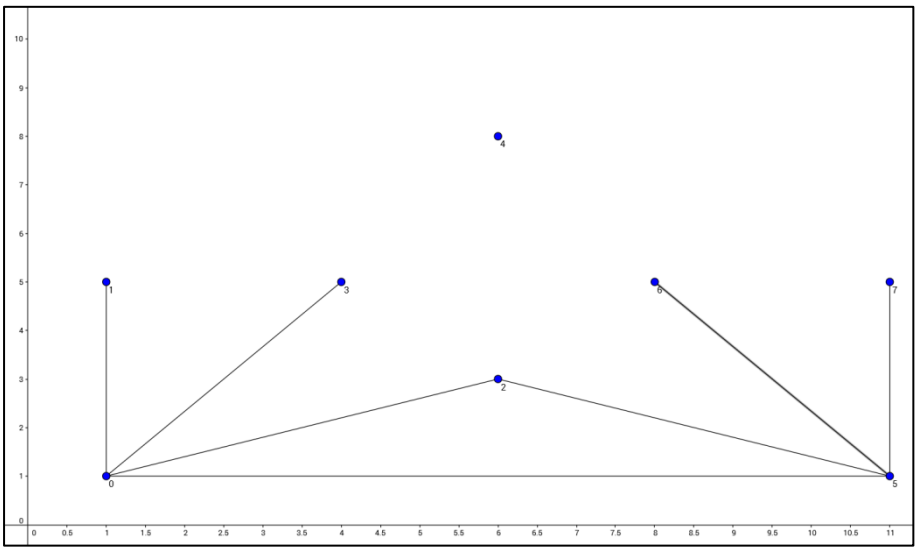
Figur 32. Punktene som skal trianguleres.

Punktene blir delt mellom prosessene. De tar i utgangspunktet $\frac{N}{P}$ hver. Men på grunn av at punktene i cohull-listen ligger sortert i en rekkefølge mot klokka blir de behandlet slik:
proc 0 has cohull-point 0
proc 0 has cohull-point 5

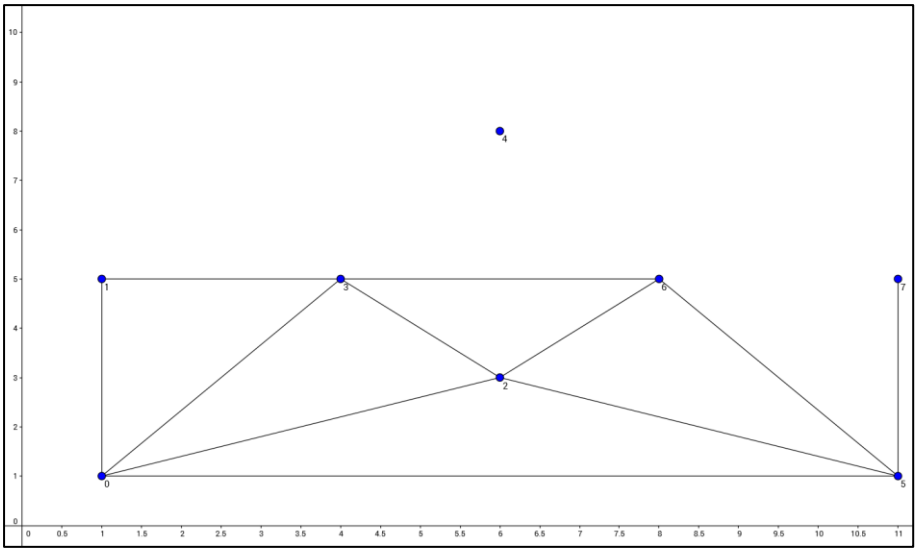
proc 1 has cohull-point 7
proc 1 has cohull-point 4
proc 1 has cohull-point 1

De indre punktene blir utdelt i stigende rekkefølge:
proc 0 has point 2
proc 0 has point 3

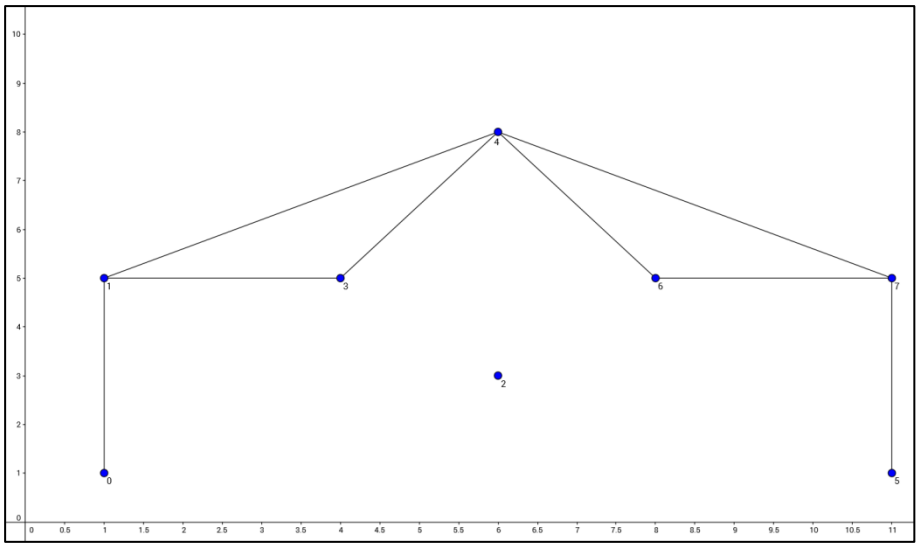
proc 1 has point 6



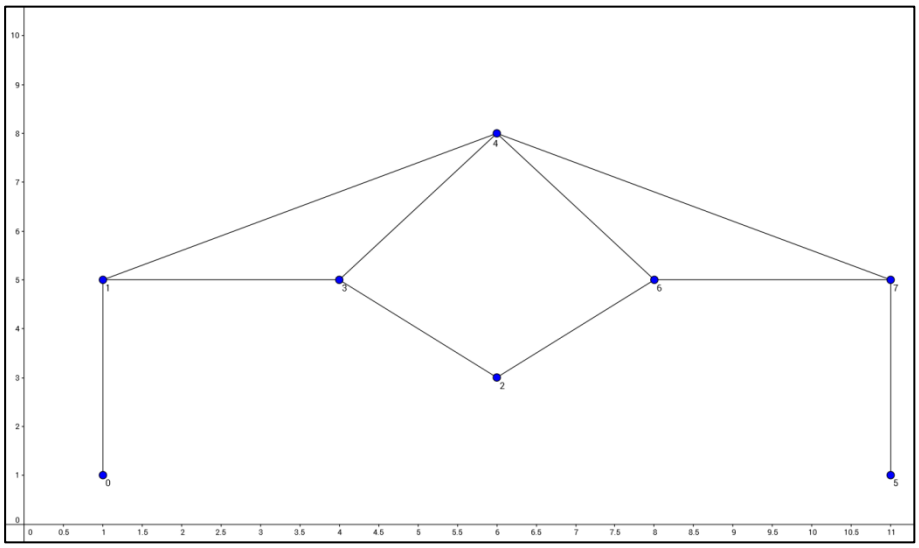
Figur 33. Etter at prosess 0 har funnet sin konvekse innhylling.



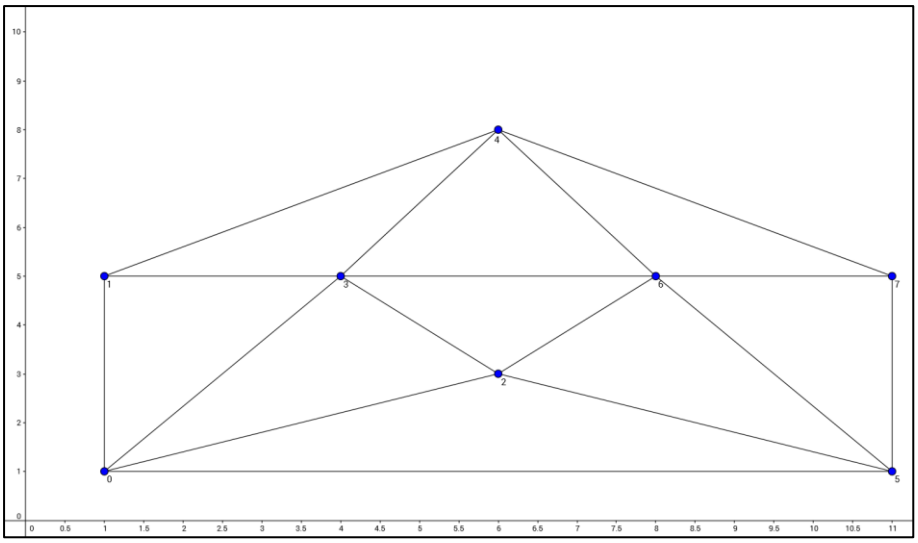
Figur 34. Etter at prosess 0 (P0) har funnet sin konvekse innhylling og behandlet sitt punkt. Det vil si at P0 har funnet sine Delaunay-kanter (DK).



Figur 35. Etter at prosess 1 har funnet sin konvekse innhylling.



Figur 36. Etter at prosess 1 har funnet sin konvekse innhylling og behandlet sitt punkt.



Figur 37. Union av grafen til P0 og P1 etter konveks innhylling og alle punkter er funnet blir en komplett DT.

Dersom vi i stedet tar snittet mellom P0 og P1, vil ikke det nødvendigvis bli *null*. Det betyr eventuelt at man har gjort for mange beregninger. Det vil si at samme beregning er utført tidligere. Her er det muligens noe man kan se nærmere på i forhold til videre optimalisering.

3.5.6. Reduksjon - samle data og skrive ut til fil

I et siste desperat forsøk på å unngå flaskehalsen med send/recv av $N \times K$ elementer, prøver vi med parallell I/O-operasjon og MPI_File_Write. I og med at alle prosesser har sitt resultat, og ikke er avhengig av andre delresultater burde dette kunne gi et forbedret resultat. Behovet for kommunikasjon blir redusert og vi får mindre overhead. Å skrive til en fil i parallell med multiple prosesser på et kontinuerlig dataområde høres mer fornuftig ut enn vanlig Unix I/O.

For å utføre denne metoden trenger vi å bruke MPI_File-typen. MPI_File_Write og MPI_File_Read er egentlig dualiteter av send/recv. Men siden vi kun gjør ett kall på MPI_File_Write per prosess, utgjør denne metoden en signifikant forskjell på resultatet. På grunn av formatet som løsningen ligger på er det vanskelig å lage en generell løsning der alle n_i ligger sortert, eller har blitt transformert til en *string* med konstant eller variabel «padding» mellom tomrom i filen.

Matrisen ser slik ut:

```
8 4 11 11 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
0 2 10 1 3 4 5 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
1 4 2 2 3 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
2 6 2 6 3 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
3 4 4 1 2 6 4 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
4 6 8 5 0 3 6 7 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
5 4 10 0 4 7 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
6 10 4 7 4 3 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
7 10 10 5 4 6 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

En tekstfil kan se slik ut:

```
8 4  11 11\n
0 2  10 1 3 4 5\n
1 4  2 2 3 0 \n
2 6  2 6 3 1 \n
3 4  4 1 2 6 4 0\n
4 6  8 5 0 3 6 7\n
5 4  10 0 4 7\n
6 10 4 7 4 3 2\n
7 10 10 5 4 6\n
```

Første linje kommer bare én gang og er nyttig dersom man skal løse inn løsning og vise den frem grafisk. Første siffer er antall punkter *n,* andre er antall bokser som må brukes. Det tredje og fjerde er maksimal lengde på x- og y-akse. Første verdi på neste linje er punktets id. De neste tallene viser punktets (x,y) koordinat. Neste tallrekke er alle naboer til det aktuelle punktet, verdien indikerer hvilken id som er nabo. De neste linjene inneholder det samme.

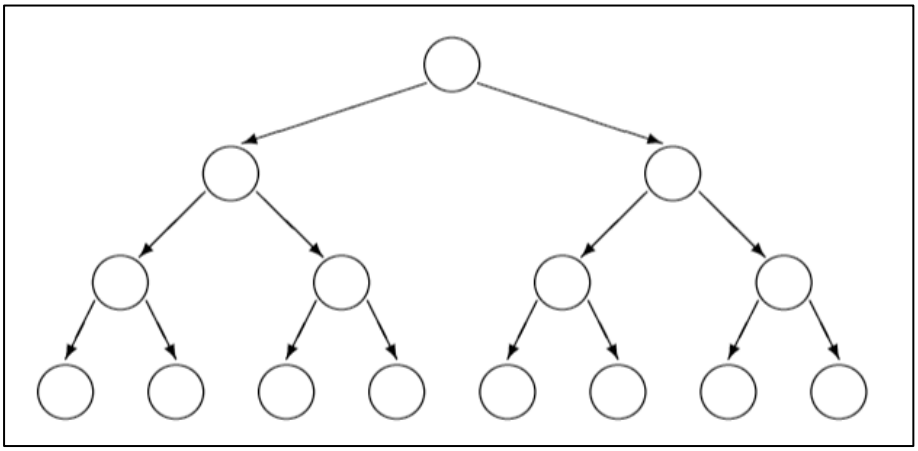
Streng eller buffer i 1 dimensjon vil se slik ut:

```
8 4  11 11\N0 2  10 1 3 4 5 X X X \N1 4  2 2 3 0 X X X X\N2 6  2 6 3 1 X
X X X\N
3 4  4 1 2 6 4 0 X X\N4 6  8 5 0 3 6 7 X X\N5 4  10 0 4 7 X X X X\N6 10 4
7 4 3 2 X X X\N7 10 10 5 4 6 X X X X\N
```

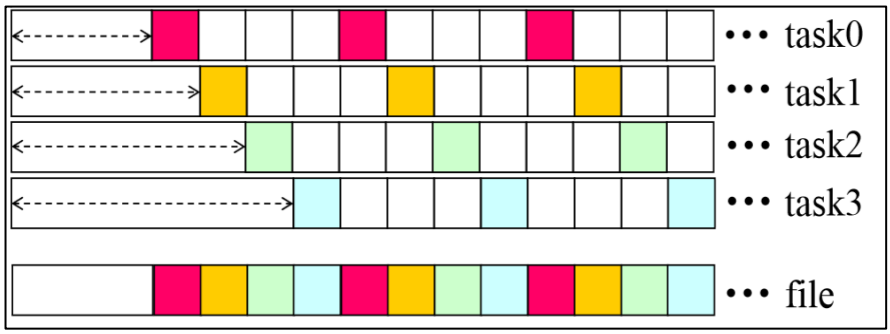
Det vil som sagt være veldig ressurskrevende å få data til å legges inn sortert. Men siden hver linje starter med id, er det ikke et problem at løsning ikke er radvis sortert. Det går fortsatt like raskt å lese inn dataene og teste løsningen fordi raden blir lagt inn på riktig sted i datastrukturer, istedenfor at de blir lagt inn med inkrementell indeks.

På grunn av at det er to eller flere prosesser som skriver til samme filområde blir det krøll med offset-skjøtingen i midten og på slutten av filen. Det løses enkelt med post-prosessering av filen ved for eksempel innlesing eller ved å lese inn i data parallelt. Løsningen på dette er å bruke streng-lengden som er skrevet til buffere og så bruke denne verdien multiplisert med prosessens id. Da brukes noe ekstra plass men det er ingen stor belastning. Det vil gi følgende intervaller for 100 punkter og 4 prosesser: P0 har punkt 0-24 +/- rest, P1 har punkt 25-49 +/- rest, P2 har punkt 50-74 +/- rest og P3 har punkt 75-99 +/- rest.

Den store fordelen med å gjøre *én* eller teoretisk sett *to* MPI_File_write-kall er at det blir eneste send/recv-kostnad i programmet. Alternativt kunne man nok pakket sammen hele resultatet til P_i, slik som vist over og sende det til root, men da måtte *root* fortsatt skrevet hele resultatet til slutt og dette hadde ført til en høy kostnad med sekvensiell diskskriving fra *root*.
For å oppsummere vil man oppnå å gå fra en ting som tilsynelatende ser ut som en sekvensiell operasjon til en ekte parallell operasjon. I sendeskjemaet til *send/recv* eller *gather* vil det bli en flaskehals for dataoverføringen, se *Figur 38*. Mens det i den andre løsningen ikke er behov for dette.



Figur 38. Vi ser 15 oppgaver totalt i sum over alle nivåer. Hver av disse tar tid t=10 for en prosess. Den maksimale arbeidsmengden som kan utføres i parallell endrer seg fra nederste til øverste nivå. Det er kun på nederste nivå alle prosessene kan jobbe i parallell.



Figur 39. Alle prosesser (med navn task i denne figuren) sitter med sin del av løsningen. De kan alle skrive sin del til en felles fil.

Ulempen er at skriving av data ganske fort blir det dominerende leddet når man øker antall arbeidere, det vil si kjerner som brukes. Den samme flaskehalsen dukker opp på nytt her[53]. Derfor trengs større data-sett (over 1 mill.) for å vise om dette er tilfellet eller ikke. For å lettere vise dette må test-data genereres i minne, fordi det blir alt for tidkrevende å lese det inn fra fil. I store datasett på over 10^6 til 10^8 punkter, er det mye lettere å avdekke hvor de regneintensive leddene ligger.

3.5.7. Innsamling av data og skriving til fil

Etter at alle prosesser har beregnet sin del av DT, må delberegningen settes sammen til en fullstendig DT. Grunnet partisjoneringen av punktene i steg 1 og hvordan den konvekse innhyllingen er utført i steg 2 vil de forskjellige intervallene av beregning kunne settes sammen uten problem.
La for eksempel A og B være vektorer med lengde N som har hver sin løsning for $\frac{N}{p}$ av DT. Da kan delløsningen settes sammen til en vektor som får den komplette løsning for DT på det gitte datasettet.

$$A = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad B = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \qquad A + B = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Etter at alle prosesser har beregnet sitt intervall av N, har som sagt alle en vektor A[i], der *i* er prosess-id. Denne vektoren kan skrives til fil. Da vil resultatet bli at vi sitter igjen med P antall filer med hvert sitt delresultat. Det er i utgangspunktet en fullverdig løsning. Når denne løsningen skal versifiseres kan filene leses inn i parallell, dette er den raskeste måten. Dersom vi ønsker å unngå dette finnes det to alternativer. Det enkleste blir å skrive til en fil i parallell. Dette tar ikke ekstra tid i forhold til måten over, men resultatet er at vi sitter igjen med én fil. Eneste bemerkning her er at vi får rader med *junk* i filen på grunn av at prosessene ikke kan vite hvor mange rader de har før programmet har kjørt. For eksempel med N = 8 og p = 2, ser vi at prosess 0 har beregnet fire rader og prosess 1 har beregnet fire rader. Det kan vi ikke vite før beregningen er utført, og derfor blir det umulig å beregne dette dynamisk uten en form for kommunikasjon mellom prosessene.

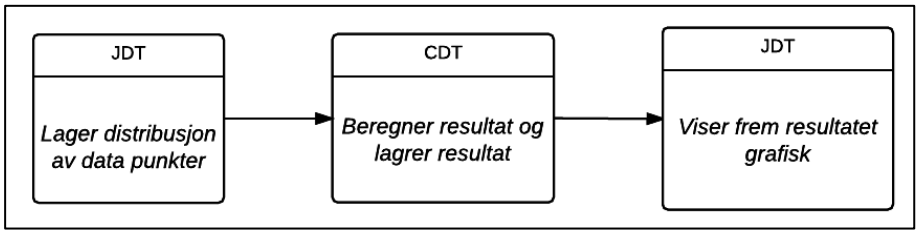
Dette kan enkelt løses med en *all_gather*. Det er en MP-funksjon som distribuerer data fra *alle-til-alle*, slik at dersom alle prosesser har en variabel for antall rader de skal skrive, vil denne distribusjonen føre til at alle prosesser har alle de andre prosessene sitt rad-antall. Write_output2 gjør dette.

3.5.8. Løsning med en serie av filer

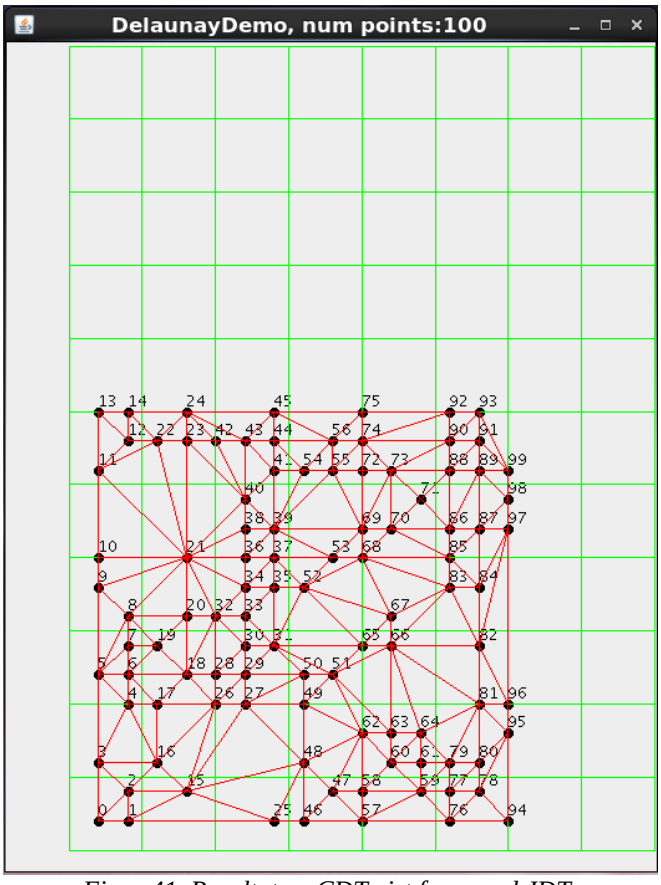
For å vise at en serie med filer som har hvert sitt delresultat kan være en god løsning, har det implantert som *write_parseq*, der vi kun dumper hver prosess sine resultater i forskjellige filer.
Det viser seg at dette er mer effektivt dersom vi kan godta post- eller pre-prosessering av resultatet før visning eller validering. Men dersom vi velger å lese data inn i parallell er ikke det nødvendig heller. Det er denne løsning de fleste resultatene i P3 kommer fra.

3.5.9. Arbeidsflyt for testing og verifisering av resultat

For å verifisere resultatet av trianguleringen finnes to metoder: Den første er en funksjon som sjekker om alle DT-kanter er lovlige som heter *sjekkDK()*. Den andre er å se på en grafisk fremstilling av trianguleringen. Sistnevnte er ofte det som ønskes, fordi den viser selve trianguleringen som eventuelt skal bruke i en annen applikasjon senere, samt at vi ganske raskt kan se om det er noe feil med mindre den ikke er for stor. Se Figur 40 for arbeidsflyt og Figur 41 for sluttresultat.



Figur 40. Arbeidsflyt for testing av portering.



Figur 41. Resultat av CDT vist frem med JDT.

3.6. P4 – Fraktaler

3.6.1. Algoritmen for Mandelbrotmengden (MS)

MS har en visuell fremstilling som er definert i det komplekse planet. Den kan defineres med følgende formel:

$$z = z^2 + c$$

Et utsnitt av MS kan produseres ved å *sample* komplekse tall og avgjøre om de går mot uendelig når de blir iterert over. Pikselen i bildet blir farget etter antall iterasjoner som blir utført dersom tallet divergerer. Det vil si at det ikke får en stabil verdi og dermed går mot uendelig, i så tilfelle blir pikselen svart. Dersom man bruker den reelle og imaginære delen av hvert tall som pikselkoordinat i bildet blir resultatet et bilde som inneholder MS. Den iterative funksjonen går gjennom alle punkter i planet, og for hver piksel-posisjon blir en farge valgt på grunnlag av antall iterasjoner som blir brukt for å finne en stabil verdi. Dersom man prøver å beregne verdien av et punkt som divergerer, og går mot uendelig, får man for eksempel en svart piksel. Verdier av *z* som konvergerer får en annen farge i 8-bit RGB spekter. Under følger en beskrivelse av algoritmen:

1. La *c* være en kompleks konstant som definerer posisjon i bilderammen.
2. La z_0 være den første verdien for iterasjonen av det komplekse tallet.
3. Først iterer over uttrykket $z_n = z_{n-1}^2 + c =$ én gang.
4. Sjekk om (I) størrelsen til *z* er over en satt verdi. Det er slik vi kan finne ut om tallet konvergerer.
Eller sjekk om (II) tall går over antall iterasjoner som er forhåndsbestemt. Gjenta (3) til (I) eller (II) forekommer.
5. Tegn opp punktet med en farge som blir bestemt ut fra en fargepalett på 8-bit (RGB) og en histogramfunksjon som avgjør hvilket iterasjonsnummer som tilhører hvilken farge i RGB-spekteret.

3.6.2. Parallellisering av MS

Dette problemet går under kategorien *pinlig parallelliserbart.* Det er fordi det ikke er behov for noe kommunikasjon mellom prosessorene under beregning. Det betyr at det er *pinlig enkelt* å parallellisere algoritmen slik at den kjører effektivt på maskinklynge. Selve beregningen gir en lineær speedup, men vi burde også legge til tiden det tar å redusere data til slutt. Derfor blir speedup begrenset av kostnad for innsamling av data. Selv om dette kan være tilfellet, er det fortsatt noen ting man må tenke gjennom for å få det implementert på en fornuftig måte. Siden alle beregninger kan forekomme individuelt, er det ikke behov for kommunikasjon under beregningen. Det er hovedsakelig dette faktumet som gjør den *pinlig parallelliserbar*. Men det betyr ikke at alle prosesser nødvendigvis får samme arbeidsmengde. I MS er det de svarte punktene som tar mest tid å beregne, fordi der brukes det flest iterasjoner. Jeg skal produsere et rasterisert bilde (digitalt bilde). Det vil si et bilde som er lagret i en matrise som indekseres med *j* kolonner og *i* rader. Når et rasterisert bilde ligger i minnet, ligger hver piksel lagret kontinuerlig etter hverandre i en vektor. Eksempelet under viser hvordan et bilde F[i][j] med svart ramme (0'ere) og en hvit prikk (255) kan representeres i én dimensjon:

$$F = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 255 & 0 \\ 0 & 0 & 0 \end{bmatrix} \rightarrow [0\ 0\ 0\ 0\ 255\ 0\ 0\ 0\ 0]$$

Dette gjør i utgangspunktet at inndeling av rader blir lettere å velge som metode for dekomposisjon av bildet. Men når man skal utføre en parallellisering av MS-algoritmen er det ønskelig å gi en rettferdig lastbalanse for prosessene. Derfor hjelper ikke dette når man velger at annenhver prosess skal ta annenhver rad, fordi det må gjøres en omstrukturering av radene senere. Dersom det gjøres en triviell inndeling i radvise blokker fra toppen av bildet til bunnen, kan noen prosesser få områder der det er mye svart eller bare svart. Derfor lar jeg hver prosess beregne en linje av bildet, det vil si en lengde av hele bildet og høyde på 1. Hver prosess gjør dette så mange ganger den trenger avhengig av bildets høyde.

Etter at jeg har utført en *gather* fra alle prosesser til root må root-prosessen gjøre om indekseringen i mottakerbuffere, slik at alle rader ligger i riktig rekkefølge. Da kan bildet vises i sin helhet med riktig rekkefølge på radene i bildet, eller som flere kopier av samme bilde nedover. Denne operasjonen tar konstant tid. Derfor er det ikke noe problem at verdiene i vektoren ligger i feil rekkefølge etter reduksjonen. Jeg oppnår også ganske jevn fordeling av arbeidsmengden fordi det er stor sannsynlighet for at neste rad ligner på den forrige. Det er rett og slett fordi det er et digitalt bilde.
Et eksempel på oppdeling: Dersom man har to prosesser og et bilde som kan deles inn i åtte rader, så gi to rader til hver prosess slik:

For prosess 0 med radvis distribusjon:

P0
(P0 skal ikke behandle denne raden det er P1 sin)
P0
(P0 skal ikke behandle denne raden det er P1 sin)

For prosess 1 med radvis distribusjon:

(P1 skal ikke behandle denne raden det er P0 sin)
P1
(P1 skal ikke behandle denne raden det er P0 sin)
P1

Etter at root-prosessen har gjort *gather* ser det slik ut:

P0
P0
P0
P0
P1
P1
P1
P1

Etter omplassering av raden blir resultatet slik:

P0
P1
P0
P1
P0
P1
P0
P1

Da ser vi at de to prosessene først beregnet annenhver rad. Deretter har root-prosessen satt vektoren sammen til et bilde der radene ligger i riktig rekkefølge.

3.6.3. Kjøring av algoritmen (MS)

I denne øvelsen skal jeg prøve å vise funksjonens selvrepeterende egenskap, slik bildeserien etter resultatene illustrerer. Dette kan naturligvis gjøres raskere og mer effektivt på en GPU. Men jeg ønsker å vise hvordan et problem som krever veldig mange iterasjoner for å beregne et punkt, det vil si en piksel i et bilde, kan parallelliseres for å oppnå ønsket resultat raskere enn i en sekvensiell versjon av algoritmen.

Vi skal se hvordan dette kan gjøres ved å starte med en sekvensiell algoritme som skal parallelliseres og kjøres på Abelklyngen. Denne

algoritmen er valgt fordi det er enkelt å vise hvordan man får en ytelsesforbedring. Problemet er her formulert slik at det faktisk ikke er mulig å gjøre den ønskede beregning på én datamaskin. Vi må ha flere, det vil si en maskinklynge, for å tilgang på nok FLOPS for å løse problemet innen rimelig tid. Som nevnt tidligere har en vanlig maskin omtrent 10 gigaFLOPS. Dersom vi skal produsere et MS-bilde som er 32000×24000 på ett minutt med 10000 iterasjoner og 10 flyttallsoperasjoner pr. gjennomkjøring av hoved-løkka, vil det kreve omtrent

$$\frac{32000 \times 24000 \times 1000 \times 10}{60} = 12800000000 = 1.28 \times 10^{12} =$$

1.28 teraFLOPS. Derfor er dette et fint eksempel på når man bør eller må bruke en maskinklynge. En fraktal som et Mandelbrot-sett er et godt eksempel på en beregning som blir veldig tung når man skal øke oppløsningen i løsningsdomenet for resultatet.

For å se om den parallelle algoritmen skalerer til høyere antall kjerner enn vi har på en vanlig maskin, kan vi utføre disse testene lokalt. Selv om vi ikke har det antall kjerner Abelklyngen har, kan kompilatoren simulere dette slik at vi slipper å vente på resultat fra køsystemet i Abelklyngen under utvikling og feilsøking. På en vanlig maskin kan vi simulere opp til 128 prosesser ved å overbooke antall logiske beregningsenheter. Det er et godt utgangspunkt for å øke antall kjerner opp mot 500 for å se at algoritmen skalerer riktig. Det er også lurt å gjøre utvikling og feilsøking med små bilder slik at det ikke tar lang tid pr. kjøring.

Jeg prøver å produsere et bilde på 64000×48000.
Her kommer jeg ikke forbi løkka i hovedberegningen. Den bare står og går fordi det tar så lang tid at minnet blir brukt opp, og det er vanskelig å få til med denne reduksjonsalgoritmen.

Når man øker bildestørrelsen må man også huske på å øke hvor mye minne som allokeres på en *node*.Jeg prøver å legge til mer minne: Tidligere ble det brukt 1GB. Jeg går nå opp til 2GB. Hver node har 64GB fordelt på 16 fysiske kjerner. Det betyr at man kan allokere opp til 4GB pr. kjerne. Men det er også mulig å «låne» minne fra de andre kjernene på noden. Men dette tar mye mer tid å få allokert i køsystemet.

Jeg gjør et regnestykke for å undersøke dette nærmere:
$64000 \times 48000 = 3.072 \times 10^9$ antall piksler, dersom jeg tar (antall piksler × RGB = $3.072 \times 10^9 \times 3 = 9.216 \times 10^9$ bytes = 9.2160) = 9.21 gigabytes (GB) før komprimering, og bruker en *z_lib* komprimering med png-formatet for å skrive bildet til disk, må jeg fortsatt ha tilgang på ca. 9 GB i minnet for å produsere bildet. Kravet til allokering av minne for MS varier også med antall prosesser som kjører. Når MS kjører med over 16 prosesser kreves det ca. 1 GB pr. kjerne. Dersom det er under 16 trenger de 2GB hver og hvis det er under 8 trengs 3GB pr. prosess.

Poenget med å lage et så stort bilde er å zoome inn på det for vise den selvrepeterende egenskapen til MS. Dette går raskere å utføre med et eget program enn å implementere funksjonalitet for det i den parallelle MS-algoritmen. Det finnes mange gode verktøy som gjør denne jobben uten at man trenger å programmere noe mer.

Testparameterne som velges er veldig viktig her. Ved feilsøking ble flere variabler endret og brukt på en gang. Dette skaper litt krøll i utviklingen av programmet. Derfor har jeg lagt opp til følgende regler for bildets størrelse, forhold og antall prosesser som skal brukes:

(I).
Vi har lagt opp til at alle bilder skal ha et forhold mellom høyde og bredde på $\frac{4}{3} = 1.33$.
Noen ganger ble det brukt et bildeforhold som er 1.5. For eksempel 1200 × 800 og 2400×1600.
Det er en feilkilde dersom man varierer bruken av disse og ikke er konsekvent fordi det blir flere problemer å løse på en gang. En løsning er at bruker skriver inn ønsket høyde på bildet. Dersom det ikke går opp i antall prosesser gjør jeg en skalering for å kompensere slik at forholdet mellom h × b er 1.33.
(II).
Den andre begrensingen er at antall prosesser som velges er delelig med høyden av bildet.
Det er fordi jeg har valgt en rad-inndeling av bildet. Det betyr at forholdet i bildet må være 1.33, eller så må dette endres.

Når jeg bruker (I) og (II) er det garantert at antall prosesser jeg vil bruke blir brukt. Men siden høyde og bredde må være 1.33 og *h* må være delelig med antall prosesser *p* slik at de ikke får noe rest må dette kompenseres for i utregning av den totale bildestørrelsen. Et eksempel på dette er at når vi vil ha et bilde med h=24000, så blir bildet 31914×23936 dersom vi skal ha p=128 antall prosesser.

Dersom jeg bruker høyden = 36000 går det også greit å kjøre. Da får jeg et bilde som er 48000×36000 og med 64 i zoom-faktor. For at root-prosessen skal slippe å sitte igjen med all I/O til slutt lar jeg hver prosess skrive ut sin del av bildet. Dette gjøres etter at root gjør *scatter*. Dette skaper en ekstra kommunikasjonskostnad, men det er ikke nevneverdig i den totale mengden tid det tar å produsere bildet.

For å sette sammen dette bildet kreves derimot ganske mye tidsbruk, opp til flere minutter. Så dette er noe som kan forbedres. Men denne operasjonen kan uansett startes mens neste beregning av nytt utsnitt for Mandelbrotmengden kan produseres. Et annet poeng jeg ønsker å vise, er hvordan ytelsen til den parallelle algoritmen skalerer med økning av antall kjerner *p*.

4. Resultater

4.1. P0 – Kommunikasjonstest

Det første som ble undersøkt var tiden det tar å sende data.

4.1.1. C-MPI med ping-test

Antall noder	**Antall kjerner pr. node**	**Totalt**	**Tid for N=10^6**
1	2	2	*20.3 ms*
2	1	2	*243.1 ms*

Tabell 8. C-MPI med ping-test.

4.1.2. Java-MPI med ping og ping-pong

Jeg satte opp flere tester for send/recv med enveiskommunikasjon (ping) og toveiskommunikasjon (ping-pong). Det var ikke mulig å sende et *int*, derfor sendte prosess 0 et heltall fra en vektor med lengde 1, til prosess 1. Jeg prøvde først å sende 10^6 (1 mill.) heltall enveis og så toveis. Alle kjøringer er utført på Abelklyngen.

4.1.3. Ping-test med J-MPI

Antall noder	**Antall kjerner pr. node**	**Totalt**	**Tid N=10^6**
1	2	2	10094.2 ms
2	1	2	1192.7 ms

Tabell 9. J-MPI med ping-test.

Allerede etter første test så jeg at J-MPI var ca. *fire* ganger tregere enn C-MPI. Andre resultater som må nevnes er at det gikk ti ganger tregere å bruke to prosesser på en node. Jeg er usikker på hvorfor det er slik, men jeg tror det kan ha noe å gjøre med at det kun er én *main-tråd* som kan kjøre i en Java virtuell machine (JVM). Når to JVM kjøres på samme node, blir det vanskelig for dem å finne hverandre, og systemet ender opp med å simulere to prosesser i én tråd. Mens når det i programmet ligger én JVM på hver node vil den underliggende MPI-implantasjonen som brukes av *Java-bindingen* fungere som normalt med unntak av litt lavere ytelse.

4.1.4. Ping-pong-test med J-MPI

Antall noder	**Antall kjerner pr. node**	**Totalt**	**Tid N=10^6**
1	**2**	**2**	**1282.9 ms**
2	**1**	**2**	**8350.1 ms**

Tabell 10. J-MPI med Ping-pong-test.

Det er litt rart at det tok mindre tid med ping-pong enn ping på J-MPI. Men det har mest sannsynlig noe å gjøre med at programmet ikke blir distribuert på riktig måte i et multikjerne-miljø.

4.1.5. Ping-pong-test med C-MPI på Abelklyngen

Antall noder	**Antall kjerner pr. node**	**Totalt**	**Tid N=10^6**
1	**2**	**2**	**333.6 ms**
2	**1**	**2**	**5774.98 ms**

Tabell 11. Ping-pong-test med C-MPI.

Hovedgrunnen til at ping-pong-testen tar mye lenger tid enn en ping-test, er at enveiskommunikasjon er veldig rimelig fordi det ikke krever noen form for synkronisering. Mens toveiskommunikasjon må synkroniseres og derfor blir det *idling* på beregningsenhetene (de står ubrukt) og det gir opphav til kostbare funksjonskall.

4.1.6. Kostnadsanalyse for send/recv og ping-pong

Tidsbruk for å kommunisere en melding med størrelse (m) i et ikke-blokkert nettverk er tiden T:
$T = (ts + tw \times m)$, der *ts* er *startuptime*, og *tw* er kostnad for hvert element som sendes. I et binær-tre som brukes av *brodcast* og *gather* blir det $T = (ts + tw \times m)\ log\ p$. Meldingstørrelsen er størrelsen på bufferen som skal sendes. Kostnaden for en melding er stor sammenlignet med tiden en beregning tar. Dette krever et gjennomtenkt design av algoritmen for at den skal kunne gi god ytelse og effektivitet.

4.2. P1 – Sortering

4.2.1. Java-MPI med mergesort på multikjerne

Noder	Kjerner	Totalt	Tid i sekvens	Tid i parallell	*Speedup*
1	2	2	0.2213 s	0.2285 s	0.9
1	4	4	0.2154 s	0.1374s	1.5
1	8	8	0.2498 s	0.1741 s	1.4
1	16	16	0.2313 s	0.1066 s	2.1

Tabell 12. J-MPI med mergesort på multikjerne.

Resultatene over viser kjøringene fra beregningsnode i køsystem. Det blir omtrent som å bruke en multikjernemaskin fordi det bare brukes én node.

4.2.2. Java-MPI med mergesort på Abelklyngen

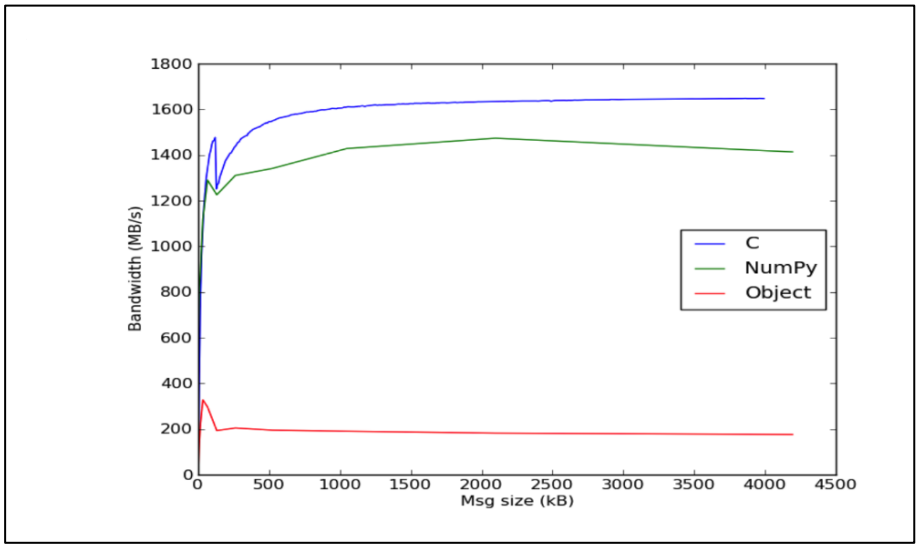
Noder	Kjerner	Totalt	Tid i sekvens	Tid i parallell	*Speedup*
2	4	8	0.2016 s	0.1833 s	1.1
2	8	16	0.1546 s	0.1261 s	1.2
4	4	16	0.1939 s	0.1787 s	1.1

Tabell 13. Java-MPI med mergesort på Abelklyngen.

Disse resultatene bør være nok til å forkaste videre utforskning av J-MPI. Men de bør kanskje gjøres på nytt allikevel for å verifisere resultatene.

4.2.3. Dårlig resultat – stopper her

På grunn av at jeg fant ut at Java er mindre effektiv med MPI enn C, og noen litt usikre resultater, valgte jeg å fortsette arbeidet på HPC med språket C. En annen grunn til at jeg ikke så mer på dette temaet, er at det ikke finnes noe oppdatert forskning på området, og det ser ut som dette har stått stille i snart ti år[54]. Jeg er usikker på den faktiske grunnen til dette, annet enn at Java på flere måter er uegnet til HPC i forhold til at programmet kjører i JMV og er minnekrevende. Derimot har Python blitt godt integrert med MPI. For videre arbeid på dette temaet bør Python brukes som høynivå språk, fordi det fungerer meget stabilt med MPI og er godt dokumentert. Det finnes også en aktiv forskningsgruppe på feltet[55]. Når det gjelder ytelsen til Python sammenlignet med C, er det slik at Python er implementert med C, og det er enkelt å kjøre kall på C og Assembler-funksjoner i Python. Andre alternativer for å bedre Python sin ytelse er å bruke *numpy* som er et vektorisert matebibliotek[56], med ytelse ekvivalent med det C leverer.



Figur 42. Grafen ytelse til MPI4Python og behov for båndbredde ved sending av objekter (Python datatype, numPy og primitive C datatyper[57]

4.2.4. Kjøring av parallell mergesort med C-MPI

Jeg prøvde med ca. samme node-konfigurasjoner som i artikkelen til ROLFE[48]. Jeg undersøkte hvor lang tid 10^6 heltall tok å sortere i sekvensiell algoritme og parallell algoritme. Resultatene under er basert på medianen av tre kjøringer.

Delt minne

Noder	Kjerner	Totalt	Tid i	Tid i	*Speedup*

			sekvens	parallell	
1	1	1	0.185		
1	2	2	---	0.101	1,831
1	4	4	---	0.069	2.655
1	8	8	---	0.048	3.784
1	16	16	---	0.039	4.706

Tabell 14. Parallell mergesort med C-MPI i delt minne.

Distribuert minne

Noder	Kjerner	Totalt	Tid i sekvens	Tid i parallell	*Speedup*
2	1	2	0.153	0.112	1.38
2	2	4		0.085	1.69
2	4	8		0.078	2.12
2	8	16		0.055	2.91
4	8	32		0.081	1.96
8	8	64		0.130	1.22
16	8	128		0.170	0.93
32	8	258		0.480	0.33
64	8	512			0.1

Tabell 15. Parallell mergesort med C-MPI i distribuert minne.

4.3. P2 – Matrisemultiplikasjon

Følgende matriser ble brukt:

$$A = m \times l = 1000 \times 500, \qquad B = l \times n = 500 \times 1000,$$
$$C = m \times n = 1000 \times 1000$$

4.3.1. Matrisemultiplikasjon på multikjerne

Kjørt på Lenovo laptop.

Noder	Kjerner	Totalt	Tid i sekvens	Tid i parallell	*Speedup*
1	2	2	5992.8 ms	2902.6 ms	2.06
1	4	4	5924.8 ms	1394.4 ms	4.25
1	8	8	5896.99 ms	712.94 ms	8.27
1	16	16	6110.63 ms	403.43 ms	15.15
1	32	32	7027.99 ms	272.05 ms	25.83

Tabell 16. Matrisemultiplikasjon med delt minne..

4.3.2. Matrisemultiplikasjon på Abelklyngen

Noder	Kjerner	Totalt	Tid i sekvens	Tid i parallell	*Speedup*
1	1	1	3749.71 ms	---	
1	2	2	---	858.76 ms	4,36
2	4	4	---	626.79 ms	5,98
2	4	8	---	460.84 ms	8,07

2	8	16	---	336.56 ms	11,14
4	8	32	---	811.86 ms	4,61
8	8	64	---	1427.46 ms	2,62
16	8	128	---	1121.40 ms	3,34
32	8	256	---	872.13 ms	4,29
64	8	512	---	1627.40 ms	2,30

Tabell 17. Matrisemultiplikasjon med distribuert minne.

4.4. P3 – Delaunay triangulering

4.4.1. Delaunay triangulering i delt minne

Alle kjøringer i dette avsnittet er kjøring av MPI-løsninger både på *laptop* og Abelklyngen. Det vil si simulering av program som kan gjøres i distribuert minne. En node hos Abelklyngen har 2 stk. Intel Xeon CPU E5-2670 0 @ 2.60GHz 16 fysiske kjerner, (32 med *hyperthreading*) og 64 GB ram. En MPI-prosess tilsvarer å bruke *én* fysisk kjerne. Når vi velger antall MPI-prosesser til å være samme antall som fysiske kjerner får hver MPI-prosess en ID tilknyttet en spesifikk kjerne. Men det er mulig å lage flere MPI-prosesser enn det er kjerner, slik som i Java der vi kan lage opptil mange tusen tråder.

Delt minne med N=10^6 og write_output_seq på Lenovo laptop

MPI-prosesser	Tid i sekvens	Tid i parallell	*Speedup*
1	16453.88 ms	---	1.00
2	---	8447.77 ms	1.94
4	---	7655.55 ms	2.14
8	---	8101.80 ms	2.03
16	---	9035.51 ms	1.82
32	---	11935.71 ms	1.37

Tabell 18. Delt minne med $N=10^6$ og write_output_seq på Lenovo laptop.

Delt minne med N=10^7 og write_output_seq på Lenovo laptop

MPI-prosesser	Tid i sekvens	Tid i parallell	*Speedup*
1	202971.32 ms	---	1.0
2	---	112345.88 ms	1.80
4	---	91158.57 ms	2.22
8	---	103038.62 ms	1.97
16	---	117475.86 ms	1.73
32	---	509265.02 ms	0.39

Tabell 19. Delt minne med $N=10^7$ og write_output_seq på Lenovo laptop.

Delt minne med N=10^8 uten å skrive til disk, med Lenovo laptop

MPI-prosesser	Tid i sekvens	Tid i parallell	*Speedup*
1	989579.82 ms (ca. 16 min)	---	1.0

Tabell 20. Delt minne med N=10^8 uten å skrive til disk, med Lenovo laptop.

Denne testen ble ikke fullført da det er tidkrevende og ikke ville gitt godt resultat.

Delt minne med N=10^6 og write_output_seq på Abelklyngen med én node

MPI-prosesser	Tid i sekvens	Tid i parallell	*Speedup*
1	5507.35 ms	---	
2	---	2701.03 ms	2,03
4	---	1353.64 ms	4,06
8	---	803.83 ms	6,85
16	---	437.48 ms	12,58

Tabell 21. Delt minne med N=10^6 og write_output_seq på Abelklyngen med én node.

Delt minne med N=10^7 og write_output_seq på Abelklyngen med én node

MPI-prosesser	Tid i sekvens	Tid i parallell	*Speedup*
1	64532.46 ms	---	
2	---	31651.22 ms	2,03
4	---	17683.66 ms	3,64
8	---	10420.64 ms	6,19
16	---	5832.0451 ms	11,06

Tabell 22. Delt minne med N=10^7 og write_output_seq på Abelklyngen med én node.

Delt minne med N=10^8 Abelklyngen med én node.

MPI-prosesser	Tid i sekvens	Tid i parallell	*Speedup*
1	921579.82 ms	---	
2	---	444983.03 ms	2.07
4	---	229434.54 ms	4.01
8	---	183185.50 ms	5.03
16	---	102731.32 ms	8.97

Tabell 23. Delt minne med N=10^8 Abelklyngen med én node. Her ses kun på distribusjon og beregningstid.

4.4.2. Delaunay Triangulering på Abelklyngen

Alle kjøringer er utført med tanke på rettferdighet mot køen og for å komme raskest mulig gjennom køen. Derfor har jeg foretatt en node-konfigurasjon

som bruker maksimalt 8 kjerner på en node, og økt antall noder for å komme til ønsket antall kjerner. Et annet alternativ hadde vært å kjøre med fulle noder (16 kjerner) og i tillegg bruke en hylleseksjon og kjøre alene (*exclusive*). Måten jeg har valgt å kjøre på her kan lettere anvendes i en kontekst fra virkeligheten. Med andre ord, det er ofte slik hverdagen er; ikke alltid ideell.

($N=10^6$) write_output_seq på Abelklyngen:

Noder	Kjerner pr. node	Totalt ant. kjerner	Tid total	Tid i distribusjon	Tid i beregning	Tid i reduksjon
1	1	1	5569.91 ms			
1	2	2	2819.09 ms	108.54 ms	2193.10 ms	499.67 ms
2	2	4	1416.14 ms	---	---	---
2	4	8	795.42 ms	---	---	---
2	8	16	545.16 ms	---	---	---
4	8	32	352.87 ms	---	---	---
8	8	64	309.38 ms	---	---	---
16	8	128	308.22 ms	---	---	---
32	8	256	176.85 ms	---	---	---
64	8	512	770.77 ms	710.63ms	4.58 ms	35.74 ms

Tabell 24. ($N=10^6$) write_output_seq på Abelklyngen. Alle mellom tider ble ikke registret.

Valget av node-konfigurasjon var basert på å få kort tid i køen. Derfor brukte jeg aldri mer enn 8 kjerner pr. node.

($N=10^7$) write_output_seq på Abelklyngen.

Noder	Kjerner pr. node	Totalt ant. kjerner	Tid total	Tid i distribusjon	Tid i beregning	Tid i reduksjon
1	1	1	64028.40 ms	1305.64 ms	52345.85 ms	10106.05 ms
1	2	2	33311.82 ms	1655.29 ms	26348.35 ms	5077.42 ms
2	2	4	17100.80 ms	1477.91 ms	12881.68 ms	2517.11 ms
2	4	8	11206.68 ms	1712.02 ms	7348.71 ms	1908.77 ms
2	8	16	6577.61 ms	1890.10 ms	3803.13 ms	6577.61 ms
4	8	32	4279.60 ms	1522.91 ms	1744.91 ms	785.06 ms
8	8	64	3149.79 ms	1869.15 ms	886.35 ms	157.28 ms
16	8	128	3268.02 ms	1989.39 ms	478.94 ms	561.70 ms
32	8	256	2672.36 ms	1998.97 ms	219.41 ms	219.52 ms
64	8	512	3306.23 ms	2542.45 ms	109.47 ms	411.44 ms

Tabell 25. ($N=10^7$) write_output_seq på Abelklyngen.

Speedup $N=10^6$ distribuert minne

MPI-prosesser	Tid i sekvens	Tid i parallell	*Speedup*
1	5569.91 ms		
2	---	2691.90 ms	2,06
4	---	1416.14 ms	3,93
8	---	795.42 ms	7,00
16	---	545.16 ms	10,21
32	---	352.87 ms	15,78
64	---	309.38 ms	18,00
128	---	308.22 ms	18,07
256	---	176.85 ms	38.92
512	---	770.77 ms	7,22

Tabell 26. Speedup $N=10^6$ distribuert minne.

Speedup $N=10^7$ distribuert minne

MPI-prosesser	Tid i sekvens	Tid i parallell	*Speedup*
1	64028.40 ms		
2	---	33311.82 ms	1,92

4	---	17100.80 ms	3,74
8	---	11206.68 ms	5,71
16	---	6577.61 ms	9,73
32	---	4279.60 ms	14,96
64	---	3149.79 ms	20,32
128	---	3268.02 ms	19,59
256	---	2672.36 ms	23.95
512	---	3306.23 ms	19,36

Tabell 27. Speedup $N=10^7$ distribuert minne

Speedup $N=10^8$ distribuert minne

MPI-prosesser	Tid i sekvens	Tid i parallell	*Speedup*
1	921579.82 ms	---	
2	---	444983.03 ms	2.07
4	---	431045.48 ms	2.13
8	---	262825.66 ms	3.50
16	---	98709.97 ms	9.33
32	---	169847.36 ms	5.42
64	---	117548.36 ms	7.84
128	---	---	---
256	---	35466.15 ms	25.98
512	---	---	---

Tabell 28. Speedup $N=10^8$ distribuert minne

Kostnadsanalyse for DT
Algoritmen for å sette opp datastrukturen tar O(N) tid, der N er antall elementer. Alle prosesser gjør dette. Hver prosess sin arbeidsmengde er N/P. Den totale tiden i parallell blir da:

$$Tpar = O\left(N + NK + \frac{N}{P} \times \log\frac{N}{p} + \log N\right) = O(N)$$

Mens den sekvensielle kjøretiden for DT er $O(N \times log(N))$.
Speedup blir da:

$$S = \frac{N \times \log N}{N} = \log N$$

Det kan da vises at effektiviteten *E* går mot null. Effektivitet kan defineres som:

$$\mathrm{E} = \frac{\text{serial runtime}}{\mathrm{P} \times \text{parallel runtime}}$$

Effektiviteten for denne implementasjonen blir følgende:

$$\lim_{p\to\infty}\left(\frac{O(N\log(N))}{p \times O(N)}\right) = \left(\frac{O(N\log(N))}{\infty}\right) = 0$$

Vi ser effektiveten gå mot null dersom vi lar antall kjerner gå mot uendelig. På den andre siden er det en logaritmisk økning i beregningshastigheten.

Kostnad ved innsamling av data - Reduksjon
Det viser seg at kostnad for å samle inn resultat var veldig dyrt, siden sending av en melding har følgende kostnad: $(ts + tw \times m)$. Alle prosesser må sende to meldinger til root, en med lengden på bufferet og en med bufferet. Dersom mange prosesser skal sende til root-prosessen, finnes det en funksjon som gjør dette på en effektiv måte. *Gather*-funksjonen til MPI bruker et sendeskjema med rekursiv dobling, det betyr at meldingen blir sendt oppover i flere nivåer, i en tre-struktur, der to barne-noder sender til forelder-noder. Gather har følgende kostnad: $T = ts \times log(p) + tw \times m \times (p-1)$. Så dersom vi øker antall prosesser som skal sende sin DT til root, blir leddet fort mer kostbart enn antatt. Det hjelper uansett ikke når andre ledd er $tw \times m \times (p-1)$, siden det i utgangspunktet var for dyrt å sende bare fra én prosess til root. Dersom vi tar tiden på å sende kun meldingen med et heltall, ser vi at det også er en stor del av kjøretiden. Under følger resultat av kjøring med *gather_all*, det vil si *alle-til-en-metoden.* Kjøring med antall punkter = 10^6 og antall prosesser = 2:

MPI-prosesser	Tid total	Tid i distribusjon	Tid i beregning	Tid i reduksjon
2	*78884.08 ms*	*1549.25 ms*	*12750.93 ms*	*64419.02 ms*

Tabell 29. DT med 1 mill. punkter og to prosesser.

Her ser vi at antall elementer som sendes er det dominerende leddet fordi pakkestørrelsen ikke ser ut til å endre tidsbruken. Under følger resultat av kjøring med parallell MPI I/O og skriving til delt fil. Antall punkter = 10^6 og antall prosesser = 2:

MPI-prosesser	Tid total	Tid i distribusjon	Tid i beregning	Tid i reduksjon
2	*16103.95ms*	*1549.25 ms*	*12750.93 ms*	*1803.77 ms*

Tabell 30. DT med 1 mill. punkter og to prosesser med MPI IO.

Dette fører til at beregning (steg 3) blir det dominerende leddet. Det betyr at jeg har oppnådd en brukbar parallell implementasjon av algoritmen. På kjøringen under er antall punkter n=10 mill. (10^7). Vi undersøker hvordan kommunikasjonskostnaden i steg 4 utvikler seg med økning av kjerner.

Noder	Kjerner	Totalt	Total tid i parallell
1	2	2	63802.3729 ms
1	4	4	47416.5881 ms
1	8	8	57716.2831 ms

Tabell 31. Økning i antall prosesser som bruke på DT.

Resultatene over viser at beregningstiden går ned, mens innsamling og skriving blir det dominerende ledd når vi øker antall prosesser. Det oppnås forøvrig en tilnærmet lineær *speedup* på beregning av triangulering, det vil si steg 3.

4.5. P4 – Mandelbrotmengden (MS)

Jeg har fokusert arbeidet på å avbilde en forstørrelse på opp mot 100x av originalbildet for å vise den repeterende egenskapen til en fraktal. Det jeg også ønsker å vise er tidsbruken dette tar på en vanlig datamaskin sammenlignet med en klyngemaskin. Hensikten er å vise hvordan vi oppnår bedre ytelse, det vil si *speedup*, og hvordan vi kan designe en parallellisering for at den skal bli mest mulig effektiv og skalerbar.

Jeg prøvde å lage et 100x forstørret bilde av originalutsnittet for å vise det repeterende elementet til fraktalen. Under følger resultater for kjøring av MS på forskjellige maskintyper:

Oppløsning	Kjerner	Noder	Type maskin	Tid
32000x24000	4	1	Lenovo bærbar – Intel i7-2620M 2.7 GHz, 8GB	653366.70 ms (ca. 11 min.)
32000x24000	1	1	Lenovo bærbar – Intel i7-2620M 2.7	15680358. 81 ms (26 min. 8

			GHz, 8GB	sek)
32000x24000	1	1	Abelklyngen	1296944.34 ms (21 min. 36 sek.)
32000x24000	16	1	Abelklyngen	91995.00 ms (ca. 1.5 min.)

Tabell 32. Kjøring av MS på forskjellige maskintyper.

Delt minne på Abelklyngen:

Noder	Kjerner	Totalt	Tid i sekvens	Tid i parallell	*Speedup*
1	1	1	1296944.34 ms		
1	2	2	---	720600.00 ms	1.80
1	4	4		333105.42 ms	3,89
1	8	8		179630.29 ms	7,22
1	16	16		75970.10 ms	17,07

Tabell 33. MS med delt minne på Abelklyngen.

Under følger resultater for kjøring av MS på forskjellige node-konfigurasjoner:

Oppløsning	Zoom	Noder	Kjerner pr. node	Totalt ant. kjerner	Tid total	Tid reduksjon
32000x24000	48x	1	2	2	716211.14ms	
32000x24000	48x	2	2	4	343988.02ms	
32000x24000	48x	2	4	8	173942.55 ms	
32000x24000	48x	4	4	16	91995.00 ms	
32000x24000	48x	4	8	32	45401.59 ms	
32000x24000	48x	8	8	64	39931.92 ms	
31914x23936	48x	16	8	128	20732.68 ms	
31744x23808	48x	32	8	256	*15900.19 ms*	

31402x23552	48x	64	8	512	18508.82 ms	

Tabell 34. kjøring av MS på forskjellige node-konfigurasjoner.

Valget av node-konfigurasjon er basert på å få kort tid i køen. Derfor brukte jeg aldri mer enn 8 kjerner pr. node. Under følger resultater fra kjøring på en 1.5x høyere oppløsning:

Oppløsning	Zoom	Noder	Kjerner pr. node	Totalt ant. kjerner	Tid total	Tid reduksjon
48000x36000	64x	---	---	---	---	---

Tabell 35. Kjøring med høyere oppløsning ble ikke loggført. Men resultat bildet er i Figur 43 til og med Figur 46.

For å lage et større bilde trengs mer minne. Abelklyngen har noen *hugemem* noder som har 32 kjerner og 1TB med minne. Jeg brukte denne videre for å beregne 64000 x 48000.

Speedup for MS

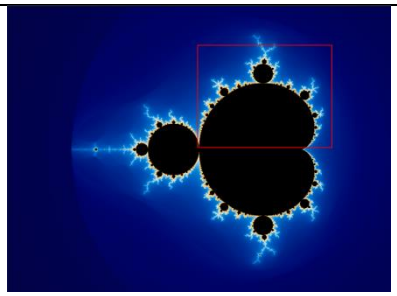
MPI prosesser	Tid i sekvens	Tid i parallell	*Speedup*
1	1296944.34 ms		1,00
2	---	716211.14ms	1,81
4		343988.02ms	3,77
8		173942.55 ms	7,45
16		91995.00 ms	14,09
32		45401.59 ms	28,56
64		39931.92 ms	32,47
128		20732.68 ms	62,55
256		*15900.19 ms*	81,56
516		18508.82 ms	70,07

Tabell 36. Speedup for MS.

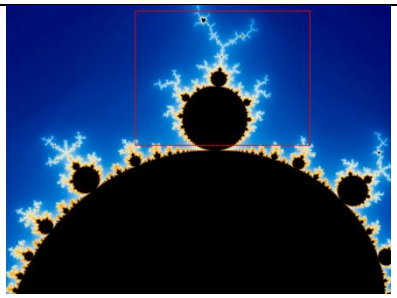
4.5.1. Drøfting av resultat

Det kan være vanskelig å avgjøre om resultatene over er gode nok i forhold til forventingen for Abelklyngens kapasitet. Derfor gjorde jeg en analyse av de forskjellige leddene i parallell MS.

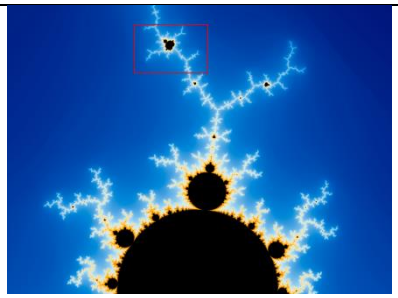
Siden det er vanskelig å få tilgang på mer enn 500 og oppover på Abelklyngen fordi det kan være kø på ressursen, gikk jeg ikke høyere på antall kjerner i utviklingsfasen. Det kan bli aktuelt senere når ønsket resultat oppnås med bruk av omtrent 500 kjerner. Dette kan også være fordelaktig ettersom det er omtrent 500 kjerner i en hylleseksjon på Abelklyngen.



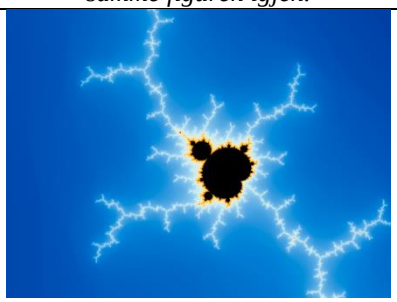
Figur 43. Originalbildet.



Figur 44. 4x forstørrelse. Her ser vi den samme figuren igjen.



Figur 45. 20x forstørrelse. Samme utsnitt som Figur 43.



Figur 46. 40x forstørrelse. Omtrent tilbake til originalbildet.

Teoretisk sett kan vi fortsette å forstørre bildet uendelig mange ganger, og det vil alltid være mulig å gi en fremstilling. Det fører til at vi ikke kan måle eksakt størrelse på figurer, noe som igjen beskriver at den er uendelig. Mandelbrotmengden er kun en observasjon og har foreløpig ikke noe teoretisk bruksområde som er kjent. Men det er en fin øvelse i å beskrive tallet uendelig med det komplekse planet. Og fordi det er fint å se på og man kan produsere uendelig mange forskjellig mønster basert på MS, blir det faktisk brukt som motiv og design på klær men ikke noe annet nyttig foreløpig.

5. Diskusjon

5.1. D1 – Drøfting av brukervennlighet

For å undersøke brukervennligheten innenfor HPC og de to typene parallell beregning som er presentert, har jeg foretatt en del praktisk arbeid. Det er enklere å beskrive sin egen erfaring enn andres, derfor har jeg sett på et utvalg av forskjellige problemtyper for å kunne gi en vurdering på brukervennlighet. Å utvikle programmer for HPC med MPI er vanskelig og tidkrevende[16]. Jeg har undersøkt dette nærmere for å kartlegge årsaken. Jeg har prøvd å måle hvor mye tid som brukes til utvikling av parallellprogrammer innen de to forskjellige typene.

Etter at det praktiske arbeidet var utført, satt jeg igjen med mange erfaringer. Et annet alternativ kunne vært å fokusere på andres arbeid. Lignende forskning er blitt utført tidligere og det kunne vært lurt å se mer på det[16]. Det blir blant annet nevnt i artikkelen til Holchstein at MPI kan utgjøre en økning på 20% i forhold til tiden det tar å utvikle applikasjonen, sammenlignet med andre typer systemutvikling.

Vurdering av praktisk arbeid
Under det praktiske arbeidet kom de største utfordringene i P3, der et Java-program skulle porteres til C. Det er flere grunner til at dette var tidkrevende og vanskelig å få til korrekt: For det første hadde programmet en høy kompleksitet av flere typer. Det besto av tre hovedsteg: sortering, partisjonering, og triangulering. For det andre hadde det et objekt-orientert design. Det tar tid å bygge om hvordan et program er designet, derfor ble det gjort forsøk på å lage en portering som lignet mest mulig på utgangspunktet. Dette var noe av den *tilfeldige kompleksiteten*[58] i P3. Størrelsen på programmet (1000 linjer) gjorde det også mer utfordrende enn først antatt. Dette var den *essensielle kompleksitet* (EK)[58] vi ikke kommer vekk fra. Metoden for å oversette JDT til CDT ble en slags testdreven utvikling uten de helt store testene. Litt som i artikkelen til Brooks der han beskriver inkrementell utvikling. Først lager vi et slags skall eller tomt program som kompilerer og kjører, men ikke gjør noe. Deretter fyller vi inn noen *dummy-funksjoner* som blir kalt på i en rekkefølge og skriver hva de skal gjøre. Siste steg er da å lage én og én funksjon slik at programmet vokser frem. På en annen side ville full kompatibilitet med MPI og Java vært et veldig kraftig utviklingsverktøy, med tanke på tiden det tar å lage C-programmer. Men dette ville gått kraftig utover ytelsen. Derfor burde MPI4Python testes og undersøkes videre i denne sammenheng[59] på grunn av at det er objekter som er kostbart å sende[57]. MPI4Python kan bruke datatyper fra *Numpy* og C og medfører derfor ikke så mye tap på ytelse slik mine tester med Java og MPI har visst.

HPC er vanskelig
Det er en del forutsetninger som må være på plass for å kunne starte å bruke HPC med MPI:

1. Erfaring med bruk av UNIX
2. Erfaring med bruk av språket C
3. Generell kunnskap om programmering
4. Generell kunnskap om algoritmer
5. Generell kunnskap om matematikk på realfaglig nivå

På UiO arrangeres et kurs som behandler tema HPC med MPI. Det heter INF3380 – Parallellprogrammering for naturvitenskaplig anvendelse.Våren 2015 hadde kurset et frafall på 44 % i forhold til antall studenter som fullførte kurset. Grunnen til dette var at studentene falt av etter å ha strøket på de obligatoriske oppgavene eller eksamen. Resultatet er tilgjengelig på *studweb* for studenter som har tatt kurset i 2015. Jeg mener det sier mye om vanskelighetsgraden for temaet, og hvor viktig det er å ha riktige forutsetninger for å lykkes. Det er en god blanding av tilfeldig kompleksitet (TK) og essensiell kompleksitet (Ek) som gjør dette vanskelig.

Tilfeldig kompleksitet:
Mengden av forkunnskap, vi kan godt si at det er fem forutsetninger. Dersom studenten mangler én eller flere vil det øke arbeidsmengden og studenten må bruke ekstra tid på å tilegne seg kunnskap den mangler.

Essensiell kompleksitet:
På grunn av at de fleste studenter på institutt for informatikk (IFI) har bakgrunn fra Java eller Python synes jeg det er greit å trekke frem denne sammenligningen. Det imperative programmeringsspråket C har et lavere abstraksjonsnivå enn Java og Python som begge er objekt-orienterte. Dette innebærer at man blir nødt til å skrive flere linjer kode for å få til samme handling. Det faktum at lengden på koden blir større er i seg selv en essensiell kompleksitet man ikke kommer utenom.

Begrensninger for implementasjon
Det mest krevende av det praktiske arbeidet var å parallellisere sekvensielle Java- og C-programmer med MPI. Å portere et Java-program til C kan være veldig tidkrevende å få til å fungere korrekt. Det kan også være tidkrevende å få et C-program til å være fri for minnelekkasjer og lignende. Å parallellisere programmet med MPI krever også mye tid. Design for distribusjon, reduksjon og partisjoner av data må defineres slik at programmet skalerer med økning i problemstørrelse og antall beregningsenheter som blir brukt. Det kan være tidkrevende å få til og noen ganger må det bli noen begrensninger for å få til det vi prøver å oppnå. Et eksempel kan være at det bare er mulig å kjøre programmet med antall prosessorer p tilsvarende en 2'er-potens slik som i P4 (Mandelbrotmengden) eller at p må være et kvadrattall. 1=(1 x 1), 4 =(2 × 2), 9 =(3 × 3), osv. slik som i P2 (Matrisemultiplikasjon). Disse begrensingene kommer på grunn av

datadistribusjonen og valg av design. Teoretisk sett kan denne typen begrensinger løses, men det vil ofte ikke gi noe nyttig resultat.

Delaunay trianguleringen i P3 var absolutt mest krevende siden alle tidligere nevnte utfordringer var der. For å understreke dette, tar jeg et eksempel der jeg skulle hente ut resultater for å understreke mine observasjoner angående P3. Under oppsummering og ferdigstillingen av program, går det mye tid til dette. Det skulle tilsynelatende virke trivielt men det er det ikke fordi det er mange parametere å stille på og det er litt kronglete satt opp, derfor går det mye tid. Se *figur 28* som viser arbeidsflyten. Alternativet kunne vært å automatisere denne prosessen med et skript som kjører alle komponentene i systemet. Men uansett vil det ta den tiden det tar fordi det er mange resultater som skal bokføres.

Imperativ programmering eller objektorientert programmering

Et annet alternativ enn C-MPI er å bruke Python og mpi4python for å gjøre utviklingsprosessen mindre tidkrevende. Det er vist at Python kan få tilnærmet samme ytelse som C dersom man bruker numpy eller datatyper fra C i koden[59]. Å bruke Python som *ducktape* for å satte samme programmet og så bruke datatyper fra numpy eller C burde testes ut mer. Jeg tror det vil gi en god forbedring på tidsbruk for utvikling. Et enkelt eksempel kan være å sammenligne signaturen til send-funksjonene i C og Python:

I programmeringsspråket C:
MPI_Send(void data, int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm comm)*

I programmeringsspråket Python:
Send(buffer, destination)

Det er ikke noe tvil om at det vil gå raskere å utvikle applikasjoner med et objekt-orientert språk[52]. Så lenge man ikke bruker objekter fra Python men kun primitive datatyper, burde ytelsen være god nok for veldig mange oppgaver. Jeg har dessverre ikke hatt mulighet til å undersøke dette nærmere med hensyn til testing av ytelse, men det kan virke som dette er veien å gå for å få ned tidsbruk for utvikling, som ikke går alt for mye utover ytelsen til applikasjonen.

Simulering av løsning

Et av målene med oppgaven var å finne ut hvor mye tid det tar å utvikle et program for delt minne sammenlignet med distribuert minne, og hvorfor det er ulikt. Her oppdaget jeg noe nyttig, nemlig at et program skrevet for distribuert minne kan simuleres og utvikles lokalt på en maskin med delt minne[46]. Det betyr i praksis at det ikke tar noe mer tid å utvikle programmet for distribuert minne, dersom man gjør det for delt minne først. Og dersom man er kjent med bruk av en maskinklynge, kan det gå ganske

raskt å få kjørt programmet. På en annen side er det ofte naturlig å gjøre dette, siden alle ikke har direkte tilgang til en maskinklynge. Men det er ikke alltid slik. På et HPC-kurs som arrangeres årlig av USIT, legger de opp til at brukerne skal utvikle programmene sine på Abelklyngen. En annen fordel er at man unngår kompleksiteten forbundet med å bruke en maskinklynge. Å bruke en maskinklynge i dag er omtrent som å bruke hullkortmaskiner i gamle dager. Gjøres det en feil med et semikolon, kan man miste mye tid fordi man faller ut av køen og må starte på nytt når man er klar.

Andre nyttige funn i denne oppgaven er ytelsen til MPI på en multikjerne. Vi ser faktisk at det kan være svært nyttig å bruke MPI selv om man ikke skal kjøre den parallelle algoritmen på en klyngemaskin. En stor fordel her er at programmet man har implementert ikke trenger å skrives om. Det eneste som må utføres, er å allokere ressurser og sette opp en node-konfigurasjon slik at programmet utnytter beregningskraften best mulig. Det gjør utvikling av et MPI-program mye enklere fordi vi slipper å forholde oss til kø-systemet. Når vi har et program som kjører fint med MPI lokalt i delt minne, er det enkelt å flytte det til et distribuert miljø og vi kan nærmest garantere at det kjører slik som forventet. Eneste forutsetning er at vi klarer å administrere maskinklyngen riktig. Det vil også være mulig å kunne få forventet resultat, siden vi allerede har kjørt en simulering. Dette blir også mye billigere for eieren av maskinklyngen.

Strøm og ressursbruk
Dersom alle brukere hadde prøvd ut programmer som er under utvikling, ville køen bli veldig lang på Abelklyngen. En maskinklynge bruker mye strøm, ca. $500w \times p$ (der p er antall noder), og det er kostbart å kjøpe timer. En høy *gjennomstrømming* er ønskelig for at maskinklyngene skal fungere optimalt, og det er ikke optimalt å spre et program på klyngen som bruker opp ressurser uten å produsere resultater. Gjennom arbeidet med denne oppgaven har jeg fått innsikt i at flesteparten av forskerne som bruker HPC har liten forståelse/kunnskap om temaet. For eksempel at en fysiker kjører en test uten å kjenne til parameterne for maskinvaren han kjører på. De får det resultatet de vil, men det man kan tjene er å spare tid og penger og få mer resultater (men kanskje er de resultatene de får gode nok). Det har de siste ti årene vært en stor økning som har gjort hardware billig og tilgjengelig. Før måtte man ha en ekspert for å kjøre, men nå har vi et web-grensesnitt som gjør at forskere kan sende inn sine programmer mye enklere. Men dette har sine begrensinger og systemet tar ikke imot programmer skrevet av brukeren, men data-sett som skal kjørers.

5.2. D2 - Drøfting av ytelse

Det er hovedsakelig *to* grunner til at vi bruker parallelle maskiner:
I.) Vi ønsker å oppnå eksakt samme resultat som i en sekvensiell løsning, men vi ønsker at det skal gå raskere å produsere resultatet.

II.) Det finnes noen problemer som er av en slik størrelse at de krever flere FLOPS eller mer hukommelse enn en vanlig datamaskin har, derfor må man ta i bruk en parallell datamaskin.
Jeg har sett på disse to utgangspunktene, og vist frem eksempler der jeg oppnår bedre ytelse i parallelle algoritmer sammenlignet med sekvensielle.

En skreddersydd løsning må til
For å undersøke hva slags ytelse og effektivitet som kan forventes innenfor HPC og de to typene parallell beregning som er presentert, har jeg parallellisert noen sekvensielle algoritmer for å måle kjøretid. Jeg har hovedsakelig startet med et sekvensielt C-program. Det sekvensielle programmet må være meget godt testet for kjøretidsfeil og minnelekkasjer. Når det fungerer slik det skal, kan man begynne å skrive om programmet til å kunne kjøre i parallell. Det må også planlegges nøye i designfasen og det er viktig å tegne opp hvordan data blir partisjonert, distribuert og redusert. Det finnes noen MPI-implementasjoner som løser lignende problem vi har sett på. Men min erfaring er at det er sjeldent man finner eksakt det man leter etter, derfor blir ofte eneste mulighet å implementere selv. På grunn av C og MPI sin kompleksitet er det ønskelig å ha flere programmer som man har utviklet selv, som gjør det enkelt å få ut de resultater man søker etter. Når det kommer til HPC generelt, har vi masse ressurser og maskinvare på universitetet (Abelklyngen), i hverdagslivet (mobiltelefoner), og i resten av verden (Internett). Det er store nettverk med datamaskiner overalt, så det er ikke vanskelig å tenke seg at ting burde gå litt fortere. Men det er ikke alltid så lett å få *speedup* selv om man har masse ny og fin maskinvare. Derfor ønsket jeg å finne ut hvilke av problemtypene som er egnet for å oppnå dette med delt og distribuert minne. Og jeg ønsket å finne ut hvilken *speedup* jeg kan få på de forskjellige problemtypene.

Optimal bruk av Abelklyngen
Etter arbeid med P0-P3 ble det klart for meg hva slags type problem Abelklyngen er best egnet til. Derfor ble Mandelbrotmengden (MS) valgt for å prøve å vise kapasiteten til Abelklyngen og speedup sammenlignet med tidligere implementasjoner (P0-P3). For å få til optimal bruk av Abelklyngen er det viktig å bruke riktig design for parallellisering av algoritmen og en problemtype som er egnet. Av de parallelle algoritmene som er prøvd ut på Abelklyngen, er det *én* problemtype som viser seg å være best egnet: Det er når data kan leses inn i parallell fra et delt område, slik at alle prosesser vil ha sin del å beregne. Da kan alle prosesser beregne i parallell uten at de er avhengige av data fra andre prosesser. Siste steg er at alle skriver data til et felles område. *To* av problemene vi har sett på egner seg for dette: DT og MS. Det er problemer der det kun er behov for *ett* eller ingen kommunikasjonsledd før beregning og ett ledd etterpå og ingen underveis. I praksis kan disse erstattes av lesing og skriving til fil. Det gjør at kommunikasjonsleddet blir skjult og vi nærmest kan se bort fra det i tidsmålingen. Et eksempel kan være når skriving av data skjer mens beregning pågår.

Ønsket resultat oppnådd
Jeg har klart å oppnå noe av det jeg ønsket, nemlig *speedup*. Derimot ble det ikke de verdiene jeg hadde håpet på i maskinklynge. Det blir en tilnærmet lineær *speedup* når jeg kjører i delt minne på *to* av problemene (P2 og P3). Men når jeg kjører i maskinklynge med distribuert minne blir det ikke slik. Til og med for problemer som er *pinlig parallelliserbare*, er det vanskelig å få noe i nærheten av lineær speedup med lineær økning av prosessorer[60]. Problemene som ikke krever noe kommunikasjon under beregningssteget, klarer å oppnå en logaritmisk speedup. Men under selve beregningssteget er det naturligvis en lineær *speedup* om vi ser bort fra distribusjonskostnadene. Siden distribusjon av data ofte er enveiskommunikasjon(*ping*), er det mindre kostbart enn dersom det hadde vært behov for synkronisering av data. Dersom dette gjelder for reduksjonssteget også, kan vi nærmest overlappe disse kostnadene med beregningsleddet av programmet ved å bruke *pipelining*. Det vil si en teknikk for å gjennomføre flere oppgaver overlappende. For å få dette til må reduksjonen utføres på en enkel måte. Jeg har funnet *to* måter som kan brukes: Skrive til delt filområde eller lage en sekvens med filer i parallell.

Beste speedup

	Delt minne	**Distribuert minne**
Sortering med n = 10^6	4.70	2.91
Matrisemultiplikasjon med n = 10^6	25.83	8.07
Delaunay triangulering med n = 10^6	12.58	38.92
Mandelbrotmengden med n = 10^7	15.42	70.07

Tabell 37. Beste speedup i delt og distribuert minne.

For å unngå flaskehalsen ved alle-til-en-kommunikasjon på reduksjonssteget[53], er det mulig å bruke et delt fil-område på det parallelle filsystemet i maskinklyngen. Det vil føre til at alle prosesser kan sende og skrive data i parallell til root-prosessen som sørger for at data havner i *én* delt fil. Den andre måten er at alle prosesser skriver data til disk i parallell. Da slipper vi å vente på synkronisering av data. Ulempen med det siste alternativet er selvsagt at vi får like mange filer som prosesser. Men dersom dataene er uavhengige av hverandre, er det ikke noe problem å sette sammen filene dersom data er 1D-partisjonert i utgangspunktet.

Den optimale måten å bruke Abelklyngen på, er å gi den «små parallell-jobber» det vil si de som bruker under 1000 kjerner. Så må den få et problem som er pinlig paralleliserbart, der resultatet ikke trenger å være sammenhengende.

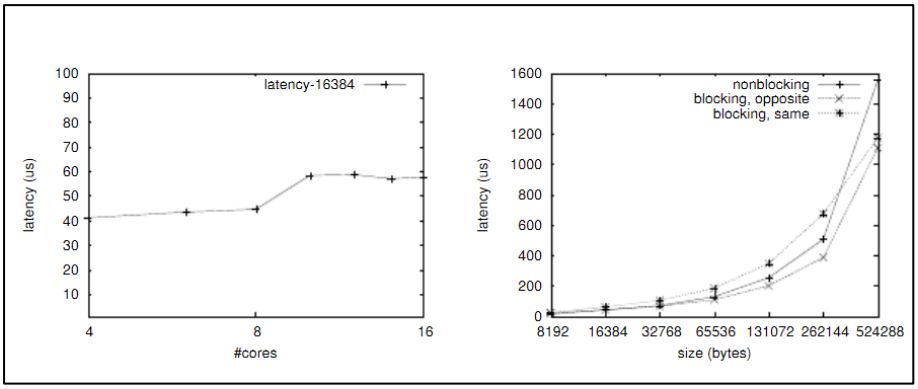
Denne observasjonen og løsningsforslaget ble oppdaget under arbeid med P3. Datastrukturen og datadistribusjonen ble vanskelig å håndtere fordi det var en veldig ujevn fordeling av data. Det gikk mye tid til å forsøke å samle inn data på en fornuftig måte, men jeg lyktes ikke. Reduksjonssteget ble veldig kostbart og det så ut til at forsøket/implementasjonene var mislykket. Samtidig undersøkte jeg MPI I/O, og oppdaget at det var mulig å skrive data i parallell siden vi jobber på en maskinklynge med et parallelt filsystem. Selv om distribusjon av data ble veldig kompleks på grunn av parallelliseringen, så var ikke dataene avhengige av hverandre. Derfor var det mulig å både skrive til delt filområde og til flere del-filer i parallell. Det løste problemet med reduksjon og synkronisering av data i P3. Dette er også såkalt *best-practice* for bruk av Abelklyngen i følge Dr. Ole Widar Saastad, senioringeniør på Abelklyngen. Det er slike jobber USIT ønsker det skal være flest av på Abelklyngen, såkalte små parallell-jobber som er pinlig parallelliserbare og som kan skrive data i parallell. Jeg kunne selvsagt spurt dem om svaret, men det er ikke sikkert jeg hadde forstått dette temaet på samme måte som nå. Det er heller ikke sikkert jeg ville brukt tid på å implementere de forskjellige problemene P0-P4. Ole Johan Dahl sa en gang at «å lære, det er å programmere» [Albregtsen 2012 & 2013]. Derfor synes jeg dette var den mest fornuftige måte å angripe problemet på.

Det er hensiktsmessig med en mer effektiv bruk av ressursene på Abelklyngen og i HPC generelt. Min erfaring er at de fleste brukerne av Abelkyngen og HPC også her jf. D1, ikke har noe særlig kunnskap om det å bruke en maskinklynge og fornuftig bruk av ressursene. De er som regel forskere som har annen bakgrunn enn informatikk. Derfor blir ofte HPC-ressurser brukt på feil måte, eller en mindre effektiv måte. Et eksempel kan være at en forsker ikke nødvendigvis har kjennskap til parallellitet eller parallell programmering. Vedkommende sender kanskje inn et program til Abelklyngen som er en sekvensiell algoritme. Kjøretiden på en vanlig maskin vil kanskje tilsvare mange timer eller flere dager. Denne situasjonen kan ligne den i P4, der vi ønsker å vise hvordan vi kan øke oppløsning på et resultat. Dette er et eksempel på hvordan HPC gir mer nøyaktighet. Det er lønnsomt og nødvendig for forskeren å sende inn programmet som en *jobb* på Abelklyngen, fordi det frigjør ressurser i laben siden en maskin da blir ledig. I utgangspunktet er det ikke noe galt med denne typen bruk av Abelklyngen, fordi i det *store bildet* blir det jo bare mange jobber som kjører i parallell. Men dersom denne jobben tar mye minne, vil det gå utover resten av kjernene på noden. Fordi det kun er 64GB pr. node for 16 kjerner (4 GB pr. kjerne), så kan dette hindre Abelklyngen i å brukes til det den er bygget for, nemlig å utføre distribuerte beregninger på opptil 1000 kjerner. Sekvensielle jobber tar minne fra noden, som gjør at det kan ta lengre tid før man får allokering til sin parallelle jobb.

God ytelse med rettferdig ressurs allokering

Jeg oppnådde en vesentlig økning i ytelsen og en speedup på nesten faktor 40 fra delt minne til distribuert. Det viser at det kan være mye å tjene på å parallellisere riktig type problem, det vil si den type problem en klynge er konstruert for. Det blir også en begrensning fordi vi alltid må tilpasse oss

maskinvaren, og det gjør at vi får færre valgmuligheter for bruksområdene ettersom hvor viktig minneallokering er når man jobber med mye data. I undervisningssammenheng er ofte oppgavene som skal løses forenklet og idealisert. Derfor er det mange som aldri møter de virkelige problemene man møter på i større systemer som er mer komplekse. I forhold til annen forskning er noen funn konsistente med tidligere observasjoner[48] når det gjelder sortering og matrisemultiplikasjon, men speedup på DT og MS motstrider til en viss grad dette. ROLFE snakker om et *straffegebyr* som kommer når programmet beveger seg fra en node til en annen[48]. Det er tydelig tilstede i sortering og matrisemultiplikasjon, men det kommer ikke like tydelig frem i Delaunay triangulering og Mandelbrotmengden. Det er fordi de bruker mye lengre tid på sitt beregningssteg sammenlignet med de andre. Kjøringen av DT og MS har en allokering som er ganske rettferdig mot resten av køen, fordi den vil ligge der kortere tid på grunn av node-konfigurasjonen. Vi går ikke over 8 kjerner pr. node, mens det er 16 tilgjengelige, men oppnår likevel en vesentlig speedup. For å få optimale resultater burde det kjøres med fulle noder, det vil si 16 kjerner allokert på hver. Men det var ikke behov for dette for å oppnå ønsket resultat.



Figur 47. Forsinkelse ved økning av antall kjerner og pakkestørrelse[53].

Problemet med å få en forventet speedup på parallelle systemer er knyttet til forsinkelsen det tar å sende en pakke fra maskin A til maskin B. Nå det gjelder denne ekstra kostnaden som kjøring av en algoritme får, er det hovedsakelig pakkestørrelsen som fører til flaskehalsen. *Figur 47* viser hvordan forsinkelsen ved økning av antall prosessorer er nærmest konstant, med unntak av et hopp der antall prosessorer er større enn 8 der pakken går ut fra noden. Ellers er det tydelig at en økning i pakkestørrelsen gir en graf som er tilnærmet eksponentiell vekst[53].

6. Konklusjon

Etter to år med forskning på og arbeid med parallellprogrammering på multikjerneprosessor og maskinklynge sitter jeg igjen med flere nyttige funn. Jeg har lært at HPC er vanskelig å mestre, og krever essensielle forkunnskaper om programmering, algoritmer og matematikk. Erfaring med UNIX og programmeringsspråket C er de viktigste forutsetninger. Å lære seg HPC på distribuert minne er mer krevende enn på delt minne, hovedsakelig på grunn av kø-systemet. Når man har lært å bruke MPI og først fått det til på laptop åpner det for bedre effektivitet på maskinklynge dersom man planlegger godt, men det begrenser seg til pinlig parallelliserbare problemer som ikke forutsetter noe kommunikasjon mellom prosessene under beregning av data.

For å oppnå størst mulig speedup må programmer kjøres på maskinklynge. Distribuert minne åpner for gode resultater på speedup under riktige forutsetninger. Parallelle programmer er normalt enklere å utvikle med objekt-orientert programmering enn med imperativ programmering. For å kjøre parallelle programmer på Abelklyngen forutsetter det derimot at programmet er laget med imperativt programmeringsspråk som C og Fortran. Man kan utvikle programmer i delt minne for å simulere løsning av problemer som kan kjøres på distribuert minne. Dette tilrettelegger for mer effektiv utnyttelse av klyngearkitektur som i Abelklyngen. Med riktig bruk gir beregninger i distribuert minne definitivt høyere oppløsning og mer nøyaktige resultater enn hva som er mulig i delt minne på grunn av mer maskinkraft i klyngen. C-MPI er betraktelig mer effektivt enn Java-MPI på både multikjerne og maskinklynge.

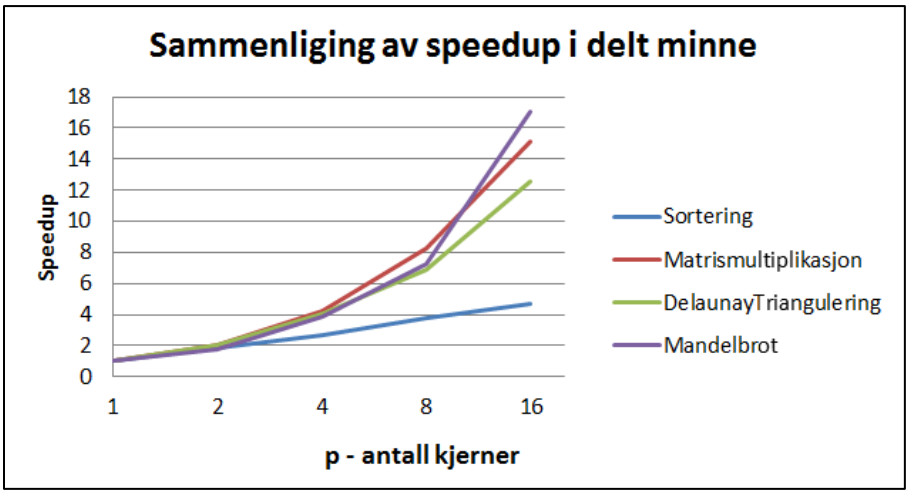
For å oppsummere, viser mine undersøkelser at en klynge er vanskelig å programmere[46]. Det ser ut som bare *én* type problemer gir bedre speedup i distribuert minne enn i delt minne, nemlig de pinlig parallelliserbare problemene som er utført på Abelklyngen og dokumentert i denne oppgaven. Jeg har kommet fram til "best practice" for bruk av Abelklyngen og konkludert med at distribuert minne er raskere og mer effektivt enn delt minne ved riktig bruk.

6.1. Brukervennlighet

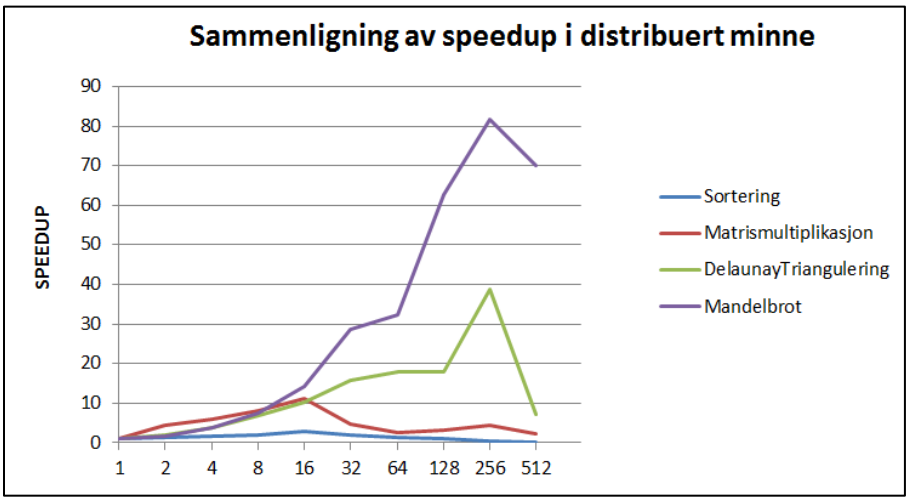
Klyngen er ikke lett å bruke, kort oppsummert er det mye å lære, lett å gjøre feil, og istedenfor interaktive kjøringer har man et system som minner om batch-prosessering fra hullkorttiden.

6.2. Ytelse

Etter mye arbeid viser implementering på en av problemtypene av flere jeg testet bedre ytelse, enn kjøring på multikjerne med delt minne. Se Figur 48 og Figur 49.



Figur 48. På en typisk multikjerne får vi ikke speed-down.



Figur 49. Det kommer en speed-down når distribusjons kostandene går opp.

For å oppsummere er to viktig poenger oppnådd:
I.)
Hvor brukbar er en maskinklynge: Det er vanskelig å bruke og det gir ikke alltid noe bra resultat.
II.)
Innsamling og reduksjon av behandlet data er en flaskehals:
Det er mest lønnsomt å skrive til et delt filområde eller lage fil-sekvenser fremfor å gjøre å gjøre *alle-til-en* reduksjon.

7. Videre arbeid

Under arbeid med denne oppgaven har det ikke skjedd stort i dette fagområdet. Bruk av MPI er fortsatt den ledende standarden i HPC. Det kan forventes at antall brukere har gått noe opp. Grunnen til dette er at HPC blir et større satsningsområde fordi det er stadig behov for å kunne løse større problemer med mer nøyaktighet i dagens samfunn. Kostnadene på

maskinvare går ned, noe som gjør at maskinklyngen stadig blir større med bedre ytelse. Under oppgaven har det blitt presentert noen forslag til hvordan man kan øke effektivitet i utviklingsfasen og for selve implementasjonene. Under følger en liten oppsummering av disse momentene.

- Bruk av MPI4Python når det kommer til å redusere utviklingstid. Det er også sannsynlig at denne metoden kan gi ønsket ytelse dersom det brukes korrekt.

- Mer fokus på hybrid-modellen som bruker delt minne på beregning intensive biter av programmet og distribuert minne for å løse større problemer. Det er hensiktsmessig for å oppnå bedre ytelse på implementasjon.

- Sammenligning av ytelsen til en parallellisering av DT i Java med CDT som kjører på Abelklyngen.

- MPI3 burde bli tatt i bruk når arbeidet foregår på en moderne klyngemaskin. Det blir stadig flere kjerner i maskinklyngene og bruk av akseleratorer som GPU, Xeon Phi blir mer vanlig. Så det er behov for å bruke en ny implementasjon som kan håndtere flere millioner MPI *ranks* og bedre støtte for bruk av hybrid-modeller.

- Kvantedatamaskin som akselerator i maskinklynge blir kanskje en stor ytelse forbedring for kombinatoriske optimaliserings problemer som TSP. Det burde undersøkes om det mulig få tilgang gjennom universiteter i USA.

Vedlegg 1

Programmene og skripts som er implementert og kjørt for å oppnå resultatene i oppgaven er tilgjengelig på:

https://bitbucket.org/galileo100584/masteroppgave-v2016

Alle implementasjoner (P0-P4) ligger i mapper på dette nettområdet.

Vedlegg 2

Ordliste

Engelsk	**Norsk**	**Forkortelse**
Array	For 1D: vektor For 2D: matrise	
Multicore	Multikjerne	
HPC	Høy-ytelses beregninger	HPC
Speeup	*Speedup*	SP
Speeddown	Speeddown	SD
Delaunay Triangulation	Delaunay triangulering	DT
	Delaunay triangulering med Java	JDT
	Delaunay triangulering med C	CDT
Struct	*Struct*	
Debugging	Feilsøking	
Parsing	Parsing	
Integer	Heltall	*int*
Deadlock	Vranglås	
Race condition	Data kappløp	
Brute force	Rå kraft	
Idle	Ledig prosessor/kjerne, ventende	
Java-MPI	Java-MPI	J-MPI
C-MPI	C-MPI	C-MPI
Convex hull	Konveks innhylling	KO
Casting	Kasting	
Rack	Hylleseksjon	
Very-large-scale integration		VLSI
Operating system	Operativsystem	OS

Ordforklaring

Absoluttverdi:
Absoluttverdien til et reelt tall er det samme som tallets verdi uten fortegn.

Kjerne:
Beregningsenhet som ligger på prosessoren. Det kan være én eller flere på én prosessor.

Delt minne:
På en multikjernemaskin er alt hovedminne delt mellom kjernene i prosessoren.

Distribuert minne:
Maskiner er satt sammen i nettverk. Hver maskin har sitt eget hovedminne. Maskinene kan ha en-kjerne prosessor eller multikjerne prosessor.

Metode:
I Java er en metode en funksjon som kan endre instansen av objektet den tilhører, samt variabler i objektets omsluttende klasser og i andre objekter via pekere.

Funksjon:
I C er en funksjon noe som kan endre en lokal variabel og global variabel i programmet, og den returnerer en primitiv type.

Prosess:
Når vi bruker C og MPI, er arbeiderne prosesser, uavhengig om det er i multikjerne eller klynge.

Tråd:
Når vi bruker Java er det tråder som fører til at vi kan gjøre parallell beregninger på en multikjerne. I C har man Pthreads.

Node:
Brukes for å omtale én av maskinene i maskinklyngen (Abelklyngen)

Hylleseksjon:
Skap med noder eller datamaskiner plassert over hverandre i høyden.

VLSI:
Very-large-scale integration er metoden som blir brukt for å produsere en integrert krets (IC) der flere elektroniske og logiske komponenter er sammen på brikke.
Mikroprosessoren er en type VLSI.

SI grunnenheter i oppgaven

Tid	sekund	s
Termodynamisk temperatur	kelvin	K
Frekvens	hertz	Hz

Effekt	watt	W
Elektrisk potensial	volt	V
Datamengde	bit	b
Datamengde	byte	B

SI prefikser i oppgaven

10^{18}	exa	E	Trillion	1 000 000 000 000 000 000
10^{15}	peta	P	Billiard	1 000 000 000 000 000
10^{12}	tera	T	Billion	1 000 000 000 000
10^{9}	giga	G	Milliard	1 000 000 000
10^{6}	mega	M	Million	1 000 000
10^{3}	kilo	k	Tusen	1 000
10^{2}	hekto	h	Hundre	100
10^{1}	deka	da	Ti	10
10^{-1}	desi	d	Tidel	0,1
10^{-2}	centi	c	Hundredel	0,01
10^{-3}	milli	m	Tusendel	0,001
10^{-6}	mikro	µ	Milliondel	0,000 001
10^{-9}	nano	n	Milliarddel	0,000 000 001
10^{-12}	piko	p	Billiondel	0,000 000 000 001

Figurer

Figur 1. 2D-mesh Topologi. Et eksempel på hvordan maskiner kan settes sammen til en klynge. 9
Figur 2. Flynn's Taxonomy er en beskrivelse av de forskjellige modellene av maskintyper for parallellberegninger med unntak av SISD som naturligvis ikke en parallell maskin. 10
Figur 3. Minnemodell for en Quad Core-prosessor.[9]. 12
Figur 4. Speedup som funksjon av antall kjerner og andel parallell kode i programmet[12] 13
Figur 5. Speedup-funksjon av antall kjerner[14] 14
Figur 6. En forenkling av Abelklyngens topologi. Rack er det samme som en hylleseksjon. Det er omtrent 24 hylleseksjoner i virkeligheten. 22
Figur 7.Delt minne. 24
Figur 8. Distribuert minne. 24
Figur 9. Pseudokode for matrisemultiplikasjon (MM). 28
Figur 10. MM for en blokk. 28
Figur 11. De to øverste linjene er pre-prosesseringsdirektiver for kompilatoren (GCC). 29
Figur 12. Delaunay triangulering med omkrets-sirkel for hvert triangel. 30
Figur 13. Her vises hvordan DT blir til datagrafikk. I denne 3D-modelllen er det variasjonen i trekantflates vinkel i forhold til lyskilden og øyepunktets plassering som skaper en illusjon av dybde i bildet[30]. 30
Figur 14. Oppdeling av punktene i rektangulære bokser[31]. 31
Figur 15.Datastruktur for punkter [31]. 32
Figur 16. Det er fem punkter på KO og totalt 10 punkter i P. 32
Figur 17. Beregning av den konvekse innhyllingen [31]. 33
Figur 18.Kandidater til punkt C i trekanten ABC [31]. 34
Figur 19. Innsamling av Delaunay triangler runde et punkt A. 35
Figur 20. Beregner neste punkt C[31]. 35
Figur 21. Her ser vi et eksempel på en kryssende kant, som ikke er lovlig i Delaunay triangulering. 36
Figur 22. Her ser vi hvordan en linje blir til et tre etter 3 iterasjoner av samme operasjon[34]. 37
Figur 23. Her ser vi fraktal geometri i et antennesystem[36]. 38
Figur 24. Den reelle og imaginære aksen i et komplekst plan med punktet z=a + ib.[40] 40
Figur 25. Tidslinje for praktisk arbeid i oppgaven. 42
Figur 26. Her vises et konseptuelt bilde av hvordan et program blir kjørt i distribuert minne. Prosess 0 til prosess (P - 1) kjører på hver sine prosessorer. En node i delt minne eller flere distribuert[45]. 44
Figur 27. Sendeskjema for MPI-prosessene. 52
Figur 28. Forskjellige nivåer av abstraksjon i programmeringsspråk[49]. Oversetting mellom et nivå er tidkrevende. 55
Figur 29. Pseudokode for DT-algoritmen. Algoritmen for å finne den konvekse innhyllingen er i Figur 30. 60
Figur 30. Pseudokode for cohull-algoritmen (den konvekse innhyllingen). 61
Figur 31. Diff sjekk mellom JDT og CDT ble brukt for å oppdage hvor feil var. ... 61
Figur 32. Punktene som skal trianguleres. 64
Figur 33. Etter at prosess 0 har funnet sin konvekse innhylling. 66
Figur 34. Etter at prosess 0 (P0) har funnet sin konvekse innhylling og behandlet sitt punkt. Det vil si at P0 har funnet sine Delaunay-kanter (DK). 66

Figur 35. Etter at prosess 1 har funnet sin konvekse innhylling. *67*
Figur 36. Etter at prosess 1 har funnet sin konvekse innhylling og behandlet sitt punkt. *67*
Figur 37. Union av grafen til P0 og P1 etter konveks innhylling og alle punkter er funnet blir en komplett DT. *68*
Figur 38. Vi ser 15 oppgaver totalt i sum over alle nivåer. Hver av disse tar tid t=10 for en prosess. Den maksimale arbeidsmengden som kan utføres i parallell endrer seg fra nederste til øverste nivå. Det er kun på nederste nivå alle prosessene kan jobbe i parallell. *70*
Figur 39. Alle prosesser (med navn task i denne figuren) sitter med sin del av løsningen. De kan alle skrive sin del til en felles fil. *71*
Figur 40. Arbeidsflyt for testing av portering. *73*
Figur 41. Resultat av CDT vist frem med JDT. *73*
Figur 42. Grafen ytelse til MPI4Python og behov for båndbredde ved sending av objekter (Python datatype, numPy og primitive C datatyper[57] *81*
Figur 43. Originalbildet. *92*
Figur 44. 4x forstørrelse. Her ser vi den samme figuren igjen. *92*
Figur 45. 20x forstørrelse. Samme utsnitt som Figur 43. ..... *92*
Figur 46. 40x forstørrelse. Omtrent tilbake til originalbildet. *92*
Figur 47. Forsinkelse ved økning av antall kjerner og pakkestørrelse[53]. *100*
Figur 48. På en typisk multikjerne får vi ikke speed-down. *102*
Figur 49. Det kommer en speed-down når distribusjons kostandene går opp. *102*

Tabeller

Tabell 1. Abelklyngens kapasitet. *23*
Tabell 2. Timer brukt i oppgaven på programutvikling. *43*
Tabell 3. Strømbruk for Abelklyngen. *49*
Tabell 4.Ping-test med C-MPI. *50*
Tabell 5. Sammenligning av C og Java syntaks. *56*
Tabell 6. JDT og CDT med $n = 105$. *63*
Tabell 7. JDT og CDT med $n = 106$. *63*
Tabell 8. C-MPI med ping-test. *79*
Tabell 9. J-MPI med ping-test. *79*
Tabell 10. J-MPI med Ping-pong-test. *79*
Tabell 11. Ping-pong-test med C-MPI. *80*
Tabell 12. J-MPI med mergesort på multikjerne. *80*
Tabell 13. Java-MPI med mergesort på Abelklyngen. *80*
Tabell 14. Parallell mergesort med C-MPI i delt minne. *82*
Tabell 15. Parallell mergesort med C-MPI i distribuert minne. *82*
Tabell 16. Matrisemultiplikasjon med delt minne.. ..... *82*
Tabell 17. Matrisemultiplikasjon med distribuert minne. *83*
Tabell 18. Delt minne med $N=106$ *og write_output_seq på Lenovo laptop.* *83*
Tabell 19. Delt minne med $N=107$ *og write_output_seq på Lenovo laptop.* *83*
Tabell 20. Delt minne med $N=108$ *uten å skrive til disk, med Lenovo laptop.* *84*
Tabell 21. Delt minne med $N=106$ *og write_output_seq på Abelklyngen med én node.* *84*
Tabell 22. Delt minne med $N=107$ *og write_output_seq på Abelklyngen med én node.* *84*
Tabell 23. Delt minne med $N=108$ *Abelklyngen med én node. Her ses kun på distribusjon og beregningstid.* *84*

Tabell 24. ($N=10^6$) write_output_seq på Abelklyngen. Alle mellom tider ble ikke registret. *85*
Tabell 25. ($N=10^7$) write_output_seq på Abelklyngen. *86*
*Tabell 26. Speedup N=*106 *distribuert minne.* *86*
Tabell 27. Speedup $N=10^7$ distribuert minne *87*
Tabell 28. Speedup $N=10^8$ distribuert minne *87*
Tabell 29. DT med 1 mill. punkter og to prosesser. *88*
Tabell 30. DT med 1 mill. punkter og to prosesser med MPI IO. *89*
Tabell 31. Økning i antall prosesser som bruke på DT. *89*
Tabell 32. Kjøring av MS på forskjellige maskintyper. *90*
Tabell 33. MS med delt minne på Abelklyngen. *90*
Tabell 34. kjøring av MS på forskjellige node-konfigurasjoner *91*
Tabell 35. Kjøring med høyere oppløsning ble ikke loggført. Men resultat bildet er i Figur 43 til og med Figur 46. ..... *91*
Tabell 36. Speedup for MS. *91*
Tabell 37. Beste speedup i delt og distribuert minne. *98*

Referanser

[1] Brombach, H. *Bransjen gir opp Moores lov*. 2016 Besøkt 10.05.2016]; http://www.digi.no/bedriftsteknologi/2016/02/16/bransjen-gir-opp-moores-lov.

[2] Hill, M.D. and M.R. Marty, *Amdahl's law in the multicore era*. Computer, 2008(7): p. 33-38.

[3] Sharma, G., A. Agarwala, and B. Bhattacharya, *A fast parallel Gauss Jordan algorithm for matrix inversion using CUDA.* Computers & Structures, 2013. **128**: p. 31-37.

[4] Bioportal. *Bioportal*. 2015 Besøkt 10.05.2016; http://www.mn.uio.no/ibv/bioportal/.

[5] CERN. *About CERN*. 2016 Besøkt 10.05.2016; http://home.cern/about.

[6] University, T.T.o.P. *Planet Lab*. 2016 Besøkt 10.05.2016; www.planet-lab.org.

[7] Institute, S. *SETI*. 2016 Besøkt 10.05.2016; www.seti.org.

[8] PRESENT, I., *Cramming more components onto integrated circuits.* Readings in computer architecture, 2000. **56**.

[9] Anders Brunland, K.H., Ole Christian Lingjærde, Arne Maus, *Rett på JAVA.* Universitetsforl., 2011: p. 350.

[10] Museum, C.H. *Gene Amdahl*. 2015 Besøkt 10.05.2016; http://www.computerhistory.org/fellowawards/hall/bios/Gene,Amdahl/.

[11] Amdahl, G.M., *Validity of the single processor approach to achieving large scale computing capabilities.* Proceedings of the April 18-20, 1967, spring joint computer conference, 1967: p. 483-485.

[12] rtcmagazine.com. *Amdahls lov*. 2016 Besøkt 13.05.2016]; http://www.rtcmagazine.com/files/images/4733/rtc1306_td_vio_fig01_original_large.jpg.

[13] Gustafson, J.L., *Reevaluating Amdahl's law.* Communications of the ACM, 1988. **31**(5): p. 532-533.
[14] Ly, P. *Gustavsons lov*. 2016 Besøkt 13.05.2016]; https://plot.ly/~astrosuf/39.png.
[15] Karp, A.H. and H.P. Flatt, *Measuring parallel processor performance.* Communications of the ACM, 1990. **33**(5): p. 539-543.
[16] Hochstein, L., F. Shull, and L.B. Reid, *The role of MPI in development time: a case study.* High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for, 2008: p. 1-10.
[17] project, G. *Fortran*. 2016 Besøkt 10.05.2016; https://gcc.gnu.org/fortran/.
[18] Cisco. *The Zettabyte Era—Trends and Analysis*. 2016 Besøkt 10.05.2016; http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/VNI_Hyperconnectivity_WP.html.
[19] BBC. *The Age of Big Data*. 2014 Besøkt 01.08.2014; http://www.bbc.co.uk/programmes/b01rt4c7.
[20] DWave. *DWave*. 2016 Besøkt 10.05.2016; http://www.dwavesys.com/.
[21] Mansfield, A., *Nasa buys into 'quantum' computer.* BBC - Science & Environment, 2013.
[22] USRA. *Quantum Computing - RFP*. 2016 Besøkt 10.05.2016; http://www.usra.edu/quantum/rfp/.
[23] Corporation, L.M. *Quantum Computing*. 2016 Besøkt 10.05.2016; http://www.lockheedmartin.com/us/what-we-do/emerging/quantum.html.
[24] Notur. *Notur - The Norwegian metacenter for computational science*. Besøkt 10.05.2016; https://www.notur.no/about.
[25] Ananth Grama, A.G., George Karypis, Vipin Kumar, *Introduction to parallel computing.* Pearson Education Limited, 2003: p. 27-30.
[26] Ananth Grama, A.G., George Karypis, Vipin Kumar, *Introduction to parallel computing.* Pearson Education Limited, 2003: p. 95.
[27] Ananth Grama, A.G., George Karypis, Vipin Kumar, *Introduction to Parallel Computing* Pearson Education Limited, 2003: p. 345-347.
[28] Woods, R.C.G.R.E., *Digital Image Processing.* Prentice Hall, 2008(3rd. ed.): p. 65-68.
[29] Delaunay, B., *Sur la sphere vide.* Izv. Akad. Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk, 1934. **7**(793-800): p. 1-2.
[30] Pierre Alliez, M.D., Jane Tournois, Camille Wormser, *Interleaving Delaunay Refinement and Optimization for Practical Isotropic Tetrahedron Mesh Generation.* ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2009, 2009. **28**(3): p. Art. no. 75.

[31] Maus, A., *Delaunay triangulation and the convex hull ofn points in expected linear time.* BIT Numerical Mathematics, 1984. **24**(2): p. 151-163.
[32] Maus, A. and J.M. Drange, *All closest neighbors are proper Delaunay edges generalized, and its application to parallel algorithms.* Proceedings of Norwegian informatikkonferanse, 2010: p. 1-12.
[33] Mandelbrot, B., *Fractals and the geometry of nature*. W. H. Freeman and Co., 1982: p. 169-181.
[34] davis.wpi.edu. *Fractal Tree sequences*. 2016 Besøkt 10.05.2016; http://davis.wpi.edu/~matt/courses/fractals/trees.gif.
[35] Cohen, N. *Fracatal Antenna Systems*. 2010 Besøkt 10.05.2016; http://www.fractenna.com/nca_cohen_bio.html.
[36] mdpi.com. *Fractal Antenna*. Besøkt 09.05.2016; http://www.mdpi.com/sensors/sensors-13-17362/article_deploy/html/images/sensors-13-17362f1-1024.png.
[37] Sharma, P. *Computer Generated Imagery*. 2014 Besøkt 10.05.2016; http://www.slideshare.net/pribhat/computer-generated-imagery.
[38] Schtick, T.H. *New paper finds climate change is explained by fractals* 2014 Besøkt 10.05.2016; http://hockeyschtick.blogspot.no/2014/08/new-paper-finds-climate-change-is.html.
[39] Mäkikallio, T.H., et al., *Prediction of sudden cardiac death by fractal analysis of heart rate variability in elderly subjects.* Journal of the American College of Cardiology, 2001. **37**(5): p. 1395-1402.
[40] UiO. *Komplekse tall og geometri*. Besøkt 10.05.2016; http://www.uio.no/studier/emner/matnat/math/MAT1001/h08/Kompendium2/Kapittel3.pdf.
[41] Parnas, D.L. and P.C. Clements, *A rational design process: How and why to fake it.* Software Engineering, IEEE Transactions on, 1986(2): p. 251-257.
[42] Sweller, J., *Cognitive load during problem solving: Effects on learning.* Cognitive science, 1988. **12**(2): p. 257-285.
[43] UiO. *LifePortal*. 2016 Besøkt 10.05.2016; https://lifeportal.uio.no/root.
[44] Jones, R. and R.D. Lins, *Garbage collection: algorithms for automatic dynamic memory management.* 1996.
[45] INF3380. *The mental picture of parallel execution*. 2011 Besøkt 10.05.2016]; http://www.uio.no/studier/emner/matnat/ifi/INF3380/v11/undervisningsmateriale/inf3380-week06.pdf.
[46] Jin, H., et al., *High performance computing using MPI and OpenMP on multi-core parallel systems.* Parallel Computing, 2011. **37**(9): p. 562-575.
[47] Debian. *The Computer Language Benchmarks Game*. 2016 Besøkt 10.05.2016; http://benchmarksgame.alioth.debian.org/u64q/compare.php?lang=gcc&lang2=java.

[48] Rolfe, T.J., *A specimen of parallel programming: parallel merge sort implementation.* ACM Inroads, 2010. **1**(4): p. 72-79.
[49] malwarebytes.org. *Levels of abstraction in programming languages*. 2012 Besøkt 10.05.2016; https://blog.malwarebytes.org/wp-content/uploads/2012/09/FlowDiagram2.png.
[50] Flanagan, D., *Java in a Nutshell.* O'Reilly Media, Inc., 2005: p. 85-86.
[51] Murphy, C. *Improving Application Quality Using Test-Driven Development (TDD)*. 2005 Besøkt 10.05.2016 [cited 2016; http://www.methodsandtools.com/archive/archive.php?id=20.
[52] Basili, V.R., et al., *Understanding the high-performance-computing community: A software engineer's perspective.* IEEE software, 2008. **25**(4): p. 29.
[53] Mamidala, A.R., et al. *MPI collectives on modern multicore clusters: Performance optimizations and communication characteristics*. in *Cluster Computing and the Grid, 2008. CCGRID'08. 8th IEEE International Symposium on*. 2008. IEEE.
[54] Carpenter, B. *THE HP JAVA PROJECT*. 2007 Besøkt 11.05.2016]; http://www.hpjava.org/mpiJava.html.
[55] Dalcin, L. *MPI for Python*. 2016 Besøkt 11.05.2016]; http://pythonhosted.org/mpi4py/.
[56] NumPy. *NumPy*. 2016 Besøkt 10.05.2016; http://www.numpy.org/.
[57] Louhivuori, M. *Python for High-performance computing*. 2016 Besøkt 13.05.2016]; https://github.com/torognes/inf9380/blob/master/python-in-hpc/lectures/mpi4py.pdf.
[58] Brooks, F., *No silver bullet*. 1987: April.
[59] Dalcín, L., et al., *MPI for Python: Performance improvements and MPI-2 extensions.* Journal of Parallel and Distributed Computing, 2008. **68**(5): p. 655-662.
[60] Flynn, M., et al. *Finding speedup in parallel processors*. in *Parallel and Distributed Computing, 2008. ISPDC'08. International Symposium on*. 2008. IEEE.