EN STUDIE AV MESH-TRAVERSERINGSMØNSTRE FOR SOPHUS-BIBLIOTEKET OG KJØRETIDSVARIASJONER PÅ SGI ORIGIN

HOVEDFAGSOPPGAVE I INFORMATIKK

AV
HOGNE HUNDVEBAKKE

2002

HOVEDFAGSOPPGAVE
I
INFORMATIKK:

„EN STUDIE AV MESH-TRAVERSERINGSMØNSTRE FOR SOPHUS-BIBLIOTEKET OG KJØRETIDSVARIASJONER PÅ SGI ORIGIN”

AV
HOGNE HUNDVEBAKKE

INSTITUTT FOR INFORMATIKK
UNIVERSITETET I BERGEN

OKTOBER 2002

INNHOLD

0. FORORD

Jeg vil gjerne takke veilederen min, førsteamanuensis ved Institutt for Informatikk, Universitetet i Bergen, Magne Haveraaen for hjelp og råd med oppgaven min og de veiledningstimene han har satt av. Året etter jeg begynte, startet han en serie Sophus-møter, slik at jeg og de andre som skulle jobbe med programvarebiblioteket Sophus skulle få kjennskap til dette. Disse møtene utviklet seg til et forum hvor resultater ble presentert og drøftet, også fra andre deler av SAGA-prosjektet, som Sophus er en del av. Jeg fikk her ideer og kommentarer under veis i arbeidet, særlig fra veileder og Krister Åhlander (under tiden post. doc.), men også fra professor ved Institutt for Informatikk, Universitetet i Bergen, Hans Munthe-Kaas, Sigurd Raubotn, Anders Abrahamsen, Otto Skrove Bagge og Ann-Kristin Åmo (de siste fire hovedfagsstudenter da jeg holdt på med hovedfagsoppgaven).

DEL 1

INNLEDNING OG BAKGRUNN

1. INNLEDNING

Hovedfagsoppgaven min er underlagt SAGA-prosjektet [w1]. Initiativtakerne til dette prosjektet var min veileder Magne Haveraaen, Hans Munthe-Kaas og Victor Madsen ved Institutt for Informatikk ved Universitetet i Bergen. Prosjektet introduserer objekt-orientert tankegang i numerisk programvare. Dette var på den tiden (begynnelsen av 90-tallet) relativt nytt innen denne typen programvare som var mye preget av prosedyre-orientert programmering, ofte utviklet i Fortran.

Kjernen i SAGA-prosjektet er det objekt-orienterte programvarebiblioteket Sophus [1] som er utviklet i C++ [2]. Mesh-klassene er lagringsklasser i dette biblioteket, og inneholder en endimensjonal datatabell med dataelementer og blir brukt mye i de seismiske simuleringsprogrammene som Sophus-biblioteket bl.a. er utviklet med tanke på. Oppgaven min går ut på å studere og endre traverseringsrekkefølgen til datatabellen for den *sekvensielle* versjonen av mesh-strukturen i dette biblioteket og se på hvordan minnebruk og kjøretid blir påvirket av dette. I empiriske studier ser jeg på enkeltmetoder i meshen, og for å se på et litt større program som benytter mesh-klassene i biblioteket, har jeg valgt ut Seismod Iso med et relativt lite datasett. Seismod Iso er én av seks seismiske simuleringsprogrammer som tar utgangspunkt i forskjellige egenskaper ved berggrunnen. Iso-versjonen er simulering av bølgene når steinene er „isotropiske" (lik i alle retninger).

Automatisk optimalisering av minnebruken ved kompilatoropsjoner vil jeg ikke gå nærmere inn på, bortsett fra noe i appendiks III, men konsentrere meg om selve traverseringsrekkefølgen.

Slik oppgaven utviklet seg, ble også ytelsesmålinger på parallelle datamaskiner sentralt. Prosessene på en slik maskin påvirker hverandre, og selv for sekvensielle programmer er variasjonen i kjøretid fra kjøring til kjøring stor. Jeg har derfor også med et kapittel om hvordan man skal empirisk sammenligne programmer på SGI Origin, som er maskintypen hvor jeg kjører alle tester i oppgaven og Sophus er utviklet mye med tanke på. En slik undersøkelse kommer også andre som jobber med Sophus-biblioteket og andre programmer på flerbrukermaskiner av denne typen til gode. Disse statistiske undersøkelsene resulterte også i en artikkel skrevet av Magne Haveraaen og meg selv [3].

1.1 Struktur på oppgaven

Oppgaven er delt inn i 5 deler. **Del 1** begynner med denne innledningen til oppgaven, og jeg har kalt denne delen „Innledning og bakgrunn". Kapittel 2 omhandler grunnleggende forståelse av det jeg jobber med i oppgaven, og hører også til denne

delen. Kapittel 3 er også grunnleggende i den forstand at den viser hvordan vi skal foreta målinger av CPU-tid, noe som er nyttig senere i oppgaven.

Del 2 (kapittel 4) består av studier av traverseringsrekkefølgen til enkeltmetoder. Dette kapittelet gir et overblikk over det eksisterende data-traverseringsmønsteret til enkeltmetoder. Traverseres data lineært? Er det eventuelt mulig å optimalisere cache-bruken ved transformasjoner på metodenivå?

Del 3 består av kapittel 5, 6 og 7 som inneholder modifikasjoner av mesh-strukturen, eller *globale programmodifikasjoner*, også kalt *globale programtransformasjoner*, dvs. modifikasjoner på klassenivå. Disse forandrer traverseringsrekkefølgen av data-tabellen på forskjellige måter og skal være tilgjengelig i kildekoden til Sophus-biblioteket.

De to neste delen er selvforklarende; **del 5** har jeg kalt konklusjon, og **del 6** er appendiks. Referansene er plassert helt på slutten av oppgaven, etter del 6.

Kapittel 3 og tredje del er strukturert på lignende måter. De består av en innledning, en eventuell teori-del, beskrivelse av implementasjonen, eksperimentdel med beskrivelse av de kjøretids- og cachebruk-eksperimentene som ble foretatt, en resultatdel og en diskusjon. Del 2 (kapittel 4) har kapitler med navn etter hvilken metode som drøftes.

Jeg har strukturert referansene på følgende måte:

[w#] En referanse som begynner med en w og etterfølges av et tall er en referanse til en internettadresse.

[#] Et tall som refererer til skriftlig materiale i form av en artikkel, dokumentasjon eller en bok.

1.1.1 Kapitteloversikt

Under følger en mer detaljert kapittelgjennomgang.

Kapittel 2 inneholder et sammendrag av mesh-klassene og en innføring i minnehierarkiet på parallelle datamaskiner. Jeg vil også utlede en egenskap ved lagringsstrukturen mesh. **Kapittel 3** omhandler de statistiske undersøkelsene av kjøretid og hvordan man skal sammenligne sekvensielle programmer på parallelle datamaskiner med minst mulig feil, anvendt på SGI Origin maskinen. I **kapittel 4** har jeg en gjennomgang av metodene i de to mesh-klassene jeg studerer i oppgaven, MeshCont og MeshScalarField, og hvordan disse traverserer mesh-data.

I **kapittel 5** studerer jeg to modifikasjoner av mesh, som har det til felles at de ikke er ment å gi en bedre cache-bruk, men heller studere et „verste tilfelle" og et antatt nøytralt tilfelle. Den ene traverserer data omvendt leksikografisk, og den andre traverserer data i meshen randomisert. Mesh-strukturen traverserer opprinnelig data leksikografisk, dvs. fra første til siste leksikografiske verdi, som er en indekseringsmetode som jeg vil komme tilbake til i neste kapittel.

Kapittel 6 tar for seg en modifikasjon av koden, Alternerende Traverseringsrekkefølge, hvor data traverseres leksikografisk ved ett metodekall og omvendt leksikografisk (studert i kapittel 5) i neste.

Kapittel 7 har jeg kalt Virtuelt Shift, og her studerer jeg hvordan programkjøringen blir påvirket av at traverseringsrekkefølgen blir endret, én metode blir mye raskere, og andre blir en del senere. **Kapittel 8** er konklusjonen.

Jeg har tre **appendiks**. **Appendiks I** er en beskrivelse av kompilering og kjøring av jobber, og er kalt Eksperimentoppsett. Referer til dette kapittelet hvis du er usikker på hvordan et eksperiment blir utført. **Appendiks II** er spesifikasjon og programkode til en

teller brukt i kapittel 7, og **appendiks III** inneholder eksperimenter gjort i samarbeid med Ann-Kristin Åmo, som faller utenfor det som var intensjonen med oppgaven, og resultater av disse.

1.2 Relatert arbeid

Mye av arbeidet innen minne-optimalisering går på optimalisering av nøstede løkker, som påpekt i [6] fra 2001. [6] gir også et godt overblikk over slike optimaliseringsteknikker.

Minimalisering av kommunikasjonskostnad ved aksess av data (parallelle dataprogrammer) og hvordan matriser skal være „aligned" i forhold til hverandre er også blitt forsket en god del på. [4] gir en beskrivelse av noen vanlige systematiske feil-„alignments" og hvordan disse kan løses. Et annet eksempel på optimalisering av minnebruken og kommunikasjonskostnaden for parallelle programmer er [5]. Her omhandles alternerende bruk av kolonne- og rad-aksess av data.

Jeg fant ikke mye arbeid når det gjelder globale program-transformasjoner (seksjon 3 i oppgaven) for sekvensielle programmer, men [6] fra 2001 studerer hvordan man kan legge inn tomme data-elementer i data-tabellen for å unngå at data mapper til samme sted i cache.

Når det gjelder sammenligning av dataprogrammer (kapittel 3), referer jeg til [3] for utfyllende informasjon og referanser.

2. SOPHUS, MESH OG MINNEHIERARKI

2.1 Innledning

Jeg vil i dette kapittelet gi bakgrunnskunnskap for forståelse av oppgaven. Jeg vil ha en kort beskrivelse av SAGA-prosjektet med vekt på Sophus-biblioteket og mesh-klassene. Datamaskinen jeg kjører tester på, en SGI Origin 3800 maskin, beskrives med vekt på minnehierarkiet.

2.2 SAGA

SAGA-prosjektet er et programutviklingsprosjekt startet rundt ideen om abstraksjoner som utgangspunkt for utvikling av numerisk programvare. Man utnytter algebraiske og formelle/matematiske programutviklingsmetoder [7] for å lage modeller, f.eks. for fysiske problemer [w2]. Disse modellene uttrykkes ved partielle differensiallikninger (PDE'er) [8]. Sophus biblioteket, som inneholder meshen jeg har studert i min oppgave, er den sentrale delen av SAGA-prosjektet. Safir og CodeBoost, som jeg ikke vil gå så nøye inn på, er andre deler av det samme prosjektet. For mer utfyllende informasjon om SAGA-prosjektet, henviser jeg til [w1].

2.2.1 Safir

Den matematiske beskrivelsen av PDE'er defineres ofte ved rekursjon. Å kode rekursive matematiske formuleringer ved hjelp av en rekursiv funksjon fører ofte til stor (eksponentiell) tids- og minne-kompleksitet i forhold til en ikke-rekursiv kode som utfører samme oppgave. Et klassisk eksempel er fibonacci tallene. Safir er et verktøy for å omforme rekursive funksjoner til effektiv ikke-rekursiv kode.

2.2.2 CodeBoost

For å gjøre kode med matematiske formler mer leselig, sammenfatter man formler isteden for å skrive dem på en ekspandert form i Sophus-biblioteket. Problemet er at den sammenfattede formen er mer tidkrevende – kompilatoren optimaliserer ikke alltid koden godt nok og oppretter flere temporære variabler. CodeBoost er et verktøy skrevet med tanke på effektivisering av kode skrevet i Sophus-biblioteket ved å omforme sammenfattet form til en ekspandert, som illustrert i figur 2.1. Ved bruk av dette verktøyet kan man oppnå mer enn 30% ytelsesforbedring.

CodeBoost har også annen funksjonalitet. Basert på layouten til data, kan den optimalisere enkelte metoder i Sophus-biblioteket, for eksempel elementvise metoder som addisjon eller mapping av data til andre verdier vha. en funksjon. Eksempel på layout kan være mesher med mange 0-elemtenter og diagonale mønstre.

CodeBoost var fortsatt under utvikling av andre studenter mens jeg holdt på med hovedfagsoppgaven.

Figur 2.1 Codeboost-optimalisering av en formel $f \in (R \to R)$.

Matematisk formel	Kode	Ekspandert form vha. CodeBoost

$f = \frac{a \cdot (b+c)}{d}$	double f = (a*(b+c))/d;	Double temp1 = b;
		temp1 += c;
		double f = a;
		f *= temp1;
		f /= d;

2.3 Sophus

Sophus-biblioteket er som sagt bygget opp ved hjelp av algebraiske utviklingsmetoder og numeriske metoder for at brukeren skal kunne produsere en datamodell for et problem ved hjelp av partielle differensiallikninger. For å implementere disse abstraksjonene, har man valgt det objekt-orienterte programmeringsspråket C++. Ofte har numerikere brukt Fortran til å utvikle numeriske biblioteker, men for å programmere abstraksjoner, er et objekt-orientert programmeringsverktøy nyttig.

Sophus-biblioteket tilfredsstiller kravene til et objekt-orientert programvarebibliotek. Det benytter seg dog bare en gang av C++-arv (i mesh-strukturen), så selve koden kan sies å være mer objekt-*basert*. På den andre siden er ikke dokumentasjonen flat, men bygget opp med arv. I en modul er de metodene som arves i dokumentasjonen skrevet om igjen. Koden er delt inn seksjoner, hvor man opplyser hva som er arvet og hva som er spesielt for modulen. *Konseptet* arv er derfor mye brukt.

Det er høye krav til dokumentasjon og utforming av komponenter som skal legges inn i biblioteket. For å kunne legge inn en modul, stilles det krav fra C++, til at utformingen av klassen følger "Sophus standarden" og til at spesifikasjonen er oppfylt. Dokumentene og koden skal valideres av ansvarlige for og medarbeidere ved utviklingen av programvarebiblioteket.

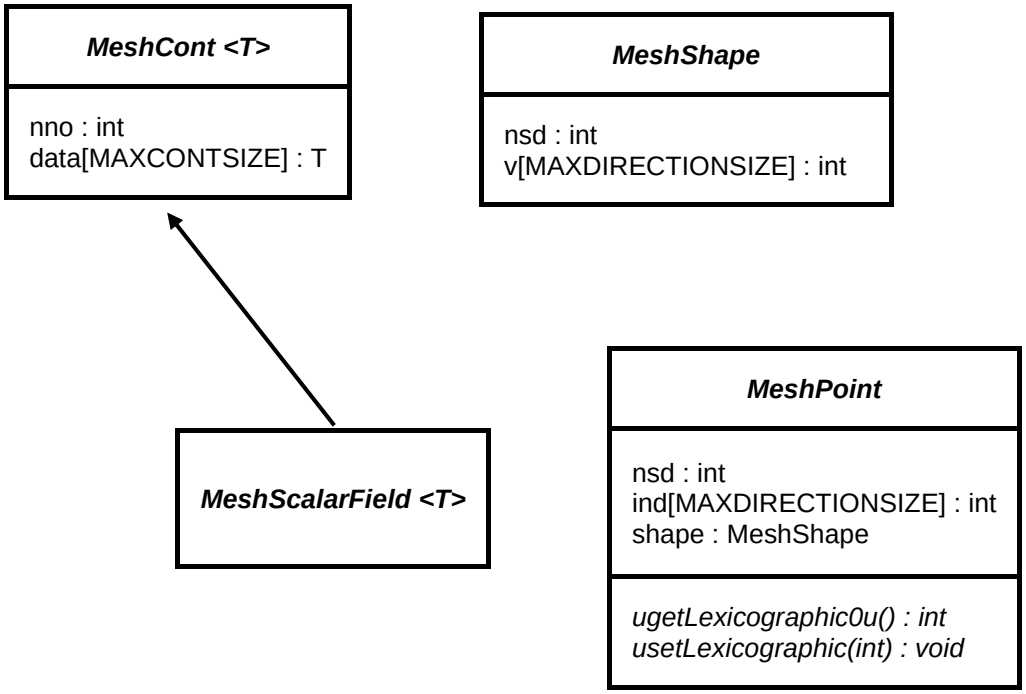
Koden er i utgangspunktet nærmest mulig de matematiske abstraksjonene den bygger på (CodeBoost gjør optimaliseringer) og implementasjonen skal være statisk for å unngå pekerproblematikk. Man skal også bare inkludere (bruke #include-direktivet for å inkludere) ett bibliotek, nemlig *SophusDebug*, i klassene i Sophus biblioteket – man inkluderer de nødvendige klassene i program som bruker Sophus-biblioteket. Videre har mange metoder datainvarianter, forkrav og etterkrav, som kan slåes av og på av bruken.

Man har prøvd å bygge opp et bibliotek hvor brukeren kan få ønsket funksjonalitet ved å bytte ut enkeltmoduler. Et eksempel er en versjon av mesh for sekvensiell og en for parallell kjøring.

En annen karakteristikk med klassene i Sophus, er at de benytter seg mye av template-klasser. For mesh-klassene betyr dette at de kan lagre flere typer objekter.

For mer informasjon om Sophus-biblioteket, henviser jeg til dokumentasjonen [1].

Figur 2.2 Klassestrukturen til mesh-abstraksjonen med utvalgte medlemsvariable og funksjoner.



Figuren er laget med UML-notasjon [9]. MAXCONTSIZE og MAXDIRECTIONSIZE er brukerdefinerte konstanter. *nno* angir antall mesh-elementer. *nsd* angir antall retninger i MeshShape og antall dimensjoner i MeshPoint. tabellen *v* i MeshShape inneholder retningene i MeshShape, og tabellen *ind* dimensjonene i MeshPoint.

2.4 Mesh

Mesh-klassene er lagringsklasser i Sophus-biblioteket [1]. I dokumentasjonen arver klassene i mesh-abstraksjonen de kartesiske abstraksjonene[1], som er en friere lagrings-struktur.

En mesh (en MeshCont eller et MeshScalarField, se figur 2.2) er en flerdimensjonal struktur, hvor hver retning (dimensjon) *i* har en størrelse k_i, som er et heltall, og hvor $0 \leq k_i$. Disse forholdene er lagret i en *MeshShape*. Volumet til en mesh er produktet

[1] side 35 i Sophus-dokumentasjonen

$k_1k_2\cdots k_d$, hvor d står for antall dimensjoner, så hvis en eller flere av retningene er 0, er meshen tom. Volumet angir hvor mange objekter av typen T en mesh kan lagre.[2]

Mesh-strukturen er implementert med klassestrukturen som vist i figur 2.2, hvor jeg har tatt med noen utvalgte medlemsvariable og metoder. MeshCont og MeshScalarField har begge en data-tabell som lagrer data av typen *T*. Det er viktig å huske at data-tabeller i C++ lagrer data sekvensielt og suksessivt etter hverandre i minnet. Det er traversering av denne tabellen jeg vil studere i denne oppgaven.

I tabellen er data ordnet i leksikografisk rekkefølge etter den leksikografiske posisjonen til det MeshPoint som et gitt data-element (av typen *T*) er mappet til. Konvertering mellom en leksikografisk posisjon og MeshPoint gjøres vha. funksjonen ugetLexicographic0u og prosedyren usetLexicographic i MeshPoint. ugetLexicographic0u returnerer den leksikografiske posisjonen til et punkt, mens usetLexicographic lager et punkt ut ifra en leksikografisk posisjon.

2.4.1 Teori

For at det skal være konsistens mellom ekstern data-aksessering (dette skjer gjennom MeshPoint), og den interne representasjonen, må disse være én-til-én. I en leksikografisk ordning, vil hvert MeshPoint være mappet til et unikt nummer, så det er opplagt at dette er en gyldig måte å lagre elementene på. Nedenfor vil jeg bevise formlene for leksikografisk konvertering.

Definisjon 2.1

(1) **MeshShape.** En MeshShape S er en n-tuppel $(s_1, s_2, ..., s_n)$, hvor $0 \le s_i$ for alle $i = 1...n$.

(2) **MeshPoint.** Et MeshPoint P med en MeshShape $S = (s_1, s_2, ..., s_n)$ er en n-tuppel $(p_1, p_2, ..., p_n)$, hvor $0 \le p_i < s_i$ for alle $i = 1...n$, og p_i er indeks i i P. Man kan også referere til indeks i i P som *P(i)*.

Lemma 2.1

Gitt et MeshPoint $P = (p_1, p_2, ..., p_n)$ med en MeshShape $S = (s_1, s_2, ..., s_n)$.

Den leksikografiske posisjonen til *P*, lex(*P*), er gitt ved

(1) $lex(P) = \sum_{i=1}^{n} (p_i \prod_{j=i+1}^{n} s_j)$.

Videre kan vi generere indeksene i punktet med den leksikografiske posisjonen *lex(P)* fra

[2] mer om mesh-abstraksjonen på side 56 i Sophus-dokumentasjonen

(2) $p_i = \left\lfloor l_i / \prod_{i+1}^{n} s_j \right\rfloor$,

og

$$l_{i+1} = l_i - p_i \prod_{i+1}^{n} s_j \text{ og } l_1 = lex(P) .$$

Bevis

Jeg viser (1) ved konstruksjon.

ll_1=lex((p_1,0,…,0)) er gitt ved antall punkter som har en leksikografisk posisjon før punktet (1,0,...,0) multiplisert med p_1, så

$$ll_1 = p_1 \prod_{j=2}^{n} s_j .$$

Videre blir antall punkter før (p_1, p_2,0,…,0)

$$ll_2 = ll_1 + p_2 \prod_{j=3}^{n} s_j .$$

Generelt får vi

$$ll_i = ll_{i-1} + p_i \prod_{j=i+1}^{n} s_j ,$$

og

$$ll_n = \text{lex}(P) = \sum_{i=1}^{n} (p_i \prod_{j=i+1}^{n} s_j) = (1).$$

Den leksikografiske ordningen har egenskapen $0 \le lex(P) < vol(S)$, hvor vol*(S)* er volumet til MeshShape til *P*. Den nedre grensen følger av at $P = (0,\ldots,0)$ gir den minste verdien til *lex(P)* siden *P* ikke kan ha negative komponenter og (1) er satt sammen av sum av produkter. $P = (s_1 - 1, s_2 - 1, \ldots, s_n - 1)$, som er punktet hvor komponentene har størst mulig verdi, er punktet som gir (1) høyest verdi. Den leksikografiske posisjonen til dette punktet blir (fra (1)):

$$\sum_{i=1}^{n} ((s_i - 1) \prod_{j=i+1}^{n} s_j) = \sum_{i=1}^{n} (s_i \prod_{j=i+1}^{n} s_j - \prod_{j=i+1}^{n} s_j) \text{-1=}$$

$$\sum_{i=1}^{n} (\prod_{j=i}^{n} s_j) - \sum_{i=1}^{n} (\prod_{j=i+1}^{n} s_j) \text{-1=} \prod_{j=1}^{n} s_j \text{ -1 =}$$

vol (S)-1.

For å generere et MeshPoint, *P* fra den leksikografiske posisjonen, gjør vi det „motsatte" av konstruksjonen av (1).

La *i=1*, slik at $l_i = lex(P)$. For at høyre side i (2) virkelig skal gi oss p_1, så må

$$p_i \prod_{i+1}^{n} s_j \le l_i < (p_i + 1) \prod_{i+1}^{n} s_j$$

være oppfylt.

Fra genereringen av den leksikografiske ordningen i bevis av (1), er $p_i \prod_{i+1}^{n} s_j \le l_i$ oppfylt, og $p_i \prod_{i+1}^{n} s_j = l_i$ hvis $P = (p_i, 0, \ldots, 0)$. Det største bidraget til den leksikografiske verdien de resterende komponentene i *P* kan gi er hvis $P = (p_i, s_{i+1} - 1, ..., s_N - 1)$. Vi må vise at dette bidraget er mindre enn $1 \cdot \prod_{i+1}^{n} s_j$, slik at høyre siden i uttrykket i (1) blir oppfylt. Det er det samme som å vise at

$$\sum_{i}^{n} (s_j - 1) \prod_{i+1}^{n} s_j < \prod_{i}^{n} s_j .$$

Hvis vi skriver ut venstresiden, får vi:

$$\begin{aligned}
&(s_i - 1) \cdot s_{i+1} \cdots s_n + \cdots + (s_n - 1) = \\
&s_i \cdots s_n + \cdots + s_n - (s_{i+1} \cdots s_n + \cdots + 1) = \\
&s_i \cdots s_n - 1 = \\
&\prod_{i}^{n} s_j - 1
\end{aligned}$$

Dermed har vi bevist at $p_i = \left\lfloor l_i / \prod_{i+1}^{n} s_j \right\rfloor$ for *i=1*.

For videre å finne p_2, trekker vi fra bidraget til den leksikografiske posisjonsangivelsen, generelt

$$l_{i+1} = l_i - p_i \prod_{i+1}^{n} s_j$$

(vi trekker fra det samme som vi i forrige avsnitt la til). Dette gir den leksikografiske posisjonen til $(p_{i+1}, 0, ..., 0)$, siden l_{i+1} er bidraget til den leksikografiske posisjonsangivelsen fra dette punktet (konstruksjonen av (1)). Slik kan vi fortsette for alle komponentene (for alle *i* opp til *n*), og få tilbake de komponentene som vi i forrige seksjon brukte til å generere den leksikografiske ordningen til et punkt. □

Jeg har nå utledet og kommet frem til de samme formlene for leksikografisk ordning som i dokumentasjonen.

2.5 SGI Origin

Ask (ask.ii.uib.no) var parallelldatamaskinen ved Para//ab [w3] på Universitet i Bergen, som jeg har brukt noe til kjøring av tester. Denne maskinen ble utfaset i løpet av oppgaven, så testene i oppgaven ble kjørt om igjen på Gridur [w4], stasjoner på Norges teknisk-naturvitenskaplige universitet. Ask er en Origin 2000 maskin fra Silicon Graphics® [w5], mens Gridur (og Embla som også stod på NTNU) er Origin 3800 maskiner fra samme leverandør.

2.5.1 Minnehierarkiet på Gridur og Embla

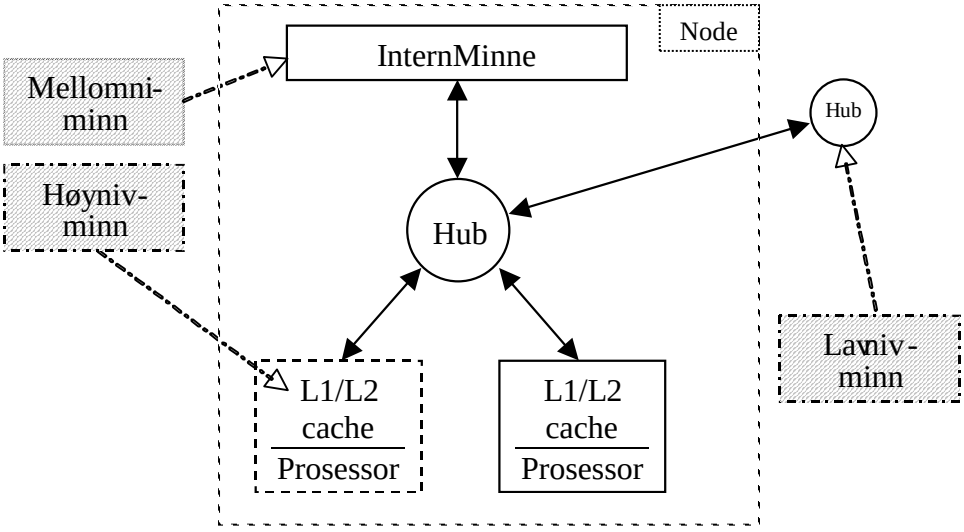
Origin 3800 maskinene er bygget opp av noder i en hyperkubestruktur; oppbyggingen er beskrevet i [w6]. En node er illustrert i figur 2.3. En node består som man ser av en hub som binder sammen internminnet og de to prosessorene. Figuren illustrerer også hvordan minne kan aksesseres via huben på en annen node.

Minnet på mellomnivå er internminnet. Gridur har 384 MIPS R14000 prosessorer og totalt 304 GB minne, mens Embla har 512 MIPS R14000 prosessorer og totalt 512 GB minne. Det laveste minnenivået er harddisken, som ikke er tatt med i figur 2.3.

2.5.1.1 Minnehierarkiet på MIPS R14000 prosessoren

Cache-minne generelt er beskrevet i [10, seksjon 4.3], og arkitekturen til R14000

Figur 2.3 Figuren viser kommunikasjon mellom de 2 prosessorene på en node (i stiplet boks) og minne på forskjellige nivå i minnehierarkiet.



Hvite piler er forklaringspiler. Svarte piler med to hoder viser hvor data går til eller fra.

spesifikasjoner ble hentet fra [11].

Denne prosessoren, som brukes på Gridur, har tilgang på to hurtigminner (cache) for *data* (man har også noe som heter *instruksjons*cache, men det vil jeg ikke omhandle her). Nivå 1 hurtigminne (også kalt nivå 1 cache eller L1 cache) sitter på prosessoren og er på 32 Kb, mens nivå 2 hurtigminne (nivå 2 eller L2 cache), som er noe tregere, er på 2, 4 eller 8 MB på SGI Origin 3800. Hvis vi ser bort ifra registeret, er disse to minnene de hurtigste på prosessoren, og kalles prosessorens hurtigminne eller cache. Hurtigminnet er viktig for at prosessoren ikke skal vente for lenge på informasjon den trenger.

Minne som brukes av prosessoren og ikke er i cache, lastes inn i cache i blokker. En slik blokk kalles en cache-linje. Dette er basert på tanken om at området rundt en minneadresse som aksesseres, vil aksesseres igjen på et senere tidspunkt (f.eks. ved traverserings av en tabell, hvor elementene lagres etter hverandre i minnet). Hvis minneområdet finnes i cache, vil det kunne leses hurtig. Å aksessere (hente data inn i registeret slik at det kan behandles av prosessoren) L1 cache tar 3-4 klokkesykler, L2 cache tar ca. 10-12 klokkesykler, mens internminnet tar mer enn 70 klokkesykler å aksessere.

Minnet kan tenkes på som inndelt i blokker på størrelse med en cache-linje. L1 har cache-linjer på 32 bytes, mens L2 har cache-linjer på 128 bytes. Hver minneblokk mapper til en cache-linje. Hvis en minneadresse som ikke er i cache aksesseres, lastes en minneblokk på størrelse med en cache-linje som inneholder denne adressen inn i L1- og L2-cache. Blokken bestemmes av den delen av minneadressen som gjenstår etter at den nederste delen av minneadressen, tilsvarende én cache-linje, er tatt bort. For en cache-linje på 32 bytes, vil de 8 siste sifrene i bit-adressen bli tatt bort. Hvor minneadressen plasseres i minne, bestemmes av

mod(minneadresse,cache-sett størrelse).

L1 og L2 cache er 2-retningers sett-assosiativ write-back cache. 2-retningers sett-assosiativ betyr at cachen er delt inn i to (like store) sett. Cachesett-størrelsen i formelen over er 16 Kbytes for L1 cache. Grunnen til at cache er delt inn i to sett, er at to forskjellige blokker i minnet som mapper til samme sted i cache fra formelen kan plasseres i hvert sitt sett. En direkte mapping, som vi finner på eldre prosessorer, bruker bare ett sett. At cachen er write-back, betyr at modifiseringer gjort på en minneadresse og skrevet til L1-cache, ikke skrives til L2-cache eller andre deler av minnehierarkiet før dette er nødvendig (f.eks. hvis cache-linjen går ut av L1-cache).

R14000 prosessoren bruker LRU (least recently used) prinsippet når den skal velge hvilket av de to settene data skal plasseres i. Den cache-linjen som sist ble aksessert blir uendret, mens den tilsvarende cache-linjen i det andre settet blir skrevet til når data hentes inn i cache.

2.5.2 IRIX

Operativsystemet som kjøres på Ask, Gridur og Embla er IRIX®, som har en del funksjonalitet for å optimalisere minnebruk [w6]. Bl.a. prøver IRIX å plassere prosesser og minne nærmest mulig. For å oppnå dette, plasseres prosesser på noder hvor mye minne er tilgjengelig på og i nærheten . Når prosessen er satt i gang, allokeres minne nærmest mulig.

IRIX har også virtuelle minneadresser. Dette gjør det mulig å benytte seg av „dynamic page migration". Programmereren jobber på samme virtuelle minneadresse, men den virkelige minneadressen (og den assosierte minnesiden) kan flyttes nærmere prosessen hvis den aksesseres ofte.

Funksjonaliteten som jeg har nevnt kan overstyres og slåes av og på av brukeren.

3. SAMMENLIGNING AV KJØRETIDEN TIL DATAPROGRAMMER

3.1 Innledning

Jeg vil i dette kapittelet gå inn på hvordan kjøretid bør måles på parallelldatamaskiner med flere brukere, representert ved en SGI Origin maskin. Grunnen til at denne problemstillingen dukker opp, er at kjøretiden på parallelldatamaskinene jeg har kjørt tester på under arbeidet med hovedfagsoppgaven varierer vesentlig fra kjøring til kjøring. En sammenligning av kjøretiden til to programmer vil derfor kunne gi forskjellig resultat fra gang til gang. Jeg vil bruke resultatene fra dette kapittelet når jeg skal sammenligne programmer senere.

Én måte å løse problemet med varierende kjøretider på, er å kjøre programmene vi ønsker å sammenligne på en datamaskin som er sperret for andre brukere. Dette var ikke praktisk mulig for oss, og er ofte ikke mulig å gjøre. Vi må altså måle hvor stor innflytelse andre brukere har på jobben (programmet) jeg kjører på maskinen. Et enkelt, dog ikke helt presist mål for dette er last, som er antall jobber i jobbkøen på maskinen.

Det jeg i korthet vil finne ut, er om jeg kan redusere usikkerheten til kjøretidsmålingene ved å kjøre flere eksperimenter (kjøre programmet flere ganger) og anvende et mål (gjennomsnitt, median eller liknende) på målingene, og hvor mange målinger jeg i så fall må ta for å oppnå en gitt usikkerhet.

3.2 Teori

For et deterministisk program, som de programmene jeg bruker i dette kapittelet, er det er tre faktorer som har innvirkning på kjøretiden, hvis vi ser bort ifra egenskaper til programmet selv.

1 Operativsystemet (IRIX på de datamaskinene jeg har jobbet på).

2 Maskintilstand.

3 Maskinarkitektur.

Punkt 2 og 3 påvirker hvordan operativsystemet eksekverer programmet på datamaskinen. Operativsystemet IRIX og maskinarkitekturen er beskrevet i kapittel 2.

3.2.1 Maskintilstand

I dette begrepet ligger prosessene og minnebruken på maskinen. Denne tilstanden preger i stor grad hvordan IRIX vil legge ut data og prosesser på datamaskinen og hvor mye data blir flyttet rundt på maskinen. Jeg bruker i dette kapittelet last som indikator på hvilken tilstand maskinen er i. Høy last er en indikator på at det er mangel på

minneressurser og at det kan være vanskelig å plassere prosesser og minne i nærheten av hverandre.

3.3 Eksperiment

Versjon 1.17 av Sophus-biblioteket ble brukt i eksperimentet. 1400 suksessive kjøringer av Seismod Iso, som er et deterministisk program for seismisk simulering, ble kjørt på Gridur med et lite data-sett med ca. 90.000 flyttall av min veileder Magne Haveraaen. Gjennomsnittlig last siste 5 minutter ble lagret etter hver kjøring sammen med eksekveringstid og CPU-tid for kjøringen (målt med /bin/time). Gjennomsnittlig last siste 5 minutter ble valgt fordi CPU-tiden til programmet var omtrent 5 minutter.

Mellom hver kjøring var det en tilfeldig pause på mellom 0 og 10 minutter, hvor pausens lengde ble bestemt av maskinklokken og var på 10*<maskinklokkens sekundteller> sekunder. Hvis maskinklokken f.eks. viste 10:04:20, ble pausen på 200 sekunder. Meningen med pausene var å eliminere systematiske innflytelser på kjøretiden som følge av andre programmer som kjørte på maskinen og påvirkninger fra siste kjøring.

På samme maskin ble det kjørt 1226 kjøringer av et testprogram som tester shift-metoden i MeshCont på en 3x5 mesh av heltall og gjentatt forskyvninger med punktet (1,3). Jeg referer til dette programmet som SpeedTest. For hver kjøring ble CPU-tiden (målt med clock()-funksjonen i /sys/time.h), og gjennomsnittlig last siste minutt ble målt, siden hver kjøring tok ca. 60 sekunder.

Alt ble kompilert vha. preprosessoren Sophus C++ preprosessor SCC versjon 1.10 som brukte MIPSpro Compilers versjon 7.3.1.2m på ask.ii.uib.no. Programmene ble deretter flyttet over til Gridur for testing som beskrevet over[3].

3.4 Resultat

Én viktig fordel CPU-tiden har fremfor kjøretid, er at den ikke påvirkes av tilfeldige pauser som kjøringene kan ta hvis jobben blir utsatt (suspended) i køen. Av denne grunn kan man observere statistiske uteliggere[4] blant kjøretidsmålingene, mens jeg ikke har observert tilsvarende for CPU-tiden (selv når kjøretiden var en tydelig uteligger for en gitt kjøring har jeg observert at CPU-tiden ikke har vært det for samme kjøring). Dette er årsaken til at jeg bruker CPU-tid videre når jeg skal presentere resultatene.

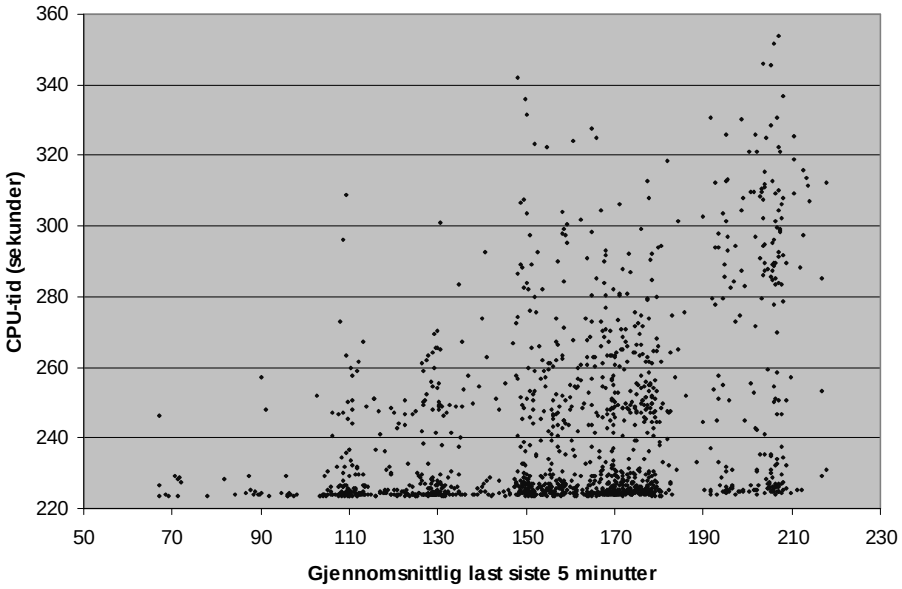
[3] Denne fremgangsmåten med å kompilere på Ask, og flytte programmene over til Gridur for kjøring blir senere i oppgaven byttet ut med å både kompilere og kjøre på Gridur.

[4] En uteligger er en observasjon i datasettet som tydelig skiller seg fra de andre observasjonene. Dette antaes i dette tilfellet å skyldes at jobber utsettes (tar en pause) i jobbkøen.

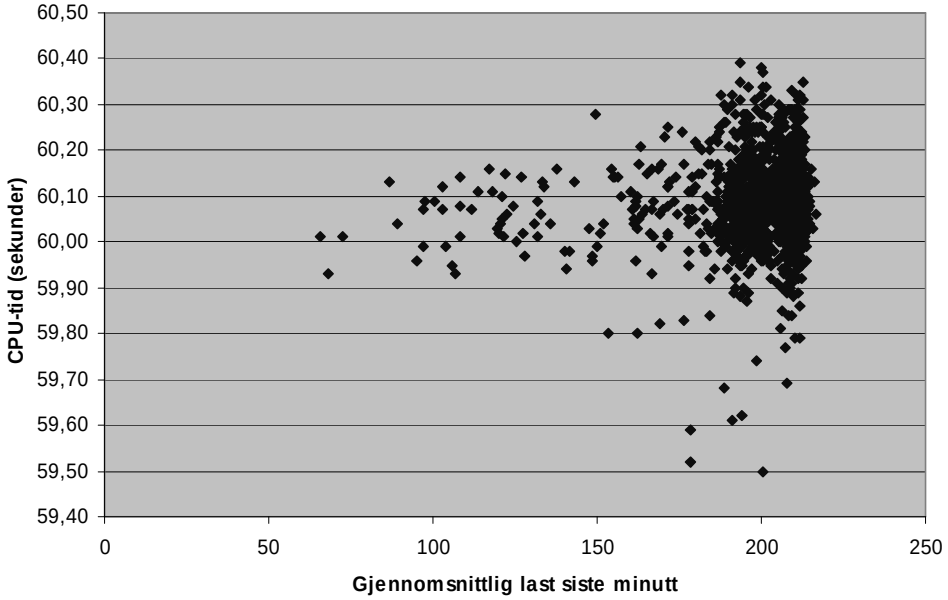
3.4.1 Sprede-plott

På neste side har jeg plottet to spredeplott for begge eksperimentene.

Figur 3.1a Sprede-plott som viser CPU-tid mot gjennomsnittlig last siste 5 minutter etter målingen for Seismod Iso-målingene



Figur 3.1b Sprede-plott som viser CPU-tid mot gjennomsnittlig last siste minutt for shift-metoden (SpeedTest)



3.4.2 Egenskaper til tallmaterialene

Tabell 3.1 Egenskaper til tallmaterialene

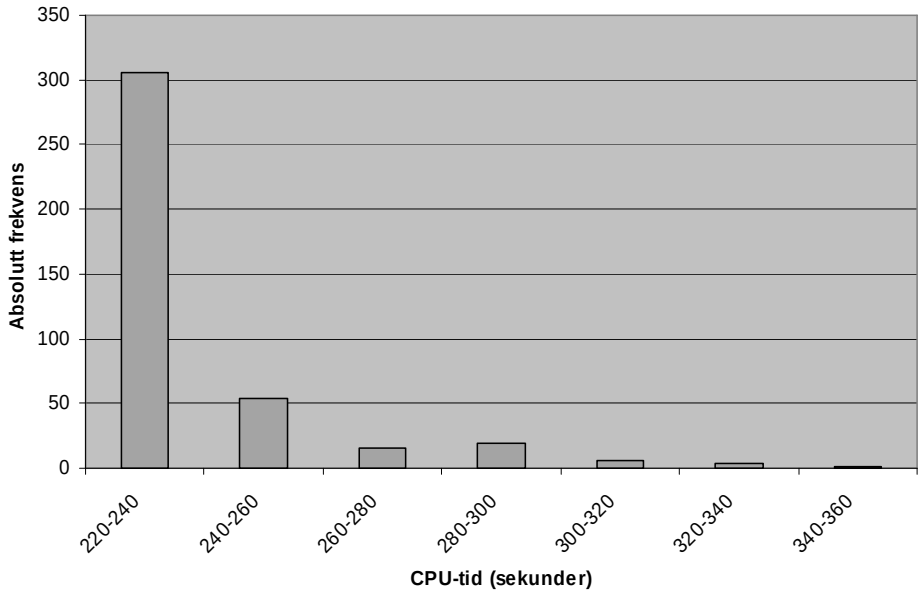
Testprogram	Min. CPU-tid (sekunder)	Min. last	Maks. CPU-tid (sekunder)	Maks. last
Seismod Iso	223,561	67,02	353,955	217,94
SpeedTest	59,5	65,69	60,39	216,82

3.4.3 Histogrammer

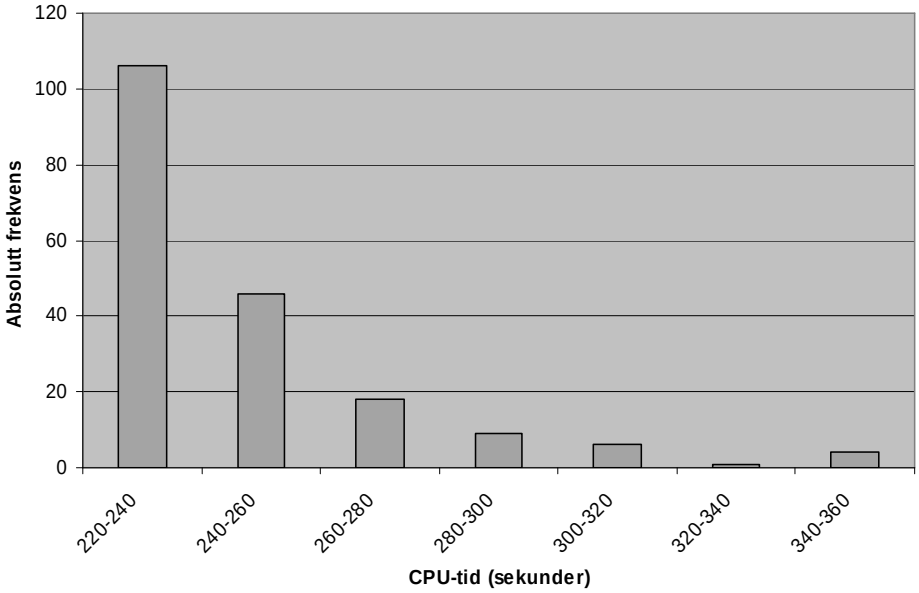
I histogrammene nedenfor har jeg delt tallmaterialet inn etter last og ser på hvordan kjøretiden fordeler seg i lastgruppene. Alle dataene i hver lastgruppe er med for alle histogrammene. For Seismod Iso har jeg delt inn i de tre lastgruppene

- lav last: [0,160>
- høy last: [160,190>
- meget høy last: [190,∞>

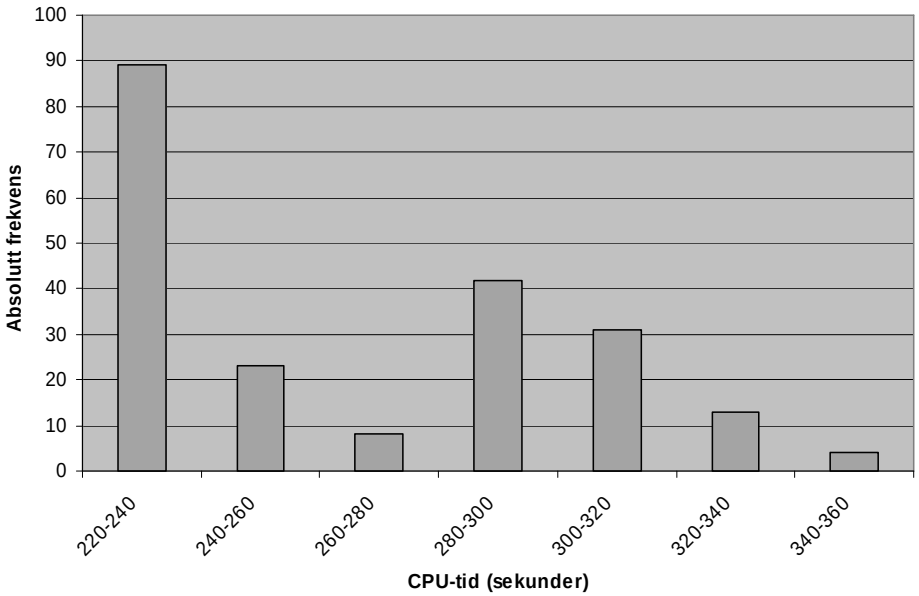
Figur 3.2a Histogram for lav last situasjon for Seismod Iso



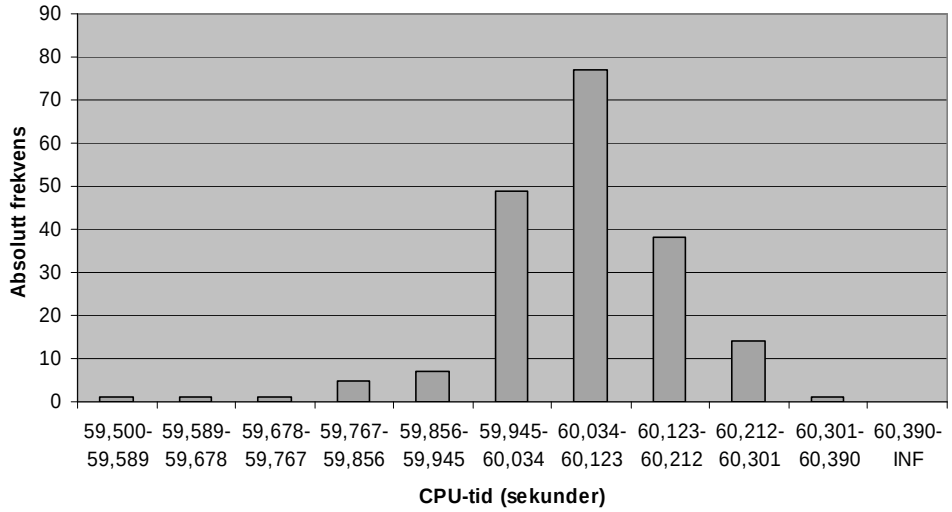
Figur 3.2b Histogram for høy last situasjon for Seismod Iso



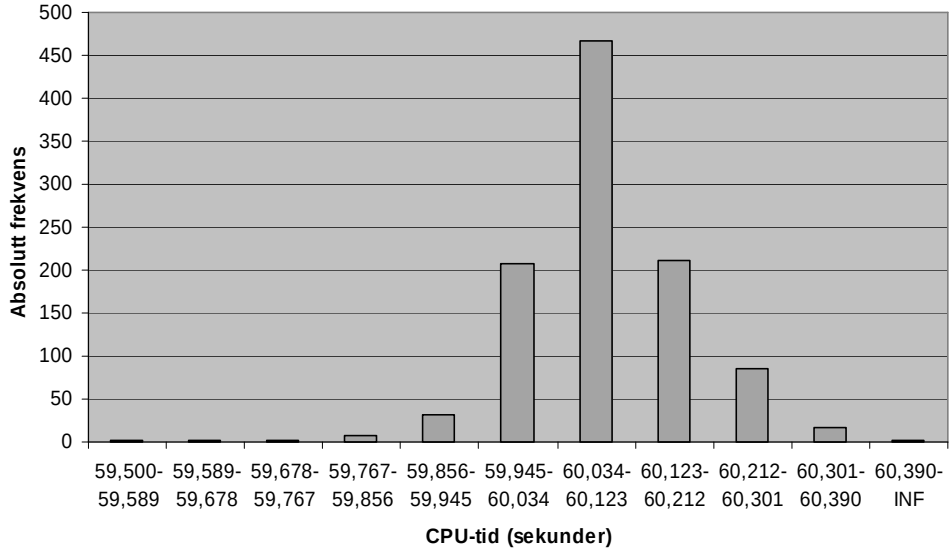
Figur 3.2c Histogram for meget høy last for Seismod Iso



Figur 3.2d Histogram for SpeedTest for høy last [0,190>



Figur 3.2e Histogram for SpeedTest for meget høy last [190, ∞>



3.4.4 Studie av målene

Jeg vil videre studere de fem målene gjennomsnitt, median, nedre kvartil (q1), minimum og maksimum for Seismod Iso og gjennomsnitt, median, nedre kvartil (q1), øvre kvartil (q3) og maksimum for SpeedTest. For et datasett på *n* elementer har jeg definert median, q1 og q3 på følgende måte:

1 Median, *m*:

$m = \lfloor n/2 \rfloor$'te største element hvis *n* er odd og

$m = \frac{(n/2\text{-}1)\text{'te element} + (n/2)\text{'te element}}{2}$ 'te største element hvis *m* er jamn.

2 En kvatil, q, definerer jeg ved

$$q = e_{\lfloor i \rfloor} + (e_{\lceil i \rceil} - e_{\lfloor i \rfloor}) \cdot (i - \lfloor i \rfloor),$$

hvor e_i er *i*'te største element og

$$i = \begin{cases} 1 \text{ hvis } n < 4 \\ \frac{1}{4} \cdot (n + a) + b \text{ ellers} \end{cases} \text{ for q1 og}$$

$$i = \begin{cases} n \text{ hvis } n < 4 \\ \frac{3}{4} \cdot (n + a) + b \text{ ellers} \end{cases} \text{ for q3.}$$

Jeg bruker a=1 og b=0.

For en gitt lastgruppe behandler jeg tallmaterialet på følgende måte for å oppnå figurene nedenfor:

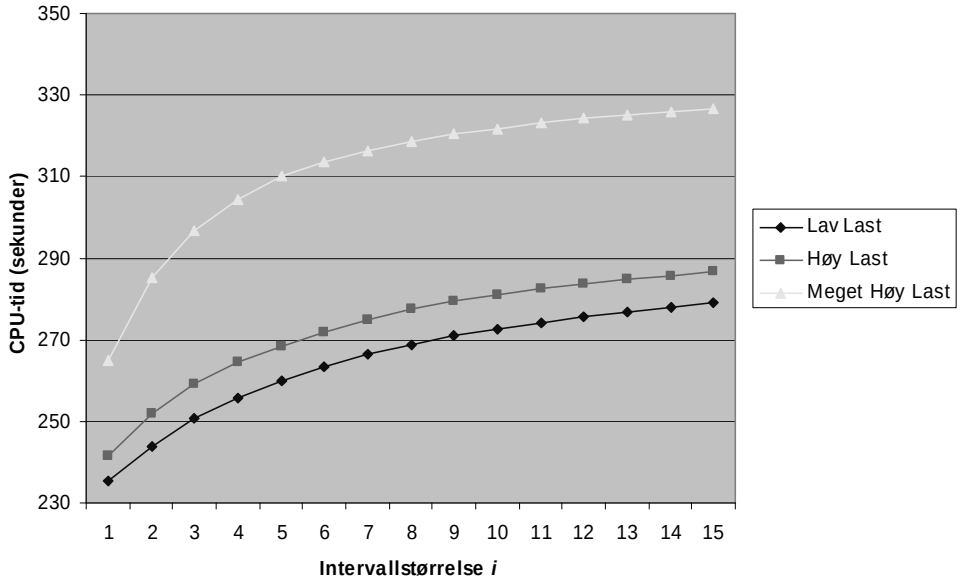
1 Plukker ut alle CPU-tider i lastgruppen. Rekkefølgen på dataene i den listen av CPU-tider som resulterer av dette utplukket var den samme som i det originale datasettet (dvs. at dataene er listet i den rekkefølge de er kjørt i).

2 For en gitt intervallstørrelse *i* plukker jeg ut elementene (kjøretidene) $k_{(x+1)\%n}, k_{(x+2)\%n}, \ldots, k_{(x+i)\%n}$ for alle *x* fra 1 til *n*, hvor *n* er antall elementer i lastgruppen jeg har plukket ut og k_i er *i*'te element (% står for modulo). Jeg får da *n* grupper, hvor hver gruppe har *i* elementer.

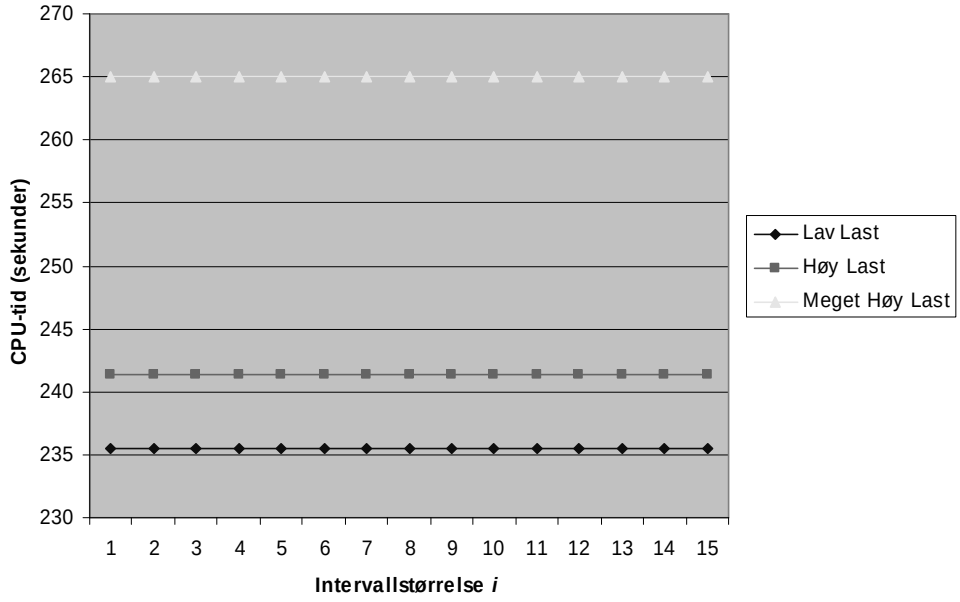
3 For hver gruppe finner jeg målet jeg er interessert i (f.eks. minimum av elementene i en gruppe). Dette resulterer i *n* nye kjøretider basert på målet. Jeg tar så gjennomsnittet av disse og finner standardavviket til dette gjennomsnittet. *Varianskoeffisient*[5] finner jeg ved å dele standardavviket på gjennomsnittet og gange med 100 for å få forholdstallet i prosent.

[5] Varianskoeffisient er en oversettelse av „coefficient of variation".

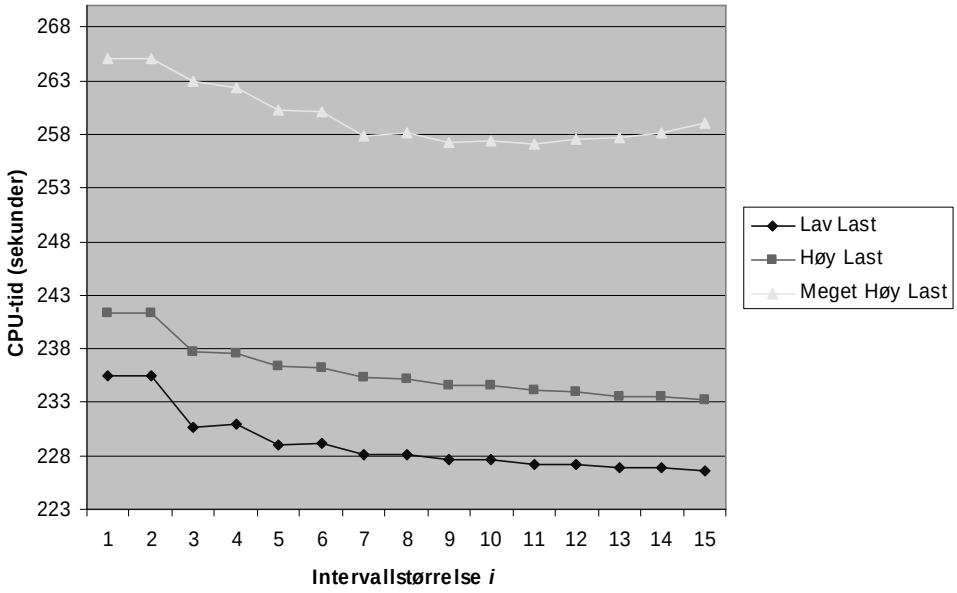
Figur 3.3a Plott av *maksimum* mot intervallstørrelse for Seismod Iso



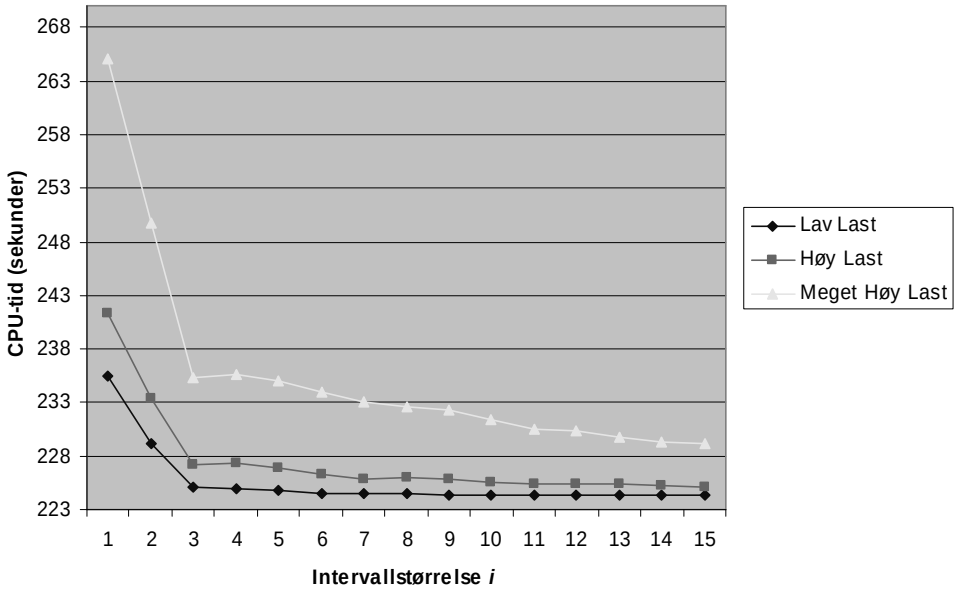
Figur 3.3b Plott av *gjennomsnitt* mot intervallstørrelse for Seismod Iso



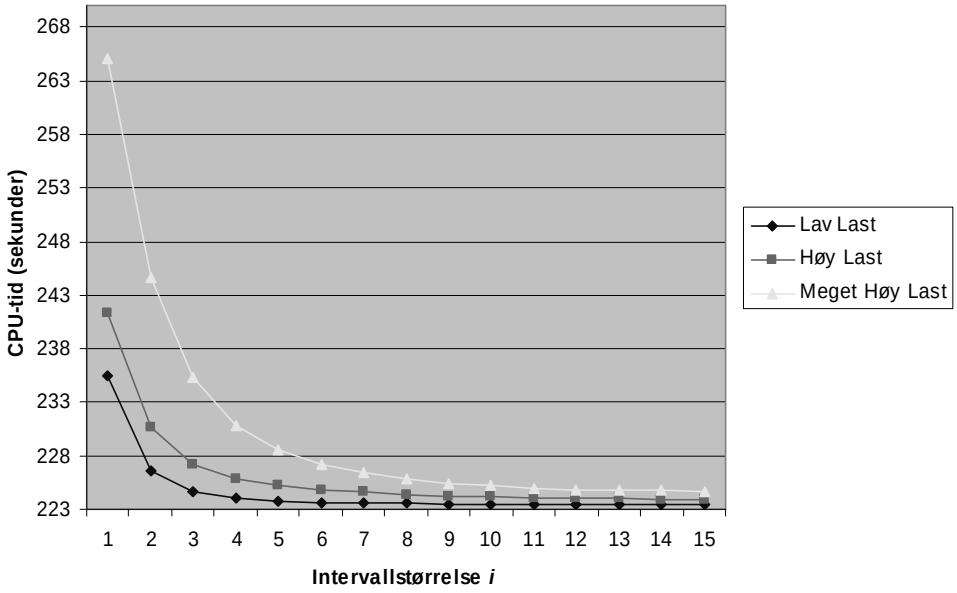
Figur 3.3c Plott av *median* mot intervallstørrelse for Seismod Iso



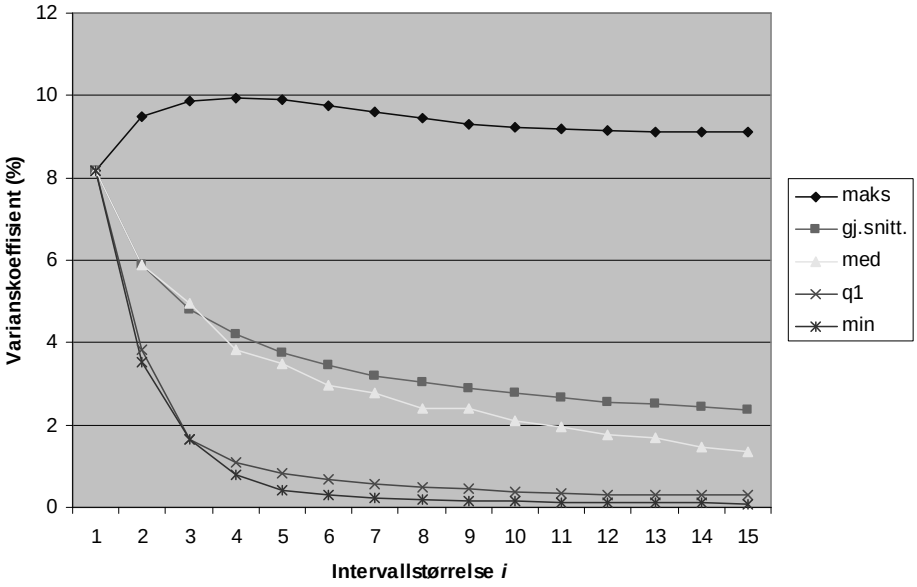
Figur 3.3d Plott av *nedre kvartil* mot intervallstørrelse for Seismod Iso



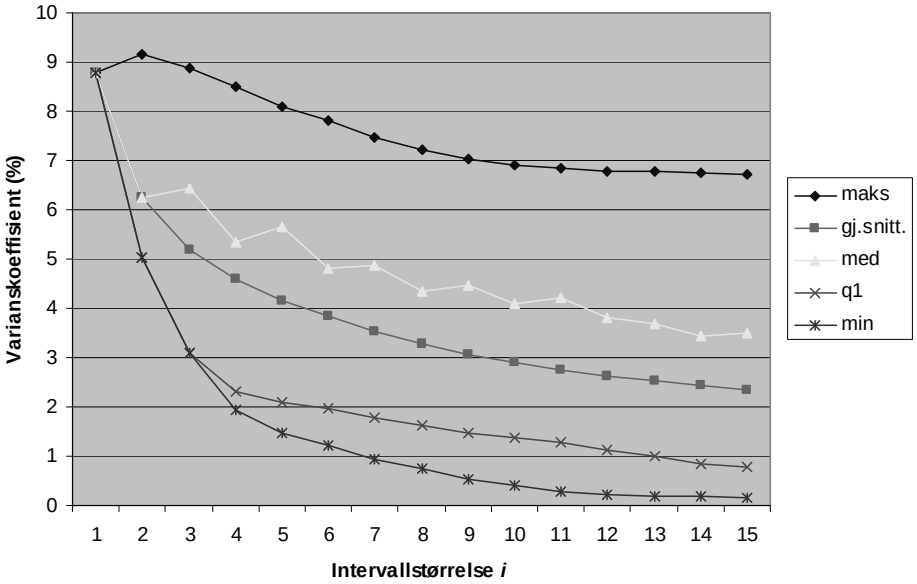
Figur 3.3e Plott av *minimum* mot intervallstørrelse for Seismod Iso



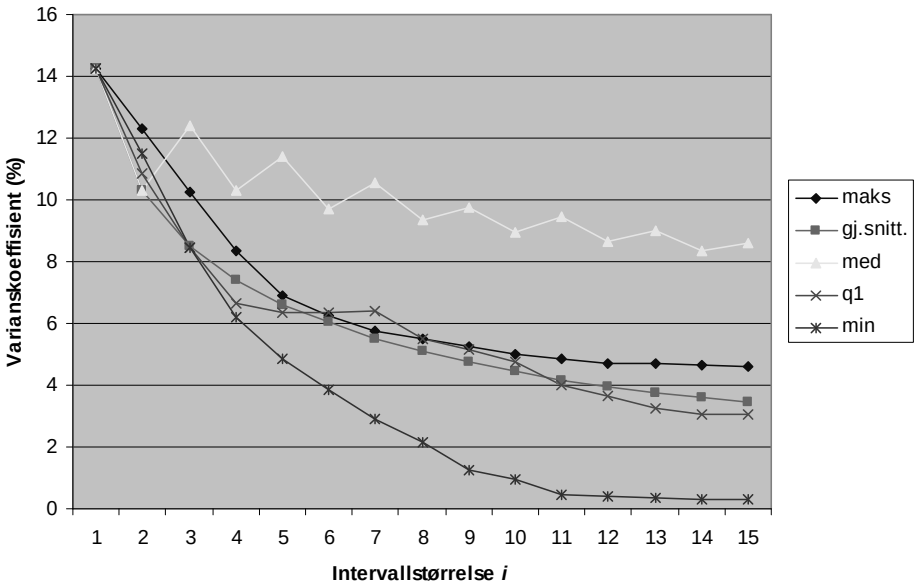
Figur 3.4a Plott av varianskoeffisient mot intervallstørrelse for lav last for Seismod Iso



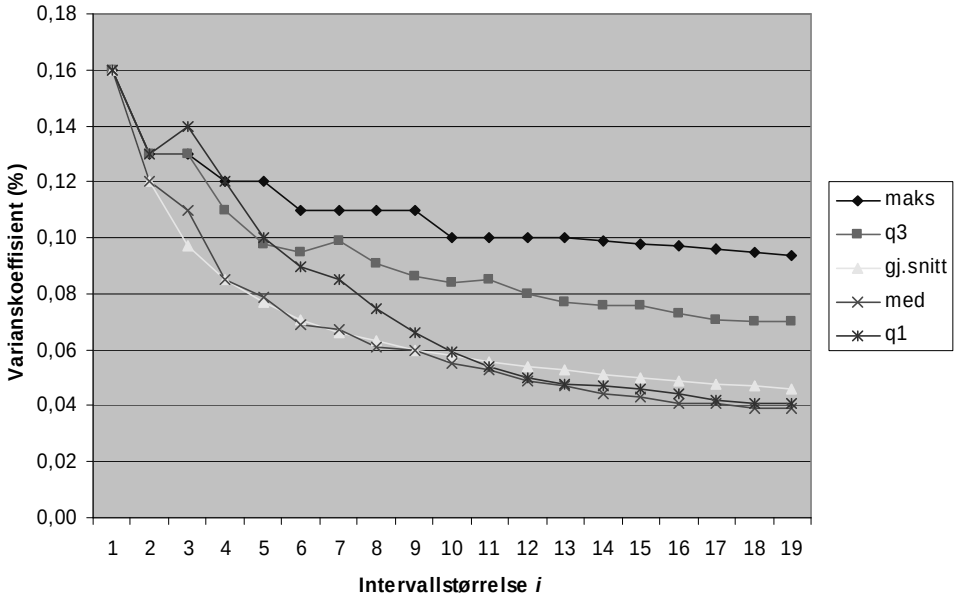
Figur 3.4b Plott av varianskoeffisient mot intervallstørrelse for høy last for Seismod Iso



Figur 3.4c Plott av varianskoeffisient mot intervallstørrelse for meget høy last for Seismod Iso



Figur 3.4d Plott av varianskoeffisient mot intervallstørrelse for last [0,∞> for SpeedTest for maks, q3 eller øvre kvartil, gjennomsnitt, median og nedre kvartil



3.5 Diskusjon

3.5.1 Seismod Iso

I spredeplottet for Seismod Iso ser vi tydelig tendenser til en oppsamling av data langs et minimum. Vi ser også en større spredning av dataene for høyere last. Det samme mønsteret ser vi igjen i histogrammene — en samling av kjøretider rundt den laveste intervallet er tydelig, og „halen" av kjøretider er mer markert for de høyeste lastintervallene.

Det vi ser i histogrammene gjenspeiler seg i figur(-ene) 3.4. For lav last situasjonen, synker både kvartil og minimum raskt mot null, median og gjennomsnittet synker saktere og samlet, men kommer aldri under 1, og maksimum *øker* litt i starten før den viser en utflatning.

For høy last situasjonen er resultatet noe av det samme, men her synker alle målene (bortsett fra maksimum) saktere. Minimumsmålet skiller seg her tydeligere fra nedre kvartil, og er veldig nært null etter mange målinger.

I situasjonen med meget høy last, skiller minimum seg enda klarere ut som målet som raskest synker til lavest variasjonskoeffisient. Medianen er her den med høyest variasjonskoeffisient, mens de andre målene har omtrent lik variasjonskoeffisient.

I figurene 3.3 kan vi se hvilke måleresultater vi kan forvente å få i de forskjellige lastsituasjonene. Hvis man ønsker å måle og sammenligne effektiviteten til programmer, så ønsker man å gjøre dette uavhengig av lastsituasjon. Det som er felles for alle figurene er at de starter med en forskjell på ca. 6 sekunder mellom lav og høy last, og 24 sekunder mellom høy last og meget høy last (lav last har en CPU-tid på ca. 235, høy last ca. 241 og meget høy last ca. 265).

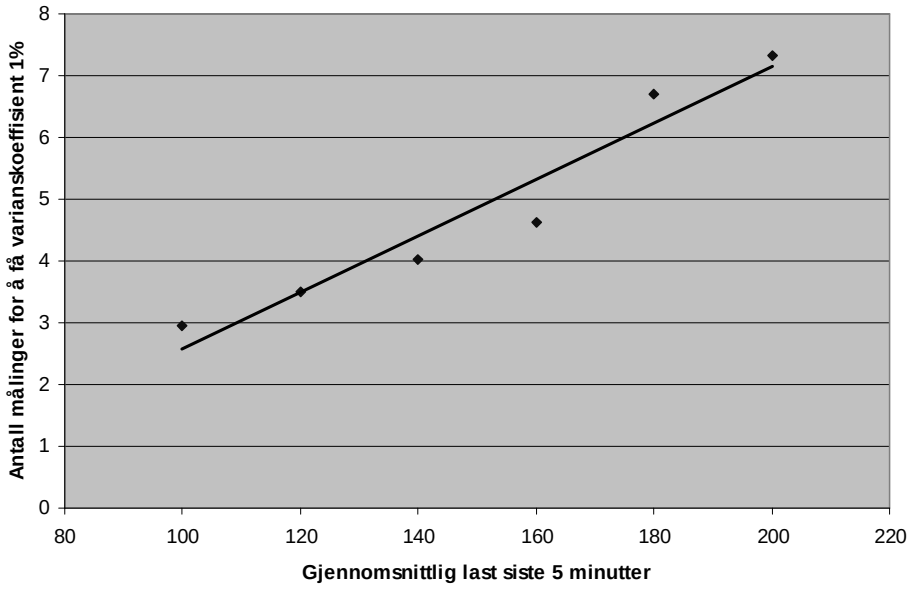
For maksimum, gjennomsnitt og median holder denne forskjellen seg omtrent konstant. For nedre kvartil, ser vi en tydelig tendens til konvergering mellom de tre last-situasjonene, men meget høy last har en CPU-tid som fortsatt etter 15 målinger er over 4 sekunder høyere enn de to andre lastsituasjonene. Minimum ser ut til å ha en enda raskere konvergering, og forskjellen etter 11 målinger er under 1,5 sekunder mellom de to lastsituasjonene som ligger lengst fra hverandre.

3.5.1.1 Sammendrag

Skal vi vurdere målene ut ifra variasjon fra måling til måling og konvergering av målene i de forskjellige lastsituasjonene, ser det ut som om minimum er det beste målet, og jo høyere lasten er, jo klarere skiller minimum seg ut. Men minimum har også andre egenskaper som gjør det godt egnet som mål for kjøretid. Det vil for det første gi oss den beste tilnærmelsen til en kjøretiden vi ville fått på en „tom" maskin (uten andre brukere). Hvis vi har kjørt et referanseprogram et stort antall ganger N og tatt minimum av disse N kjøringene, har denne verdien meget liten usikkerhet. Hvis man senere skal kjøre et program og sammenligne dette med referanseprogrammet, og man kjører x målinger, hvor $x << N$, og tar minimum av disse, vil man ved en bedre minimumsverdi for dette programmet være meget trygg på at dette programmet virkelig er mer effektivt, og at det ikke skyldes en målefeil, så lenge maskinkonfigurasjonen eller kompilatoren ikke er forskjellig. Dette er fordi minimum har stor sannsynlighet for å være lavere om man har flere kjøringer.

For å få en følelse av hvor mange kjøringer (målinger) som må til for å få et gitt standardavvik for minimum, har jeg plottet antall målinger jeg må foreta for å få en varianskoeffisient under 1% i figur 3.5. At det er en slik lineær sammenheng har jeg ikke bevist, men punktene i figuren kan antyde en slik sammenheng.

Figur 3.5 Plottet viser antall målinger som må til for å få 1% varianskoeffisient for en gitt last for målet minimum. Linjen er basert på lineær regresjon gjennom punktene.



For å komme frem til antall målinger for å få varianskoeffisient på 1% (punktene i grafen), har jeg brukt CPU-tidene i lastintervallene [*l*-10,*l*+10>. Lasten jeg har plottet for et punkt er gjennomsnittlig last i dette intervallet. For å finne antall målinger for et punkt som må foretaes for å få en varianskoeffisient på 1%, har jeg brukt regresjon mellom det antall målinger som gir noe over 1% varanskoeffisient og det antall målinger som gir noe under 1% varianskoeffisient. Hvis f.eks. vi får variasjonskoeffisient på 1,2% ved 4 målinger og 0,9% variasjonskoeffisient ved 3 målinger, vil vi ha 1% variasjonskoeffisient ved litt over 3 målinger, ca. 3,33.

3.5.2 SpeedTest

Histogrammene for SpeedTest (figur 3.2e og 3.2d) viser at det er liten forskjell på høy og meget høy lastsituasjon. Vi har en markant topp rundt intervallet 60,034-60,123, og noe som kan se ut som en normalfordeling med senter i dette intervallet. Situasjon med lav last hadde jeg for få data til å tegne fornuftige histogrammer for.

Pga. likheten mellom de to last-situasjonene, valgte jeg å behandle dem samlet videre når jeg studerte variasjonskoeffisienten i plott 3.4d. Av 3.4d ser vi at gjennomsnittet og medianen for små intervallstørrelser raskt stabiliserer seg på en meget liten verdi, mens nedre kvartil kommer etter på en intervallstørrelse på 10 og utover. At median og gjennomsnittet har små avvik er i god overensstemmelse med normalfordelingen i histogrammene.

Det som man også kan merke seg, er at hele datasettet har en meget lav varianskoeffisient (på ca. 0,16%). For å få en varianskoeffisient på under 0,10%, må vi ha 3 målinger for gjennomsnittet. Hvis vi ønsker en lavere varianskoeffisient, kan det se ut som om medianen er det beste målet.

DEL 2

LOKALE TRAVERSERINGS-MØNSTRE

4. TRAVERSERINGSREKKEFØLGEN TIL ENKELTMETODER

4.1 Innledning

Jeg vil i dette kapittelet se på cache-bruk til metoder som traverserer data-tabellen som man finner i MeshCont. Dette betyr at jeg vil studere metodene i MeshCont og MeshScalarField, siden MeshScalarField arver data-tabellen fra MeshCont.

Det man umiddelbart ser når man skal studere cachebruk for enkeltmetoder, er at mange traverserer data-tabellen lineært. Et eksempel på dette er gitt nedenfor i programutsnitt 4.1.

I kapittel 6 om Alternerende Traverseringsrekkefølge, presenterer jeg en modifikasjon som benytter seg av dette til å forbedre cachebruken, men i dette kapittelet vil jeg fokusere på traverseringsmønsteret for ett metodekall. Sett fra dette synspunktet, virker cachebruken for en lineær traversering optimal, siden enhver cache-linje utnyttes fullt ut før en ny hentes inn. Bom i cache oppstår dermed ikke mer enn nødvendig. Jeg kan heller ikke tenke meg en bedre cache-bruk for metoder som traverserer data-elementene suksessivt og parallelt (element *i* aksesseres i begge tabellene), f.eks. ved kopiering av elementene i den ene tabellen over i den andre; hele cache-linjene som lastes inn brukes opp før det lastes inn nye. Hvis de to tabellene er overlappende i cache, vil man på Gridur ikke få noe problem med dette, siden vi har en to-retningers sett-assosiativ

Programutsnitt 4.1. Prosedyre i MeshCont for speiling av alle punktene i en Mesh rundt origo

```
template<class T>
inline void MeshCont<T>::umap2c(void (*uF)(T& x, const T& y), const T& v)
//------------------------------------------------------------------------------
{
  IF_PRETRACE(umap2c);
  IF_PREINV(FATAL);
  IF_PREINV2(v,ERROR);

  for ( int i = 0; i < nno; i++) {
    (*uF)(data[i],v);
  }

  IF_RETINV(FATAL);
  IF_RETTRACE(umap2c);
}
```

I programutsnittet er nno antall elementer i en mesh (elementene er lagret leksikografisk fra 0 til nno-1 i data-tabellen). „IF_"-setningene er makroer som brukes under feilsøking.

cache.

De fleste metodene har altså en god cache-bruk, men for noen, er det mindre opplagt hvordan cachebruken er. Jeg vil i de følgende avsnittende studere disse metodene nærmere.

4.2 Speiling

Det er to prosedyrer for speiling i MeshCont. For nærmere dokumentasjon av disse prosedyrene og de andre metodene nevnt i dette kapittelet og resten av oppgaven, henviser jeg som vanlig til Sophus-dokumentasjonen [1].

4.2.1 Speiling rundt origo

Den ene av speilingsprosedyrene speiler hvert punkt i en mesh rundt origo vha. en MeshShape *S* som fungerer som et „bitmap" for speilingen. Hvis retning *i* i *S* er 0, vil

Programutsnitt 4.2. Prosedyre i MeshCont for speiling av alle punktene i en mesh rundt origo

```
//-------------------------------------------------------------------------
template<class T>
inline void MeshCont<T>::umirror(const MeshShape& S)
//-------------------------------------------------------------------------
{
  IF_PRETRACE(umirror);
  IF_PREINV(FATAL)
  IF_PREINV2(S,ERROR);
  IF_PRECOND(shape.uisMirror0u(S), FATAL);
/* Speiler samtlige punkt i MeshShapen vha. speilet S i origo og lar verdien
     (T verdien) vaere den samme i speilpunktet */

  if (shape.uvolume0u()!=0){
    MeshPoint P;
    P.usetShape(shape);
    T helpdata[MAXCONTSIZE];
    for (int i=0; i<nno; i++) {
      P.usetLexicographic(i);
      P.umirror(S);
      helpdata[P.ugetLexicographic0u()] = data[i];
    }
// har speilet og maa overfoere helpdata til data
    for (i=0; i<nno; i++) {
      data[i] = helpdata[i];
    }
  }
  IF_RETINV(FATAL);
  IF_RETTRACE(umirror);
}
```

komponent *i* i punktet etter speiling bli $-p_i + s_i$ hvis $-p_i$ er ulik null og 0 ellers. p_i er komponent *i* i punktet før speiling. Hvis retning *i* i *S* er ulik null, vil komponent *i* i punktet etter speiling være det samme som før speiling.

Koden for den opprinnelige speilingsprosedyren er gitt i programutsnitt 4.2. Som vi ser, benytter prosedyren en hjelpetabell, speiler data over i denne og kopierer data tilbake. I begge trinnene traverseres tabellene som data skal leses fra lineært, og cache benyttes derfor effektivt i begge for-løkkene for *data*-tabellen.

helpdata-tabellen traverseres ikke lineært i første for-løkke. Mønsteret til denne traverseringen er beskrevet i tabell 4.1, og tilsvarer traverseringen til siste løkke i min modifiserte speilingsprosedyre vist under.

Jeg vil nedenfor vise at det er mulig å traversere data-tabellen bare én gang, og dermed slippe å opprette en hjelpedatatabell. Jeg vil sammenligne cache-bruken til denne modifikasjonen med den originale koden (Sophus kode 1.16). Under bruker jeg definisjonen av speiling gitt i Sophus-dokumentasjonen, som jeg også skisserte over.

Lemma 4.1

$mirror(mirror(P,S),S) = P$, hvor *P* er et MeshPoint, og *S* er en MeshShape med like mange retninger som *P* sin MeshShape.

Bevis

La MeshPoint *P* før speiling være $P = (p_1, p_2, ..., p_n)$ og ha MeshShape $S' = (s_1', s_2', ..., s_n')$ og la $S = (s_1, s_2, ..., s_n)$.

Hvis s_i er ulik null, endres ikke komponent *i* under speiling. Hvis retning *i* i *S* er lik null, vil komponent *i* etter $mirror(mirror(P,S),S)$ bli $-(-p_i + s_i') + s_i' = p_i$ hvis p_i er ulik null og 0 ellers. □

4.2.2 Modifisert Speilingsprosedyre

Lemma 4.1 viser at det i speilingsprosedyren er mulig å bytte om to og to MeshPoint som er speil av hverandre for data i *data*-tabellen i MeshCont (forventet ut ifra den intuitive oppfattelsen av hva et speil er). Dette gjør jeg ved å bruke en tabell av typen boolean som holder styr på hvilke data i data-tabellen som er byttet. Prosedyren traverserer ellers alle dataene i data-tabellen og bytter så fremt det ikke er markert i boolean-tabellen at punktet allerede er speilet. Jeg markerer et punkt som speilet kun hvis det følger etter det punktet jeg har kommet til i traverseringen. Grunnen til dette, er at data-tabellen traverseres i stigende rekkefølge. Grovskissen er vist i grovskisse 4.1.

4.2.3 Cache-bruk og tidskompleksitet

Jeg vil nå sammenligne cache-bruken i min speilingsprosedyre og den opprinnelige speilingsprosedyren.

Grovskisse 4.1 Modifisert speilingsprosedyre

```
if <MeshShape til MeshCont har mer enn 0 dimensjoner>
    boolean done[MAXCONTSIZE];
    <Initialiser alle elementer fra 0 til nno-1 i done til false>
    for i = 0 to nno-1 do
        if (!done[i])
            MeshPoint P =  <speilet av det MeshPoint'et med
                           leksikografisk posisjon i>
            <bytt data data[i] og data[P's leksikografiske posisjon]>
            if <P's leksikografiske posisjon> > i
                done[P's leksikografiske posisjon] = true;
```

4.2.3.1 Min speilingsprosedyre

Min speilingsprosedyre har to løkker. Den første initialiserer alle elementene i *done*-tabellen. I seismiske simuleringer med Seismod inneholder *data*-tabellen elementer av typen *double.* Siden tabellen bruker mindre plass i minnet (boolean-tabellen bruker 1/8 av plassen en double-tabell bruker), vil traverseringen og tilordningen av den boolske tabellen *done* ventes å ta mindre tid enn traversering av *data*-tabellen (som i den originale speilingsprosedyren) under seismisk simulering. Færre cache-linjer lastes også inn enn i den første for-løkken i den originale speilingsprosedyren. At tabellen er så liten, vil også si at det er stor sjanse for at den blir lagret høyt i minnehierarkiet hvis vi sammenligner med hjelpetabellen som brukes i den originale speilingsprosedyren. Dessuten vil den også påvirke cache-bruken mindre under bytting av elementer i den neste for-løkken. Selv om den faller ut av cache, vil en cache-linje romme 8 ganger så mange elementer som en cache-linje av *data*-tabellen, og dermed vil den bidra mye mindre til cache-bom.

Den neste for-løkken bytter *data*-elementer. Avstanden mellom elementene som er speil av hverandre hvis begge dimensjonene speiles i en 16x12 mesh er vist i tabell 4.1.

Avstanden er avhengig av den MeshShape $S = (s_1, s_2)$ som meshen har, hvis vi ser på en 2-dimensjonal mesh, som Seismod-programmene bruker. Når første komponent i punktet som speiles er liten, men ulik null, er avstanden stor, noe som er negativt for cache-bruken. Jo nærmere denne komponenten er $s_1 / 2$, jo mindre blir avstanden, og jo bedre forventes cache-bruken å bli. Når vi har traversert den første halvdelen av elementene (leksikografisk sett), er alle elementene speilet. Alle *data*-elementene, utenom i de tilfellene speilet har samme leksikografiske posisjon som det punktet som skal speiles, er aksessert.

I tillegg til dette, vil aksesseringen av data skje innenfor s_1 bolker. Elementene ligger nært hverandre innenfor disse bolkene, som igjen er positivt for cache-bruken (gjenbruk av data).

Hvis bare første dimensjon speiles, forventes L1 cache-bruken ikke å bli så mye bedre enn hvis begge speiles, men noe forbedring burde det bli, siden man slipper hoppet fra f.eks. (15,0) til (15,11). Isteden aksesseres data omvent leksikografisk.

Hvis bare andre dimensjon speiles, vil dataene ligge nært hverandre på midten av hver av de s_1 bolkene. Cache-bruken forventes derfor å bli bedre.

Tabell 4.1 *data*-elementpar som byttes ved speiling av en 16x12 Mesh (begge dimensjoner speiles)

MeshPoint *P*	**Leksikografisk**	**Speilet av *P***	**Leksikografisk**
(0,0)	0	(0,0)	0
(0,1)	1	(0,11)	11
(0,2)	2	(0,10)	10
...	...	...	...
(1,0)	12	(15,0)	180
(1,1)	13	(15,11)	191
(1,2)	14	(15,10)	190
...	...	...	...
(8,0)	96	(8,0)	96
(8,1)	97	(8,11)	107
(8,2)	98	(8,10)	106
...	...	...	...
(15,0)	180	(1,0)	12
(15,1)	181	(1,11)	23
(15,2)	182	(1,10)	22
...	...	...	...

4.2.3.2 Opprinnelig speilingsprosedyre

Programutsnitt 4.2 viser den opprinnelige speilingsprosedyren. Cache-bruk er bestemt av to for-løkker. Begge henter inn data lineært i cache. I den første løkken skrives data i en rekkefølge som er bestemt av speilingen.

Siden det er to tabeller som benyttes, vil ikke man dra nytte av at møtepunkter (punkter der den leksikografiske posisjonen data hentes fra er nært). To tabeller vil også si at dobbelt så mye minne brukes, som igjen kan føre til flere cache-bom.

4.2.3.3 Sammenligning

Cache-bruken til min speilingsprosedyre, er som vi har sett avhengig av dimensjonene til meshen. Sammenligning av bom i nivå 1 cache[6] og kjøretiden til speilingsprosedyrene er vist nedenfor. Jeg bruker samme mesh-dimensjoner som for Seismod-programmene.

Som vi ser av tabell 4.2, er L1-cachebruken noe bedre for min speilingsprosedyre enn for den opprinnelige. Jeg går ut ifra at det er byttingen av data-elementer som står for de fleste cache-bommene for min speilingsprosedyre. Vi ser at cache-bom reduseres hvis speiling foretaes i bare én av dimensjonene. Tidsbruken reduseres også.

[6] L2 cachebruk varierte så mye fra måling til måling, men holdt seg godt under L1 cache-bom (under 1/10). Dette tolker jeg som et tegn på at data stort sett holder seg i L2-cache eller høyere for de mesh-størrelsene jeg har sett på her, og ikke må hentes fra minne ved L1 cache-bom. Bom i L1-cache så ut til å være stabilt for kjøringer med samme program, selv under variable lastforhold. Jeg ser derfor på L1 cache her.

Tabell 4.2 Sammenligning av kjøretid og bom i L1-datacache for speilingsprosedyrene

	L1-cache bom for min speilings-prosedyre (1 speiling)	**L1-cache bom for original speilings-prosedyre (1 speiling)**	**CPU-tid for 200 speilinger med min speilings-prosedyre (sekunder)**	**CPU-tid for 200 speilinger med original speilings-prosedyre (sekunder)**
Speiling i begge dimensjoner	165244	208776	17,7	31,7
Speiling kun i første dimensjon	158042	209441	16,9	28,9
Speiling kun i andre dimensjon	133941	212589	14,6	28,9

Gjennomsnittet av 3 kjøringer er brukt i tabellen. Jeg har brukt template-variabler av typen double i MeshCont.

Speilingsprosedyren min ser ut til å være nesten dobbelt så rask som den opprinnelige speilingsprosedyren. Min speilingsprosedyre har fordelen av bedre minne-kompleksitet, siden hjelpe-tabellen av data-elementer ikke må opprettes, men en mindre sannhetsverditabell (tabell av typen boolean) brukes istedenfor. Tidskompleksiteten er også noe bedre; det tar mindre tid å tilordne verdier i en sannhetsverditabell enn å kopiere elementer over fra en hjelpetabell. Antall tilordninger er derimot det samme.

Kjøretiden påvirkes positivt av at det er færre L1-cachebom. Hvor mye dette betyr er litt vanskelig å fastslå, siden jeg fikk variable L2-cachebom (se fotnoten), og dermed ikke er sikker på hvor i minnehierarkiet data hentes fra når L1-cachebom oppstår. Fra målinger med perfex –y på parallelldatamaskinen Ask, ser det ut som om disse cache-bommene har lite å si for kjøretiden (2-3 sekunder). Testene for hvor lang tid cache-bom tar ble forsøkt kjørt på Gridur, men perfex –y virket ikke der.

4.2.4 Speiling relativt til et punkt

Den andre speilingsprosedyren i MeshCont er programmert veldig likt den første. Eneste forskjell mellom denne og den andre, er at den bruker en annen MeshPoint-metode for speiling av et punkt relativt til et annet punkt. Denne MeshPoint-metoden er definert i Sophus-dokumentasjonen ved aksiomet

$$mirror(P,Q,S) = Q + mirror(P-Q,S),$$

hvor P og Q er to MeshPoint med samme MeshShape, og S er en MeshShape.

Lemma 4.2 viser at speiling oppfører seg som forventet; speiling av et punkt to ganger relativt til det punkt Q gir tilbake det samme punktet.

Lemma 4.2

$mirror(mirror(P,Q,S),Q,S) = P$, hvor P og Q er MeshPoint med samme MeshShape, og S er en MeshShape med like mange retninger som P sin MeshShape.

Bevis

La MeshPoint P før speiling være $P=(p_1, p_2,..., p_n)$ og $Q=(q_1, q_2,..., q_n)$ og ha MeshShape $S'=(s_1', s_2',..., s_n')$ og la $S=(s_1, s_2,..., s_n)$ være en MeshShape.

Hvis s_i er ulik null, endres ikke komponent i under speiling. Hvis retning i i S er lik null, vil komponent i etter $mirror(P,Q,S)$ bli $q-(p-q)+m\cdot s'-n\cdot s'=2q-p+n\cdot s'-m\cdot s'$, hvor $p=p_i, q=q_i, s'=s_i'$, og m, og n er heltall. Faktoren $n\cdot s'-m\cdot s'$ brukes til å justere komponenten i MeshPointet tilbake, slik at den ligger i intervallet $[0, s\rangle$; kravet til et lovlig punkt. Hvis vi speiler en gang til, vil komponenten endres til

$$2q-(2q-p)+k\cdot s'-l\cdot s'=$$
$$p+k\cdot s'-l\cdot s'$$

Justeringsfaktoren $k\cdot s'-l\cdot s'=0$, siden p er et lovlig MeshPoint og denne ellers ville justert p utenfor intervallet $[0, s\rangle$. □

Pga. denne egenskapen nå er bevist å gjelde for speiling rundt et punkt, kan vi bruke tilsvarende modifikasjon som vist i grovskisse 4.1.

4.2.5 Cache-bruk og tidskompleksitet

Data-aksesseringsmønsteret, illustrert i tabell 4.1 for speiling rundt origo endres noe. En komponent p i et MeshPoint blir $-p$ etter speiling rundt origo og $2q$-p (se beviset over) etter speiling relativt til et MeshPoint hvor komponenten i samme dimensjon som p er q.

Avstandsmønsteret som beskrives i tabell 4.1 forskyves altså med $2q$ for hver indeks. Dette betyr at møtepunktene, hvor cache-bruken er bra, forskyves. Møtepunktene i tabell 4.1 er (8,x) og (0,x). For speiling i enkeltdimensjoner, vil møtepunktene også forskyves, men fortsatt oppstå. Cache-bruken forventer jeg derfor skal bli omtrent den samme.

Den opprinnelige speilingsprosedyren forventer jeg skal ha omtrent samme cache-bruk som før. Eneste forskjell er at man har de samme forskyvningene som for min prosedyre i den første av de to for-løkkene.

4.2.5.1 Sammenligning

Eksperimentell sammenligning av cache-bruk er gitt i tabell 4.3. Som vi ser er cache-bruken veldig lik den andre speilingsprosedyren (se tabell 4.2). Dette stemmer godt med diskusjonen over.

CPU-tiden er noe større enn for speiling rundt origo. Kompleksiteten er større, siden MeshPoint-prosedyren for speiling relativt til et punkt tar lengre tid enn speiling rundt origo. Den originale speilingsprosedyren sin CPU-tid øker også mer enn CPU-tiden for min speilingsprosedyre. Dette skyldes at den originale speilingsprosedyren kaller speilingsprosedyren i MeshPoint på hvert punkt, mens min bare kaller den for halvparten av punktene. Dette betyr at en økning i kompleksitet til speilingsprosedyren i MeshPoint påvirker den originale speilingsprosedyren mer enn min speilingsprosedyre.

Tabell 4.3 Sammenligning av kjøretid og bom i L1-datacache for speiling relativt til et punkt

	L1 cache-bom for min speilings-prosedyre (1 speiling)	**L1 cache-bom for original speilings-prosedyre (1 speiling)**	**CPU-tid for 200 speilinger med min speilings-prosedyre (sekunder)**	**CPU-tid for 200 speilinger med original speilings-prosedyre (sekunder)**
Speiling i begge dimensjoner	166507	208998	19,7	36,9
Speiling kun i første dimensjon	166283	210138	18,6	34,3
Speiling kun i andre dimensjon	135541	209041	17,5	31,9

Gjennomsnittet av 3 kjøringer er brukt i tabellen. Jeg har brukt template-variabel av typen double i MeshCont. Jeg speiler en mesh av samme størrelse som i tabell 4.2 (850x620) relativt til punktet (72,133).

4.3 To map-prosedyrer med nøstede for-løkker

Map-prosedyrene i MeshCont og MeshScalarField har stort sett 1 forløkke som mapper alle elementene vha. en funksjon. Unntakene er de to prosedyrene umap2tc og umap2tp som har to nøstede for-løkker. Som de to speilingsprosedyrene, ligner disse map-prosedyrene veldig på hverandre.

4.3.1 Umap2tc

Denne map-prosedyren tar inn 3 argumenter: en funksjon og to MeshCont-objekter. Prosedyren er vist i programutsnitt 4.3.

I korthet tar denne prosedyren ut template-verdien for hvert MeshPoint i *this* (det MeshCont-objektet som prosedyren kalles fra) i leksikografisk rekkefølge. For hver av disse template-verdiene, tar den ut hver template-verdi i CP i leksikografisk rekkefølge, anvender prosedyren på de to template-verdiene, og setter denne inn i MeshCont d i posisjonen bestemt av concatenate(Q,QP), hvor Q er det MeshPointet som man har kommet til i traverseringen av *this* og QP er MeshPointet man har kommet til i traverseringen av CP. concatenate(Q,QP) lager et MeshPoint som er Q etterfulgte av retningene i QP, med en MeshShape som er MeshShapen til Q etterfulgt av den til QP, så d traverseres altså også leksikografisk.

For å illustrere cache-bruken, kaller jeg *data*-tabellen i *this* for a, *data*-tabellen i CP for b, og *data*-tabellen i d for c. Jeg antar at størrelsen til a og b er hhv. n og m. c har da

Programutsnitt 4.3. Map-prosedyren umap2tc0g i klassen MeshCont

```
//-------------------------------------------------------------------------
template<class T>
inline void MeshCont<T>::umap2tc0g(void (*uF)(T& x, const T& v),
const MeshCont<T>& CP, MeshCont<T>& d) const
//-------------------------------------------------------------------------
{
  IF_PRETRACE(umap2tc0g);
  IF_PREINV(FATAL);
  IF_PREINV2(CP,ERROR);
  IF_PRECOND(&d!=this,FATAL);

  d.shape = concatenate(shape,CP.shape);
  d.nsd = d.shape.udomainDimension0u();
  d.nno = d.shape.uvolume0u();

  if (d.shape.uvolume0u()!=0){
    MeshPoint Q;
    Q.usetShape(shape);
    MeshPoint QP;
    QP.usetShape(CP.shape);

    for(int i=0;i<nno;i++) {
      Q.usetLexicographic(i);
      for(int j=0;j<CP.nno;j++) {
        QP.usetLexicographic(j);
        T value = getData0g(*this,Q);
        (*uF)(value,getData0g(CP,QP));
        d.usetData(concatenate(Q,QP),value);
      }
    }
  }

//IF_RETINV(FATAL);
  IF_RETINV2(d,ERROR);
  IF_RETTRACE(umap2tc0g);
}
```

størrelsen $m \cdot n$[7]. Hvis a og b er endimensjonale tabeller, ser grovskissen for umap2tc0g ut som vist i grovskisse 4.2.

Jeg fokuserer på en prosessor med samme cache-egenskaper som jeg har beskrevet på slutten av kapittel 2. Jeg ser mest mulig bort ifra effekten andre programmer har på kjøringene.

[7] Med tabell-størrelse mener jeg antall elementer i meshen. Programteknisk kan antall elementer i meshen være mindre enn de statisk deklarerte tabellene. Størrelsen på disse tabellene er bestemt av brukeren av Sophus.

Grovskisse 4.2 Illustrasjon av map2tc0g for mapping av to MeshCont med én dimensjon til en MeshCont med to dimensjoner

```
for i=0 to nno-1 do
    for j=0 to nno-1 do
        c(den leksikografiske verdien til MeshPoint (i,j)) = f(a[i],b[j])
```

4.3.1.1 Tilfelle 1: Maksimalt 2 av tabellene overlapper hverandre i cache

Cache-bruken er best i det tilfellet der maksimalt to tabeller mapper til en lokasjon i cache (2-retningers sett-assosiativ cache). Da vil data i de to overlappende tabellene lagres i to forskjellige sett.

La oss se på L1 cache på 32 Kilobyte, og anta at a og b inneholder mer enn ett element. Hvis a og b er lagret slik at de ligger inntil hverandre i minne, vil de til sammen kunne utgjøre maksimalt 16 Kilobyte hvis a, b og c ikke skal være overlappende. Da vil det andre settet være fritt til å lagre c-elementer.

4.3.1.2 Tilfelle 2: De 3 tabellene overlapper hverandre (delvis eller helt) i cache.

Alle b-tabellens elementer aksesseres for hver *i* i den ytterste for-løkken i grovskisse 4.2. Disse brukes om igjen for hver iterasjon, og vi ønsker at disse skal bli i cache. Ett og ett element fra a-tabellen aksesseres for hver *i*, så cache-linjene fra a-elementer (som ikke lenger trengs) er de som stort sett blir erstattet av c-elementer hvis man har valget mellom en cache-linje av ”gamle” a-elementer og av b-elementer (LRU prinsippet beskrevet på slutten av kapittel 2).

Det er når cache-linjen hvor a-elementet man har kommet til i for-løkken overlapper med cache-linjen til c-elementet som skal fylles inn, og man samtidig har en overlapping med b-tabellen, at elementer som på et senere tidspunkt vil bli lastet inn i cache vil bli kastet ut. Når dette oppstår kan cache-linjer med a- eller b-elementer kastes ut:

a. Siden a-elementet hentes inn på nytt for hver iterasjon i den innerste for-løkken, vil det bare være når b-elementet som man har kommet til har en cache-linje som overlapper med cache-linjen til dette elementet og samtidig overlapper med cache-linjen til c-elementene som skal oppdateres, at cache-linjen til a-elementet kastes ut av cache.

b. Ellers er det cache-linjen med b-elementer som kastes ut.

I neste iterasjon vil man kunne få kastet ut c-elementer som skal brukes igjen senere fra cache.

c-tabellen traverseres fortløpende i den innerste for-løkken, mens a-elementet bare økes i den ytterste, så det vil bare være én cache-linje for hver cache-sett størrelse hvor tilfellet beskrevet over kan forekomme. Siden intervallet er så sjelden, regner jeg cache-bruken til prosedyren som ganske nært det optimale. Det som også taler for en god cache-bruk, er at b-tabellen i de fleste tilfeller holdes i cache. Siden b-tabellen brukes om igjen for hver *i*, ønsker man at denne skal holdes i cache. c-tabellen traverseres lineært, det samme med a-tabellen, så gjenbruk av elementer etter at en cache-linje er brukt er ikke så interessant.

Hvis tabellene blir meget store, kan mer komplekse situasjoner oppstå i L1 cache (en tabell kan overlappe med seg selv). Jeg vil ikke gå nærmere inn på det. I seismiske simuleringene som bruker Sophus-biblioteket som jeg har sett på vil dette ikke forekommer.

4.3.1.3 Forslag til modifikasjon

En modifikasjon jeg har sett på er å traversere tabellene intervallvis tilpasset cache-linjestørrelsene. Tanken er å utnytte cache-linjene fullt ut før ny lastes inn, noe som kan tenkes å være en fordel hvis data fra andre programmer skriver over data i cache-linjen som benyttes av mesh-metoden. Grovskisse for denne modifikasjonen er gitt i grovskisse 4.3.

Modifikasjonen har to ulemper:

1. Det er ikke sikkert en cache-linje vil bli truffet eksakt (det er faktisk en mindre sannsynlighet for at den blir truffet eksakt enn at den ikke blir det for enkle datatyper, siden disse er små). Man kan f.eks. risikere å benytte siste del av en cache-linje og første del av neste, men begge må lastes inn i sin helhet. Dette er en ulempe, siden data fra andre programmer kan da overskrive denne cache-linjen, og man siden må laste den inn på nytt.
2. Ekstra kompleksitet i administrasjon av den nye traverseringsrekkefølgen.

Fordel:

1. En fordel med modifikasjonen, er at den gjør seg ferdig med små bolker av a og b av gangen. Det er ikke like stor sjanse for at a og b går ut av cache på et senere stadium pga. innflytelse fra andre programmer eller hvis tabellene blir så store at de overlapper seg selv. Man kan også øke bolkestørrelsen slik at den ikke er lik cacherlinjestørrelsen, og samtidig bevare noe av den samme effekten samtidig som man reduserer ulempe 1 og 2.

4.3.1.4 Resultater

IRIX prøver å finne en ledig node å kjøre programmer på. Hvis lasten blir for høy, kan køsystemet utsette kjøringen. Dette kan være noen av årsakene til at jeg ikke så noen forbedring i kjøretid for denne modifikasjonen (i tillegg til de to ulempene nevnt ovenfor).

4.3.1.5 Videre arbeid

Jeg kunne prøvd å bruke verktøyet *dplace* til å allokere minne og plassere prosessen på en spesifikk node. Jeg kunne da kjørt flere programmer samtidig for å overbelaste cachen og se om det var forskjell på min modifikasjon og den originale prosedyren. Dette har ikke blitt gjort.

Grovskisse 4.3 Modifisert Umap2tc basert på grovskisse 4.2

```
cachelinesize = <elementer som passer inn i en cachelinje>
for l=0 to nno-1 step cachelinesize
    for k=0 to nno-1 step cachelinesize
        for i=l to min(nno-1,l+cacheinesize) do
            for j=k to min(nno-1,k+cachelinesize) do
                c(den leksikografiske verdien til MeshPoint (i,j)) = f(a[i],b[j])
```

4.3.2 Umap2tp

Denne prosedyren har som eneste forskjell at den introduserer en hjelpevariabel. Ellers er cache-bruken som for umap2tc.

4.4 Shift

Shift-prosedyren flytter alle data i en mesh i forhold til et MeshPoint. I Sophus-dokumentasjonen er den, som speilingsprosedyrene, definert i klassen CartGroupCont som arves av MeshCont i dokumentasjonen. I C++-implementasjonen av biblioteket, ligger shift-prosedyren i MeshCont.

Shift er definert ved definisjonslikningene[8]

$$\text{getShape0u(shift(C,P))} == \text{getShape(C)}$$

og

$$\text{getData0u(shift(C,P))} == \text{getData0u(C,P+Q)) for alle lovlige Q.}$$

Jeg vil i dette avsnittet konsentrere meg om empirisk studie av cache-bruken til shift-prosedyren sammenlignet med en mye enklere modifisert shift-prosedyre. Jeg vil studere forskyvning (anvendelse av shift) i én og to dimensjoner på en to-dimensjonal Mesh.

4.4.1 Modifisert shift-prosedyre

Programutsnitt 4.4 viser denne

4.4.2 Cache-bruk og CPU-tider

Jeg kjørte tester som sammenlignet CPU-tiden og cache-bruken til modifikasjonen og til den originale shift-prosedyren for å illustrere hvor optimalisert den var. Jeg brukte verdier for forskyvning (shift) og mesh som er nært de brukt i seismisk simulering. Tallene er vist i tabell 4.4.

Det ser ut som om det er gjort en god jobb med å optimalisere cache- og CPU-tidsbruken til denne prosedyren. CPU-tiden ligger godt under min modifikasjon og det samme gjelder cache-bruken. Man har i den originale shift-prosedyren plukket ut de elementene som forskyves, og i tillegg ikke gått via usetLexicographic-prosedyren, ugetLexicographic0u-prosedyren og operator+= slik jeg gjør i modifikasjonen (som er en mer høynivå og direkte oversettelse av definisjonen). Man har isteden funnet ut hvor langt data i data-tabellen skal forskyves og utnyttet dette. En forklaring på hvordan dette gjøres kommer jeg tilbake til i kapittelet om Virtuelt Shift-modifikajonen i kapittel 7.

[8] Shift er brukt som en funksjon i definisjonslikningene over. Man kan bruke shift som en funksjon ved å kalle funksjonen ved navn shift, og som en prosedyre når man kaller den ved navnet ushift i et program som bruker Sophus-biblioteket. Jeg kaller shift en prosedyre, siden den er definert i klassen MeshCont som en metode som tar inn ett argument og returnerer *void*.

Programutsnitt 4.4. Modifisert shift-prosedyre

```
template<class T>
inline void MeshCont<T>::ushift(const MeshPoint& P)
//-------------------------------------------------------------------------
{
        IF_PRETRACE(ushift);
        IF_PREINV(FATAL);
        IF_PREINV2(P,ERROR);
        IF_PRECOND(ulegalPoint0u(P), FATAL);
        IF_PRECOND(uniDirectional(P),ERROR);

        int i;
        T helpdata[MAXCONTSIZE];

        // Traverserer Meshen og shifter

        for (i=0; i<nno; i++)
        {
                MeshPoint tP;
                tP.usetLexicographic(i);
                tP+=P;
                helpdata[tP.ugetLexicographic0u()] = data[i];
        }

        for (i=0; i<nno; i++)
        {
                data[i] = helpdata[i];
        }

        IF_RETINV(FATAL);
        IF_RETTRACE(ushift);
  }
```

4.5 GetSubCont

ugetSubCont0g har tre overlastede versjoner. De tar ut utsnitt av en MeshCont basert argumentene. Hvis vi ser bort ifra forkrav (IF_PRECOND-makroer) og preinvarianter (IF_PREINV-makroer), som varierer fordi prosedyrene har forskjellige argumenter å anvende dem på, er prosedyrene bygget opp på samme måte bortsett fra den ene setningen som bestemmer utsnittet.[9] Den ene av prosedyrene er gitt i programutsnitt 4.5. Det er bare en del av setningen som er forskjellig fra en overlastet versjon til en annen, og denne er vist med grå bakgrunn.

[9] Det er to programtekniske forskjeller også. Den ene prosedyren bruker CP.shape som er lik S istedenfor S når MeshShape til MeshPoint Q skal settes, og en av prosedyrene bruker en annen måte å finne ut om CP.nno er ulik null på.

Tabell 4.4 Sammenligning av kjøretider og cache-bruk for modifikasjon og original

	CPU-tid (sekunder) ved 500 metodekall			L1 cache-bom
	Shift med MeshPoint (2,0)	**Shift med MeshPoint (0,3)**	**Shift med MeshPoint (1,3)**	**Ett metodekall av shift med hhv. (2,0), (0,3) og (1,3)**
Modifikasjon	4,86	4,81	5,76	881359
Original	1,32	1,46	1,34	598167

I forsøkene har jeg brukt en 850x620 mesh.

CP er den MeshCont som oppdateres med utsnittet av MeshConten *this*. CP får samme MeshShape-argumentet *S*, og CP traverseres leksikografisk når den fylles med utsnittet fra *this*. Data fra CP hentes inn i cache med minst mulig cache-bom (leksikografisk traversering).

Forkravet

IF_PRECOND(shape.usubShape0u(S),ERROR)

forteller oss at *S* er subShape av *this* sin shape. Dette betyr at alle retningene i *shape* er større eller lik retningene i *S*, hvor *shape* er medlemsvariabel til MeshCont og holder på *this* sin MeshShape.

4.5.1 Utsnitt basert på embed(Q,shape)

Den overlastede versjonen som tar utsnitt basert på embed(Q,shape), er vist i programutsnitt 4.5. CP traverseres som nevnt med god cache-bruk. Spørsmålet er hvordan data-tabellen i *this* traverseres.

embed-funksjonen er en omskrivning av uembed foretatt av SCC, som er en preprosessor til C++-kompilatoren CC utviklet slik at (blant annet) alle prosedyrer som begynner med u kan brukes som en funksjon.

inline void MeshPoint::uembed(const MeshShape& S1)

setter MeshShape til MeshPointet til S1, og tilpasser dimensjonene til MeshPointet til S1 ved å anvende modulo retningen i S1 (for alle retninger hvor dimensjonstallet og retningstallet er likt). Men siden *S* er subShape til MeshShapen til *this* vil modulo ikke endre noen av dimensjonene i *Q*. *this* traverseres altså økende den også, men med noen hopp i *data*-tabellen såfremt *S* er ulik *shape*. *this* traverseres også på en gunstig måte i forhold til cache-bruk.

4.5.2 Utsnitt basert på embed(P,Q)

I denne overlastningen baseres utsnittet på

inline void MeshPoint::uembed(const MeshPoint& Q).

Denne prosedyren plusser alle dimensjonene til MeshPointet med dimensjonen med samme dimensjonstall i Q. Deretter tilpasser den alle dimensjonene i MeshPointet ved å ta modulo mot retningene i MeshShapen til MeshPointet. embed(P,Q) fungerer på samme måte som addisjon av P og Q, men tillater at Q har en annen MeshShape enn P.

Argumentlisten til denne versjonen er

const MeshPoint& P, const MeshShape& S, MeshCont<T>& CP.

Argumentet P, som fra forkravet har samme MeshShape som MeshConten *this*, bestemmer hvordan utsnittet skiller seg fra utsnitt gjort med embed(Q,shape). Hvis alle dimensjonene i P er null, vil man få det samme utsnittet som ved forrige overlastning. Hvis alle dimensjoner utenom første er lik null, blir traverseringsmønsteret omtrent som over. Traverseringen vil starte på en høyere indeks i *data*-tabellen, og man kan risikere at man midt i traverseringen går fra siste delen av *data*-tabellen til første indeks. Dette kan være negativt for cache-bruken, siden man kan risikere å ikke få utnyttet hele cache-linjen. På den andre siden ville man hatt et hopp i traverseringen på dette punktet uansett, fordi man traverserer stykker lineært. Hvis disse systematiske hoppene er lange nok, noe som avhenger av hvor mye MeshShapen til *this* avviker fra MeshShapen *S* i argumentlisten, ville en ny cache-linje måtte hentes inn uansett.

Hvis flere dimensjoner i P er ulik null, vil cache-bruken kunne bli enda litt dårligere, siden man vil introdusere flere hopp fra en høy til en lav indeks i traverseringen av *this*.

4.5.3 Utsnitt basert på scale(embed(P,Q)-P,SF)+P

scale(embed(P,Q)-P,SF) har en annen MeshShape enn P, så dette må være en feil i dokumentasjonen (og koden); det er ikke lovlig å utføre +-operasjonen mellom to punkt

Programutsnitt 4.5. getSubCont-prosedyre

```
//-------------------------------------------------------------------------
template<class T>
inline void MeshCont<T>::ugetSubCont0g(const MeshShape& S,
 MeshCont<T>& CP) const
//-------------------------------------------------------------------------
{
  IF_PRETRACE(ugetSubCont0g);
  IF_PREINV(FATAL);
  IF_PRECOND(&CP!=this,FATAL);
  IF_PRECOND(shape.usubShape0u(S),ERROR);
  IF_PRECOND(S.uvolume0u()<=MAXCONTSIZE,ERROR);

  CP.shape = S;
  CP.nno = S.uvolume0u();
  CP.nsd = S.udomainDimension0u();

  if (CP.nno!=0){
    MeshPoint Q;
    Q.usetShape(S);

    for(int i=0;i<CP.nno;i++) {
      Q.usetLexicographic(i);
      CP.usetData(Q,getData0g(*this,embed(Q,shape)));
    }
  }

//IF_RETINV(FATAL);
  IF_RETINV2(CP,ERROR);
  IF_RETTRACE(ugetSubCont0g);
}
```

Programutsnitt 4.6. setSubCont-prosedyre

```
//-----------------------------------------------------------------------------
template<class T>
inline void MeshCont<T>::usetSubCont(const MeshCont<T>& CP)
//-----------------------------------------------------------------------------
{
  IF_PRETRACE(usetSubCont);
  IF_PREINV(FATAL);
  IF_PREINV2(CP,ERROR);
  IF_PRECOND(shape.usubShape0u(CP.shape), FATAL);

  if (CP.nno !=0){
    MeshPoint Q;
    Q.usetShape(CP.shape);

    for (int i=0; i<CP.nno; i++) {
      Q.usetLexicographic(i);
      usetData(embed(Q,shape),getData0g(CP,Q));
    }
  }

  IF_RETINV(FATAL);
  IF_RETTRACE(usetSubCont);
}
```

med forskjellig form. Jeg vil derfor ikke gå nærmere inn på denne prosedyren.

4.6 SetSubCont

På liknende måte som ugetSubCont0g, har usetSubCont tre overlastninger. En av dem er gitt i programutsnitt 4.6.

Disse tre prosedyrene skiller seg fra hverandre på en tilsvarende måte som getSubCont-prosedyrene. Hvis vi ser bort ifra setningene med prefiks IF_PRE, så er det kun koden vist med grå bakgrunn over som varierer. Vi har en „embed(Q,shape)”-, en „embed(P,Q)”- og en „scale(embed(P,Q)-P,SF)+P”-versjon. I usetSubCont traverseres CP gjennom alle elementene og det er *this* som får et utsnitt av CP. Cache-bruken blir akkurat som for ugetSubCont.

4.7 BorderEmbed og GetBorder

Metodeheaderne for de to metodene er

Programutsnitt 4.7a. borderEmbed

```
//------------------------------------------------------------------------
template<class T>
inline void MeshCont<T>::uborderEmbed(const MeshCont<T>& CP,const
MeshShape& SP)
//------------------------------------------------------------------------
{
 IF_PRETRACE(uborderEmbed);
 IF_PREINV(FATAL);
 IF_PREINV2(CP,ERROR);
 IF_PREINV2(SP,ERROR);
 IF_PRECOND(isEmbeddableBorder0u(shape,SP),ERROR);
 IF_PRECOND(CP.shape==getBorder(SP,SP),ERROR);

 MeshPoint Q;
 Q.usetShape(CP.shape);
 for(int i=0;i<CP.nno;i++) {
  Q.usetLexicographic(i);
  usetData(borderEmbed(Q,shape,SP),getData0g(CP,Q));
 }

 IF_RETINV(FATAL);
 IF_RETTRACE(uborderEmbed);
}
```

uborderEmbed(const MeshCont<T>& CP,const MeshShape& SP)

ugetBorder0g(const MeshShape& SP,MeshCont<T>& CP) const

Begge har en forløkke som traverserer CP fra 0 til og med nno-1. Cache-bruken for traversering av CP er altså bra. Traverseringen av *this*, bestemmes av borderEmbed(Q, shape,SP), hvor Q er et punkt i CP som brukes til å traversere CP leksikografisk. De to metodene er gitt i programutsnitt 4.7a og 4.7b.

En grense (border) MeshShape *SP* for en MeshShape *S* har like mange retninger som *S*. Grensen er den MeshShape som fremkommer når vi konkatenerer alle retningene i *S* hvor *SP* er ulik null. En grense er „embedable" hvis *SP* er en subShape av *S*.

void MeshPoint::uborderEmbed(const MeshShape& S,const MeshShape& SP)

opererer på et punkt som har grensen av *SP* for *SP* som MeshShape. MeshShape til det nye punktet blir *S*. *SP* er en embeddeble border til *S*, og resultatet etter uborderEmbed for dimensjonene i punktet $(p_1, p_2, ..., p_n)$ er

$$p_i = \begin{cases} p_j \text{ hvis retning } i \text{ i } SP \text{ er ulik } 0 \\ 0 \text{ ellers} \end{cases}.$$

I både getborder og borderembed i MeshCont settes eller er MeshShape til CP den grensen *SP* er for seg selv, dvs. lik en konkatenering av de ikke-tomme retningene i *SP*. SP er en embedeble border til shape ifølge forkravet til MeshPoint-prosedyren uborderEmbed. Resultatet av kallet borderEmbed(Q,shape,SP), er et punkt med en MeshShape shape, hvor alle retningene er større eller lik retningene i SP, og hvor alle retningene hvor SP er lik null er fylt ut med 0 også i punktet. Begge disse forholdene kan medføre hopp i traverseringen.

Programutsnitt 4.7b. getBorder

```
//------------------------------------------------------------------------
template<class T>
inline void MeshCont<T>::ugetBorder0g(const MeshShape&
SP,MeshCont<T>& CP) const
//------------------------------------------------------------------------
{
  IF_PRETRACE(ugetBorder0g);
  IF_PREINV(FATAL);
  IF_PREINV2(SP,ERROR);
  IF_PRECOND(isEmbeddableBorder(CP.shape,SP),ERROR);

  CP.shape = getBorder(SP,SP);
  CP.nsd = CP.shape.udomainDimension0u();
  CP.nno = CP.shape.uvolume0u();

  if (CP.nno !=0){
    MeshPoint Q;
    Q.usetShape(CP.shape);

    for(int i=0;i<CP.nno;i++) {
      Q.usetLexicographic(i);
      CP.usetData(Q,getData0g(*this,borderEmbed(Q,shape,SP)));
    }
  }
  IF_RETINV2(CP,ERROR);
  IF_RETTRACE(ugetBorder0g);
}
```

4.8 Sammendrag

Jeg har gått igjennom de metodene hvor man ikke trivielt kan se at cache-bruken er god og man har en leksikografisk traversering av data av en eller to tabeller. En ny speilingsprosedyre ble laget. Tidsforbruket til denne var nesten halvparten av den originale for både speiling rundt origo og rundt et punkt. Cache-bruken ble også noe forbedret, men tidsforbruket skyldes antakeligvis i størst grad et redusert antall operasjoner.

De to map-prosedyrene som ble studert viste seg å ha god cache-bruk i de fleste tilfeller. Det er når man med en 2-retningers sett-assosiativ cache har 3 overlappende tabeller i cache at det kan oppstå problemer. Dette oppstår sjelden, så cachebruken er optimal i de fleste tilfeller. Når jeg studerte shift-algoritmen mot en annen og enklere shift-implementasjon, ser jeg at denne også er godt implementert med hensyn til cache-bruk.

I get- og set-subcont prosedyrene, er det en tabell som traverseres optimalt, mens man kan ved noen av disse prosedyrene få cache-messig ugunstige hopp i traverseringen, avhengig av parametrene til prosedyrene. Det samme er tilfelle for borderEmbed og getBorder. Disse hoppene skyldes at et utsnitt av *data*-tabellen traverseres.

Alt i alt er det 85 metoder i MeshCont og MeshScalarField. Noen av disse traverserer ikke datatabellen, og det er ikke særlig mange minneoperasjoner og potensial for forbedring av cache-bruken. Blant de 35 metodene som traverserer data-tabellen, ser det ut som om alle utenom speilingsprosedyrene har en meget bra cache-bruk og et gunstig traverseringsmønster i de fleste tilfeller.

DEL 3

GLOBALE TRAVERSERINGS-MØNSTRE

5. TO IKKE-LEKSIKOGRAFISKE ORDNINGER

5.1 Innledning

Jeg vil i dette kapittelet se på hvordan cache-bruken endrer seg hvis data ikke lagres leksikografisk. Lagringsposisjonen i *data*-tabellen bestemmes her av en én-til-én mapping av den leksikografiske posisjonen. På grunn av én-til-én mappingen av leksikografisk posisjon til MeshPoint bevist i kapittel 2, vil vi fortsatt ha en én-til-én mapping mellom den nye måten indekseringen er gjort på og MeshPoint. Jeg vil studere mappingene beskrevet nedenfor.

- *Omvendt leksikografisk:* Data lagres i omvendt rekkefølge av den leksikografiske. Data med leksikografisk posisjon lex, blir lagret i posisjon $n - lex - 1$ i data-tabellen, hvor n er antall data i meshen.
- *Randomisert:* Sekvensen $L = (0,1,...n-1)$ stokkes om „tilfeldig". Et data-element med leksikografisk posisjon lex, blir lagret i posisjon L_{lex} (element nr lex i L, hvis vi nummerer fra 0 og oppover) i data-tabellen.

Disse mappingene medfører en annen traversering av data-tabellen, og jeg vil studere den nye cache- og tids-bruken.

5.2 Implementasjon

Begge implementasjonene innebærer forandring i MeshCont- og MeshScalarField-klassene. Jeg baserer meg på en modifikasjon hvor alle direkte oppslag i *data*-tabellen er erstattet med kall til:

obs C. prosedyre ugetData0g (obs int i, upd T d);
Forkrav: $0 \le i < n$
Resultat: Returnerer verdien på usetLexicographic(P,*i*).

upd C. prosedyre usetData (obs int i, obs T d);
Forkrav: $0 \le i < n$
Resultat: Setter verdien på setLexicographic(P,*i*).

n over er antall verdier i meshen, C er MeshCont-objektet *this*.

Når alle oppslag går via getData- og setData-prosedyrer (det er to til, som allerede ligger i Sophus-biblioteket), kan man styre lagringsrekkefølgen ved å endre hvor disse prosedyrene henter og lagrer data i *data*-tabellen.

5.2.1 Omvendt leksikografisk

Endringene i getData- og setData-prosedyrene. som ligger i MeshCont er beskrevet nedenfor.

obs C.prosedyre ugetData0g (obs int i, upd T d);
Eksakt: Returnerer verdien på $n-i-1$.

upd C. prosedyre usetData (obs int i, obs T d);
Eksakt: Setter verdien på $n-i-1$.

obs C. prosedyre ugetData0g (obs MeshPoint P, upd T d);
Eksakt: Returnerer verdien på $n-P.ugetLexicographic0u()-1$.

upd C. prosedyre usetData (obs MeshPoint P, obs T d);
Eksakt: Setter verdien på $n-P.ugetLexicographic0u()-1$.

Jeg lager også en *referanse*, hvor lagringsrekkefølgen er leksikografisk. For at forskjellen i cache-bruk og tidsbruken mellom omvendt leksikografisk og referansen i størst mulig skal skyldes lagringsrekkefølgen, introduserer jeg en variabel int nno_, initialiserer denne til 0 i konstruktørene og erstatter nno-x-1 (hvor *nno* er antall elementer i meshen) med x-nno_.

5.2.2 Randomisert

Jeg legger til en tabell

```
protected:
        int maptable[MAXCONTSIZE];
```

som medlemsvariabel i MeshCont. Denne lagrer en omstokket versjon av *L*. Omstokkingen gjøres ved kall på innebygde randomiseringsmetoder i standardbiblioteket stdlib.h til C++. Dette gjøres i en ny prosedyre

upd C.prosedyre ucreateMaptable ();

For å initialisere *maptable*, brukes koden vist i programutsnitt 5.1 i begge konstruktørene til MeshCont (som pga. arv også kalles når et MeshScalarField-objekt opprettes), samt alle steder meshen får en ny MeshShape (et eksempel er ugetSubCont-metodene beskrevet i forrige kapittel, i tillegg til operator=).

getData- og setData-prosedyrene i MeshCont for denne modifikasjonen er beskrevet under:

obs C.prosedyre ugetData0g (obs int i, upd T d);
Eksakt: Returnerer verdien på L_i.

upd C. prosedyre usetData (obs int i, obs T d);
Eksakt: Setter verdien på L_i.

obs C. prosedyre ugetData0g (obs MeshPoint P, upd T d);
Eksakt: Returnerer verdien på $L_{P.ugetLexicographic0u()}$.

upd C. prosedyre usetData (obs MeshPoint P, obs T d);
Eksakt: Setter verdien på $L_{P.ugetLexicographic0u()}$.

Også her lager jeg en *referanse*. Referansen skiller seg fra randomisert-modifikasjonen bare ved at jeg erstatter setningen

Programutsnitt 5.1. Initialisering av maptable-tabellen.

```
//-------------------------------------------------------------------------
template<class T>
inline void MeshCont<T>::ucreateMaptable()
//-------------------------------------------------------------------------
{
  IF_PRETRACE(ucreateMaptable);
  IF_PREINV(FATAL);

  srandom(time(NULL));
  int i;
  for (i=0; i<nno; i++)
     maptable[i] = i;

  int temp;
  int x;
  for (i=0; i<nno; i++)
    {
     x = random()%nno;
     temp = maptable[i];
     maptable[i] = maptable[x];
     maptable[x] = temp;
    }

  IF_RETINV(FATAL);
  IF_RETTRACE(ucreateMaptable);
}
```

```
maptable[x] = temp;
```

med

```
maptable[i] = temp;
```

i for-løkken i programutsnitt 5.1

5.3 Eksperiment

Eksperimentet består i å teste modifikasjonene mot sine referanser. Jeg ser på L1 cache-bom og CPU-tid i alle eksperimentene. L2 cache-bom var enten mye lavere enn L1 cache-bom, eller de var så variable fra kjøring til kjøring at det var umulig å komme frem til en pålitelig verdi for sammenligning, og variasjonen korrelerte ikke heller med kjøretiden.

Jeg bruker Seismod Iso som testprogram fra Seismod-pakken. For omvendt leksikografisk tar jeg minimum av 5 kjøringer når jeg skal komme frem til CPU-tid og cache-bom. Fremgangsmåten er i samsvar med kapittel 3 for CPU-tidsmålinger. Når det gjelder cache-bom, bruker jeg samme resonnement som brukt om CPU-tid i kapittel 3 –

minimum er det målet som gir en cache-bom som er nærmest det man ville fått på en maskin hvis det ikke var andre brukere til stede, og høyere cache-bom må skyldes innflytelse fra andre prosesser.

Randomisert er litt spesiell, siden lagringsrekkefølgen og dermed cache- og CPU-tidsbruken er avhengig av ordningen til *L*. Jeg bruker derfor *gjennomsnittet* av CPU-tid og cache-bom for 5 kjøringer. Seismod Iso skriver til disk under kjøring, så jeg kjørte ikke kjøringer fra samme katalog samtidig for noen av kjøringene.

Jeg testet også enkeltmetoder med en 850x620 mesh, som er i samme størrelsesorden som og til dels større enn dem som brukes i Seismod Iso, som bruker forskjellige meshstørrelser. Jeg valgte ut de tre mesh-metodene operator=, som kopierer en mesh til en annen mesh, umapc, som traverserer alle data i data-tabellen og anvender en unær funksjon $f : T \to T$ på template-verdiene og umirror som speiler data, og er beskrevet i begynnelsen av kapittelet om cache-bruken til enkeltmetoder. Jeg kjører her også 5 kjøringer for hver metode, og bruker minimum for omvendt leksikografisk og gjennomsnitt for randomisert. f som brukes i umapc er $f(x) = (\lfloor x \rfloor + 2)\%14$, $f : double \to double$ (double brukes som template for alle meshene). Speiling ble foretatt i begge dimensjoner.

ucreateMaptable ble også testet. Når jeg skulle måle CPU-tid repeterte jeg metodekallet 200 ganger for å få kunne måle tidsbruken nøyaktig nok. Ved måling av cache-bruk, brukte jeg én iterasjon.

5.4 Resultat

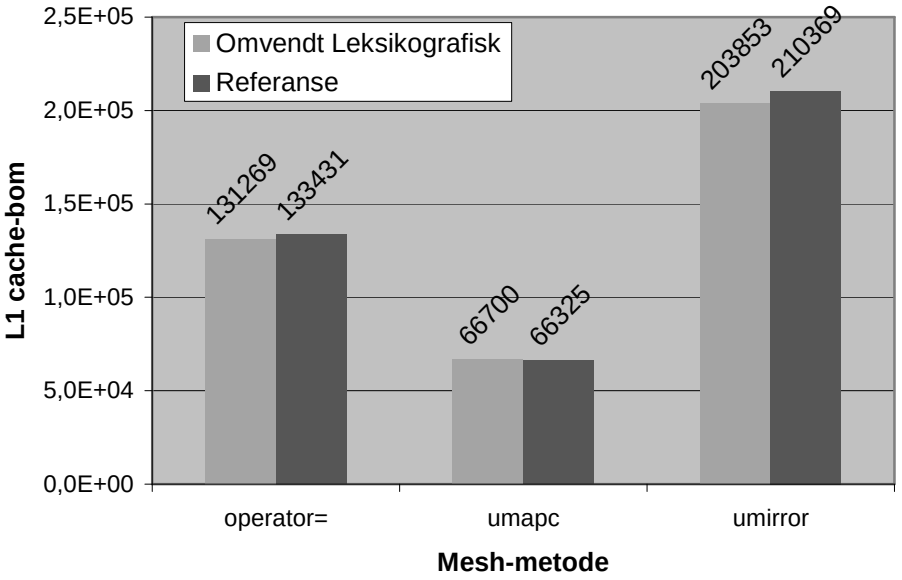
CPU-tidene og cache-bom er gitt i grafene og tabellene nedenfor.

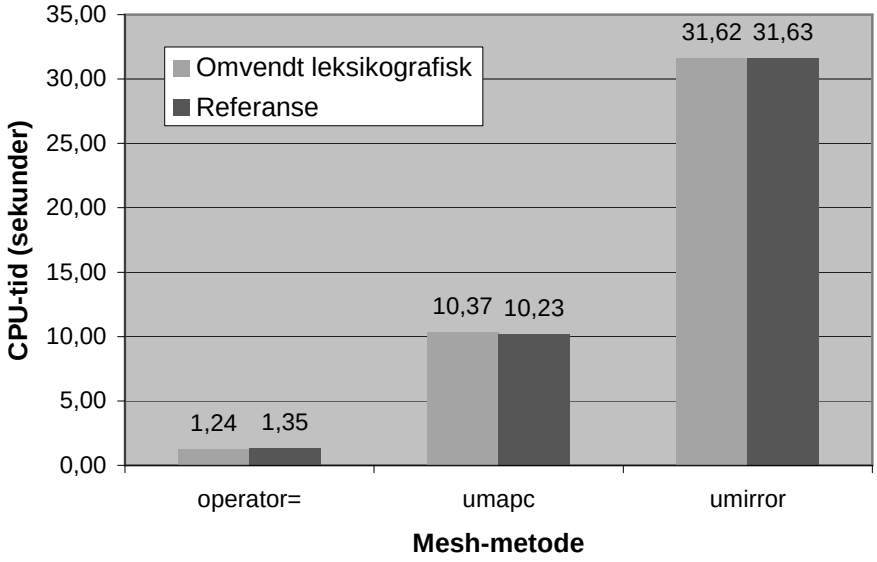
5.4.1 Omvendt leksikografisk

Tabell 5.1 *Omvendt Leksikografisk:* CPU-tid og cache-bruk for Seismod Iso

	Omvendt leksikografisk	**Referanse**
CPU-tid (sekunder)	300	325
L1-cache bom	$406 \cdot 10^7$	$406 \cdot 10^7$

Figur 5.1 *Omvendt Leksikografisk:* Cache-bom og CPU-tid for mesh-metoder



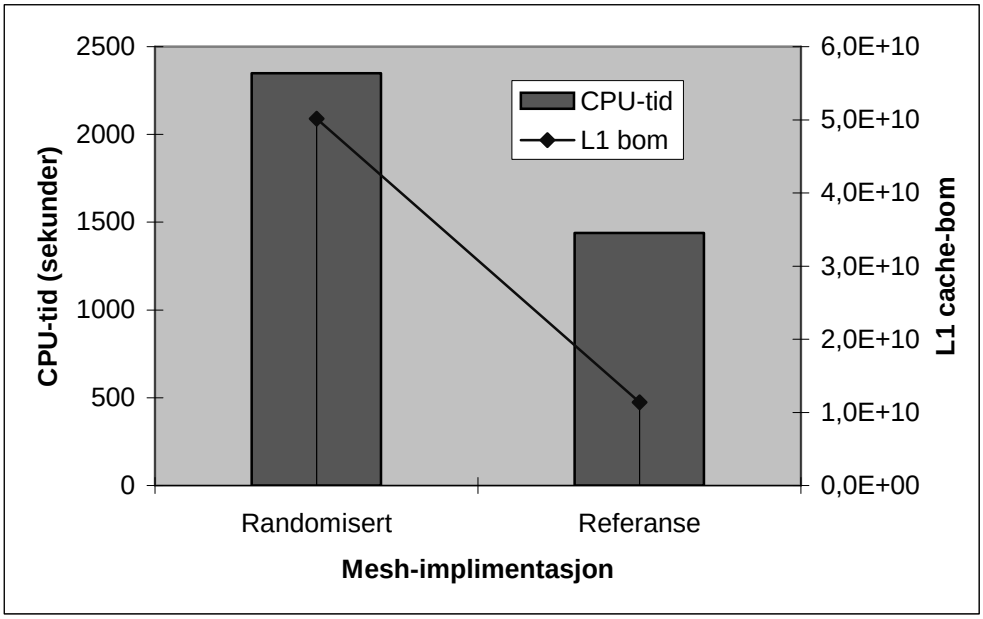


Antall cache-bom (for 1 iterasjon) og CPU-tid (for 200 iterasjoner) er vist over søylene.

5.4.2 Randomisert

Resultatene for den randomiserte mesh-implementasjonen så ikke ut til å variere vesentlig mer enn ellers for noen av kjøringene, selv om vi for hver kjøring har en ny ordning. Jeg går ut ifra at dette skyldes at det er 527000 elementer i meshen. Dette er såpass mange elementer, at „randomiseringsgraden" blir omtrent lik fra gang til gang.

Figur 5.2 *Randomisert:* CPU-tid og cache-bom for Seismod Iso



Kombinert graf med to akser, Cache-bom i høyre akse, og CPU-tid i venstre.

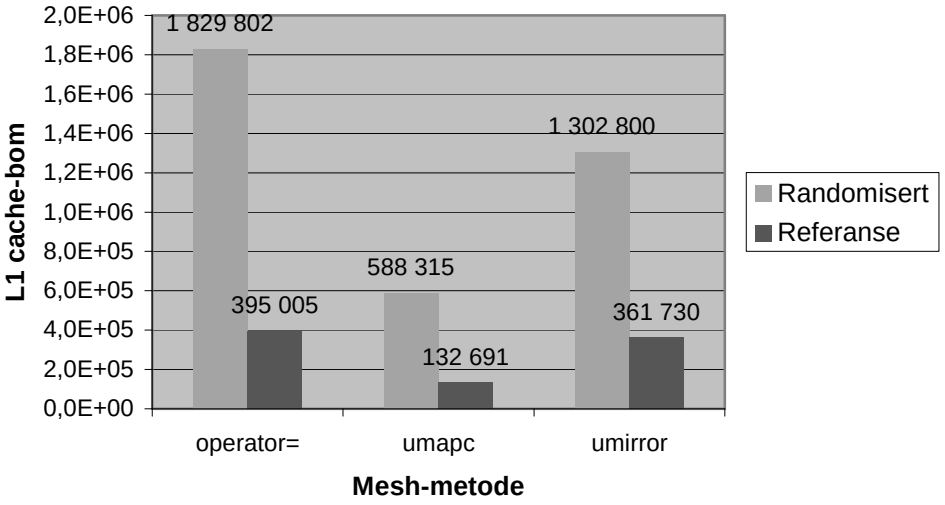
	Randomisert	**Referanse**
CPU-tid (sekunder)	2349	1439
L1 cache-bom	$503 \cdot 10^8$	$114 \cdot 10^8$

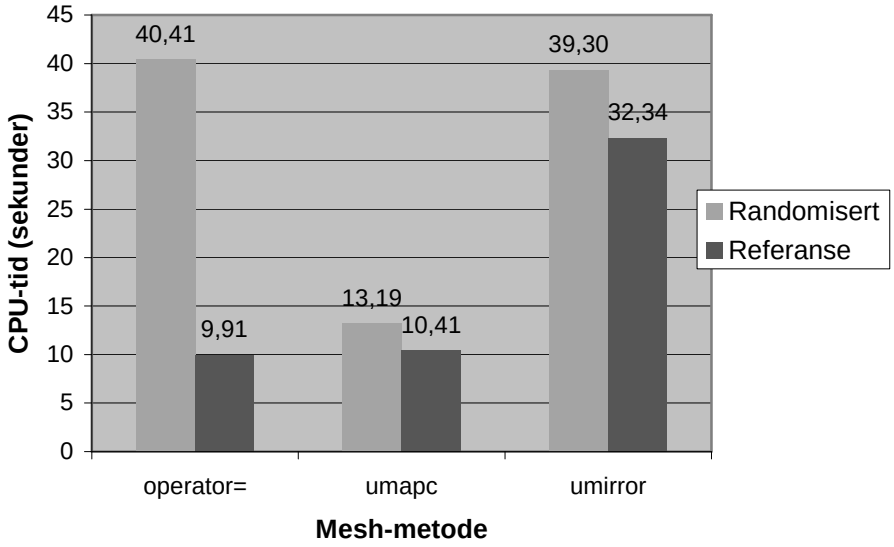
Dataene i figure 5.2 i tabellform

Tabell 5.2 *Randomisert:* CPU-tid og cache-bom for ucreateMaptable

	Randomisert	**Referanse**
CPU-tid (sekunder)	17,1	7,46
L1 cache-bom	$650 \cdot 10^3$	$131 \cdot 10^3$

Figur 5.3 *Randomisert:* Cache-bom og CPU-tid for mesh-metoder





5.5 Diskusjon

5.5.1 Omvendt leksikografisk

Seismod Iso målingene viser at L1 cache-bom er lik med en nøyaktighet på 3 signifikante sifre. Likheten skyldes blant annet at uansett om man leser bakfra eller forfra, så vil cache-linjene bli utnyttet helt før en ny lastes ved en suksessiv og leksikografisk traversering, som det finnes mye av i MeshCont (jamfør kapittelet om mesh-metoder, kapittel 4). umapc og operator= er metoder som hører til denne kategorien mesh-metoder, og disse har omtrent likt antall cache-bom (innenfor det jeg regner som feilmarginene). umirror er en metode som er studert i kapittel 4, og cache-bruken påvirkes heller ikke her signifikant.

CPU-tidene påvirkes noe. Operator= ser ut til å påvirkes mest (i favør av omvendt leksikografisk), mens CPU-tidene kan for de to andre metodene betraktes som like innenfor feilmarginene. Operator= er en metode som stort sett bruker tid på å kopierer data i en *data*-tabell til en annen, i alle fall ved de mesh-størrelsene som er studert her. Cache-bruken er derfor viktig for denne metoden, samt kallene til get- og set-data metodene, siden det er mange minneoperasjoner relatert til CPU-operasjoner. Jeg regner derfor med at CPU-tidsforskjellen i favør av omvendt leksikografisk, må skyldes at kallene til get- og set-data metodene tar noe lengre tid for referansen. Jeg regner med at det er dette som gjør at også Seismod Iso med omvendt leksikografisk mesh-implementasjon bruker noe kortere tid enn referansen, til tross for likt antall cache-bom. Forskjellen er imidlertid også her liten.

5.5.2 Randomisert

Randomisert modifikasjon er et forsøk på å lage en cache-messig dårlig modifikasjon. Ofte vil en ny cache-linje lastes inn etter at bare ett element er lest fra forrige cache-linje. Når man hopper rundt i *data*-tabellen på denne måten og laster inn nye cache-linjer hele tiden, er sjansen stor for at man må kaste ut en cache-linje som skal brukes senere.

Figur 5.3 viser også en stor forskjell i cache-bruk mellom den randomiserte og referansen. Cache-bom for referansen er 22 og 23 prosent av cache-bom for den randomiserte for umapc og umirror, mens den er 28 prosent av referansen for operator=. Operator= har kall til ucreateMaptable, siden den lager en ny mesh, og hvis vi justerer for forskjellen i cache-bom for ucreateMaptable mellom den randomiserte og referansen, som gjort i tabell 5.3, blir prosenttallet for operator= 22 prosent. Vi har altså

Tabell 5.3 *Randomisert:* CPU-tid og cache-bruk for operator= justert for ucreateMaptable-kall

	operator=	
	Randomisert	**Referanse**
CPU-tid (sekunder)	22,7	2,15
L1 cache-bom	$118 \cdot 10^4$	$264 \cdot 10^3$

Tabell 5.4 *Randomisert:* Forhold mellom forskjell i cache-bom og CPU-tidsforskjell

		Randomi-sert	Referanse	Randomi-sert – Referanse	CPU-tid/cache-bom (sekunder)
umapc	**CPU-tid (sekunder)**	13,19	10,41	2,78	6,11E-06
	L1 cache-bom	588 315	132 691	455 624	
umirror	**CPU-tid (sekunder)**	39,30	32,34	6,96	7,40E-06
	L1 cache-bom	1 302 800	361 730	941 070	
Operator=	**CPU-tid (sekunder)**	22,70	2,15	20,55	2,24E-05
	L1 cache-bom	1 180 000	264 000	916 000	
Seismod Iso*	**CPU-tid (sekunder)**	2349	1439	910	2,34E-8
	L1 cache-bom	$503 \cdot 10^8$	$114 \cdot 10^8$	$389 \cdot 10^8$	

* For Seismod Iso har jeg ikke justert CPU-tid eller cache-bom for kall til ucreateMaptable

en rimelig konstant forskjell; randomisert dataaksess gir omtrent 4,5 ganger så mange cache-bom som leksikografisk for alle de tre metodene.

Eneste forskjell mellom umapc og umirror, er at ordningen (og aksessmønsteret) av *data*-tabellen er forskjellig. Dette resulterer i en viss forskjell mellom CPU-tidene i favør av referansen. Et høyere antall L1 cache-bom gir en høyere CPU-tid. Følgende tabell bruker tallene i tabell 5.3 for operator=, og tallene i figur 5.3 for umapc og umirror, får vi følgende tabell som illustrer forholdet mellom forskjell i cache-bom og forskjell i CPU-tid.

CPU-tidene i tabellen i viser også at den randomiserte blir påvirket av cache-bom, men i forskjellig grad avhengig av metode. Siste kolonne har jeg prøvd å vise hvor lang tid hvert cache-bom bruker. Mens umapc og umirror har et forhold som ligger i omtrent samme område, skiller operator= seg kraftig ut, og Seismod Iso skiller seg ut i enda større grad. En forklaring på at operator= skiller seg ut, kan være at prosessorens evne til å kompensere for utestående cache-bom er mindre for denne metoden. Dette gjelder i så fall i enda større grad for Seismod Iso.

Forholdet mellom cache-bom for referansen og den randomiserte føyer seg bra inn blant forholdet for metodekallene. Antall cache-bom for referansen er omtrent 23 prosent av cache-bom for den randomiserte. CPU-tiden som vi ser i figur 5.2 påvirkes en god del av dette, og referansen har en CPU-tid som er ca. 61 prosent av den randomiserte. Men så er traverseringsmønsteret også svært ugunstig.

5.5.3 Sammendrag

Å traversere *data*-tabellen omvendt leksikografisk, så ikke ut til å ha noe effekt på cache-bom eller CPU-tid, hvis vi ser bort ifra administrasjonskostnader. Randomisert traversering gav en meget stor forskjell i cache-bruk. Et konstant forhold mellom cache-bom for den randomiserte og for referansen ble observert; den randomiserte hadde omtrent 4,5 ganger så mange cache-bom som referansen, men resulterte i omtrent 63% økning i CPU-tid.

Det var ikke mulig å finne et fast forhold mellom CPU-tidsforskjellen mellom referansen og den randomiserte og forskjellen i antall cache-bom.

6. ALTERNERENDE TRAVERSERINGSREKKEFØLGE

6.1 Innledning

De sist traverserte data i tabellen *data*, som er en medlemsvariabel i MeshCont, vil være lagret på et høyt nivå i minnehierarkiet relativt til de andre dataene i datatabellen. Aksess av disse dataene vil være rask, og dette ønsker jeg å utnytte ved å traversere data i alternerende rekkefølge. Hvis data traverseres fra første til siste element i én metode i meshen, vil neste traversering (i en annen metode eller i samme metode) være fra siste element til første element i Alternerende Traverseringsrekkefølge. Merk at vi i forrige kapittel studerte omvendt leksikografisk traversering, og sett at dette ikke påvirker minnebruken.

6.2 Implementasjon

Sophus-implementasjon 1.16 ble brukt som utgangspunkt for modifikasjonen. For å holde styr på traverseringsrekkefølgen introduserte jeg en sannhetsvariabel som medlemsvariabel for å holde på forrige traverseringsrekkefølge. Jeg lagde to implementasjoner av denne modifikasjonen. I den ene er valget av traverseringsrekkefølge innebygd i løkketesten og i den andre bruker jeg en if/else-konstruksjon for å finne traverseringsrekkefølge.

Programutsnitt 6.1 Kode-eksempel fra operator = i MeshCont som illustrerer valg av traverseringsrekkefølge og „hack" for å omgå const.

```
        operator = (const MeshCont<T>& mc)
        …
        for (int i = mc.from_start ? 0 : nno-1 ; mc.from_start ? i < nno : i >= 0 ;
mc.from_start ? i++ : i-- )
            {
                IF_ASSERT(i<nno&&i>=0,FATAL);
                data[i] = mc.data[i];
            }

            from_start = !mc.from_start;
            MeshCont<T> *p_mc = (MeshCont<T>*) &mc;
            p_mc->from_start = from_start;
        …
```

Programutsnitt 6.2 Kode-eksempel fra operator = i MeshCont som illustrerer valg av traverseringsrekkefølge ved if/else og „hack" for å omgå const.

```
operator = (const MeshCont<T>& mc)
….
 if (mc.from_start)
     for (int i = 0 ; i < nno ; i++)
        data[i] = mc.data[i];
 else
     for (int i = nno-1 ; i >= 0 ; i-- )
        data[i] = mc.data[i];

   from_start = !mc.from_start;
   MeshCont<T> *p_mc = (MeshCont<T>*) &mc;
   p_mc->from_start = from_start;
…
```

6.2.1 Valg av traverseringsrekkefølge for hver iterasjon („?-implementasjon")

Programutsnitt 6.1 illustrerer hvordan traverseringsrekkefølgevalget foretaes i selve løkke-testen. *from_start* i programutsnittet er sannhetsvariabelen. Løkke-kroppen er uendret. Etter løkken inverterer jeg *from_start*. Jeg brukte også et „hack" for å omgå *const*, slik at jeg slapp å forandre Seismod Iso, men likevel få en god test på hastighetsforbedring ved disse modifikasjonene.

6.2.2 if/else valg av traverseringsrekkefølge („if/else-implementasjon")

Programutsnitt 6.2 nedenfor viser den samme mesh-metoden som programutsnitt 6.1. Her velger jeg altså traverseringsrekkefølge vha. en if/else-konstruksjon.

6.3 Eksperiment

Jeg kjørte 11 Seismod Iso kjøringer og tok CPU tiden for hver av kjøringene. Jeg gjorde dette for kompilering med standard opsjoner, se appendiks I. Jeg gjorde kjøringene så tett i tid at det er trolig at ingen endringer av konfigurasjonen på datamaskinen kan ha funnet sted. Jeg sammenlignet også resultatfilene med de resultatfilene som fremkom fra Seismod Iso med mesh-implementasjon 1.16.

Jeg ville også studere nærmere kjøretidsforskjellene mellom mesh-implementasjon 1.16 av Sophus-koden og Alternerende Traverseringsrekkefølge, så jeg kjørte 10 kjøringer av Seismod Iso med de to nye mesh-implementasjonene og 10 med original-koden (Sophus 1.16), og brukte /bin/time for å finne CPU-tid. For hver av disse testene, brukte jeg 5 til å samtidig telle L1 cache-bom med perfex, som teller datamaskintellere, og de 5 andre til å telle L2 cache-bom (disse to kan ikke telles samtidig).

Jeg studerte også de tre metodene operator=, umapc og umirror. Jeg gjorde CPU-tidsmålinger med 5 kjøringer for hver metode, og målte antall L1 cache-bom, også dette med 5 kjøringer for hver metode.

Eksperimentene ovenfor bruker en 850x620 mesh. Jeg kjørte også 7 kjøringer for hver størrelse av if/else- og original-implementasjon og målte CPU-tid og L1 cache-bom for følgende mesh-størrelser:

mesh-størrelse	Antall mesh-elementer	Antall repetisjoner
25x10	250	421600
100x10	1000	105400
170x124	21080	5000
340x248	84320	1250
510x372	189720	556
680x496	337280	313
850x620	527000	200

Antall repetisjoner angir hvor mange ganger jeg kaller en metode. Jeg kjørte testene for metodene operator=, umapc og umirror. Forholdet mellom antall mesh-elementer og antall repetisjoner er konstant (innenfor avrundingsfeil), slik at antall mesh-elementer som traverseres av hver metode skal være mest mulig likt.

6.4 Resultat

Ved å kjøre Seismod Iso og sammenligne resultatfilene, kunne jeg verifisere at alle kompileringer med de to nye mesh-implementasjonene gav korrekt resultat ved bruk med Seismod Iso. En slik verifisering blir gjort for alle modifikasjoner jeg har lagd.

Jeg bruker som vanlig minimumsverdien av kjøretidene. Fra kjøringene med perfex, fikk jeg sett på cache-bruken til de to modifikasjonene, og sammenlignet dem med originalen. Jeg brukte i hvert tilfelle den kjøringen som gav minst verdi.

Tabell 6.1. Seismod Iso kjøringer for de to modifikasjonene og original

	L1 cachebom	Tidsbruk for L1 cachebom (sekunder)*	L2 cachebom	Tidsbruk for L2 cachebom (sekunder)*	CPU-tid (sekunder)
Alternerende - If/else	3 198 258 353	31,98	198 537 027	1,99	164,5
Alternerende - ?-implementasjon	3 197 442 825	31,97	198 738 758	1,99	273,0
Original	3 245 282 268	32,45	198 079 369	1,98	181,7

„?-implementasjon" er implementasjonen illustrert i programutsnitt 6.1. „if/else" er implementasjonen illustrert i programutsnitt 6.2. Original er kjøring med mesh-versjon 1.16.
* Jeg antar at en cache-bom i L1-cache tar 5 klokkesykler (data må hentes fra L2-cache) og L2-cachebom tar 100 klokkesykler (aksess direkte til minne, se kapittel 2). Testene er foretatt på gridur.ntnu.no, som er en maskin med 500 MHz prosessor, og klarer 500 millioner sykler per sekund.

Under har jeg sammenlignet if/else versjonen av alternerende med original for tre mesh-metoder.

Tabell 6.2. Sammenligning av CPU-tid og cache-bom for 3 mesh-metoder

	Alternerende	Original	Absolutt forskjell mellom Alternerende og Original	Hvor mange prosent Original er av Alternerende (%)
operator= (sekunder)	0,86	0,93	0,07	92,9
umapc (sekunder)	5,31	5,51	0,20	96,4
umirror (sekunder)	31,11	31,45	0,34	98,9
L1 cachebom	91 660 367	92 074 992	414 625	99,6

Målingene av enkeltmetodene gjelder 200 repetisjoner av operasjonen. L1 cache-bom gjelder 200 kjøringer av alle operasjonskallene. Alternerende er if/else-implementasjonen.

Tabell 6.3. CPU-tid og L1 cache-bom for noen 7 forskjellige mesh-størrelser

	CPU-tid for operator= (sekunder)		CPU-tid for umapc (sekunder)		CPU-tid for umirror (sekunder)	
størrelse	**Ori**	**Alt**	**Ori**	**Alt**	**Ori**	**Alt**
250	0,65	0,54	5,30	4,98	30,80	30,49
1000	0,63	0,53	5,31	4,97	30,78	30,51
21080	0,84	0,71	5,57	5,17	30,97	30,82
84320	0,85	0,76	5,64	5,27	31,10	30,94
189720	0,85	0,76	5,57	5,30	31,09	31,05
337280	0,89	0,80	5,64	5,35	31,52	30,56
527000	0,91	0,84	5,51	5,36	31,25	30,86

	CPU-tid for alle de tre metodene (sekunder)		L1-cachebom*	
størrelse	**Ori**	**Alt**	**Original**	**Alt**
250	36,76	36,01	8528	8520
1000	36,71	36,01	27457	31333
21080	37,38	36,70	89028611	79210913
84320	37,59	36,97	91411865	89009787
189720	37,51	37,11	92073572	90896448
337280	38,04	36,71	92292654	91613198
527000	37,68	37,06	92218497	91810065

størrelse i tabellene er mesh-størrelse, Ori står for original, og Alt står for Alternerende.
* L1 cache-bom for alle metodene samt initialisering. I andre L1 cachebom tellinger for enkeltmetoder i denne oppgaven, er initialisering tatt bort.

6.5 Diskusjon

Tabell 6.2 viser resultater for enkeltmetoder. Det ser ut til å være en liten forskjell for hvert metodekall i favør av Alternerende. Den prosentvise forskjellen for operator= er størst. Denne metoden er den mest cache-sensitive metoden. Den traverserer 2 tabeller pr. repetisjon og det er for det meste minneoperasjoner for foregår (kopiering av elementer fra en tabell til en annen), noe som betyr at en forbedring i minnebruken slår større ut for denne metoden enn de fleste andre mesh-metodene. Ved Alternerende blir traverseringsrekkefølgen snudd i begge tabellene for hver repetisjon, og dette gir seg utslag i noe forbedret CPU-tid, og en større forbedring per tidsenhet enn de to andre metodene.

Umapc forandrer alle elementene i meshen vha. en funksjon. Den prosentvise forskjellen mellom en modifikasjon som bare forbedrer minnebruken og den originale vil avhenge av hvor lang tid dette funksjonskallet tar. En større forbedring enn operator= er imidlertid ikke å vente pga. dette funksjonskallet.

Umirror har et kall til speilingsprosedyren i MeshPoint, så dette regner jeg med er grunnen til at den originale prosentvis ligger å tett den Alternerende for denne metoden; en mindre prosentandel av metoden er minneoperasjoner.

Forskjellene mellom CPU-tidene er ganske små, men godt innenfor feilmarginene hvis vi skal bruke de avvikene som er beskrevet for „speedtest” i kapittel 3. Men hvis vi

sammenligner L1 cachebom for Alternerende og Original, så ser vi at forskjellen ikke er så stor. L2 cachebom ble også målt, men denne varierte ganske mye og var under 1/100 av L1 cache-bom, så jeg har ikke tatt med dette. L1 cachebom holdt seg rimelig konstant.

Det som blir kalt Alternerende i tabell 6.2, er mesh-implementasjonen med if/else valg av traverseringsrekkefølge. I denne tabellen ser vi at ?-implementasjonen har en dårligere CPU-tid enn original og if/else, mens cache-bruken er ganske lik, noe som indikerer at minnebruken til de to modifikasjonene er ganske lik. Selv om ?-implementasjonen er mer elegant kodemessig, så har den ytelsesmessig en ulempe, siden testen for traverseringsretning foregår i hver løkkeiterasjon.

CPU-tiden til Alternerende (if/else) er 90,5 prosent av CPU-tiden til den Originale. Dette er litt under de tilsvarende tallene for enkeltmetoder. Seismod Iso er et større program og bruker mer minne enn kall til enkeltmetoder bruker (mange mesher opprettes).

De estimerte tallene for hvor lang tid cache-bom tar er interessante. De viser at den største forbedringen kjøretiden kan få ved en ren forbedring av cachebruk, under de forutsetningene som er beskrevet under tabell 6.1, er 34,4 sekunder. Det vi også ser, er at den lille cache-gevinsten vi får ved Alternerende Traverseringsrekkefølge bare medfører en gevinst på omtrent et halvt sekund, mens CPU-tiden forbedres med omtrent 17 sekunder. Antall L2 cache-bom er omtrent lik, faktisk med en liten fordel til den Originale, så den eneste forklaringen på CPU-tidsforskjellen jeg kan se, er at den Alternerende henter data i større grad fra lokalt minne ved L2 cache-bom enn den originale gjør (jeg regner med at forklaringen ligger i minnebruken, siden det bare er traverseringsrekkefølgen jeg forandrer).

Minne blir håndtert i sider (pages). Sider som ikke er brukt på en stund, kan bli flyttet nedover i minnehierarkiet. Når Alternerende snur traverseringsrekkefølgen, er det liten sannsynlighet for at de siste sidene som er aksessert er flyttet nedover i minnehierarkiet. Dette kan, ved siden av færre L1 cache-bom, være grunnen til at if/else versjonen av denne modifikasjonen har bedre CPU-tid enn Original. En typisk kjøring med Seismod Iso viser også at mye virtuelt minne blir brukt (maksimalt virtuelt minne som blir brukt under en kjøring er ca. 843 MB). Virtuelt er minne som ikke er lagret lokalt i RAM.

6.5.1 Mesh-størrelse, CPU-tid og L1 cache-bom

Tabellen under viser hvor stor prosentandel Alternerende sin CPU-tid og cache-bom er av Original.

størrelse	operator=	umapc	umirror	Totalt*	L1 cache-bom
250	83	94	99	98	100
1000	84	94	99	98	114
21080	84	93	99	98	89
84320	89	94	99	98	97
189720	90	95	100	99	99
337280	90	95	97	96	99
527000	92	97	99	98	100

* For alle tre metodene

Gevinsten av modifikasjonen ser ut til å holde seg for alle størrelsene. operator= ser ut til å ha en CPU-tidsgevinst som er omvendt proporsjonal med meshstørrelsen. Man kan se noe av det samme for umapc, mens det for umirror ikke er mulig å se et slikt forhold.

Forholdet mellom CPU-tiden for Alternerende og Original er påvirket stort sett av umirror, siden den har lengst CPU-tid, så her er heller ikke dette forholdet tydelig.

L1 cache-bom for 250 og 1000 er veldig usikker; det var stor variasjon fra kjøring til kjøring. Dette er trolig forklaringen på at Alternerende har flere cache-bom enn Original for størrelse 1000. For de andre størrelsene, ser det ut som om L1 cache-bom forskjellen blir mindre. Det er rimelig å anta at data i større grad blir lagret lavere i minnehierarkiet (L2-cache eller lavere) etter hvert som størrelsen øker.

7. VIRTUELT SHIFT

7.1 Innledning

Jeg vil i denne modifikasjonen studere effekten av å effektivisere shift-metoden [1,side 51] på bekostning av langsommere aksess av data. Tanken bak et Virtuelt Shift, er å lagre forskyvningen i en MeshPoint-variabel (medlemsvariabel i MeshCont.h) og korrigere for denne verdien når man ønsker å aksessere data „lagret på" et gitt MeshPoint. Håpet er at en raskere shift-prosedyren vil kompensere for langsommere aksessering av data lagret på et gitt MeshPoint og ikke-leksikografisk traversering av data-tabellen.

7.2 Teori

I en implementasjon av Virtuelt Shift, vil data i data-tabellen i MeshCont ikke ligge leksikografisk ordnet hvis ikke forskyvningen er 0. Hvis man skal ha tak i data på et visst MeshPoint eller på en viss leksikografisk posisjon, må man regne om vha. medlemsvariabelen som lagrer forskyvningen for å finne indeksen i data-tabellen. Aksesseringen blir dermed langsommere enn i originalkoden. Jeg vil i denne teoridelen se på noen egenskaper til MeshPoint og prøve å finne frem til en effektiv måte å aksessere data i en Virtuelt Shift-implementasjon.

7.2.1 Akkumulering av forskyvningen

I den originale Sophus-koden, realiseres forskyvningen med en gang. I Virtuelt Shift akkumuleres forskyvningen for deretter å realiseres. Jeg vil nedenfor vise at forskyvning med en akkumulert forskyvning gir samme resultat som trinnvis forskyvning (og at faktisk Virtuelt Shift er en idé som er verdt å forfølge).

Jeg viser først at MeshPoint har en distributiv egenskap.

Lemma 7.1 Distributivitet

Hvis *A*, *B* og *C* er MeshPoint med samme MeshShape, så er

$$A + (B + C) = (A + B) + C\,.$$

Bevis

For at de to sidene skal være like, må vi ha

$$a_i + (b_i + c_i) + n \cdot s_i = (a_i + b_i) + c_i + m \cdot s_i$$

hvor a_i, b_i og c_i er indeks i i hhv. A, B og C, s_i er indeks i i MeshShapen til MeshPointene og n og m er heltall som justerer punktet slik at det er lovlig og befinner seg i intervallet $[0, \mathrm{s_i} >$. $a_i + (b_i + c_i) = (a_i + b_i) + c_i$ er trivielt oppfylt. Men da må vi også ha at $\mathrm{n} \cdot \mathrm{s_i} = \mathrm{m} \cdot \mathrm{s_i}$, siden *i'te* indeks P(i) i et MeshPoint underlagt MeshShapen s_i må ligge i intervallet gitt ved $0 \leq \mathrm{P(i)} < \mathrm{s_i}$ □

Lemma 7.1 brukes til å vise at forskyvning av et akkumulert shift er det samme som å forskyve trinnvist i lemma 7.2.

Lemma 7.2

Gitt et MeshCont *m*. Hvis m_1^n og m_2^n er gitt ved

$$m_1^n = m.shift(P_1 + P_2 \cdots + P_n)$$
$$m_2^n = m.shift(P_1).shift(P_2)...shift(P_n)$$

hvor P_i er et MeshPoint i MeshCont, så er $m_1^n = m_2^n$.

Bevis

Jeg beviser dette ved induksjon.

Fra definisjonene har vi at $m_1^1 = m.shift(P_1), m_2^1 = m.shift(P_1) \Rightarrow m_1^1 = m_2^1$, og basis-tilfellet er trivielt oppfylt.

Vi antar at $m_1^i = m_2^i$. Vi må vise at $m_1^{i+1} = m_2^{i+1}$. m_1^{i+1} og m_2^{i+1} er gitt ved

$$(1)\ m_1^{i+1} = m.shift(P_i^s + P_{i+1})$$
$$m_2^{i+1} = m_2^i.shift(P_{i+1})$$

hvis $P_i^s = \sum_{j=1}^{i} P_j$. På grunn av induksjonshypotesen, kan vi skrive m_2^{i+1} som

$$(2)\ m_2^{i+1} = m.shift(P_i^s).shift(P_{i+1}).$$

Å vise at (1) = (2) er det samme som å vise at $(P_i^s + P_{i+1}) + S = P_i^s + P_{i+1} + S$. Dette følger fra lemma 7.1. □

7.2.2 Indeksering på MeshPoint

Hvis man i den originale Sophus-implementasjonen skal aksessere data lagret på et gitt MeshPoint, konverterer man dette til en leksikografisk posisjon og indekserer i data-tabellen på denne posisjonen. I Virtuelt Shift er eneste forskjellen at man må justere den leksikografiske posisjonen, for deretter konvertere på samme måte til leksikografisk

Grovskisse 7.1 Grovskisse for beregning justering av leksikografisk posisjon *lex* til tabellposisjon ved det akkumulerte shiftet $S = (s_1, s_2, ..., s_n)$.

```
1    MeshPoint P;
2    P.usetLexicographic(lex);
3    P+=S;
4    lex = P.ugetLexicographic();
```

posisjon og indeksere. Å justere et MeshPoint for en forskyvning (lagret som et MeshPoint) gjøres effektivt ved å plusse på forskyvningen, som vist i linje 3 i grovskisse 7.1.

7.2.3 Indeksering på leksikografisk posisjon

Denne konverteringen kan gjøres som vist i grovskisse 7.1 for en leksikografisk posisjon *lex*. Jeg vil i dette avsnittet prøve å finne en mer effektiv måte å finne *indeks(P) = lex(P) + S* enn dette (*indeks(P)* er tabellposisjonen til MeshPoint *P*, *lex(P)* er den leksikografiske posisjonen til *P*, og *S* er den akkumulerte forskyvningen).

En slik optimalisering kan ikke gjøres ved en enkel addisjon eller subtraksjon av indeksene i et MeshPoint. Et eksempel for en MeshShape (4,3) er at (1,2) + (2,1) = (3,0), mens *lex(1,2)* = 5 og *lex(2,1)* = 7 og *lex(3,0)* = 9 (og addisjon eller subtraksjon mellom 5 og 7 gir ikke 9). Grunnen til dette, er at vi har „wrap around" på indeksene isolert, og ikke en oppjustering av punktet til leksikografisk sett neste punkt. Dette betyr at 3 i indeksposisjon 2 justeres til 0 i (3,3), som er summen av (1,2) + (2,1), og ikke (0,0) som ville vært neste leksikografiske posisjon hvis vi hadde „wrap around" på volumet til MeshShapen ((5+7)%12) = 0 = *lex(0,0)*).

For å finne en mer effektiv indeksering, vil jeg se på noen egenskaper ved shift. I lemma 7.3 viser jeg at man kan gjøre forskyvningen i hver retning for seg selv. Senere vil jeg vise hvordan dette gir en mer effektiv måte å indeksere på i Virtuelt Shift enn det som er vist i grovskisse 7.1.

Lemma 7.3

Gitt MeshPoint *P* og *Q*

$$P = (i_1, i_2, ..., i_n)$$
$$Q = (j_1, j_2, ..., j_n)$$

Da er

$$P + Q = P + (j_1, 0, ..., 0) + (0, j_2, 0, ..., 0) + \cdots + (0, ...0, j_n)$$

Bevis

Vi ser at

$$(j_1, 0, ..., 0) + (0, j_2, 0, ..., 0) + \cdots + (0, ..., 0, j_n) = Q \,.\ \square$$

Figur 7.1 Forskyvning av alle elementer i en mesh med MeshShape (3,4,2) og forskyvning (0,1,0).

Grunnbase for alle punkt (2,y,z).

Base

Før		Etter	
MeshPoint	*lex*	MeshPoint	*lex*
(0,0,0)	0	(0,1,0)	2
(0,0,1)	1	(0,1,1)	3
(0,1,0)	2	(0,2,0)	4
(0,1,1)	3	(0,2,1)	5
(0,2,0)	4	(0,3,0)	6
(0,2,1)	5	(0,3,1)	7
(0,3,0)	6	(0,0,0)	0
(0,3,1)	7	(0,0,1)	1
(1,0,0)	8	(1,1,0)	10
(1,0,1)	9	(1,1,1)	11
(1,1,0)	10	(1,2,0)	12
(1,1,1)	11	(1,2,1)	13
(1,2,0)	12	(1,3,0)	14
(1,2,1)	13	(1,3,1)	15
(1,3,0)	14	(1,0,0)	8
(1,3,1)	15	(1,0,1)	9
(2,0,0)	16	(2,1,0)	18
(2,0,1)	17	(2,1,1)	19
(2,1,0)	18	(2,2,0)	20
(2,1,1)	19	(2,2,1)	21
(2,2,0)	20	(2,3,0)	22
(2,2,1)	21	(2,3,1)	23
(2,3,0)	22	(2,0,0)	16
(2,3,1)	23	(2,0,1)	17

Jeg vil nå prøve å finne ut hvordan man skal forskyve i en gitt dimensjon. Figur 7.1 viser hvordan vi „forskyver" i dimensjon 2 for en mesh med MeshShape (3,4,2) og forskyvning (0,1,0). Siden første indeks i forskyvningen ikke forandrer seg, finner vi *lex* etter forskyvning i intervallet $\lfloor lex(x,0,0), lex(x+1,0,0) \rangle$ for alle $x < s_1$, hvor s_1 er indeks nummer 1 i MeshShapen og i intervallet $\lfloor lex(x,0,0), \text{volumet til meshen} \rangle$ ellers.

For et gitt MeshPoint (x, y, z), er $groundbase_{\text{lex}} = lex(x,0,0)$ og $base = lex(1,0,0)$. Groundbase er avhengig av den leksikografiske posisjonen som skal transformeres, og jeg indekserer derfor denne med lex. For en gitt leksikografisk posisjon *lex*, kan jeg subtrahere *lex* med $groundbase_{\text{lex}}$ og dermed havne i intervallet $\lfloor 0, base \rangle$. Deretter kan jeg transformere resultatet på samme måte som jeg vil transformere en *lex* i intervallet $\lfloor 0, base \rangle$, for så å addere med $groundbase_{\text{lex}}$ igjen for å finne den forskjøvne leksikografiske posisjonen. Grovskisse 7.2 viser denne fremgangsmåten.

Base er kun avhengig av hvilken *form* (MeshShape) den aktuelle MeshCont har, og ved en gitt dimensjon *k* er *base* gitt ved

Grovskisse 7.2 Grovskisse for forsyvning av den leksikografiske posisjonen *lex* med forskyvningen $(d_1, d_2, ..., d_n)$ i en MeshCont med MeshShape $(s_1, s_2, ..., s_n)$

```
1       for (int k=1 ; k<=n ; k++)
2           grunnbase_lex = < grunnbase for dimensjon k >;
3           base = < base for dimensjon k >
4           lex -= grunnbase_lex
5           lex += < leksikografisk posisjon til (0,...,0,d_k,0,...,0) >
6           lex %= base
7           lex += grunnbase_lex
```

$$base_k = \prod_{i=k}^{n} s_i$$

Den leksikografiske posisjonen til $(0,...,0,d_k,0,...,0)$ er gitt ved

$$d_k \cdot base_{k+1}$$

hvor $base_n = 1$. *Groundbase* er gitt ved summen av største antall *base*, slik at groundbase er mindre eller lik lex. Groundbase for en gitt dimensjon *k* er gitt ved

$$groundbase_k = \lfloor lex / base_k \rfloor \cdot base_k$$

For å erstatte modulo-metoden i grovskisse 7.2, bruker jeg følgende lemma.

Lemma 7.4

Gitt to MeshPoint $(i_1, i_2, ..., i_k)$ og $(j_1, j_2, ..., j_k)$ med samme MeshShape $(s_1, s_2, ..., s_k)$. Hvis vi definerer *X* og *Y* ved

$$X = (i_1, i_2, ..., i_k) + (j_1, j_2, ..., j_k)$$
$$Y = (i_1, i_2, ..., i_k) - (s_1 - j_1, s_2 - j_2, ..., s_k - j_k)$$

så er $X = Y$.

Bevis: Siden vi har forskyvning med „wrap around", så er en indeks x_i i *X* gitt ved:

$$x_i = (i_i + j_i) \text{ hvis } i_i + j_i < s_i$$
$$x_i = (i_i + j_i) - s_i \text{ hvis } i_i + j_i \geq s_i$$

En indeks y_i i *Y* er gitt ved:

$$y_i = (i_i - s_i + j_i) + s_i \text{ hvis } (i_i - s_i + j_i) < 0$$
$$y_i = (i_i - s_i + j_i) \text{ hvis } (i_i - s_i + j_i) \geq 0$$

De to ligningene for *X* er de samme som for *Y*, så $X = Y$ □

Hvis vi bruker dette lemmaet, og finner den leksikografiske posisjonen til $(0,...,0,s_k - i_k,0,...,0)$ ved

Grovskisse 7.3 Grovskisse for shifting av den leksikografiske posisjonen *lex* med shiftet $(d_1, d_2, ..., d_n)$ i en MeshCont med MeshShape $(s_1, s_2, ..., s_n)$

```
1       for (int k=1 ; k<=n ; k++)
2           grunnbase_lex = < grunnbase for dimensjon k >;
3           base = base_k
4           lex -= grunnbase_lex
5           lex -= < leksikografisk posisjon til (0,...,0, s_k − d_k ,0,...,0) >
6           if (lex<0) lex += base
7           lex += grunnbase_lex
```

$$(s_k - i_k) \cdot base_{k+1}$$

får vi grovskisse 7.3.

7.3 Implementasjon

Jeg vil her beskrive litt av koden i til Virtuelt Shift. Jeg bruker en implementasjon med bare get- og set-data aksess av data som utgangspunkt for Virtuelt Shift, på samme måten som i kapittel 5, men i ugetData og usetData bruker jeg to ugetRealPos0u-funksjoner som overlaster hverandre og er dokumentert nedenfor for å indeksere i datatabellen (C er MeshCont-objektet som metodene opererer på):

1 protected : obs C.funksjon ugetRealPos0u (obs int lex)
Forkrav: lex er en lovlig leksikografisk posisjon i MeshCont C
Resultat: Returnerer posisjonen i data-tabellen til verdien til P.usetLexicographic(lex)

2 protected : obs C.funksjon ugetRealPos0u (obs MeshPoint P)
Forkrav: P er et lovlig MeshPoint i MeshCont C
Resultat: Returnerer posisjonen i data-tabellen til verdien til P

Disse funksjonene konverterer til indeksposisjoner som diskutert i teoridelen.

I tillegg til disse funksjonene har jeg følgende nye (beskyttede) medlemsvariabeldeklarasjoner:

1 MeshPoint mp_shift
Inneholder: Den akkumulerte forskyvningen

2 int mp_shift_ind[MAXDIRECTIONSIZE+1]
Inneholder: Indeksene i forskyvningen

3 int base[MAXDIRECTIONSIZE+1]
Inneholder: Basene i alle dimensjoner for

4 int lexpos[MAXDIRECTIONSIZE+1]
Inneholder: Leksikografiske posisjoner eller reverserte leksikografiske posisjoner som trengs i linje 5 i grovskisse 7.2 og 7.3

5 boolean is_shifted
Inneholder: Usann hviss forskyvningen mp_shift er (0,0,...,0)

Medlemsvariabel 1 brukes til å holde på forskyvningen, og oppdateres i shift-prosedyren i MeshCont. Her oppdaterer jeg også de andre medlemsvariablene, slik at kompleksiteten til ushift blir O(*nsd*), hvor *nsd* er antall dimensjoner til MeshCont-objektet. Dette er for å effektivisere ugetRealPos0u-funksjonene.

I getsetData-versjonen[10] skjer all aksessering av data i data-tabellen gjennom ugetData- og usetData-prosedyrene. Dette er ikke nødvendig i alle tilfeller. En unær map-prosedyre, som mapper alle dataverdiene i en mesh (en MeshCont eller en MeshScalarField) vha. en funksjon som ikke tar argumenter er eksempel på en metode hvor jeg aksesserer data direkte i data-tabellen. Jeg har brukt en slik direkte aksessering i de metodene der dette både er mulig *og* dette er gjort i den originale Sophus-koden.

I noen operasjoner er det ikke mulig å gjøre et slikt direkte oppslag i data-tabellen. Ett eksempel er operator-= i MeshScalarField, hvor to MeshScalarField-objekter subtraheres fra hverandre ved elementvis minus-operasjoner på alle elementene i data-tabellen; jeg må passe på å subtrahere elementer som befinner seg i samme leksikografiske posisjon. I slike tilfeller bruker jeg aksess på samme måte som i getsetData-versjonen.

7.4 Eksperiment

7.4.1 Modulo operasjonen

Jeg ønsker å teste effektiviteten til linje 6 i grovskisse 7.2 mot den samme linjen i grovskisse 7.3 (for å finne ut hvilken implementasjon jeg skal bruke). Dette gjør jeg ved et enkelt testprogram. Dette testprogrammet tester effektiviteten til modulo-operasjonen mot en if-test og en += operasjon. Alle testene ble kjørt på under ett minutt, og fra gang til gang var resultatene de samme, så usikkerheten blir anslått til å være veldig liten ut ifra dette og resultatene i kapittel 3. Jeg brukte 5 kjøringer. Akkurat disse testene ble utført på Ask.

7.4.2 Profil

Jeg ønsker å finne „profilen" til Seismod Iso for Virtuelt Shift koden, dvs. hvor mange ganger metoder i MeshCont og MeshScalarField kalles. SophusDebug modifiseres slik at den teller metodekall isteden for å skrive ut metodenavn når debug-opsjonen PRETRACE settes på. counter.h, beskrevet i appendiks II, brukes til å telle. Jeg teller antall kall totalt til get- og set-data metoder, ushift-metoden og get- og set-data metoder når meshen er forskjøvet.

7.4.3 CPU-tid og cache-bom

Det jeg referer til som Virtuelt Shift i denne seksjonen, er den versjonen som korresponderer med grovskisse 7.3 (på grunn av modulotestresultatene på neste side).

[10] Det jeg kaller getsetData-modifikasjonen er en modifikasjon hvor eneste forskjell fra Sophus 1.16, er at all aksess av data skjer gjennom get- og set-data metoder (som aksesserer data-tabellen dirkete).

Jeg kjører 10 CPU-tidsmålinger, som samtidig måler L1 cache-bom, for umirror, operator= og umapSc0g på en forskjøvet mesh. Forskyvningen er (1,3) og (2,4) for de to meshene i operator= og umapSc0g, og (1,3) for umirror (umirror jobber bare med én mesh.

Jeg gjør samme type kjøringer for ushift-kall for punktene (2,0), (0,1) og (1,3) for de 3 mesh-implementasjonene Virtuelt Shift, Original og getsetdata. umirror, operator= og umapSc0g itererer 200 ganger for alle mesh-implementasjonene, mens forskyvningene itererer 2000 ganger for ushift-kallene for getset-data og Original og 10 millioner ganger for Virtuelt Shift.

Jeg bruker 10 kjøringer av Seismod Iso for Virtuelt Shift og Original. 5 av disse måler L1 cache-bom og 5 måler L2 cache-bom og alle 10 måler CPU-tid.

7.5 Resultat

7.5.1 Modulo

Tabell 7.1 Sammenligning av modulo og if-test

Operasjon	CPU-tid (sekunder)
a=3; a%4	3,23
a=3; a%4	3,26
e=3; if (a<b) e+=5	2,62

a, b og c er variabler av typen int. I siste operasjoner er a=3 og b=5 satt på forhånd.

7.5.2 Profil

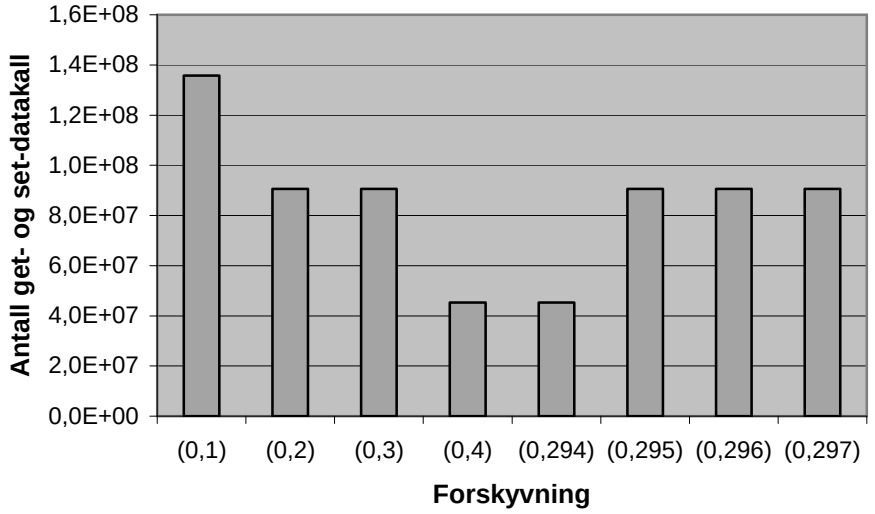
Profilen til en Seismod Iso er gitt ved to tabeller. Tabell 7.2 viser hvor mange ganger en forskjøvet mesh kaller get- eller set-data (altså hvor mange ganger ucreateMapTable trenger å forskyve data). Figur 7.2 viser hvor mange ganger disse metodene blir kalt for en viss forskyvning.

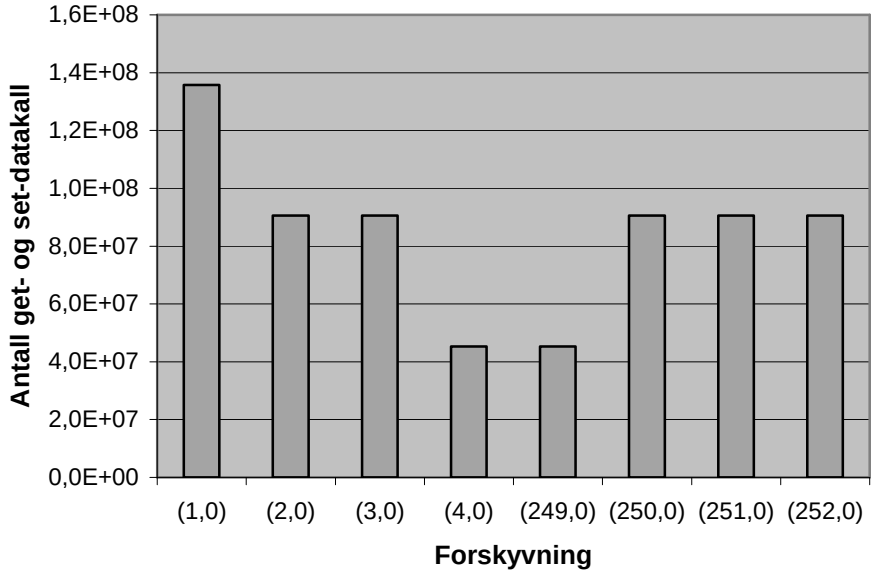
Tabell 7.3 viser hvor mange ganger noen enkeltmetoder blir kalt.

Tabell 7.2 Antall kall til get- eller set-data i en forskjøvet mesh for Seismod Iso

	Totalt for en forskjøvet mesh	For en mesh med forskyvning bare i retning 1	For en mesh med en forskyvning bare i retning 2
Antall kall til get- eller set-data ved forskyvning	1 358 147 516	679 073 758	679 073 758

Figur 7.2 Antall kall til get- eller set-data i en forskjøvet mesh for Seismod Iso





Tabell 7.3 Antall metodekall for Seismod Iso

	shift	setData int	getData int	usetData MP	getData MP
Antall metodekall	18 014	17 503 949 217	19 466 310 727	1 722 156	1 741 354

Tabellen viser antall kall til de to overlastede versjonene av getData- og setData-metodene. int står for den versjonen som tar inn leksikografisk posisjon, mens MP er den versjonen som tar inn MeshPoint.

7.5.3 Enkeltmetoder

CPU-tids- og cachebom-målingene for enkeltmetoder er gitt under

Tabell 7.4 CPU-tid og cache-bom målinger for umirror, operator= og umapSc0g i en forskjøvet mesh og ushift

	CPU-tid (sekunder) for metode			
	umirror	**operator=**	**umapSc0g**	**L1 cache-bom**
get-/set-data	33,04	0,96	35,16	118 709 249
Original	32,26	0,96	31,10	118 709 601
Virtuelt Shift	51,15	9,99	49,99	118 584 306

	CPU-tid (sekunder) for ushift med gitt forskyvning			
	(2,0)	**(0,1)**	**(1,3)**	**L1 cache-bom**
get-/set-data 1000 kall	2,48	2,81	2,57	199 284 369
Original 1000 kall	2,50	2,90	2,60	199 710 888
Virtuelt Shift, 10 mill. kall	2,49	2,54	2,66	547 566

L1 cache-bom i figurene er cache-bom for 3 metodekall samt initialisering av disse (som innebærer få cache-bom). 200 iterasjoner er brukt i metodekallene.

7.5.4 Seismod Iso

Tilsvarende Seismod Iso målinger er vist under.

Tabell 7.5 CPU-tid og cache-bom målinger for Seismod Iso

CPU-tid (sekunder)		L1 cache-bom		L2 cache-bom	
Original	**Virtuelt Shift**	**Original**	**Virtuelt Shift**	**Original**	**Virtuelt Shift**
182,12	351,08	3 245 210 604	3 703 981 853	197 745 747	197 075 894

7.6 Diskusjon

7.6.1 Virtuelt Shift implimentasjonen

Som tabell 7.1 viser, er modulo-operasjonen, selv når den ikke trenger å gjøre noe (3%4=3), mindre effektiv enn en if-setning og en inkrementering. Jeg valgte derfor grovskisse 7.3 som utgangspunkt for videre studier av Virtuelt Shift.

Det er i hovedsak tre ting som utgjør forskjellen mellom en CPU-tidsmåling for Original og Virtuelt Shift (hvis vi ser bort ifra faktorene nevnt i kapittel 3). Det ene er når kall ikke kan gjøres direkte i data-tabellen, men man må gå via get- eller set-datametodene, det andre er forskjellig traverseringsrekkefølge i en forskjøvet mesh og det tredje er forskjell i tidsbruk ved forskyvning av meshen.

7.6.1.1 Kall til get- og set-datametoder

Tabell 7.4 viser at det er en viss forskjell mellom CPU-tidene for get-/set-data-implementasjonen av mesh og Original, særlig for umapSc0g, som for hver iterasjon i traverseringen av data-elementer har kall til både get- og set-data. operator= og umirror viser ingen og liten forskjell. Dette viser at for noen metoder er det en forskjell i CPU-tid i disfavør av Virtuelt Shift selv når meshen ikke er forskjøvet.

CPU-tiden for Virtuelt Shift er betydelig høyere enn for Original. Særlig operator= har en stor prosentvis forskjell (over ti ganger så høy CPU-tid). operator= er som vi har sett tidligere meget sensitiv for aksess i data-tabellen. Forskjellen er stor også for umapSc0g og umirror.

Økningen per kall av get- og set-data er vist i tabell 7.6.

7.6.1.2 Traverseringsrekkefølge

Tabell 7.4 viser at det er veldig liten forskjell når det gjelder L1 cache-bom. Dette kan tyde på at for de forskyvningene som er studert har traverseringsrekkefølgen lite å si. På den andre siden viser Seismod Iso noe færre cache-bom for Original.

7.6.1.3 ushift-metoden

Tabell 7.4 viser CPU-tid for ushift. ushift i Original tar omtrent ti tusen ganger så lang tid som ushift i Virtuelt Shift. CPU-tiden til metodekall i virtuelt shift og get-/set-data subtrahert CPU-tiden til de samme kallene i Original er vist i tabell 7.7.

Forskjellen mellom get-/set-data og Original er positiv (Original bruker lengre tid).

Tabell 7.6 CPU-tidsforskjell mellom getsetData-versjon og Original og forskjellen mellom Virtuelt Shift og Original per kall til get- og set-data

	CPU-tidsforskjell (sekunder)			
	umirror	**operator=**	**umapSc0g**	**gjennomsnitt**
get-/set-data	3,72E-09	9,49E-12	1,93E-08	7,66E-09
Virtuelt Shift	8,96E-08	8,57E-08	8,96E-08	8,83E-08

I tabell 7.4 blir hver metode kalt 200 ganger på en 850x620 mesh. umirror og umapSc0g har 2 kall til get- eller set-data, mens operator= har ett kall til en av metodene per iterasjon.

Tabell 7.7 CPU-tidsforskjell mellom getsetData og Original og forskjellen mellom Virtuelt Shift og Original per kall til ushift

	CPU-tidsforskjell (sekunder) for ushift			
	(2,0)	**(0,1)**	**(1,3)**	**gjennomsnitt**
get-/set-data	2,10E-05	8,40E-05	3,50E-05	4,67E-05
Virtuelt Shift	-2,50E-03	-2,90E-03	-2,60E-03	-2,67E-03

I tabell 7.7 brukes 1000 kall til ushift for get-/set-data og Original og 10 millioner kall til ushift for Virtuelt Shift. Dette for å få en målbar CPU-tid. I alle tilfeller justeres tallene ned til CPU-tid per kall.

Dette regner jeg med skyldes måleunøyaktig. Forskjellen er også svært liten.

7.6.2 Seismod Iso

Som vi ser i figur 7.2, er forskyvningene for Seismod Iso fordelt over et lite spekter, og bare én av dimensjonene er forskjøvet. Hvis forskyvningen er i retning én, vil dette bare medføre at traverseringen får én „wrap around" per traversering av data-tabellen, noe som medfører lite for minne-bruken. En forskyvning i retning to medfører at vi får en „wrap around" for hver førsteretning ved traversering, noe som er mer negativt for minnebruken.

I tabell 7.8 har jeg forsøkt å regne ut minimum CPU-tidsforskjell mellom Seismod Iso med Virtuelt Shift og Seismod Iso med Original implementasjon. Utregningen tar ikke hensyn til CPU-tiden som blir brukt ved kall til get- og set-data i en mesh som ikke er forskjøvet. Dette varierte fra metode til metode, som kall til get- og set-data i get-/set-data-implementasjonen er et mål på. Grunnen til dette er at forskjellen mellom get-/set-data-implementasjonen og Original varierte veldig fra metode til metode, som vi ser i tabell 7.4.

Den målte forskjellen er på omtrent 169 sekunder. Å øke antall kall til ushift per kall til get- eller set-data kan redusere denne forskjellen, men det er tydelig at å gå omveien om get- og set-data-metodene også tar en god del tid, siden den estimerte CPU-tidsforskjellen, som ikke tar hensyn til kall til get- eller set-data metoder utenom når disse skjer i en forskjøvet mesh, er 72 sekunder.

Antall kall til get- og set-data i en ikke-forskjøvet mesh er stort sammenlignet med kall til disse metodene i en ikke-forskjøvet mesh. Disse tallene kan vi komme frem til ved å trekke totalt antall kall til get- og set-data i en forskjøvet mesh, vist i tabell 7.2 fra antall kall til get- eller set-data metodene vist i tabell 7.3. Jeg regner bare med antall kall til de versjonene av get- og set-data som tar inn leksikografisk posisjon, og ikke de som tar inn MeshPoint; det er svært få kall til disse, og jeg regner med at CPU-tidsforskjellen er

Tabell 7.8 Estimering av minimum CPU-tidsforskjell mellom Virtuelt Shift og Original basert på tabell 7.6 og 7.7.

	Kall til get- eller set-data i forskjøvet mesh		**Kall til ushift**		
	Antall Kall	**Estimert CPU-tid (sekunder)**	**Antall Kall**	**Estimert CPU-tid (sekunder)**	**CPU-tids-gevinst (sekunder)**
Virtuelt Shift	1,36E+09	-120	18 014	48,0	-72

Jeg har brukt gjennomsnittsverdiene oppgitt i tabell 7.6 og 7.7 som grunnlag for estimert CPU-tidsforskjell.

mindre siden det bare er snakk om en addisjon mellom to MeshPoint for å justere for forskyvningen av meshen. Fra beregningen av antall kall og CPU-tiden som er vist i tabell 7.6, kommer vi frem til følgende tabell som viser CPU-tidsøkningen ved bruk av Virtuelt Shift som følge av kall til get- eller set-data i en ikke-forskjøvet mesh.

Antall kall til get- og set-data i en ikke-forskjøvet mesh	**CPU-tidstap hvis alle kall til get- eller set-data foretaes gjennom følgende metode (sekunder)**			
	Umirror	**Operator=**	**umapSc0g**	**Gjennomsnitt et av disse 3 metodene**
36 970 259 944	138	0,35	714	283

Tabellen viser oss at det er veldig stor variasjon fra metode til metode. Man kan ikke bruke gjennomsnittet og regne med å få et ganske bra mål på CPU-tidstapet.

7.6.3 Sammendrag

Virtuelt Shift kan ha noe for seg hvis man har mange kall til shift, men raskt går tilbake til en ikke-forskjøvet mesh. Bildet kompliseres noe av at CPU-tiden ikke bare påvirkes av antall kall til shift og hvor mange kall man har til get- og set-data i en forskjøvet mesh, men også av hvor mange kall man har til andre metoder og hvilke metoder man kaller.

Forholdet mellom gevinsten som følge av raskere ushift-metode og tapet pga. tregere kall get- og set-data er 30238. Dette forholdstallet gjelder bare for en 850x620 mesh. Har vi en annen mesh-størrelse, vil forskjellen mellom shift-metoden i Virtuelt Shift og Original forandre seg. Denne metoden skiller seg vesentlig fra de andre metodene i meshen, så det er lite trolig at forandringen vil relativt sett være like stor som forandringen i forskjell mellom CPU-tiden til metodkall i Virtuelt Shift og Original.

Hvis vi fortsatt holder oss til en 850x620 mesh, vil forholdstallet mellom gevinst pga. raskere shift-metode og tap for ett metodekall med ett kall til get- eller set-data per iterasjon (som for eksempel operator=, men ikke umirror og umapSc0g) bli omtrent 0,06. Det betyr at man må ha langt flere kall til shift-metoden enn til disse andre metodene som traverserer data-tabellen gjennom get- eller set-data metodene hvis denne modifikasjonen skal ha noe for seg, til tross for optimaliseringene som ble foretatt.

DEL 4

KONKLUSJON

8. KONKLUSJON

Eksisterende traverseringsmønster og lokale og globale transformasjoner av dette ble studert for en endimensjonal data-tabell i Sophus-biblioteket. Vi så også på variasjonene i kjøretid fra kjøring til kjøring av samme program på SGI Origin.

For enkeltmetoder viste det seg at traverseringsmønsteret allerede var godt – for en stor del lineært – men for speilingsmetoden ble en lokal transformasjon foretatt med godt resultat.

Alternerende Traverseringsrekkefølge er en global transformasjon som snur traverseringsrekkefølgen for hver traverserings av data-tabellen. Dette gav en liten, men signifikant forbedring i kjøretid. Kjøretids-forbedringen skalerte også bra over forskjellige datatabellstørrelser.

En annen global transformasjon, Virtuelt Shift, demonstrerte hvordan en metode som flytter data systematisk i data-tabellen kan elimineres mot en høyere kostnad ved aksess av data. En slik global programmodifikasjon kan ha noe for seg i de tilfellene data traverseres i en liten periode i ikke-suksessiv rekkefølge eller hvis omregningen av aksesseringsposisjon er rask nok. I dette tilfellet viste det seg at aksesseringen ikke var rask nok.

Hvordan metodene bruker minne og hvor mange ganger de blir kalt er vesentlig når noen av de globale transformasjonene skal gjøres. Randomisert data-aksess gav en flerdobling i antall cache-bom. Økningen var lik for forskjellige metoder, men mens én gav en flerdobling i CPU-tid, gav to andre under 30% økning. Dette viser at forskjellige metoder har forskjellig minnesensitivitet.

En slik forskjell ble også påvist for Alternerende Traversering. Alternerende Traversering ville gitt større utslag på kjøretiden om programmet (Seismod Iso) var mer minnesensitivt i forhold til data-traversering, og i Virtuelt Shift ville vi fått en forverret CPU-tid med større minnesensitivitet.

Sammenligning av kjøretiden til dataprogrammer på parallelle datamaskiner med variabel last er et problem som jeg ikke har sett nevnt i litteraturen, selv om problemet er relevant ved sammenligning av programmer på parallelle maskiner med flere brukere Dette problemet ble studert på én maskinarkitektur (SGI Origin). Minimum CPU-tid av noen kjøringer viste seg å være det målet med de beste statistiske egenskaper for en programkjøring på noen minutter både ved høy og lav last for testprogrammet. Dette målet har også den egenskapen at det er den beste tilnærmingen av CPU-tid på en maskin uten last, og det målet som gir mest sammenlignbare CPU-tidsresultater ved forskjellige last-situasjoner, og mye bedre enn målet gjennomsnittet. For en kjøring av et program som tok noen sekunder på å bli ferdig, viste det seg at gjennomsnittet og median var et godt mål på CPU-tid, men variasjonen her var liten fra kjøring til kjøring.

8.1 Videre arbeid

Undersøkelse av traverseringsrekkefølgen, anvendelse av de globale og lokale transformasjonene og de statistiske undersøkelsene av hvordan CPU-tid bør måles kan undersøkes for flere programmer og flere datamaskinarkitekturer. Kjøretiden ved andre datatyper enn double kan også studeres.

Blant de globale transformasjonene, virker særlig Alternerende Traverseringsrekkefølge å være en modifikasjon som enkelt kan benyttes for å forbedre kjøretiden for andre programmer hvor datatabeller traverseres suksessivt.

DEL 5

APPENDIKS

I. APPENDIKS: EKSPERIMENTOPPSETT

I.1 Kompilering

Jeg har i hele oppgaven brukt minimumsverdiene når jeg har kjørt tester med Seismod Iso, og gjennomsnittet når jeg har kjørt tester for enkeltmetoder, i henhold til kapittel 3. Under eksperimentering har jeg brukt MIPSpro kompilator, CC, versjon 7.3.1.3m der annet ikke er nevnt sammen, med SCC versjon 1.10, som er en preprosessor til CC. Kompilatorkommandoen jeg har brukt var

SCC -n32 -I. -IWORKSET -IProgrammes -O -OPT:Olimit=0 <programnavn> -o <ut-fil>

for testing av enkeltmetoder og

SCC -I. -IWORKSET -IProgrammes -O -OPT:Olimit=0 -DTIMING -DDATASIZEX=1050 -DDATASIZEYZ=800 -DEXPLOSIONSIZE=300 -DGEOPHONESIZE=400 -DMAX_SAMPLES_PER_TRACE=50000 -DMINNOPROCESSORS=6 -DMAXNOPROCESSORS=10 -U_FDM_ Seismod-app-cod.C -lm -D_ISO_ -o Seismod-app-cod_Iso

for testing av Seismod Iso. I kapittel 7 var det nødvendig med 64 bits kompilering for at telleren skulle bli stor nok, så her brukte jeg også kompilatoropsjonen –64 og -D_MIPS_SZLONG=64 (det siste for at det matematiske biblioteket math.h).

Med noen få unntak ble eksperimentene kjørt på parallelldatamaskinen Gridur ved Norges Teknisk Vitenskaplige Universitet. Datamaskinen kan forandre konfigurasjon under eksperimentering hvis f.eks. ingeniørene på maskinen gjør endringer i parametere på maskinen, så jeg prøvde å kjøre relaterte eksperimenter så tett som mulig i tid.

I.2 Kjøring av jobber

Når jobber skal kjøres på Gridur, må de legges inn i et køsystem. Jeg brukte enten single-køen eller express-køen for å kjøre jobber. Single-køen er for langvarige jobber som bare bruker én prosessor. Express-køen er for jobber med høy prioritet som kjøres over en kort tidsperiode.

Kommandoen som ble brukt for å sende jobber til jobbkøen, var

bsub –q <kø> -m ”gridur.ntnu.no+” –o <ut-fil> <program/skript>.

-m opsjonen brukte jeg for å tvinge jobben til å kjøre på Gridur.

Typiske kjøreskript for Seismod Iso og for enkeltmetoder er vist under.

Programutsnitt l.1a Kjøreskript for Seismod Iso.

```
#!/bin/csh

@ i = 0
@ n = $2

while ($i<$n)
      /bin/time perfex -e 25 $1 -compute seismod-app-test/* >>& L1_$1.dat
      /bin/time perfex -e 26 $1 -compute seismod-app-test/* >>& L2_$1.dat
      @ i = $i + 1
      sleep 5
end
```

Programutsnitt l.1b Kjøreskript for enkeltmetoder.

```
#!/bin/csh

set files = (`ls alternerende* | grep -v .dat`)

@ i = 0
@ n = $1

while ($i<$n)
  foreach f ($files)
    bsub -q express -o $f.dat -m "gridur.ntnu.no+" perfex -e 25 $f
  end
  sleep 150
  @ i = $i + 1
end
```

Kjøreskriptet i 1a tar seismod-filen som innparameter $1 og antall kjøringer som innparameter $2 og kjører Seismod Iso kjøringene suksessivt. Seismod Iso skriver til fil, og resultatet kan bli påvirket hvis to Seismod Iso kjøringer kjører samtidig.

Kjøreskriptet i 1b tar antall kjøringer som innparameter. /bin/time, som måler bl.a. CPU-tid, er ikke nødvendig her, siden jeg bruker C++-metoden clock() i C++-biblioteket time.h for å måle CPU-tid for hver enkelt metode.

Dette kjøreskriptet kjører flere kjøringer samtidig. CPU-tidsmålinger og L1 cache-målinger av enkeltmetoder var meget stabile, og det virket ikke som om de lot seg påvirke av å bli kjørt samtidig. De skriver heller ikke til resultatfiler slik Seismod Iso gjør.

perfex –e 25 teller antall L1 cache-bom, og perfex –e 26 måler L2 cache-bom. For å måle cache-bruken for en spesifikk metode, laget jeg et modifisert program som bare itererte over en metode, og trakk fra cache-bruken for en referanse hvor alt var likt bortsett fra at denne metoden ikke ble kalt.

II. APPENDIKS: TELLER

II.1 Innledning

I kapittel 7, Virtuelt Shift, har jeg brukt en teller for å telle antall cache-bom. Spesifikasjonen av klassen teller er gitt under.

II.2 Spesifikasjon av Counter

Summary of Class Counter

Declaration:

class Counter;

Uses:

class Element;

Properties: Stores strings and the number of times they are counted.

Purpose

This class keeps track of how many times strings are counted (by calling the procedure count()).

Description

Types

class Counter;

Operations

Constructors:

constructor Counter;

Result: Makes an empty Counter.

Generators:

upd procedure count (obs char* s);
Result: Increments the counter for string s with 1 (starting at 0)

Text operations:

obs procedure print ();

Result: Prints a formated output of the strings and the assosiated number of times it has been counted.

II.3 Listeimplementasjon av teller

Liste-implementasjonen til denne er gitt under.

Programutsnitt II.1 Listeimplementasjon av teller

```
// Counts number of elements with equal key
// Based on a list with a first dummy node

#include <string.h>
#include <iostream.h>

#ifndef counter_h
#define counter_h

#ifndef _MAXSTRING_
#define _MAXSTRING_ 50
#endif

class Element
{
 public:
  char* key;
  unsigned long value;
  Element *next;

    Element()
      {
       key = new char[strlen("")+1];
       strcpy(key,"");
       value = 0;
       next = 0;
      }

    Element(char* k)
      {
       key = new char[strlen(k)+1];
       strcpy(key,k);
       value = 1;
       next = 0;
      }
};
```

```
class Counter
{
private:

   Element *first;

public:
   Counter()
      {
         first = new Element("");
      }

   void count(char* key)
      {
         Element  *e,*p;
         e = find(key,p);
         if (e)
            e->value+=1;
         else
         {
            cout << key << endl;
            e = new Element(key);
            p->next = e;
         }
      }

   void print()
      {
         Element* pivot;
         // Finds the longest value element in the list
         pivot=first->next;
         int wl=0;
         while (pivot)
         {
            wl = (wl <= strlen(pivot->key)) ? strlen(pivot->key) : wl;
            pivot = pivot->next;
         }
         // Prints
         pivot=first->next;
         while (pivot)
         {
            cout << pivot->key << " ";
            int i;
            for (i=0; i<=(wl-strlen(pivot->key)+2) ; i++)
               cout << '.';
            cout << " " << pivot->value << endl;
            pivot = pivot->next;
         }
      }

   ~Counter()
      {
```

```
            print();
            Element *pivot,*next;
            pivot = first;
            while (pivot)
            {
               next = pivot->next;
               delete[] pivot->key;
               delete pivot;
               pivot = pivot->next;
            }
         }

   private:
      Element* find(char* key,Element*&previous)
         {
            previous = first;
            while (previous->next)
            {
               if (!strcmp(key,previous->next->key))
                  break;
               previous = previous->next;
            }
            return previous->next;
         }
   };

   #endif
```

III. APPENDIKS: ANNET ARBEID

III.1 Innledning

I samarbeid med hovedfagsstudent Ann-Kristin Åmo, gjorde jeg noe arbeid som i ettertid viste seg å falle på siden av oppgaven. Jeg vil her gi et overblikk over dette arbeidet og resultatene som vi kom frem til.

III.2 Kompilatoropsjoner

Vi ville se om kompilatoropsjoner førte til høyere hastighet på Seismod Iso versjon 1.16. Testmaskinen var Gridur.

III.2.1 Resultat

O-opsjonen er, som vist i appendiks I, oppgitt under kompilering, og dette er det samme som å bruke O2-opsjonen. Når jeg brukte optimalisering med O3, fikk jeg korrekt resultat med Seismod Iso, men ingen hastighetsforbedring. Hvis jeg brukte Ofast, fikk jeg 5% forbedring, men resultatfilene til Seismod Iso var forandret.

For å oppnå forbedringen med Ofast, men samtidig ikke få forandrede resultatfiler, gjorde vi følgende tillegg i kompilatoropsjoner:

- *-OPT:roundoff=0* gir mindre avrundingsfeil, og dette gav korrekte resultatfiler uten forandringer i hastighetsforbedringen.
- *-OPT:IEEE_arithmetic=1:div_split=OFF:roundoff=0* gir samme regnenøyaktighet som –O2, og fortsatt fikk vi samme hastighetsforbedring og korrekte resultatfiler.
- *-O2 –OPT:alias=typed* (uten –Ofast) gav en 3-4 prosent hastighetsforbedring. *–OPT:alias=typed* betyr at pekere av forskjellig basetype antas å peke til forskjellige objekter. Dette kan gi problemer ved arv.

III.3 Inlining

„Inlining” av en metode medfører at instruksjonene for metoden legges inn i programfilen på de stedene den kalles istedenfor at den legges inn ett sted og man hopper til dette stedet i programfilen hver gang den kalles (hvis den ikke er „inlined”).

C++-nøkkelordet inline er brukt konsekvent foran metodedefinisjoner i Sophus 1.16-implementasjonen (og også den nyere 1.19-implementasjonen). Det er allikevel opp til kompilatoren om den bruker „inlining" eller ikke, og selv om man instruerer kompilatoren om å bruke inlining, er det begrensninger på hvor langt ned i kall-hierarkiet (de metodene en metode kaller) „inlining" skal gå, og også hvor stor programfilen skal bli. Man kan også angi spesifikke metoder som skal bli „inlined". Vi så i hovedsak på „inlining" hvor størrelsen på programfilen var begrensningen.

III.3.1 Resultat

Utgangspunktet var Sophus 1.16-kompilering som allerede hadde en hastighetsforbedring på 5% i forhold til O2, som vist i forrige avsnitt. Det var ingen hastighetsmåling med opsjonen *–INLINE=ON*, men størrelsen økte fra 0,28 MB til 0,3 MB. all_inlined sammen med en størrelsesbegrensning på 100 gav en hastighetsøkning på 5%, og en størrelse på 0,35 MB. Hvis jeg prøvde all_inlined og ingen størrelsesbegrensning, fikk jeg 3-4% hastighetsøkning fra utgangspunktet, og en 1,2 MB stor programfil.

III.4 Konklusjon

Ved bruk av kompilatoropsjoner, fikk jeg en hastighetsforbedring på omtrent 5%. Ved „inlining" fikk jeg ytterligere 5% forbedring. Det så også ut til å være en balansegang mellom programstørrelse og „inlining" og programstørrelse, siden full „inlining" uten programstørrelsesbegrensning gav en mindre hastighetsøkning enn full „inlining" og en begrenset programstørrelse.

REFERANSER

[w1] SAGA-prosjektet, http://www.ii.uib.no/saga

[1] Magne Haveraaen og Sophus-gruppen. *Sophus Library – Selected Specifications and Designs.* 1.7 Sophus-doc.tex, Library date: 4. mars 2000.

[2] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3d edition, 1997.

[3] Magne Haveraaen og Hogne Hundvebakke. *Some Statistical Performance Estimation Techniques for Dynamic Machines.* Norsk Informatikkonferanse – NIK'01, side 176-185, 2001.

[4] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S.-H. Teng. *Automatic array alignment in data-parallel programs.* Proceedings, 20th Annual ACM Symposium on Principles of Programming Languages, side 16-28, 1993.

[5] B.S. Chlebus, A. Czumaj, L. Gasieniec, M. Kowaluk, W. Plandowski. *Algorithms for the parallel alternating direction access machine*. Theoretical Computer Science 2000, Vol 245, Iss 2, side 151-173, 2000.

[6] C. Grelck. *Improving cache effectiveness through array data layout manipulation in SAC.* Implementation of functional languages 2001, Col 2011, side 231-248, 2001.

[7] Magne Haveraaen, Helmer Andre Friis, Tor Arne Johansen. *Formal software engineering for computational modeling.* Reports in Informatics, University of Bergen, Norway, rapport nummer 173, 1999.

[w2] Seismod, http://www.ii.uib.no/saga/Seismod

[8] Magne Haveraaen, Victor Madsen og Hans Munthe-Kaas. *Algebraic programming technology for partial differntial equations.* Norsk Informatikkonferanse – NIK'92, side 183-192, 1992.

[9] Grady Booch, Ivar Jacobson, James Rumbaugh. *Unified modeling language user guide*. Pearson Professional Education, 1998.

[w3] Para//ab ved Universitet i Bergen, http://www.parallab.uib.no

[w4] High Performance Computing ved NTNU, http://hpc.ntnu.no

[w5] SGI, http://www.sgi.com

[10] William Stallings. *Computer Organization and Architecture*. Prentice-Hall International, Inc. 5th edition, 2000.

[11] Seminar av Igor Zacharov. *Origin Optimalisation and Parallelisation Training*, Januar 2002, tilgjengelig i skrivende stund på http://www.nsc.liu.se/files/support/userguide/user_guide_sgi3k.pdf

[w6] David Cortesi, Origin (TM) 2000 and Onyx2® Performance Tuning and Optimization Guide. © 2000-2001, Silicon Graphics, Inc, http://ask.ii.uib.no/ebt-bin/nph-dweb/dynaweb/SGI_Developer/OrOn2_PfTune/