SENSE - delt lager på netværk af arbejdsstationer

Anders Sewerin Johansen

15. august 1999

Abstract I denne opgave gennemgås først det teoretiske grundlag for implementation af distribueret delt lager (DSM) på netværk af arbejdsstationer. Dernæst diskuteres konkrete arkitektoniske og programmeringsmæssige løsninger på kendte ydelsesmæssige problemer, og der argumenteres for at en ren softwarebaseret DSM-implementation vil være ydelsesmæssigt sammenlignelig med de mere udbredte sidebaserede DSM-systemer. Afslutningsvis implementeres et DSM-system — SENSE — der adskiller sig fra størstedelen af de eksisterende DSM systemer ved at være 100% softwarebaseret, ikke kræve understøttelse fra en specialiseret oversætter, have mulighed for tre varianter af lazy release consistency (update, invalidate og hybrid) og være meget portabelt.

SENSE benytter en dynamisk delingsenhed, og lider således ikke under ydelsesproblemer der skyldes falsk deling. Metoden til implementation af dette, de såkaldte Low Level Objects, er inspireret af tidligere arbejde med DSM-systemet ORDER udført på DIKU af Michael Svendsen.

Ydelsesmålinger med benchmarkprogrammet TSP indikerer at SENSE for dette program leverer en ydelse og skalérbarhed, der tilsvarer den der opnås med det sidebaserede DSM-system TreadMarks. Dette resultat er ikke endegyldigt, da det ikke var muligt at udføre en direkte sammenligning på samme testplatform.

Resultater fra kørsler med benchmarkprogrammet 3DFFT giver dårlige resultater, men dette at skyldes at programmet ydviser et ydelseshæmmende tilgangsmønster.

SENSE blev designet, implementeret og verificeret med et tidsforbrug på 5 mandeuger.

Dedikation At denne opgave blev til, skyldes først og fremmest Maz Spork, som fik mig til at se det *interessante* ved DSM-systemer.

At opgaven blev færdig, skyldes først og fremmest min vejleder Jyrki Juhani Katajainen, som udviste en engleblid tålmodighed, kombineret med en evigt usvækket tro på, at selv det umuligt kunne lade sig gøre.

At ydelsesevalueringen blev gennemført skyldes hjælp fra DIKU's DistLab gruppe i almindelighed, og Povl T. Koch og Jørgen S. Hansen i særdeleshed. Jeg skylder dem en stor tak for deres råd og vejledning... og tålmodighed!

For mange lange timer brugt på diskussion af konsistensmodeller, Lamport-tid, ORDER og Low Level Objects skal også lyde en stor tak til Michael Ø. Svendsen.

Indhold

1 Indledning

I dette kapitel præsenteres rationalet bag opgaven, og der redegøres for opgavens opbygning samt den anvendte typografi.

1.1 Rationalet bag denne opgave

En af datalogiens store udfordringer er, og har i mange år været, at finde en effektiv måde at udnytte parallellisme i praksis. Emnet er med tiden blevet mere og mere relevant, da vi tilsyneladende er ved at støde ind i fundamentale grænser indenfor fysik og programmeringssprog. Hvis det hidtidige tempo indenfor udviklingen i pris/ydelse forholdet for datamater skal opretholdes, er det nødvendigt at finde nye måder at forøge datamaternes udeevne på. Da vi med de nuværende programmeringssprog nærmer os grænserne for, hvor megen parallellisme der kan afdækkes maskinelt af oversættere og processorer, er det oplagt at vende blikket mod metoder til udnyttelse af parallellisme på en højere semantisk niveau.

Når man skal behandle dette emne, kan man vælge mellem to meget forskellige indfaldsvinkler: En praktisk og en teoretisk. Den teoretiske tilgang beskæftiger sig med parallelle miljøer på et abstrakt niveau, og vil typisk udmynte sig enten i udvikling af nye maskin- eller programmeringsmodeller, eller en evaluering af eksisterende forskning på området. Den praktiske tilgang vil derimod rette sig mod at forbedre ydelsen for eksisterende parallelle miljøer og programmer.

Jeg har i denne opgave valgt at forholde mig pragmatisk til et eksisterende praktisk problem: Den forholdsvis dårlige ydelse der opnås med eksisterende DSM-systemer (distribueret delt lager, Distributed Shared Memory, DSM), der giver adgang til delt lager på netværk af arbejdsstationer (Shared Memory on Network of Workstations).

Emnet er interessant af flere grunde. Først og fremmest er programmeringsmodellen meget tiltrækkende, da den ikke kræver en større omstilling for en programmør, der i forvejen er trænet indenfor de eksisterende imperative modeller, og har prøvet at udvikle flertrådede — eller multiprocess — programmer. Sekundært fordi der allerede eksisterer en stor mængde programmer, der er skrevet til netop denne programmeringsmodel. Disse eksisterende programmer — ofte (uformelt) kaldet "støvede hulkort" (Dusty decks) — er udviklet til hardwarebaserede DSM systemer, og det er et af målene for DSM på netværk af arbejdsstationer at afvikle disse programmer med tilstrækkelig ydelse på billigere systemer — i dette tilfælde et eksisterende netværk af arbejdsstationer.

Formålet med denne opgave er således at forsøge at komme med en frisk indgangsvinkel på DSM-systemer på netværk af arbejdsstationer. Langt størstedelen af den eksisterende litteratur om udviklingen af DSM-systemer er baseret på evaluering eller implementering af enten en konsistensmodel, en protokol eller en underliggende mekanisme. I denne opgave vil jeg i stedet foretage en analyse med udgangspunkt i de grundlæggende problemstillinger vedrørende distribueret programmering, DSM og arkitektur, med henblik på at evaluere eksisterende løsninger og foreslå nye strategier eller mekanismer.

Analysen forventes at munde ud i et eller flere designpunkter, som siden skal implementeres i form af et fungerende DSM-system. Jeg forventer undervejs at kunne påvise den teoretiske nytte af en hybridiseret invalidate/update protokol, og forhåbentligvis at afslutte med at eftervise en positiv effekt. Andre interesseområder er håndtering af låse og barrierer og den praktiske værdi af oversætterstøtte til distribuerede systemer.

1.2 Opgavens opbygning

Jeg har valgt at opdele opgaven i følgende kapitler:

1. Indledning
2. Baggrund, herunder nomenklatur, beskrivelse af de relevante karakteristika for nutidens datamater, datanet og programmeringssprog og gennemgang af de centrale problemstillinger i parallel programmering på distribuerede systemer.
3. Diskussion af de eksisterende løsninger på de kendte problemer i implementation af Shared Memory på netværk af arbejdsstationer, samt forslag til håndtering af de tilbageværende problemstillinger.
4. Beskrivelse af implementation af SENSE.
5. Evaluering og benchmarking af SENSE.
6. Diskussion af resultater.
7. Konklusion.

Da målet med opgaven ikke blot er at beskrive opbygningen af et nyt DSM-system, men også at give læseren en samlet oversigt over de kendte problemstillinger og eksisterende systemer, vil der være en del gentagelser af fakta. Jeg har derfor lagt specielt vægt på, at de enkelte afsnit i kapitel 2 og 3 i de fleste tilfælde kan læses som en helhed, og de er derfor disponeret som selvstændige og afrundede meningsytringer.

En sideeffekt er, at man kan møde den samme argumentation flere steder i opgaven. Til gengæld håber jeg at teksten kan bibringe læseren det jeg har savnet under arbejdet med opgaven: En samlet informationskilde om implementation og implementationer af sidebaseret delt lager.

1.3 Typografi

I det følgende vil der nødvendigvis blive brugt en del forkortelser og fagudtryk. Når et specialudtryk optræder for første gang vil det blive præsenteret således:

`<dansk beskrivelse> (<fagudtrykket>,<forkortelsen>)`

Eksempelvis:

> "udveksling af beskedder (Message Passing, MP)"

2 Introduktion til datamatarkitektur og distribueret delt lager

I det følgende gives en kort introduktion til distribueret delt lager (Distributed Shared memory, DSM) og moderne datamatarkitektur. For arkitekturens vedkommende gives en kort gennemgang af de væsentligste egenskaber i nutidens processorer, hukommelseshierakier, datanet og programmeringssprog i det omfang det er relevant for implementation af distribuerede parallelle miljøer. Der bliver beskrevet de grundlæggende problemstillinger i parallel programmering og DSM, og hvilke tiltag der er blevet forsøgt for at håndtere disse.

2.1 Nomenklatur

Da jeg i denne opgave benytter en del termer, som ikke normalt optræder i litteraturen om emnet, er det på sin plads at give en kort forklaring her, som læseren kan referere tilbage til.

De fleste termer og forkortelser, der optræder i det følgende, vil blive forklaret første gang de tages i brug, eller antages at være i så gængs brug, at deres mening er kendt. Som det fremgår af definitionerne, har jeg gjort meget ud af at adskille DSM– og synkroniserings-mekanismer, da disse ikke altid er afhængige af hinanden.

Det har her ikke været muligt at undgå at bruge termer og forkortelser i forklaringerne, som optræder senere i disse afsnit. Det kan derfor anbefales at læse samtlige disse definitioner i deres helhed.

2.1.1 Generelle termer vedrørende DSM

Distribueret bruges i denne opgave i sin mest altfavnende betydning. Her betragtes f.eks. kommunikation med en ydre enhed (program og disk, eller CPU og hovedlager) som distribueret, da det involverer to separate arbejdsopgaver, kommunikation og synkronisering.

Knude er et element der indgår i en distribueret beregning. Knudens beskaffenhed (program, ydre enhed, andet) vil ikke blive beskrevet nærmere, hvis det ikke har relevans for den sammenhæng udtrykket bruges i.

Distribueret miljø er de omgivelser der tilvejebringer de grundlæggende faciliteter der skal være til rådighed for at distribuerede programmer kan udvikles og udføres. Et minimum af faciliteter er en mekanisme til at udveksle information mellem de involverede knuder, og en mekanisme til synkronisering af knudernes arbejde.

Distribueret delt hukommelse (Distributed Shared Memory, DSM) bruges i denne opgave primært som beskrivelse af den del af et distribueret miljø, der varetager udveksling af information mellem knuder.

DSM-system er et distribueret miljø med et specifikt formål, nemlig at stille faciliteter til rådighed der specielt retter sig mod at kunne udføre parallelle beregningsopgaver så hurtigt og effektivt som muligt. Et DSM-system stiller både DSM— (dvs. informationsudveksling) og synkroniserings–faciliteter til rådighed for knuder. Bemærk at et DSM-system kan stille flere DSM-strategier til rådighed for programmøren. Eksempelvis giver ORDER [18] adgang til to strategier — Shared Memory og Message Passing — så programmøren har mulighed for trinvist at raffinere sit program ved anvendelse af MP for højere ydelse.

DSM-strategi er den overordnede plan, eller *paradigmet*, for informationsdeling mellem knuderne. De mest udbredte strategier er Shared Memory (SM), Message Passing (MP) og Mobile Objects (MO). DSM-strategien kan være baseret på antagelsen om at en specifik synkroniseringsmekanisme er til rådighed i DSM-systemet, og dette er typisk tilfældet. Der er dog eksempler på strategier der er baseret på en speciel synkroniseringsmekanisme bliver brugt i DSM-systemer der stiller yderligere synkroniseringsmekanismer til rådighed. Et eksempel på dette er et DSM–system der baserer sig på MP paradigmet, hvor meddelelsesudveksling er synkroniseringsmekanismen, men som yderligere stiller synkroniseringsprimitiver såsom låse, barrierer og semaforer til rådighed for programmøren. Disse ekstra synkroniseringsfaciliteter er ikke essentielle for strategiens korrekthed, men er at betragte som et tilbud til programmøren.

DSM-mekanisme er implementationen af den overordnede DSM-strategi. Eksempelvis kan strategien være SM, og den underliggende mekanisme er Lazy-release protokol med multiple skrivere, implementeret ved brug af diffs og MMU funktioner, som man ser i TreadMarks [2]. Bemærk at dette er et relativt term, da mekanismen i en MO-strategi f.eks. kan være at benytte et underliggende SMLB system, altså at det overordnede DSM-system (MO) benytter det underliggende DSM-system (SMLB) som mekanisme.

DSM-protokol er den del af DSM-mekanismen der sikrer at strategien overholdes. DSM-protokollen er reglerne for hvilke meddelelser der kan udveksles mellem knuderne i DSM-systemet, og hvornår disse kan bruges. Bemærk at en DSM-strategi kan være baseret på antagelsen om at global kommunikation (broadcast) ikke er nødvendig, selvom den underliggende DSM-protokol benytter denne kommunikationsform.

Synkronisering er betegnelsen for den handling at sikre to eller flere knuder udfører en del af deres arbejde på veldefineret vis. Dette gøres ved at give programmøren adgang til en metode til at få en eller flere knuder i et parallelt system til at vente på hinanden på et givent punkt i deres programudførsel.

Synkroniseringsmekanisme er paradigmet for synkronisering i et DSM-system. Der eksisterer både eksplicitte mekanismer (såsom låse) og implicitte mekanismer (Message Passing). Jeg har valgt ikke at formalisere synkronisering på samme måde som de ovennævnte DSM-termer, da der ikke hersker tilsvarende forvirring om brugen af udtrykkene. Jeg betragter det således som acceptabelt at undlade opsplitning i synkroniserings-strategi og —mekanismer.

Load balancing (LB) er et begreb der dækker over arbejdsfordelingen på knuderne der samarbejder om at løse en opgave i et distribueret miljø. Som "dårlig" LB betragtes en situation, hvor arbejdsfordelingen er skæv, og en, eller meget få, knuder udfører arbejde imedens størstedelen er inaktive.

2.1.2 Alment brugte forkortelser

Her opremses kort de forkortelser jeg benytter mig af uden forklaring, men hvis betydning det er vigtigt at kende for at kunne følge med i det følgende.

SM er en DSM-strategi, som benytter delt lager (Shared Memory) som programmeringsparadigmet. Her udveksles information mellem knuderne ved skrivninger til et delt lager, og altså uden explicit udveksling af information mellem knuderne.

SMLB er den klasse af SM strategier som antager at *låse og barrierer* er synkroniseringsmekanismen i DSM-systemet. Stort set alle SM implementationer benytter denne strategi.

MP er en DSM-strategi der benytter Message Passing, dvs. eksplicit kommunikation mellem knuder som både DSM— og synkroniserings–mekanisme.

MO er en DSM-strategi der benytter mobile objekter (Mobile Objects) som DSM-strategi.

NW — Network of Workstations. Brugt som i SMLB, NW, hvilket betyder et SMLB-baseret DSM-system på et netværk af arbejdsstationer

HW — HardWare-based. Brugt som i SMLB, HW, hvilket betyder et SMLB-baseret DSM-system på specialdesignet hardware.

2.1.3 Andre termer

CPU bruges her kun til at beskrive en processor, og er altså ikke et synonym for arbejdsstation, arkitektur eller lignende.

Processor bliver brugt som synonym for CPU.

Uniprocessor er betegnelsen for en datamat, der på et givent tidspunkt kun har én aktiv instruktionsstrøm. Mere uformelt er det en "almindelig" arbejdsstation med en enkelt CPU, hvis arkitektur (hardware og OS) ikke er baseret på distribueret programmering.

Arbejdsstation bruges som synonym for uniprocessor.

Imperative sprog er den oprindelige, eller "klassiske", type af programmeringssprog, såsom FORTRAN, C, assembler og Pascal. De er kendetegnet ved at være baseret på en overordnet tanke om programmering som værende at sammensætte en detaljeret beskrivelse af problemløsningen i form af på hinanden følgende ordrer. Til forskel kan nævnes funktionsprogrammeringssprog som Miranda og ML.

API/Programmeringsparadigme , eller bare *paradigme*, er den model for problemløsning programmøren præsenteres for af et givent udviklingsmiljø. I denne opgave typisk en kombination af programmeringssproget, samt DSM-systemets faciliteter.

Latenstid eller *opstartstid* er den ventetid der optræder fra et program eller en enhed rekvirerer udførslen af en given opgave, og til denne opgave iværksættes. Udtrykket bruges typisk i forbindelse med læsninger og skrivninger til RAM, netværk eller harddisk.

Datanet eller blot *netværk* eller *net* bruges i denne opgave som generel betegnelse for lokalnet som Ethernet eller ATM.

Protokol er formelt betegnelsen for en række handlinger, tilstande og regler for transition mellem tilstande, som definerer reglerne for samarbejde mellem to eller flere entiteter. Udtrykket bruges mest til at beskrive regler for kommunikation, eksempelvis TCP-protokollen.

Oversætter (engelsk: compiler). Et program der oversætter fra et sprog til et andet. Her brugt i forbindelse med oversættelse fra højniveausprog som C eller Pascal til maskinkode.

Brugerprogram er et program, der bliver eksekveret af et operativsystem eller — i denne opgave — et distribueret miljø. Eksempelvis det parallelle benchmarkprogram TSP, udført på et SMLB DSM-system.

IO er en forkortelse for Input/Output. Dette dækker over at ind- eller udlæse data fra eller til en ydre enhed, såsom en terminal eller en harddisk.

TLB er en forkortelse for Translation Look-aside Buffer. Dette er en mekanisme i CPU'en, der håndterer oversættelse af virtuelle lageradresser til faktiske lageradresser.

RAM er det lager på en arbejdsstation som indeholder de programmer der eksekveres, og deres arbejdssæt af data.

Cache er et lille, hurtigt lager, som arkitekturmæssigt ligger mellem f.eks. CPU og lokalt RAM lager.

2.2 Hvorfor DSM på arbejdsstationer?

Hvis man vil have løst et beregningstungt problem, og man ikke er tilfreds med ydelsen på en uniprocessor har man grundlæggende tre valg:

- Vent på at en hurtigere uniprocessor er tilgængelig, eller billig nok.
- Optimering af det eksisterende program.
- Forsøg at få flere uniprocessorer til at samarbejde om at løse problemet.

I mange tilfælde er den sidste mulighed den mest tillokkende, da visse problemer enten ikke kan løses med tilstrækkelig ydelse på eksisterende (eller fremtidige) uniprocessorer eller fordi den eksisterende hardware allerede er tilgængelig. Optimering af et eksisterende program kan være en meget tidskrævende process, og kan i mange tilfælde ikke udføres uden at medføre en sænkning af præcissionen i beregningerne [4, column 5].

DSM-systemer er udviklet med henblik på at kunne stille et distribueret programmeringsmiljø til rådighed for programmører — primært med henblik på denne slags opgaver, som uformelt kaldes "talknusning". Der eksisterer både specialdesignede DSM-maskiner (hardwarebaserede), softwareløsninger og en hel række af hybrider imellem disse to ekstremer. Der er også en lang række forskellige — og i nogen tilfælde konkurrerende — paradigmer til rådighed for programmører på disse DSM-systemer. Formålet med disse DSM-systemer er dog altid det samme: Enten at udføre distribuerede beregninger med så høj ydelse som muligt (praktisk brug), eller at forske i teorier eller mekanismer der kan forbedre ydelsen af fremtidige DSM-systemer.

2.3 Ydelse, hvad er det?

Ydelse er et begreb, som der gennem tiderne har været mange definitioner af. Problemet er, at ydelse ikke kan defineres objektivt. For eksempel risikerer man at ignorere responstid, hvis man baserer definitionen af ydelse på jobgennemstrømning [6, p. 50]. Det er altså i forvejen en ikke-triviel opgave at definere ydelse på en uniprocessor.

På distribuerede systemer indføres der yderligere en række parametre, man kan vælge at definere ydelse fra. De vigtigste af disse er *skalérbarhed* og *speedup*. Speedup er den faktor eksekveringstiden for et givent program og inddatasæt formindskes med, når det udføres på det parallelle system. Skalérbarhed er et givent distribueret systems evne til at opnå speedup ved at øge mængden af knuder i systemet. Man kan altså tale om, at et system opnår et maksimalt speedup X ved brug af Y knuder, for programmet Z med inddatasættet W. I så tilfælde skalerer systemet for programmet Z med inddatasættet W til Y knuder.

Uheldigvis er en sammenligning af DSM-systemer alene på grundlag af speedup og skalerbarhed ikke mulig, da man risikerer at ignorere eksekveringstiden. Man kan jo næppe sige at system A er bedre end system B fordi det kan skalere på flere knuder, hvis system B med færre knuder løser problemet på den halve tid.

Jeg har derfor valgt at benytte samme definition på ydelse, som Hennessey og Patterson anvender i "Computer Architecture: a Quantitative Approach". Det er ganske simpelt som følger:

> Ydelse er et udtryk for, hvor hurtigt *i tid* et givent program bliver udført på en given platform. Hvis det bliver udført hurtigere på platform A end på platform B, har platform A en bedre ydelse for dette program [6, p. 18].

Denne simple og yderst pragmatiske definition har en del implikationer, der ikke umiddelbart er åbenlyse. Den fremmer først og fremmest en *kvantitativ og helhedsorienteret* analyse af en platform — i denne opgave et DSM-system. Kvantitativ, da det væsentlige er om vi kan måle en forskel på to DSM-systemer i form af hurtigere eksekvering af programmer. Helhedsorienteret, fordi det i denne sammenhæng kan være en fejl at basere en vurdering af et DSM-system på målinger af en enkeltstående funktions ydelse, såsom evnen til at overgive kontrol over en kritisk region til en anden knude (låsehåndtering). Ydelsesmæssigt kan det system, der har den "dårligste" låsehåndtering, sagtens være det mest attraktive. Den relativt langsomme låsehåndtering kan for eksempel skyldes et design, der tilbyder bedre skalérbarhed på bekostning af langsommere håndtering af låse!

Tilbage står kun at vælge det programmer og inddatasæt der skal lægges til grund for tidsmålingen. Her anvender Hennessy og Patterson SPEC's samling af benchmarks [6, p. 21], da de menes at repræsentere de mest almindelige former for programmer man kunne ønske at udføre på en uniprocessor. Indenfor distribuerede systemer anvender man normalt en række programmer, der er udvalgt efter de samme kriterier — de repræsenterer et bredt udsnit af de opgaver, og dermed programmer, som man i praksis benytter DSM-systemer til at løse. De mest brugte programmer er følgende:

n-body , som simulerer n legemers bevægelser i et tyngdefelt.

Fast Fourier Transform eller *FFT*, som udfører Fouriertransformation.

Traveling Salesman Problem eller *TSP*, som finder den korteste tur over vægtede veje gennem en række byer.

Water er en simulation af molekylers bevægelse i vand.

Alle fire problemer vides teoretisk at kunne parallelliseres effektivt.

Der er ikke tale om en etableret standardsamling af testprogrammer, men snarere en de-facto standard, i den forstand at nogen systemer er testet med flere programmer [10, pp. 39..44], og andre med færre [18, p. 67]. Disse fire programmer er dog gennemgående i næsten alle evalueringer af parallelle systemer, da deres opførsel er meget velkendt, og har vist sig at være repræsentativ for næsten alle former for programmer i DSM-miljøer.

2.4 Nutidens datamatarkitektur

Det er en gammel sandhed indenfor enhver form for programmering, at man må tage hensyn til de styrker og svagheder den foreliggende platform har, hvis man vil opnå optimal ydelse. På intet tidspunkt i datalogiens historie har dette været så sandt som i dag. Nutidens datamater og netværk bliver udviklet med henblik på at udføre visse former for opgaver så effektivt som muligt — af og til på bekostning af de mindre hyppige opgaver. Det er derfor på sin plads her at gennemgå nutidens datamater, datanet og programmeringssprog. Denne gennemgang bliver på ingen måde udtømmende, og vil snarere fokusere på de arkitekturmæssige karakteristika der har relevans for opgavens emne. Specielt vil gennemgangen af datanet være koncentreret om lokalnet, da internetværk stort set aldrig bruges til at forbinde knuderne i et DSM-system.

2.4.1 Datamater

Hvis man skal fremhæve de væsentligste udviklinger idenfor datamatarkitektur i de sidste 20 år, må der specielt være to områder der er af interesse:

- Hukommelseshierakiet
- Instrukssionssæt

For at starte med hukommelseshierakiet, har udviklingen især været præget af to ting: Hukommelse er blevet billigere, og forskellen mellem CPU hastighed og almindelig RAM er vokset betragteligt. Dette har ført til udstrakt brug af cachelager, ofte i flere niveauer. Hvor en almindelig arbejdsstation i prislaget kr. 10.000 – 20.000 i 1980 var udrustet med 1-2 MB hukommelse og ikke havde nogen cache mellem CPU og RAM, vil en arbejdsstation i samme prisklasse i dag (1999) være udstyret med 128-256 MB lager, og have mindst to niveauer af cache mellem CPU og RAM.

Cachelageret er af absolut nødvendighed for at opnå en god ydelse på moderne datamater, da forskellen i hastighed mellem læsning fra cache og almindeligt lager kan være en faktor 20-30 [6, p. 384]. Grunden til at datamater for 10-20 år siden sjældent var udstyret med CPU cache var, at læsning i RAM typisk kunne udføres med næsten samme hastighed, som tilgang til registre i processoren. Et cachelager var altså ikke nødvendigt på disse maskiner.

Et konkret eksempel er en sammenligning mellem Intels 80386 og Pentium Pro processorer. 80386 blev introduceret i oktober 1985 med en clockfrekvens på 16MhZ, havde ingen cache og blev typisk brugt i arbejdsstationer med 1-2 MB RAM. Pentium Pro blev introduceret i november 1995 ved 150MhZ, havde to niveauer af cache indbygget på chippen (16Kb 1. level og 256Kb 2. level) og blev typisk brugt i arbejdsstationer med 64-128 MB RAM. Tidsmæssigt er der ca. 10 år mellem dise to typer arbejdsstationer. Ydelsesmæssigt nærmer forskellen sig en faktor 50, selvom forskellen i clockfrekvens kun er omkring en faktor 10. Dette er i parantes bemærket stort set i overensstemmelse med Moores "lov", der indikerer at pris/ydelse forholdet for datamater forbedres med en faktor to på ca. 18 måneder. Begge processorer implementerer det samme instruktionssæt, men på Pentium Pro er der en markant større forskel i eksekveringstid mellem den hurtigste og langsommeste instruktion. Således er Pentium Pro's ydelse langt mere afhængig af sammensætningen af de instruktioner den eksekverer.

De fleste moderne processorer benytter et langt mere simpelt instrukssionssæt end Intels 80386 arkitektur, da det er nemmere for oversættere at generere effektiv maskinkode, når arkitekturen er simpel og ortogonal. Dette gælder bla. CPU'er som Alpha, MIPS, SPARC og PA-RISC. Udviklingen imod simplere instruktioner og forøget afhængighed af oversætteren fortsætter tilsyneladende, hvilket Intel og HP's Merced chip er et glimrende eksempel på. Med mindre, simple instruktioner, dybe pipelines, hurtig cache og gode oversættere er det muligt at opnå høj ydelse med de mest brugte typer af applikationer. Til gengæld er den relative ydelsesnedsættelse ved brug af dårlige oversættere, eller eksekvering af programmer der udviser cache-fjendtlig opførsel stigende, da de fleste applikationsprogrammer (og dermed benchmarks) ikke udviser denne opførsel.

Hvad er så cache-fjendtlig opførsel? En cache er opbygget ud fra en antagelse om, at programmer udviser temporær og spatiel lokalitet i deres lagertilgang og eksekveringssti. Ydermere udløser et kontekstskift en tømning af både cache og TLB-registre, da mange caches benytter *virtuelle* adresser, som ikke er gyldige i et andet hukommelsesdomæne — det være sig ved skift til kernen eller en anden brugerprocess [20, pp. 563..564].

Der er som nævnt ingen tegn på, at denne udvikling vil ophøre. Specielt går udviklingen indenfor CPU-design i retning af Very Long Instruction Word (VLIW) og dybere pipelines. Hukommelsestyper der indfører højere båndbredde end normal RAM på bekostning af latenstiden — RAMBUS — bliver allerede brugt på visse arbejdsstationer.

For at opsummere: Det er karakteristisk for nutidens datamater, at de er ekstremt afhængige af karakteren af de programmer de udfører [6, p. 29]. Ansvaret for at levere høj ydelse er i langt højere grad end tidligere langt på oversætterne [6, p. 6]. Specielt kan ydelsen skades af cache-fjendtlig opførsel, dvs. programmer der ikke udviser temporær og spatiel lokalitet, fylder cachen med data der ikke er relevante for programudførsel i den nærmeste fremtid eller ugyldiggør cachelageret ved at skifte til et andet lagerdomæne.

2.4.2 Datanet

De former for datanet der er i almindelig brug i dag, adskiller sig kun fra dem man brugte for 20 år siden på et punkt: Båndbredden — hvor mange bits der kan sendes pr. sekund (Mbit/sek eller blot Mbit) — er højere. Mere præcist er udviklingen gået fra 10 Mbit Ethernet til 100 Mbit Ethernet. Der er altså rent hardwaremæssigt tale om ikke-forbindelsesorienteret kommunikation over et delt medie, som giver mulighed for enten punkt-til-punkt eller broadcast kommunikation. Alle andre former for kommunikation over Ethernet vil blive håndteret af protokoller, men er altså ikke den "naturlige" kommunikationsform netværket understøtter. Mere moderne former for datanet, såsom det cellebaserede ATM-system, har endnu ikke slået igennem i stor stil. Således er det endnu ikke normalt at datanettet som sådan stiller faciliteter som garanteret båndbredde (quality of service, QoS) og gruppekommunikation (multicast) til rådighed.

Den form for kommunikation der er mest nyttig indenfor distribueret programmering — forbindelsesorienteret en-til-en kommunikation over en stabil kanal — bliver altså for de fleste arbejdsstationers vedkommende stillet til rådighed af den del af operativsystemet, der håndterer kommunikation over datanet: protokolstakken. Den mest udbredte protokol med de nævnte egenskaber er TCP protokollen, der simulerer en kanal af denne type gennem brug af IP protokollen, der er forbindelsesløs.

Det er langt fra trivielt at implementere en protokol af denne type effektivt, og det har da også taget lang tid for TCP at opnå sit nuværende meget høje niveau af effektivitet. I dag opnår man med TCP en udnyttelse af båndbredden på kanalen der nærmer sig det teoretiske maksimum i langt de fleste tilfælde [20, p. 567].

Udveksling af data via datanet adskiller sig fundamentalt fra skrivning og læsning til RAM på tre punkter: API, latenstid og skjulte ydelsesmæssige omkostninger.

Når et program udveksler data med det lokale lager — RAM — sker det med en implicit antagelse af, at der kun er én mulig modtager. I kommunikation over netværk er det langt fra tilfældet, da modtageren kan være en, flere eller alle maskiner tilknyttet nettet. Dette faktum afspejler sig selvfølgelig i de funktioner der stilles til rådighed for programmøren. Både transmission og modtagelse af data via datanet kan udføres *synkront* eller *asynkront* — også kaldet *blokerende* og *ikke-blokerende*. Ved blokerende kommunikation sættes programmet i dvale af operativsystemet indtil transmissionen er overstået, hvorimod programmet ved asynkron kommunikation kan fortsætte eksekveringen, og således overlappe kommunikation og beregning. Til gengæld stiller brugen af asynkron kommunikation større krav til programmøren, da ansvaret for at sikre at transmissionen er overstået ikke varetages af operativsystemet og protokolstakken. Forskellen på disse to metoder er illustreret i pseudokode i afsnit A.1.

Den høje latenstid på kommunikation over netværk i forhold til læsning og skrivning til lokalt lager skyldes både de langt større afstande, og at den basale kanal ikke er stabil. Der er ingen garanti for at data bliver transmitteret fejlfrit over et datanet, og ved forbindelsesorienteret kommunikation skal man endvidere etablere at modtageren overhovedet eksisterer, og er i stand til at modtage. Alt dette medfører at latenstiden for transmission over net typisk er to til tre størrelsesordener højere end for kommunikation mellem processor og lokalt lager på en arbejdsstation.

De skjulte omkostninger ved kommunikation over net, skyldes at både transmission og modtagelse af data delvist varetages af et program — protokolstakken — i begge ender af kanalen. Der udløses altså mindst et kontekstskift — typisk til kernetilstand — og både afsender og modtager afbrydes i deres andre gøremål.

Disse tre problemer optræder ikke kun i forbindelse med Ethernet, men vil også være aktuelle med andre typer af datanet.

2.4.3 Programmeringssprog

Indenfor programmeringssprog har de seneste 20 år budt på hele to revolutioner — objektorientering og funktionsprogrammeringssprog — uden det har haft den store indflydelse på de sprog den almindelige programmør bruger. Der er stadig tale om imperative sprog, omend fokus er skiftet fra Pascal og FORTRAN til C og C++. Det er dog et beklageligt faktum at de færreste programmører udnytter muligheder som nedarvning, og i stedet kun benytter objektorienterede faciliteter såsom indkapsling. Der er altså udenfor de deciderede datalogiske forskningsmiljøer i bund og grund tale om den samme slags programmer, som man har skrevet de sidste 30-40 år.

Brugen af denne type sprog medfører en lang række ydelsesmæssige problemer, primært i forbindelse med oversættelsen til maskinkode. Da de giver adgang til sprogfaciliteter, der har såkaldte *sideeffekter*, hæmmer de oversætterens muligheder for at afdække et programs læse- og skrivemønstre (dataflow) gennem statisk analyse. Et klassisk problem er de såkaldte *aliaseffekter*, som ofte optræder i forbindelse med indirekte reference til lageret (dereferering). Nedenstående eksempel i pseudo-Pascal illustrerer problemet:

```
Program Aliaseffekt;

Var
  a, b : Integer;

function alias(var n) : Integer;
Begin
  n := n+1;
End;

Begin
```

```
  a:=1;
  b:=2;
  alias(a);
  alias(b);
  Writeln(a, b);
End.
```

I funktionen alias er det på oversættelsestidspunktet umuligt at afgøre om *n* refererer til *a* eller *b*. Det er derfor umuligt for oversætteren at optimere den resulterende maskinkode ved at placere nogen af variablene i registre. Der er tale om et særdeles velkendt problem, som har plaget konstruktørerne af oversættere i mange år [1, pp. 648..660], [3, sektion 17.5], [6, p. 94]. Problemet kan løses i visse specifikke tilfælde — i dette eksempel f.eks. ved at *inline* to udgaver af funktionen alias — men der eksisterer ikke nogen generel løsning. I praksis optræder disse effekter så hyppigt, at det normalt er umuligt at udføre registeroptimering på tværs af kodefragmenter (basic blocks). Disse kodefragmenter er typisk af størrelsesordenen 20-50 maskinkodeordrer.

Den eneste kendte løsning er, at benytte programmeringssprog, hvor denne type effekter ikke kan opstå. Dette er netop de såkaldte *rene funktionsbaserede* sprog [3, sektion 17.5]. Det løser dog ikke problemet med de eksisterende programmer, der er skrevet i "gammeldags" imperative sprog, og de mange programmører der er trænet i brugen af disse, men aldrig har lært at benytte funktionelle sprog.

2.4.4 Distribuerede programmeringsmodeller

Der eksisterer som nævnt en del DSM-strategier for udveksling af data mellem knuder i et distribueret miljø. De tre vigtigste er beskedudveksling (Message Passing, MP), delt lager (Shared Memory, SM) og mobile objekter (Mobile Objects, MO). I programmeringseksemplet i afsnit A.2 er vist et simpelt parallelt program implementeret med disse tre forskellige paradigmer. En alternativ metode er regionbaseret programmering, som er implementeret i CRL [8], hvor de delte lager opdeles i regioner, som programmøren explicit skal erklære at programmet på et givent tidspunkt skal have adgang til. Denne metode minder dog så meget om MO i praksis, at den ikke vil blive nævnt yderligere her.

Som det fremgår af eksemplet, minder MP meget om almindelig kommunikation over netværk (se til sammenligning eksemplet i afsnit A.1), og SM og MO programmerne er meget ens i opbygning. Dette skyldes at kommunikationen mellem knuderne for SM og MO varetages af systemet, hvilket er en meget stor lettelse for programmøren.

To meget vigtige fakta fremgår *ikke* af eksemplet: MP programmeringsmodellens overlegne fleksibilitet og delingsproblemerne for MO programmet.

En programmør der arbejder indenfor MP paradigmet har langt flere frihedsgrader, i og med at han har direkte kontrol over *hvad* der transmitteres, *hvor* det sendes til og *hvordan* det transmitteres. Programmører der benytter SM eller MO må i stedet forlade sig på, at DSM-systemet foretager transmission af data mellem knuderne effektivt. Således er det muligt at lave mere effektive programmer i MP, omend denne effektivitetsgevinst ofte opnås på bekostning af overskueligheden. Problemstillingen minder om forskellen på høj- og lavniveau sprog indenfor almindelig programmering.

Med hensyn til delingproblemer i MO eksemplet, gælder det for de fleste MO DSM-systemer, at de benytter en *single-writer* protokol. Dette betyder at kun én knude af gangen har skriveadgang til et objekt af gangen. I praksis vil skrivningen til det delte objekt i eksemplet altså blive *serialiseret*, og der vil ikke opnås en speedup ved at bruge to eller flere knuder. Denne svaghed skyldes implementationen af paradigmet, og ikke paradigmet som sådan.

2.5 Distribueret programmering — de grundlæggende forhold

Når man programmerer i et distribueret system, dvs. *parallelt*, støder man ind i en række forhold som man skal tage hensyn til, og som man normalt ikke oplever under almindelig programmering på en uniprocessormaskine. De forhold jeg beskriver i det følgende er for størstedelens vedkommende gældende for alle distribuerede miljøer, og altså ikke kun for DSM-systemer — de er relevante for alle processer der involverer fordeling af arbejde på to eller flere knuder. De følgende afsnit er derfor at betragte som grundforudsætninger for udvikling eller evaluering af DSM-systemer.

Vægten er primært lagt på delt lager som DSM-mekanisme og låse og barierrer som synkroniseringsmekanisme, da denne opgave netop er fokuseret på effektiv løsning af praktiske problemer i implementation af denne type systemer. Alle betragtninger er som udgangspunkt tænkt i konteksten distribuerede systemer, dvs. løstkoblede, snarere end tætkoblede parallelle systemer.

2.5.1 Faser i programmet

For det første er forløbet i et parallelt program mere kompliceret end i et almindeligt program. I et normalt (imperativt) program på en uniprocessor gennemgår man overordnet set følgende faser:

1. opstart
2. indlæsning
3. beregning
4. uddata
5. afslutning

Til forskel er de typiske faser i er parallelt program som følger:

1. opstart
2. indlæsning
3. distribuering af arbejde på knuder
4. beregning
5. synkronisering
6. uddata
7. nedlukning af knuder

8. afslutning

Udover at der er flere faser i et parallelt program, er det også meget sjældent at se et interaktivt parallelt program, og IO foregår typisk kun under indlæsnings— og uddatafaserne.

2.5.2 Delt lager, synkronisering og data races

Ydermere skal parallelle programmer udvikles så de undgår *væddeløbstilstande*, eller *data races*. Hvad er så en væddeløbstilstand? Uformelt er der tale om en væddeløbstilstand når resultatet af samarbejde mellem 2 eller flere programmer (processer), eller enheder (såsom et program og en harddisk) — her for nemhedens skyld kaldt *knuder* — kan blive "ukorrekt" afhængig af i hvilken rækkefølge knuderne udfører deres arbejde.

Følgende eksempel illustrerer problemet: To knuder, her programmer, skal samarbejde om at løse en opgave. Den ene knude, producenten, leverer arbejde til den anden, konsumenten. De to knuder kommunikerer ved at deles om adgang til en arbejdskø (FIFO). Det går fint i langt de fleste tilfælde, undtagen når konsumenten prøver at fjerne et element fra starten af køen samtidigt med at producenten tilføjer et. Så længe det er sikret at kun en af knuderne har adgang til arbejdskøen af gangen, er der ikke noget problem. De tidspunkter under eksekveringen hvor der kan opstå data-races, kaldes for *kritiske regioner*. Et program, hvor alle kritiske regioner er sikret med synkronisering kaldes for *Data-Race-Free* (DRF). Problemet kendes fra alle systemer, hvor arbejde kan udføres i parallel (også kaldt *concurrent programming*). En udgave af producent/konsument programmet i pseudokode kan beskues i afsnittet A.3.

Ud fra dette simple eksempel kan vi se de to grundlæggende behov et DSM *miljø* skal tilfredsstille — deling af information mellem knuder (DSM) og *synkronisering* af knuders arbejde. De nødvendige elementer er et DSM-system, baseret på en DSM-strategi, som er implementeret, dvs. udfører strategien, ved hjælp af en DSM-mekanisme. Strategien kan implicit, altså uden programmøren skal indsætte specielle kommandoer for det, håndtere synkronisering, eller det kan gøres eksplicit. Et eksempel på dette er netværkskommunikation, der som nævnt i afsnit 2.4.2 kan udføres synkront (*blokerende*) eller asynkront (*ikke-blokerende*). Hvis et program modtager synkront, stoppes det af operativsystemet, indtil der er modtaget data, og synkroniseringen er hermed implicit en del af modtagelsen. Hvis det modtager asynkront, er modtag-kommandoen kun en besked til operativsystemet om, hvor modtagne data skal sendes hen, og det er programmørens opgave at sikre — altså explicit synkronisere — at programmet ikke læser data der endnu ikke er modtaget

På en almindelig arbejdsstation — en *uniprocessor* — oplever man sjældent væddeløbstilstande som et problem, men det er kun fordi den underliggende hardware præsenterer et miljø der er renset for dette. Væddeløbstilstande kan opstå i alle tilfælde hvor der kommunikeres mellem processer eller enheder, og er altså ikke kun et faktum i DSM systemer.

Væddeløbstilstande er altså ikke noget der "skyldes" det distribuerede miljø — det er et universelt faktum i enhver form for samarbejde. Det er trods alt kun kærester og tegneseriefigurer der kan spise den samme strimmel spaghetti samtidigt...

2.5.3 Illusionen om samtidighed

Det andet store universelle faktum man skal forholde sig til i et distribueret miljø er *manglen på samtidighed*. Der er ikke noget der hedder absolut tid, når to knuder skal samarbejde — der er kun tale om at det ene sker før det andet. Når man programmerer i et imperativt sprog, programmerer man under antagelse af at ordre nr. 9 udføres før ordre 10, f.eks. at en ændring i lageret udført af ordre 9 kan ses af ordre nr. 10. Selv på en uniprocessor med en enkelt aktiv process, der ikke kommunikerer med andre processer, er det et problem der skal håndteres. I det tilfælde er strategien at ordre 10 ikke får lov til at begynde før ordre 9 er afsluttet — en strategi som er medvirkende til at sænke den faktiske ydelse i forhold til den teoretiske på moderne processorer, der som oftest forsøger at udføre ordre så hurtigt som muligt ved at påbegynde en ny før den igangværende er afsluttet (pipelining) [6, pp. 137..161].

Man kan sige at alle skrivninger i en uniprocessor gøres globalt kendte, og er blokerende, i den forstand at ingen ordre der afhænger af skrivningen får lov til at fortsætte forbi det punkt hvor de afhænger af skrivningen før de har adgang til resultatet (indholdet) af skrivningen.

Denne strategi kan i princippet også benyttes i et DSM system, og var også en af de oprindelige strategier for hardware-baserede DSM-systemer [15]. I praksis er denne strategi dog langt fra nødvendig, så længe programmet er synkroniseret korrekt (*data-race-free*). I det ovenstående eksempel fra uniprocessoren kunne man sagtens klare sig uden avancerede mekanismer til bremsning af læsningen, hvis man bare sikrer sig at der enten er passende langt imellem skrivninger og læsninger til at skrivningen er effektueret før læsningen påbegyndes, eller hvis programmøren (eller oversætteren) har indført et explicit check for, hvorvidt skrivninger er udført før læsninger påbegyndes.

2.5.4 Afslappede DSM-mekanismer og happens–before–relationer

I et moderne SMLB baseret DSM-miljø benytter man sig af langt mere afslappede modeller end den ovennævnte. Ikke fordi de er nemmere at programmere til, men fordi det er uforholdsmæssigt dyrt i tid at effektuere alle skrivninger globalt. Kommunikation tager tid, og bør derfor begrænses til et minimum. Desuden kan den tid kommunikationen tager som nævnt i afsnit 2.4.2 opdeles i *opstartstid* og *overførselstid*. Opstartstiden — også kaldet *latenstiden* — er en konstant (stort set), og er grundomkostningen for overhovedet at kommunikere mellem to knuder. Efterhånden som netværks overførselshastighed (*bitrate*) stiger, bliver det mere og mere attraktivt at *amortisere* denne ved at sende så meget relevant information som muligt når der kommunikeres. Princippet er det samme som når man betaler kr. 5,- ekstra hos McDonalds for at få en stor Cola og Megafritter — det syner ikke af så meget i forhold til grundprisen for en BigMac Menu (tm).

Derfor benytter et moderne DSM-system sig af DSM-mekanismer, der kun kræver udveksling af lagerændringer ved synkronisering. Intuitivt er det kun nødvendigt at udveksle information ved synkroniseringspunkter, hvis ellers programmet er synkroniseret så det er data-race-frit. Hvis dette medfører at programmet ikke udfører sit arbejde korrekt, er det fordi det ikke er synkroniseret korrekt!

Den mest brugte af disse *konsistensmodeller* er doven opdatering ved frigivelse (Lazy Release Consistency, LRC). I denne model meddeles lagerændringer først til andre knuder, når man forlader en kritisk region (release) og en anden knude anmoder om adgang til regionen — deraf tilnavnet "doven", da meddelelserne først udveksles i sidste øjeblik. Denne model har vist sig at være den mest effektive, da mængden af meddelelser og dataoverførsel generelt holdes på et lavere niveau end ved brugen af andre konsistensmodeller [10].

Denne form for strategier medfører at der skal skabes orden i skrivningerne. Problemet fremgår af følgende eksempel der findes i pseudokode i afsnit A.4:

> 3 knuder arbejder sammen om at løse et problem. Knude 0 starter med at skrive til lagercellerne 1-3. Herefter overgives kontrollen til først knude 1, og derefter til knude 2. Knude 1 ved at lagercelle 1-3 er ændret, men ved ikke at knude 2 er orienteret. Inden knude 1 og 2 kommunikerer, har knude 2 ændret indholdet af lagercelle 1. Når knude 1 og 2 mødes, hvad skal der så stå i lagercelle 1?

Her er tydeligvis brug for at kunne ordne skrivningerne på en eller anden måde, da det er intuitivt klart at knude 2's skrivning til celle 1 bør tage præcedens over den oprindelige initialisering på knude 0. Det er ikke muligt at bruge et tidsstempel, da maskinernes ure kan være forskelligt indstillet (eks. hvis knude 2's ur er stillet 1 time tilbage, er skrivningen til lagercelle 1 tidsstemplet til at være sket *før* knude 0 initialiserede den). Det er godt nok muligt at lade alle skrivninger foregå centralt, f.eks. på knude 0, men så udløser *samtlige* skrivninger og læsninger kommunikation med knude 0. Dette koster ikke bare den målbare tid kommunikationen tager, men også at man afbryder knude 1 i dens (forhåbentligvis relevante) arbejde — en *skjult* pris for denne strategi.

En mulig løsning er, at konstatere det vigtige ikke er hvornår, men *i hvilken rækkefølge* skrivningerne sker. Hvis vi nummererer skrivningerne efter dette sker følgende: den oprindelig skrivning til celle 1-3 har nummer 0, skrivningen på knude 2 har nummer 1. Når knude 1 og 2 mødes, ses det straks at knude 2's skrivning til lagercelle 1 skete efter den oprindelige initialisering. Dette er en såkaldt *happens-before* relation [19, pp. 120..124].

2.5.5 Lamport-tid og bogføring af skrivninger til delt lager

Den årvågne læser spørger så straks, hvad der så sker hvis knude 3 nu også havde ændret indholdet af celle 1. Svaret er for det første, at der så er tale om et data-race, og samarbejdet mellem knuderne altså ikke er korrekt synkroniseret. For det andet kan man indføre en kausal ordning på hendelser ved at benytte den såkaldte *Lamport-tid* [13] [14]. Uformelt fungerer Lamport-tid således, at knuder der mødes ved et synkroniseringspunkt først stiller deres "ur" til den højeste værdi de kan blive enige om, og derefter tæller den op med 1. I eksemplet starter alle med at have Lamport-tid (L) 0. Efter knude 0 og 1 har synkroniseret har de begge L=1 (begge har 0, dette tælles 1 op). Efter knude 0 og 2 har synkroniseret har de begge L=2 (1 er højere end 0, læg 1 til). Med dette system får knude 2's skrivning til lagercelle 1 L-stemplet 2.

Dette er dog ikke en sikring mod utilstrækkelig synkronisering. Hvis knude 1 nu havde kommunikeret med den hidtil ukendte knude 3 (L er herefter 2 for dem), og *derefter* skriver til celle 1, har vi stadig to skrivninger til celle 1 med samme L-tid når knude 1 og 2 mødes. Denne situation kan iøvrigt være en indikator af manglende synkronisering, men langt størstedelen af de eksisterende DSM-systemer undersøger ikke om dette opstår af ydelseshensyn.

Der er en ydelsesmæssig pris for at benytte Lamport-tid til stempling af skrivninger. For det første er der selvfølgelig et administrationsarbejde og det lagerforbrug der følger med. Desuden skal DSM-systemet implementere en mekanisme til at fange (bogføre) skrivninger til det fælles lager (*write-tracking*). Også denne mekanisme kan laves explicit (ved at kræve at programmøren benytter specielle primitiver til at skrive til lageret), eller implicit (ændringer fanges uden programmørens aktive medvirken, typisk ved brug af datamatens Memory Management Unit (MMU) eller ved compileres mellemkomst). Begge metoder kræver selvfølgelig mere tid end en normal skrivning til lageret på en uniprocessor.

Dette tidsforbrug kan være endog meget svært at opgøre, da det kan indeholde skjulte omkostninger — her tænkes specielt på at indeholdet af TLB og cache i datamatens processor kan blive ugyldiggjort ved brug af MMU-baseret write-tracking, eller at cachebuffere fyldes op med information fra DSM-systemets lagerstyringssystem. I begge tilfælde sænkes den generelle ydelse af det egentlige program i forhold til normal ydelse på en uniprocessor, som beskrevet i afsnit 2.4.1.

Lamport-tid er langt fra den eneste metoder der kan benyttes til at indføre en kausal ordning af skrivninger, men er uden tvivl den mest udbredte.

2.5.6 Konsistensmodeller og protokoller

Reglerne for hvornår og hvordan denne viden blandt knuderne om lagerændringer og deres placering i henhold til happens-before relationen (konsistensinformation) skal synkroniseres kaldes for *konsistensmodeller*.

I SMLB systemer er kan de kategeriseres som *entry* eller *release* consistency, afhængigt af om synkronisering af lageret udløses af indgang eller udgang af en kritisk region. Desuden findes der såkaldte *dovne* (lazy) varianter, hvor synkroniseringen først udføres ved overgivelse af ejerskabet til en anden knude. Hvis en konsistensmodel ikke er doven, kaldes den *ivrig* (eager).

Udover disse "rene" konsistensmodeller, findes også en række systemer der giver adgang til hybridiserede modeller. Et system som Midway [5] giver adgang til både release- og entry consistency, og i ORDER [18] kan benyttes en kombination af normal SMLB konsistensmodel og MP.

I en større sammenligning af de forskellige varianter af konsistensmodeller for SMLB-systemer [12] fandt man at den dovne release-consistency (Lazy Release Consistency, LCR) i langt de fleste tilfælde gav den bedste ydelse, da den begrænsede mængden af dataudveksling til et minimum i forhold til de andre protokoller.

2.5.7 Update og invalidate protokoller, read faulting

Et emne der følger naturligt af det foregående afsnit, er valget mellem at rundsende skrivninger (*active update*-protokol, eller blot *update*-protokol) eller blot orienteringer om skrivningerne (*invalidate*-protokol), og i så fald om dette skal gøres explicit eller implicit [12] [18, p. 7]. Det er meget tænkeligt at en knude ikke har brug for at kende indholdet af de lagerceller en anden knude har opdateret. Så længe det vides *hvilke* lagerceller der er ændret, *hvem* der har ændret dem, og hvad deres L-tids stempel er, kan der præsenteres et opdateret billede af lageret. Hvis man nøjes med at rundsende orienteringer, kan knuderne så udbede sig indholdet af en lagercelle i området fra ejeren, hvis der bliver brug for det. På denne måde spares der på tidsforbruget på overførsel af informationer, under forventning af at de oplysninger man undlader at rundsende for langt størstedelens vedkommende er irelevante. Da emnet vil blive behandlet mere grundigt senere, vil jeg her kun orientere om problemstillingens eksistens, samt gøre opmærksom på at brugen af invalidate-metoden kræver indførelsen af muligheden for læsefejl (*read-faulting*). Behovet for læsefejl er indlysende, da en læsning af oplysninger fra et område af lageret der vides at være ikke-opdateret — knuden har modtaget besked om at denne del af lageret er ændret ved en forudgående synkronisering, men har ikke modtaget det ændrede indhold — selvfølgelig skal udløse en anmodning om at få tilsendt det opdaterede indhold af lageret fra en knude, der er i besiddelse af dette.

Situationen minder på mange måder om implementation af virtuel hukommelse, i og med at de tidsmæssige omkostninger der er involveret ved rundsendelse er så høje, at det kan betale sig at bruge en del tid på at vælge den bedst mulige strategi i den foreliggende situation [6, p. 441].

Det ses straks at en invalidate-protokol har ydelsesmæssige problemer i en situation hvor store mængder *relevant* information produceres, hvorimod en update-protokol har sin svaghed når der produceres store mængder *irrelevant* information.

Read-faulting kan, præcis som write-tracking, implementeres implicit eller explicit, og ved brug af de samme metoder.

Det er også værd at bemærke sig, at Lamport-tid *ikke* opdateres ved en read-fault, og den deraf følgende kommunikation. Lamport-tid opdateres *kun* ved synkroniseringspunkter!

2.5.8 Irelevante opdateringer

Et specielt problem for SMLB-systemer der benytter en eller anden form for update-protokol, er risikoen for rundsendelse af irelevante opdateringer. Problemet er illustreret med et programeksempel i afsnit A.5. Her opdateres dele af lageret på begge knuder, men kun en lille del disse opdateringer er nødvendige for at få korrekt uddata fra programmet.

Nu er det tydeligt for et menneske der studerer eksempelprogrammet, at oplysningerne i matricen A kun er af interesse for den enkelte knude, og derfor højest sandsynligt ikke burde have været erklæret som delt lager, men problemet optræder også i forbindelse med "almindelige" distribuerede SMLB-programmer.

Problemet skyldes at programmøren ikke har nogen mulighed for at fortælle DSM-systemet, hvilke dele af lageret det er nødvendigt at rundsende, og evt. hvilke man kan nøjes med at meddele er ændrede (invaliderede), *selv hvis programmøren har denne viden*. Da SMLB-paradigmet ikke giver mulighed for at gennembryde denne semantiske mur, og da det er umuligt at udføre en udtømmende statisk analyse af delingsmønstre på oversættelsestidspunktet, må systemet forlade sig på statiske regler, eller opsamlet heurestik.

2.5.9 Fastslåelse af ejerskab af kritiske regioner

Endnu en grundlæggende problemstilling er distribuering af viden om *ejerskab* af kritiske regioner. Det er indlysende at en knude der ønsker adgang til en kritisk region, skal spørge om lov hos den knude der sidst har haft ejerskab af regionen. Hvordan skal en knude finde ud af, hvilken af de andre knuder der tager del i beregningen der er tale om? Der er tre mulige metoder:

Statisk En statisk analyse, dvs. foretaget *før* programmet er startet, har afsløret det. Denne strategi er i langt de fleste tilfælde umulig at gennemføre i praksis, på grund af de alias-problemer der beskrives i afsnit 2.4.3.

Velkendte mellemmænd En eller flere knuder får fast ansvar for at bogføre ejerskab af kritiske regioner.

Broadcast En knude rundsender (broadcaster) til alle andre knuder når den opnår ejerskab af en kritisk region. Alternativt kan ønsket om at opnå ejerskab broadcastes.

Da den statiske metode (som nævnt) er stort set ubrugelig i praksis, benytter de fleste systemer sig af enten broadcast eller velkendte mellemmænd. Hvilken af de to mekanismer der er at foretrække, afhænger i bund og grund af den foreliggende situation, dvs. hvilken løsning der giver den bedste ydelse for det aktuelle system og programmel. Generelt kan siges følgende om deres egenskaber:

Velkendte mellemmænd

- Klare ulemper: Højere latenstid, da mellemmanden skal videresende ønsket om at overtage ejerskabet af den kritiske region til ejeren.
- Klare fordele: Afbryder kun 1 knude i dens arbejde — ejeren af den kritiske region. Mulige fordele: Mulighed for at oprette en kø af ventende knuder i tilfælde af stor konkurrence om adgang til den kritiske region, og dermed fremsendelse af ventende knuder for sænkning af latenstiden. Mellemmandsfunktionen kan varetages af en knude som ikke deltager i beregning, og således sikre at en beregningsknude ikke påføres skjulte omkostninger.

Broadcast

- Klare ulemper: Skjulte udgifter på at afbryde andre knuder i deres (forhåbentligvis nyttige) arbejde — i værste tilfælde spild af N-2*tiden at håndtere forespørgslen ved optimal distribuering, da overførsel af ejerskabet er uvæsentlig for de N-2 andre knuder i systemet.
- Klare fordele: Simpelt at implementere, kortere latenstid på at opnå ejerskab.

Hvis målet er at opnå en generelt god skalerbarhed i DSM-systemet som sådan, er velkendte mellemmænd altså at foretrække. Med denne metode risikerer man godt nok flaskehalsproblemer, hvis alle knuder konkurrerer om adgangen til en enkelt kritisk region, men har i det mindste muligheden for at ændre programmet så det fordeler arbejdet mere effektivt på knuderne ved brug af andre datastrukturer.

2.5.10 Opsummering — De grundlæggende DSM spørgsmål: Hvem, Hvad, Hvor

På grundlag af det forudgående, kan vi nu forsimple problemstillingerne til et mere praktisk niveau som følger:

> Ethvert DSM-system stiller faciliteter til rådighed for en mængde knuder, typisk processer, der skal samarbejde om at udføre en beregning. Overordnet set skal et DSM-system give en knude den *for knuden relevante* information om
>
> - Hvilken knude laver hvad lige nu, dvs. *synkronisering*.
> - Hvad er indholdet af en given lagercelle?
> - Er lageret opdateret, og i fald det ikke er, hvor får man det opdateret?

Disse spørgsmål lyder simple at besvare, men det viser sig i praksis at være knapt så simpelt — især hvis man vil opnå tilfredsstillende ydelse fra et DSM system.

2.6 De forskellige klasser af parallelle programmer

Der er utallige måder at parallelisere en algoritme på, men størstedelen af dem udmønter sig i enten stratificering (eks. FFT), arbejdskøer (eks. TSP) eller zonificering (eks. water og N-body).

Stratificerede algoritmer opdeler inddata i (næsten) lige store dele, og udfører derefter et vist antal iterationer af synkron beregning. Denne form for algoritmer benyttes typisk til forskellige former for matricemanipulation.

Arbejdskøer (worklists) dækker over den klasse af algoritmer der blandt andet benytter den såkaldte branch and bound metode (af og til kaldt "divide and concour" eller "divide and crush"). Det indledende problem opdeles i delproblemer, og paralleliseringen foregår ved at de enkelte knuder ud fra et (eller flere) delproblem(er) enten konstruerer et eller flere nye delproblemer, eller mulige løsninger.

Zonificerede algoritmer tager som inddata en grundtilstand, og konstruerer en ny tilstand ud fra denne. Dette kan gentages, hvis man ønsker at studere processen snarere end den nye tilstand, eller hvis man for nøjagtigheden af sluttilstanden må foretage flere mellemliggende tilstandsberegninger. Paralleliseringen foregår her oftest ved at en tilstand opdeles i zoner, som så kan indeholde få eller mange datapunkter der skal beregnes. Fordelingen i zoner kan skifte under eksekveringen, afhængigt af deltilstanden (eg. i en n-body beregning kunne en "klump" af masser udgøre en zone). Et andet navn for denne gruppe af algoritmer kunne være "dynamisk stratificerede", men jeg mener zonificerede er et mere indlysende valg — især fra programmørens perspektiv. Denne form for algoritmer bruges typisk til fysiske simulationer, såsom partiklers indvirkning på hinanden.

Det er ikke muligt at udtale sig kategorisk om, hvilket API der egner sig bedst til hvilken klasse af algoritmer, men man kan godt generelt sige flg. om hvilke mønstre for lagerdeling og load balancing der er typiske for dem:

Stratificering er karakteriseret ved at arbejdsbyrden (Load Balancing) er meget jævnt fordelt på knuderne. Det er nemt at statisk (dvs. på oversættelsestidspunktet) forudsige hvilke knuder der skal udveksle hvilken information ved et givent synkroniseringspunkt. I et SMLB system er den primære synkroniseringsmetode en barriere. Stratificerede algoritmer er typisk de nemmeste at implementere i et MP DSM-system.

Arbejdskøer er karakteriseret ved at de samarbejdende knuder deles om en datastruktur, typisk en liste, en kø (FIFO) eller et træ, og tilgang til denne delte datastruktur er en kritisk region beskyttet med en lås (Lock) eller barriere (Barrier). Det er oftest umuligt statisk at forudsige hvilke knuder der skal udveksle hvilke oplysninger, men det er til gengæld typisk muligt at forudsige *dynamisk* (dvs. under eksekveringen). I et SMLB system er den primære synkroniseringsmetode en lås.

Zonificering er karakteriseret ved at Load Balancing ofte er ujævn. Deling af informationer mellem knuder afhænger i høj grad af inddata, og er derfor svær at forudsige statisk. Det er også relativt svært at forudsige deling dynamisk. Man kan ikke sige disse algoritmer har en primær synkroniseringsmetode i et SMLB system.

3 Distribueret programmering med shared memory, hvorfor og hvordan?

I dette kapitel vil jeg argumentere for brugen af SMLB paradigmet for distribueret programmering, redegøre for de kendte problemstillinger, evaluere tre SMLB systemer (TreadMarks, Midway og ORDER), og fremsætte konkrete forslag til at forbedre brugervenlighed, speedup og/eller ydelse for fremtidens SMLB systemer.

3.1 Shared memory — fordele og ulemper

SMLB paradigmet for distribueret programmering er blevet ganske udbredt, især fordi programmeringsmodellen svarer til den, der kendes fra brug af letvægtstråde og delt lager på en uniprocessor. De enkelte processer er ikke ansvarlige for eksplicit at udveksle information, men skal blot være synkroniseret korrekt — det der kaldes data-race free (DRF). Programmeringsmodellen med letvægtstråde der har adgang til et delt adresserum bruges i mange sammenhæng — bla. under UNIX — og er altså på forhånd kendt af en stor gruppe programmører.

Ved brug af letvægtstråde på en uniprocessor kører trådene selvfølgelig ikke i parallel, men skedulering fra operativsystemet opretholder illusionen. Her deles processerne — dvs. trådene — om et fælles adresserum, som fysisk er placeret på en enkelt maskine.

Ved burg af SMLB udføres de enkelte processer ægte parallelt på hver deres maskine, men det fælles adresserum er ikke lokaliseret på en enkelt maskine. I stedet opretholder hver af maskinerne, der er vært for en proces, en kopi af det delte lager, som holdes tilstrækkeligt opdateret til at sikre, at et program der er DRF eksekverer korrekt. Man kan sige at det lokale lager fungerer som en cache for det delte lager.

Det delte lager eksisterer ikke i én samlet opdateret form, men er en logisk konstruktion der opretholdes af DSM-systemet. Den lokale repræsentation af det delte lager på en knude består ikke nødvendigvis kun af data, men kan også ved brug af invalidate helt eller delvist udgøres af henvisninger til knuder, der er i besiddelse af disse data.

Det er altså DSM-systemets ansvar, at holde de lokale kopier af det delte lager tilstrækkeligt opdateret til at sikre at programmer der er DRF eksekverer korrekt. Dette kan, som nævnt i teoriafsnittet i kapitel 2, gøres på mange måder, men forudsætter som minimum bogføring af skrivninger (write tracking). De fleste systemer er også i stand til at udføre bogføring af læsninger (read tracking), da dette giver mulighed for at benytte en invalidate protokol, hvor der blot rundsendes information om at skrivninger har fundet sted, uden at fremsende de faktiske data der blev skrevet.

Det er primært mekanismerne til håndtering af denne bogføring, der adskiller de forskellige SMLB systemer fra hinanden. Der eksisterer groft sagt tre forskellige metoder:

- Compilergenereret tracking
- MMU baseret tracking
- Læsning og skrivning til delt lager gennem primitiver (funktioner)

En fjerde metode er en ren hardwareløsning på specialbyggede maskiner, hvilket er en dyr, men hurtig, metode. Da jeg koncentrerer mig om helt eller delvist softwarebaserede systemer i denne opgave, vil jeg ikke behandle denne type systemer yderligere.

De primære kendte årsager til lav ydelse fra SMLB systemer er følgende:

- Udlægning af objekter i lageret kan modvirke effektiv deling
- Algoritmer der ikke tager hensyn til cache-forurening på processoren
- Krydsninger af protection domæner
- Dårlig vægtning af fremsendelse af data og invalidering af dele af lageret
- Administration af rundsendelse eller rekvirering af opdateringer
- Grov opløsning i deling kan afføde stafetproblemet (eller ping-pong effekten)
- Tidsforbrug på bogføring af læsninger og skrivninger
- Administration af synkroniseringsprimitiver

Disse problemer bliver i det følgende behandlet grundigt, og vil ikke blive taget yderligere op i dette afsnit af opgaven.

Et andet problem, som ikke umiddelbart kan løses med bedre teknologi, skyldes at udvekslingen af data mellem processerne er implicit. Kommunikation mellem knuderne i et distribueret system er som nævnt i afsnit 2.4.2 en relativt dyr operation i forhold til skrivning til lokal hukommelse, især med hensyn til latenstiden for første skrivning. At programmøren i et SMLB system ikke udtrykkeligt skal udbede sig kommunikation med en anden knude, har en tendens til at medføre at programmer skrevet i et sådanne system udfører mere kommunikation end et tilsvarende program implementeret med message passing. Man kan sige at det skyldes at SMLB-paradigmet fokuserer på algoritmen, snarere end kommunikation og deling af data mellem knuder.

Endvidere vil et SMLB program stort set altid være langsommere end et tilsvarende MP program, da et SMLB system faktisk udfører præcist det samme arbejde som programmøren i et MP system selv ville være ansvarlig for. Dette arbejde medfører et tidsforbrug på administration, som MP systemet klarer sig uden. Da programmer i MP hhv. SMLB som illustreret i eksempelprogrammerne i afsnit A.2 er meget forskellige, er det umuligt at udføre en direkte sammenligning. Det er dog typisk tilfældet at et grundigt optimeret MP program vil være et grundigt optimeret SMLB program overlegent, da SMLB programmet til en hver tid vil transportere mindst lige så meget data mellem knuderne, og synkronisere mindst lige så mange gange. MP er som det fremgår af afsnit 2.4.2 den underliggende model for netværkskommunikation i alle systemer, og SMLB er blot et forsøg på at skjule den kompleksitet som programmering til MP medfører. Det er indlysende at dette må medføre et ekstra tidsforbrug (overhead) på administration.

Den store forskel på MP og SMLB er, at under MP skal programmøren eksplicit angive både hvad der skal sendes, og hvilken knude der skal modtage det sendte. Ingen af delene er nødvendigt i SMLB, da det bliver håndteret af systemet. Prisen er som nævnt en forøget administration, og en større mængde kommunikation mellem knuderne. Ved programmering i MP er det programmørens ansvar at vide, hvilken knuder der har besidder ejerskabet af en kritisk region, hvorimod denne opgave udføres af systemet i SMLB med ekstra kommunikation som følge. Denne ekstra kommunikation kan som nævnt i afsnit (ref: mellemmaend) være i form af broadcast, eller fremsendelse gennem velkendte mellemmænd.

Fordelene ved SMLB består altså udelukkende i at programmeringsmodellen er mere tilgængelig for programmører. Dette er dog ikke at foragte, da en nedsættelse af tidsforbruget på programudviklingen og en lettere fejlfinding kan nedbringe omkostningerne ved udvikling af et parallelt program betragteligt. At ydelsen og skalérbarheden for SMLB generelt er lavere end for MP [16] er ikke nødvendigvis et problem. En oplagt mulighed at omgå denne begrænsning på, er at udvikle et program baseret på clustering, hvor de enkelte clusters består af en gruppe maskiner der benytter SMLB til deres del af beregningen, men bruger MP til kommunikation mellem clusters. Alternativt kan man benytte clustering, og lade en enkelt knude i hver cluster have adgang til to SM-domæner — et for clusteren, og et til udveksling af information mellem clusters. Med denne metode kan opnås speedup der nærmer sig en ren MP implementation, samtidigt med at en stor del af kompleksiteten i MP programmering skjules for programmøren. Opgaven bliver således løst med høj ydelse, og på et højere abstraktionsniveau end i ren MP.

Men netop at hæve abstraktionsniveauet programmøren kan bevæge sig på, har jo været en af datalogiens mærkesager siden den første assembler blev taget i brug.

3.2 Kendte problemstillinger

Som nævnt er der en række kendte problemstillinger, som de eksisterende implementationer af SMLB har afdækket. I det følgende beskriver jeg de vigtigste indgående.

3.2.1 Udlægning af objekter i lageret

I visse tilfælde kan den måde objekter allokeres i lageret have en negativ effekt på deres evne til at blive delt mellem knuder i et SMLB system. Årsagen er, at alle kendte SMLB systemer har langt nemmere ved at håndtere deling af få, store lagerområder der er placeret fortløbende i lageret end mange små der er spredt over hele lageret.

Et klassisk eksempel er en algoritme, hvor en gruppe processer skal dele data der er organiseret i en N x M matrice. Hvis matricen allokeres så rækkerne ligger i fortløbende i lageret, men algoritmen der tilgår data deler dem på en pr. kolonne basis udløser det en opdeling af lageret i N*M delingszoner snarere end blot N/antal knuder. Situationerne kan illustreres som følger:

```
Rækkevis allokering, rækkevis adgang, 5x5 matrice
0000011111222223333344444

Rækkevis allokering, kolonnebaseret tilgang, 5x5 matrice
0123401234012340123401234
```

En afart af problemet kendes fra udviklingen af optimerende oversættere, da de fleste moderne processorer er langt hurtigere til at udføre læsninger og skrivninger til fortløbende lageradresser. Her forsøger man at løse problemet ved maskinel transformation af algoritmen, så dens tilgangsmønstre ændres [6, pp. 408..410]. Metoden kaldes blocking, og har vist sig at være yderst effektiv. Uheldigvis kan den kun benyttes i de tilfælde, hvor det står klart på oversættelsestidspunktet hvilke grænser der gælder for de løkker der håndterer gennemløb af matricen. Der kendes mig bekendt ingen generelle metoder til håndtering af tilsvarende problemer i andre datastrukturer end matricer.

I SMLB sammenhæng kan man godt forestille sig brugen af blocking og tilsvarende metoder til at begrænse problemet, men opgaven besværliggøres af det faktum at det sjældent på oversættelsestidspunktet er muligt at afgøre hvilke dele af data en given knude skal håndtere, endsige hvor mange knuder der indgår i beregningen.

Alternativet er at lægge ansvaret for effektiv udlægning af objekter i lageret hos programmøren. Man kan til en vis grad hævde at dette allerede er tilfældet for programmeringssprog som C, der har veldefinerede regler for hvordan objekter allokeres. På den anden side er formålet med SMLB at skærme programmøren for kompleksiteten i distribueret programmering, snarere end at pålægge hende yderligere arbejdsopgaver.

3.2.2 Cache-forurening

Moderne arbejdsstationer er som nævnt i afsnit 2.4.1 ofte udstyret med to niveauer af cache mellem CPU og RAM. Årsagen er, at kløften mellem ydelsen for CPU og RAM er stigende [6, p. 374]. Hvor man for få år siden bekymrede sig over en enkelt "wait state" på tilgang til lageret, er situationen i dag at et cache-miss udløser 10-20 clockcyklers ventetid før data er til rådighed for processoren. For at sikre at ydelsen forbliver høj under disse omstændigheder, har man altså øget størrelsen på cache lagrene, og indført et dybere hukommelseshieraki.

Under normale omstændigheder, dvs. under udførsel af "almindelige" programmer, kan man forvente at over 95% af alle læsninger fra lageret leveres fra cachelager uden ventetid [6, p. 384]. Men forudsætningen for dette er, at cachelageret ikke fyldes med data der ikke er relevante for udførslen af programmet — en situation der kaldes cache-forurening.

Denne effekt er en af de vigtigste årsager til, at kopiering af data i dag er en relativt dyr operation, i forhold til f.eks. matematiske funktioner. Problemet er ikke at CPUen venter på lageret, da dette håndteres ved at læse data ind i cachelageret i store blokke. Sagen er, at når kopieringen er afsluttet, har de data der netop er blevet kopieret forstødt de oplysninger der før var placeret i cachelageret.

Problemet kendes især fra operativsystemer og udvikling af kommunikationsprotokoller. I begge tilfælde må man forsøge at undgå kopiering af data for at opretholde en høj ydelse, hvilket ikke er en triviel opgave, da både operativsystemet og protokoller udfører mange opgaver der involverer transport af data mellem processer og maskiner.

I et SMLB-system udføres der også mange opgaver, der involverer kopiering, søgning og transformation af data. Uheldigvis er mange af de eksisterende systemer designet i en tid, hvor cache-forurening ikke var nær så stort et problem som det er i dag, og tager altså ikke nogen forholdsregler for at undgå det. Et eksempel er den operation der kaldes udarbejdelse af *diffs* [2, pp. 13–14]. En diff er en datastruktur, der afspejler ændringer et program har foretaget i en lagerside, og den skabes ved at kopiere siden, og senere sammenligne kopien med sidens nye indhold. Da dette involverer kontekstskift, kopiering og store mængder betingede hop, er det på moderne maskiner en yderst tidskrævende operation. Prisen for oprettelse af en diff er stigende i forhold til eksekvering af almindelige operationer — tilgang til lageret er i dag en flaskehals.

Et andet tilfælde hvor cache-forurening kan give utilsigtede sideeffekter, er opslag i data ved brug af hashing. I et SMLB-system der udfører opslag i det delte lager med en hash-tabel, vil man risikere at se lavere ydelse end et system hvor man bruger binære træer. Årsagen? Opslagene i det delte lager vil sandsynligvis udvise spatiel og temporær lokalitet, da de skyldes et kørende program. Hvis opslagene udføres med et binært træ, vil en god del af de knuder i træet man skal passere for at finde en given datablok allerede være i cachelageret, da data i et træ er placeret tæt på hinanden hvis deres nøgler er tæt på hinanden. Ved brug af hashing fordeler man tilgangen jævnt over hele hashtabellen — ellers ville det ikke være en god hash-funktion der blev benyttet. Risikoen for at fylde cachelageret med oplysninger der ikke kan genbruges i den nærmeste fremtid, dvs. cache-forurening, er altså langt større her.

Der er ikke udsigt til, at cache-forurening vil blive et mindre problem i fremtiden — tværtimod. Forskellen i ydelse mellem CPU og RAM er stadigt stigende, og ny forskning i RAM teknologi retter sig i de fleste tilfælde mod at forbedre båndbredden på overførslen mellem RAM og cache-lager, snarere end at forbedre latenstiden. I fremtiden vil vi altså formentlig se arkitekturer der i endnu højere grad forlader sig på brug af cachelager, og hvor cache misses medfører en endnu større penalty.

Konklusionen må være, at man i fremtiden bør stile mod at benytte algoritmer, der begrænser kopiering og gennemsøgning af store lagerområder, og i stedet i højere grad udviser cache-venlig opførsel: regional og temporær lokalitet.

3.2.3 Krydsninger af protection domæner

En anden sideeffekt af moderne CPUers afhængighed af cachelageret er, at kontekstskift — især mellem user kode og kerne kode — er blevet en meget dyr operation. Et skift til kernedomænet (protect- eller supervisor-mode), medfører nemlig i mange tilfælde at information i cache og transition lookaside buffers (TLB) bortkastes. Sagen er, at brugerprogrammer typisk opererer med virtuelle adresser, som ikke nødvendigvis svarer til de virkelige adresser i RAM som de oversættes til. Dette har bla. ført til brugen af letvægtstråde der ikke håndteres af kernen — og dermed ikke udløser kontekstskift — indenfor multiprogrammering på uniprocessorer [19, p. 179].

Da næsten al kommunikation over netværk som nævnt i afsnit 2.4.2 udløser skift til kernetilstand, rammer denne effekt altså også DSM-systemer på netværk af arbejdsstationer. Da den eneste måde at undgå dette problem på er, at integrere både DSM-systemet og det program det eksekverer i kernen, kan denne tidsmæssige omkostning nok næppe undgås.

En mængde kontekstskift der *kan* undgås, er dem der udløses i et SMLB system der bruger arbejdsstationens MMU-hardware til at spore læsninger og skrivninger til delt lager. Der fremføres to argumenter for at benytte denne metode:

- Det letter programmørens arbejde ved at lade tilgang til delt lager foregå på samme måde som almindelig tilgang til lokalt lager
- Tilgang til delt lager kan udføres med samme hastighed som almindelig tilgang til lokalt lager.

Det første argument er i høj grad gyldigt, da det er et ekstraarbejde for programmøren at benytte primitiver eller objektmetoder til at tilgå elementer i delt lager, og dermed en fejlkilde. Men med hensyn til ydelsen, er det klart at dette ikke vil være tilfældet på en moderne arbejdsstation. Det er ikke umiddelbart indlysende, hvordan det kan lade sig gøre at opgøre de skjulte omkostninger ved skift til kernetilstand præcist — den eneste praktiske metode er at sammenligne med et lignende DSM-system, der blot benytter en anden metode til bogføring af læsninger og skrivninger til delt lager.

3.2.4 Vægtning af fremsendelse af data og invalidering af dele af lageret

Et SMLB-system's lagermekanisme har som sin primære opgave at opretholde et billede af det delte lager på den enkelte knude, som er tilstrækkeligt opdateret til, at et DRF program vil eksekvere korrekt. At opretholde et opdateret billede af det delte lager kan som nævnt i afsnit 2.5.5 gøres ved brug af en bogføringsmekanisme til sporing af skrivninger og læsninger, en passende protokol der implementerer en konsistensmodel (eks. LRC) og en kausal ordning af skrivninger (eks. lamporttid). Et af de store problemer med effektiv udførsel af denne opgave er, at afdække delingsmønstre mellem knuderne — deres dataflow — og dermed i hvilken grad det lokale billede af lageret skal bestå af egentlige data, snarere end blot meddelelser om, på hvilken knude opdaterede data befinder sig.

Der er altså tale om en balancegang mellem (evt. ivrig og/eller spekulativ) fremsendelse af oplysninger, hvilket nedbringer antallet af beskeder der skal udveksles mellem knuderne, og blot meddelelse om opdateringer, hvilket nedbringer både behovet for båndbredde på nettet, og administrationsarbejdet på opdateringen af lageret. Nøglen til høj ydelse på dette område er, at afdække hvilke oplysninger en knude har behov for at være i besiddelse af, for at kunne fuldføre den næste fase af beregningen. Denne information skal helst være til rådighed så tidligt som muligt, så data kan fremsendes samlet, eller endog i samme meddelelse som en synkroniseringsmeddelelse — en teknik der kaldes *piggy-backing*, og som bruges i mange kommunikationsprotokoller til at hæve ydelsen [20, pp. 202..203].

I enhver distribueret beregning er der behov for denne overførsel af information mellem de involverede knuder. I et SMLB-system ligger ansvaret for rundsendelse af den nødvendige og korrekte information som nævnt i afsnit 2.4.4 hos DSM-systemet, i modsætning til f.eks. MP, hvor ansvaret påhviler programmøren. Det er selvfølgelig en lettelse af programmørens arbejde, men medfører en risiko for, at DSM-systemet enten rundsender for meget eller for lidt information. Rundsendes for meget information — dvs. information som den modtagende knude ikke behøver for at kunne foretage beregningen — koster det tid, som i stedet kunne være blevet brugt produktivt. Rundsendes for lidt, f.eks. i et tilfælde hvor systemet benytter en ren invalidate protokol, spildes der tid på kommunikation, der kunne være undgået.

For at tage et konkret eksempel, vil der i forbindelse med eksempleprogrammet i afsnit A.5 ved brug af en ren invalidate-protokol blive udløst følgende beskeder ved knude 1's ankomst til barrieren:

1. Fra knude 1 til knude 0: Ankommet, invalidate count[1] og A[1,1..10000]

2. Fra knude 0 til knude 1: Alle knuder ankommet ved barriere — fortsæt eksekvering.

3. Fra knude 0 til knude 1: Rekvirering af indholdet af count[1].

4. Fra knude 1 til knude 0: Opdatering: Indholdet af count[1].

Ved brug af en ren update-protokol, vil antallet af beskedder være mindre, men store mængder data bliver overført uden der er behov for dem. Her vil følgende blive udvekslet:

1. Fra knude 1 til knude 0: Ankommet, opdatering: Indholdet af count[1] og A[1, 1..10000]

2. Fra knude 0 til knude 1: Alle knuder ankommet ved barriere — fortsæt eksekvering.

Hvis DSM-systemet var i stand til at benytte en hybridiseret invalidate/update protokol, og var udstyret med en tilstrækkeligt effektiv mekanisme til at forudsige delingsmønstre, kunne man godt nøjes med samme antal beskedder som i den rene update-løsning, men mængden af data ville blive begrænset betragteligt:

1. Fra knude 1 til knude 0: Ankommet, opdatering: Indholdet af count[1], invalidate A[1,1..10000]
2. Fra knude 0 til knude 1: Alle knuder ankommet ved barriere — fortsæt eksekvering.

Det er dog værd at bemærke sig at brugen af en hybridiseret protokol forudsætter at en mekanisme til bogføring af læsninger (read faulting) indføres, og læsninger af delt lager hermed vil udføres langsommere end ved bug af den rene update protokol.

Fremsendelse af irrelevante data ville ikke nødvendigvis være et problem på et hardwarebaseret DSM-system, da man her kunne indbygge en enhed til håndtering af lageret som kørte i parallel med selve beregningen. I et softwarebaseret DSM-system tager opdatering af det delte lager derimod processortid fra den egentlige beregningsopgave, og medfører ydermere cacheforurening der sænker den generelle ydelse på knuden.

Det centrale problem er, at et SMLB system opererer uden semantisk viden om, hvilke objekter lagercellerne repræsenterer, og hvad deres delingsmønstrer er. Netop det faktum at systemet håndterer rundsendelse af data uden programmørens aktive medvirken, forhindrer systemet i at have denne viden fra beregningens start. Det rene SMLB-paradigme giver simpelthen ikke programmøren mulighed for at videregive en sådan viden til systemet.

Forskning i protokoller for SMLB systemer peger på, at hverken den rene update eller den rene invalidate protokol er svaret. I stedet vendes opmærksomheden mod hybridiserede protokoller, og metoder til at uddrage delingsmønstre og mobilitet for det delte lager.

Tre metoder har været taget i brug til løsning af dette problem:

- Statisk analyse ved oversættelsestidspunktet [5], [18].
- Grænseværdier for øvre- og nedre- størrelse for rundsendelse af opdaterede lagerblokke [18]
- Dynamisk opdateret heuristik [10].

I det følgende gennemgås de tre metoder grundigt.

Statisk analyse Den optimale løsning på problemet ville selvfølgelig være, at man allerede på oversættelsestidspunktet havde tilstrækkelig viden til at kunne forudse delingsmønstre mellem knuderne. En oversætter, kunne udføre en sådan statisk analyse, ville — i teorien — sikre optimal ydelse i form af den mindst mulige mængde udveksling af beskeder og data mellem knuder. Teorien er besnærende, men støder i praksis ind i alvorlige problemer.

Først og fremmest gælder det for mange distribuerede programmer, at deres delingsmønstre er voldsomt afhængige af de inddata programmet præsenteres for. Dette er især tilfældet for programmer som TSP, der benytter arbejdskøer. Visse programmer — igen kan programmer baseret på algoritmer der benytter arbejdskøer og branch and bound nævnes — kan også udvise vidt forskellige delingsmønstre afhængig af små forskelle i tidsforbrug på løsning af delproblemer på de enkelte knuder.

Et andet, og måske mere alvorligt, problem er, at den statiske analyse som nævnt i afsnit 2.4.3 aldrig kan være fuldstændig for programmer skrevet i et sprog, der giver mulighed for aliasing af en variabel, hvilket gælder alle sprog med adgang til pointere og andre former for reference-baseret tilgang til variable. Aliasing er navnet for de situationer, hvor flere variable kan referer til den samme lagerplads, typisk i tilfælde af dereferering af pointere. Nogle af disse alias-situationer kan selvfølgelig håndteres gennem tilstrækkelig grundig analyse af programmet, men visse klasser af aliasproblemer kan ikke analyseres statisk. I praksis er det altså nødvendigt enten at begrænse anvendelsen af disse sprogfaciliteter på data placeret i det delte lager, eller at supplere den statiske analyse med en dynamisk bogføring af læsning og skrivning til det delte lager. At modificere en oversætter til at udføre den statiske analyse er altså en ikke-triviel opgave.

Aliasproblemet er et kendt problem indenfor udviklingen af optimerende oversættere. Der kendes ingen effektiv løsning, udover anvendelse af såkaldt rene funktionsbaserede sprog, hvor aliaseffekter ikke optræder. Men ved brug af traditionelle sprog som Pascal, C og C++, er man nødt til at opgive al viden om lagerets tilstand når man støder på en mulig aliasing af en variabel, da man ellers ikke kan garantere at den genererede maskinkode er semantisk ækvivalent med det program oversætteren er blevet præsenteret for. Resultatet er, at selv en god aliasanalyse sjældent strækker sig længere end et enkelt kodefragment (basic block). I SMLB samenhæng betyder det, at man i praksis enten må ignorere problemet (dvs. erklære at aliasing af data i delt lager giver udefinerede resultater), eller benytte en supplerende mekanisme til at bogføre skrivninger og læsninger i delt lager, og varetage udvekslingen af disse med andre knuder i systemet.

En anden faktor er programmets hastighed i de dele, der ikke direkte vedrører tilgang til delt lager. En oversætter rettet mod distribuerede programmer kan ikke tillade sig at ignorere disse optimeringer, da det kan medføre en kraftig nedsættelse af ydelsen. Forskellen i ydelse på programmer oversat med og uden optimering kan være dramatisk — tit ses en forbedring med en faktor 10. Alene udviklingen af en oversætter der er i stand til at optimere på dette niveau er en meget stor opgave, der kan kræve hundredvis af mandeår og grundig indsigt i målmaskinens arkitektur. Her risikerer man altså at blive ramt af Amdahls Lov [6, pp. 29..32].

Konklusionen må være, at selvom en statisk analyse i teorien kunne give et fuldstændigt billede af dataflow, og dermed deling af data mellem knuder, på oversættelsestidspunktet, er det et uopnåeligt ideal. Bare at udføre en brugbar (dvs. en målbart ydelses-fremmende) statisk analyse, kræver omfattende arbejde med udvikling af en oversætter, eller evt. modifikation af en eksisterende. Det er i den forbindelse værd at bemærke sig, at det ikke har været muligt for mig at finde eksempler på systemer der faktisk forsøger at implementere denne metode, selvom nogle systemer [18] baserer sig på en antagelse af, at en brugbar statisk analyse kan udføres i praksis.

Ivrig fremsendelse af seneste skrivninger En statisk analyse kan altså ikke give et fuldstændigt billede af, hvilke delingsmønstre der opstår under eksekvering. Et alternativ er at forsøge at nedbringe mængden af læsefejl, ved at ivrigt — dvs. *spekulativt* — at fremsende data ved synkronisering ved at fastsætte en ønsket minimumsstørrelse af en besked mellem knuder [18, p. 26]. Metoden bygger på en iagttagelse af, at forholdet mellem opstartstid og båndbredde for kommunikation på datanet med den sigende båndbredde favoriserer en protokol der begrænser mængden af beskeder. Ved at fremsende data der ikke eksplicit er blevet rekvireret af den modtagende knude, forsøger man at undgå eventuelle rekvireringer af data, udløst af forsøg på læsning af invaliderede hukommelsesområder.

Metoden har den store fordel, at den er yderst simpel at implementere. Fastsættes den ønskede mindstestørrelse af en besked til f.eks. 8Kb, fremsendes de seneste 8Kb opdaterede data som passagerer (piggy-backed) på den besked, der anderkender frigivelse af en lås eller barriere. Endvidere er det en rimelig antagelse at de senest modificerede data er af interesse for knuden der overtager ejerskab af en lås eller barriere, da alle programmer i større eller mindre grad udviser spatiel og temporær lokalitet.

Simpeltheden er uheldigvis også algoritmens svaghed. I praksis er den optimale minimumstørrelse af beskeden voldsomt afhængig både af det eksekverede program, og ofte af de inddata det behandler. Metoden gør intet forsøg på dynamisk at opnå viden om det aktuelle programs opførsel, men baserer sig udelukkende på en antagelse om, at der er en fordel i spekulativ fremsendelse af oplysninger.

Der er ingen tvivl om at ivrig og spekulativ fremsendelse af data kan hæve ydelsen for distribuerede programmer, men man bør være yderst forsigtig med at udføre denne fremsendelse ud fra for simple kriterier fremsendelse af data er nemlig ikke ubetinget et gode. Selvfølgelig er der ofte behov for udveksling af data ved en synkronisering mellem knuder, men disse data er ikke nødvendigvis de senest opdaterede. Et eksempel er et program, der benytter en arbejdskø til opbevaring af partielle og fuldstændig løsninger. Delproblemer tages på vanlig vis fra hovedet af køen, og nye delproblemer og løsninger tilføjes i halen. Ved overgivelse af låsen, der sikrer udelelig adgang til køen, er det sikkert at lagerblokken med head- og tailpointer til køen er af interesse for den modtagende knude, men de data der er tilføjet køens hale er irrelevante.

Brugen af fremsendelse af seneste lagerændringer kan altså i visse tilfælde føre til, at ydelsen faktisk sænkes betragteligt, da man her løber en stor risiko for fremsendelse af irrelevante oplysninger. Hvad værre er, skader dette ikke bare ydelsen, men også skalérbarheden af systemet. Da der ikke er nogen mulighed for feedback i algoritmen, er det ikke muligt at undersøge om brugen af denne metode virker hæmmende på systemets ydelse, eller for den sags skyld om størrelsen af minimumsbeskeden med fordel kan justeres.

Dynamisk opdateret heuristik En anden metode til at opnå fordelene ved amortisering af opstartstiden ved netkommunikation er, at benytte en dynamisk heuristik til at bestemme, hvilke data der spekulativt skal fremsendes. Som nævnt kan man med rimeligehed antage at distribuerede programmer tilgår lageret med spatiel og temporær lokalitet, men det er ikke givet at de seneste opdateringer af lageret alle er af interesse for den knude der overtager ejerskabet af en kritisk region. I stedet kan man forsøge dynamisk at opbygge en viden om delingsmønstre for dele af lageret, og basere den spekulative fremsendelse på dette.

Det er oplagt at basere denne heurestik på en eller flere af følgende antagelser:

- En del af lageret der har været mobil i den nærmeste fortid, vil have en tendens til at være mobil i den nærmeste fremtid.
- En del af lageret der har været modificeret af en given knude i den nærmeste fortid, vil have en tendens til at være af interesse for denne knude i den nærmeste fremtid.
- En del af lageret der er blevet rekvireret i forbindelse med overgivelse af en given kritisk region i den nærmeste fortid, vil have en tendens til at være mobil i forbindelse med den lås der håndterer adgang til den region i fremtiden.

Denne form for opførsel kalder jeg i denne opgave under ét for *affinitet*, og i de tre nævnte tilfælde kan man altså tale om at en del af lageret kan være affin for generel mobilitet, for en given knude eller for en given lås eller barriere. Udfordringen for DSM-systemet ligger her i at afgøre opdelingen af lageret i blokke med hver deres affinitet, og evt. i at gennemføre en passende feed-back, for at dæmpe rundsendelse af oplysninger.

I praksis er dynamisk heuristik kun blevet brugt i en begrænset form, og det endda kun i enkeltstående tilfælde [7] [12]. En vurdering af denne metodes styrker og svagheder må derfor bero på et skøn, snarere end praktiske erfaringer. Det er dog klart at afdækning og brug af mobil affinitet ville gavne programmer som FFT, og at affinitet til en kritisk region ville gavne programmer som TSP og Water. Om affinitet til en knude vil være gavnlig er mindre klart. Metoden er forsøgt brugt i praksis i en udgave uden mulighed for feedback [12], og havde tilsyneladende en gavnlig effekt på udførslen af visse typer programmer. Uheldigvis var prisen for fremsendelse af irrelevante data på det anvendte system så høj, og den mindste delingsenhed så stor (en hel lagerside) på det målte system, at en ren invalidate protokol gav bedre resultater selv om den udløste udveksling af et markant større antal beskeder. Dette fremgår ikke umiddelbart af artiklen, men da anden forskning i et tilsvarende system [11] har givet det overraskende resultat, at en serialisering af skrivninger til lagersider ingen nævneværdig forskel gjorde på ydelsen, må man konkludere at den benyttede diff-metode til indførelse af dynamisk delingsenhed er så ineffektiv, at den maskerer en eventuel negativ effekt af serialiseringen.

Opsummering Et SMLB-system bør som udgangspunkt have en mekanisme til håndtering af rundsendelse af data, som et fleksibel nok til at give en god allround ydelse for alle former for distribuerede programmer. Målet med SMLB er jo netop at være et generelt værktøj til distribueret programmering, og hvis systemet kun egner sig til visse former for programmer, har man sat generaliteten over styr. I praksis er der formegentlig umuligt at lave et system der egner sig til alle typer programmer, men det bør være det der er sigtet med forskningen.

Det er klart at det ikke er muligt at basere et DSM-system alene på en statisk analyse. Der er ikke bare tale om en manglende forsknings- og programmeringsindsats, men snarere om at en perfekt statisk analyse er teoretisk umulig at gennemføre. Konsekvensen er enten at benytte statisk analyse sammen med en anden mekanisme, eller at acceptere dårlig ydelse da aliasproblemer vil tvinge systemet til at invalidere hele det delte lager ved afslutningen af næsten hver eneste kodefragment. Fremtidig forskning i statisk analyse bør altså koncentrere sig om at udvikle metoder der støtter, snarere end erstatter, en supplerende dynamisk strategi.

Hvis målet er en god all-round ydelse på alle former for distribuerede programmer, er det altså nødvendigt at fortsætte forfiningen af de adaptive protokoller. Den rene invalidate-løsning medfører udveksling af alt for mange meddelelser mellem knuderne, og den rene update protokol begrænser ydelsen ved rundsendelse af irrelevant information.

Dynamisk heuristik er derfor nok det bedste bud på en god protokol til SMLB systemer. Erfaringer fra implementation af virtuel hukommelse i operativsystemer viser, at der med fordel kan benyttes avancerede strategier til udvælgelse af data til transport over en langsom kanal — en situation der på mange måder er analog til denne problematik. Udfordringen ligger i at udvikle metoder til afdækning af lagerets opdeling i regioner med forskellig opførsel, og afgøre hvilken affinitet de enkelte blokke har.

3.2.5 Størrelse af delingsenhed

Det er som nævnt normalt at DSM-systemer kun understøtter programmer der er DRF. Det er både et spørgsmål om at dette tillader brug af mere afslappede og højtydende modeller for opdatering af lageret, og at programmer der ikke er DRF sjældent kan garanteres korrekt udførsel, selv på systemer der håndhæver sekventiel lagerkonsistens. En interessant sideeffekt er, at på et system der udfører et program der er DRF, vil de lagerområder to knuder ønsker at synkronisere enten være disjunkte ud fra en kausal ordning, eller et overlap mellem to opdaterede lagerområder med samme tidsstempel vil ikke kunne påvirke programmets uddatas korrekthed [2, p. 13].

En lagerstyringsmekanisme med en tilstrækkeligt fin opløsning vil altså kunne udføre en utvetydig synkronisering af lageret mellem to knuder der synkroniserer. Er den mindste delingsenhed (Unit of Constency eller Unit of Coherence, UC) større end programmets faktiske mindste delingsenhed, er der risiko for delingsproblemer [19, pp. 336..337] [18, p. 6..7]. Udfordringen for udvikleren af et DSM-system ligger altså enten i at kunne opretholde et tilstrækkeligt finkornet billede af det delte lager, eller at sikre at et system der benytter en grovkornet opløsning vil kunne fungere korrekt.

Det klassiske delingsproblem kaldes for ping-pong effekten eller falsk deling [2, 7..9]. Hvis to knuder ønsker at ændre hver deres objekt samtidigt, og disse to objekter er placeret indenfor samme delingsenhed, vil denne blive sendt frem og tilbage mellem de to knuder i et system der giver eksklusiv skriveret til en enkelt knude af gangen (Single Writer, SW). Delingenheden — typisk en lagerside — spiller "ping-pong" mellem knuderne.

Mange SMLB-systemer benytter en ret grovkornet opløsning i deres billede af det delte lager. Dette gælder især de systemer der benytter en almindelig arbejdsstations MMU som mekanisme til bogføring af skrivninger — de sidebaserede systemer. De første systemer af denne side benyttede en single-writer protokol, hvor kun en knude af gangen havde tilladelse til at skrive til en given side i lageret [19, pp. 346..353]. Metoden var inspireret af de eksisterende hardwarebaserede DSM-systemer, men var altså i praksis hæmmet af den grovere opløsning i håndtering af lageret.

Problemet kendes også fra systemer der ikke er sidebaserede. Et eksempel er objektbaserede DSM-systemer, hvor skriveadgang til objekter i det delte lager typisk er eksklusiv for en enkelt knude af gangen [19, pp. 368..369]. Her kan effekten som nævnt i afsnit 2.4.4 være endnu mere udtalt, da der effektivt ikke en nogen øvre grænse på delingsenhedens størrelse. Der er altså en reel risiko for, at en programmør der ikke er klar over disse mulige sideeffekter kommer til at opleve alvorlige ydelsesproblemer.

Som nævnt er serialisering af skrivninger til områder i det delte lager absolut ikke en forudsætning for korrekt udførsel af programmer der er DRF. En serialisering af skrivninger til en region af lageret er altså ofte en uønsket effekt, og hæmmer selvfølgelig skalerbarheden. I et forsøg på at forbedre ydelsen har de fleste moderne SMLB-systemer indført metoder at tilvejebringe den nødvendige fine opløsning i lageradministrationen.

For de sidebaserede systemer vedkommende opereres med en såkaldt *diff*. En diff oprettes ved at tage en kopi af en given lagerside før første skrivning effektueres, og senere samholde kopien med sidens indhold ved synkronisering. Differencen — deraf navnet diff — er så de ændringer der er lavet på denne lagerside. At oprette en diff er en relativt tidskrævende operation, da den involverer store mængder kopiering og sammenligning af data, og i systemer der benytter denne metode udløses oprettelse af en diff da også kun i de tilfælde hvor to eller flere kunder faktisk signallerer ønske om at ændre samme lagerside.

Uheldigvis stiger behovet for oprettelsen af en diff med mængden af samarbejdende knuder. Det er endvidere intuitivt indlysende, at sandsynligheden for at to knuder skal skrive til samme lagerside (og der derfor skal oprettes en diff) er stigende med antallet af knuder, størrelsen af sider og antallet af elementer lageret opdeles i, under antagelse af tilfældig fordeling af elementer på knuderne.

Da der er visse tegn på, at størrelsen på lagersider er stigende i moderne arkitekturer i et forsøg på at holde størrelsen af TLB i processorerne nede på et effektive niveau på trods af den generelt voksende mængde RAM arbejdsstationer udstyres med [6, pp. 446..447], vil dette problem formegentlig ikke blive mindre på fremtidens datamater.

En anden metode er at benytte en dynamisk enhed for deling. En metode der benyttes i ORDER [18] — Low Level Objects, LLO — opdeler lageret i disjunkte områder med en opløsning der kan være så fin som et enkelt maskinord. Opdelingen foretages på grundlag af det kørende programs delingsmønstre, og opdateres konstant under eksekveringen. Med denne metode undgås falsk deling fuldstændigt, hvis ellers brugerprogrammerne er DRF.

Ulemperne ved brug af LLO er, at man risikerer at administrationsbyrden for DSM-systemet bliver en hæmsko. Da lageret principielt kan blive opdelt i et objekt pr. maskinord, skal systemet typisk administrere hundredetsinder eller måske endda millioner af objekter, hvor et sidebaseret system kun skulle administrere et par tusinde lagersider. Endvidere forudsætter denne metode, at det er muligt at bogføre tilgang til lageret med en granularitet på et enkelt maskinord om nødvendigt. Da en udtømmende statisk analyse af tilgang til lageret på oversættelsestidspunktet som nævnt er uladsiggørlig på grund af aliasproblemer, er dette på almindelige arbejdsstationer kun muligt ved anvendelse af MMU-hardware og diffs, eller ved at forudsætte at programmøren benytter primitiver til at tilgå det delte lager.

Konklusion Det er klart, at sidebaserede DSM-systemer har store skalérbarhedsproblemer, på grund af den store delingsenhed, og metoden der benyttes til bogføring af skrivninger. Disse problemer kan forventes at blive mere udtalte med tiden, da udviklingen indenfor datamatarkitektur vil medføre en endnu større ydelsessænkning i forhold til almindelig programkode ved anvendelse af disse metoder.

Brugen af en dynamisk delingsenhed ser altså mere og mere attraktiv ud, omend metoden forudsætter en større grad af diciplin fra programmørens side. Eksperimenter med brugen af LLO til implementation af en dynamisk delingsenhed har vist en tilfredsstillende ydelse ved simple testprogrammer, men var ikke i stand til at opretholde dette forspring i forhold til sidebaserede systemer ved udførsel af større benchmark-programmer [18, p. 80]. Årsagerne til dette misforhold er ikke umiddelbart klare, men formodes at være det specifikke systems dårlige håndtering af barrierre-synkronisering.

Dette indikerer at det vil være muligt at implementere et LLO-baseret system, der vil være de sidebaserede systemer overlegne i ydelse på dagens arbejdsstationer og datanet, hvis det er muligt at løse de meget specifikke ydelsesmæssige problemer der kendes fra ORDER.

3.2.6 Tidsforbrug på bogføring af læsninger og skrivninger

Et SMLB-system kan ikke implementeres uden en mekanisme til sporing af læsning eller skrivning til det delte lager, da det ellers er umuligt at sikre et tilstrækkeligt opdateret billede af det delte lager på den enkelte knude uden programmørens mellemkomst. I et hardwarebaseret DSM-system håndteres dette af et lager, der er i stand til at spore tilgang til lageret med en meget høj opløsning, i visse tilfælde helt ned til et enkelt maskinord.

En softwarebaseret SMLB-implementation på almindelige arbejdsstationer har også adgang til hardware der kan udføre denne funktion, da alle moderne processorer er udstyret med en memory-management unit (MMU), så de er i stand til at tilbyde virtuel hukommelse til programmer. Uheldigvis er denne MMU optimeret til virtual memory, og det betyder i praksis at den har en langt grovere opløsning i sporing af skrivninger end en MMU designet til DSM-formål. Som udvikler af SMLB-systemer på netværk af arbejdsstationer må man enten benytte den eksisterende MMU — et såkaldt sidebaseret DSM-system — og finde en supplerende metode til håndtering af de begrænsninger og problemer det medfører, eller benytte en alternativ mekanisme.

Indenfor de sidebaserede systemer finder man to typer: Dem der kun tillader én knude af gangen at have skriveadgang til en given side (single-writer), og dem der tillader flere knuder skriveadgang til samme side samtidigt (multiple-writers). Generelt har systemerne med multiple writers protokol bedre ydelse end signle-writer systemerne, men det er ikke klart at dette skyldes protokollen — i [11] argumenteres for at det kan skyldes at disse systemer er mere moderne, og benytter mere afslappede protokoller som LRC.

I multiple-writers protokoller på sidebaserede systemer benyttes en såkaldt *diff* til at afdække ændringer i lageret. Alle lagersider markeres som skrivebeskyttede, og første skrivning til en side udløser et interrupt. Her tages en kopi af siden, som senere — ved synkronisering — kan sammenlignes med siden. På den måde kan man afgøre, hvilke dele af siden er blevet ændret.

Denne mekanisme tillader at læsning til lageret kan udføres med normal hastighed, men skrivning vil udløse et administrationsarbejde, med deraf følgende tidsforbrug. En yderligere omkostning er det førnævnte interrupt, der både tager tid, og vil medføre skjulte omkostninger i form af TLB— og cacheforurening. Hvis systemet benytter en invalidate protokol er der også administrationsmæssige omkostninger i form af at markere om lagersiderne er opdateret lokalt, og sætte de nødvendige læsetilladelser herefter. Tidsforbruget på dette må dog antages at være minimalt.

Blandt de systemer, der ikke er sidebaserede, findes tre metoder: Compilergenererede annotationer, suppleret af annotationer udført af programmøren [5] [18], annotationer indsat af programmøren [8] og single-writer protokol baseret på objekter [19, sektion 6.6.3].

Et alternativ til disse metoder er tilgang til delt lager gennem primitiver. Dette kræver enten at programmøren annoterer samtlige tilgange til delt lager, eller at systemet stiller typer til rådighed der implementerer denne funktionalitet transperant. Fordelen ved denne metode er, at programmøren ikke som sådan skal have et perfekt overblik over programmets tilgangsmønstre, da denne vide bliver opsamlet under eksekveringen. Ulempen er, at disse oplysninger skal registreres, hvilket medfører et forøget administrationsarbejde i forhold til de andre softwarebaserede metoder. Dersom denne metode kan implementeres transperant, vil den tilbyde en stor grad af komfort for programmøren.

3.2.7 Administration af synkroniseringsprimitiver

Administration af synkroniseringsprimitiver kan være et ydelsesmæssigt kritisk punkt i et SMLB-system. Der indgår to forskellige arbejdsopgaver i dette: Overgivelse af ejerskab og synkronisering af lager mellem knuderne.

For at starte med overgivelse af ejerskab er dette primært et problem i forbindelse med låse, da det ved barrierer vides at alle knuder skal ankomme til synkroniseringspunktet før eksekvering kan fortsætte, og der derfor ikke skal etableres viden om ejerskab af en kritisk region. For at kunne overtage ejerskab af en lås og den tilhørende kritiske region, skal en knude kontakte den nuværende ejer af låsen. Dette kan enten gøres ved brug af broadcast kommunikation, eller ved brug af en *velkendt mellemmand*, der er i besiddelse af denne viden. Da brug af broadcastkommunikation er hæmmende for skalérbarheden i et SMLB-system, er den velkendt mellemmand i de fleste tilfælde den bedste løsning.

Hvis brugerprogrammet benytter mange låse, medfører brugen af velkendte mellemmænd risiko for, at der opstår en flaskehals i systemet. Dette kan afhjælpes ved at benytte flere mellemmænd, som hver har ansvaret for deres gruppe af låse.

I mange SMLB-systemer implementeres barrierer ved brug af låse, men denne praksis kan ikke anbefales. Der er tale om to fundamentalt forskellige operationer, i og med at overgivelse af ejerskab af en lås er en 1 til 1 kommunikation, og ankomst til en barriere er n til n kommunikation. Barrierer giver desuden mulighed for at synkronisere lageret fuldstændigt på alle knuder, hvilket med fordel kan udføres med en specialiseret mekanisme — i denne situation er det f.eks. ikke nødvendigt at overføre viden om, hvilke knuder der har set hvilke lageropdateringer (copysets), da *alle* knuder kommer til at se alle lageropdateringer.

Tilbage er arbejdet med at afgøre, hvilke lageropdateringer der skal overføres ved synkronisering. I de fleste systemer benyttes en kombination af copysets og Lamport-tid (eller tilsvarende mekanisme, såsom vektortid) til at afgøre hvilke opdateringer der skal registreres på knuderne for at opfylde happens-before relationen. Men måden man finder kandidaterne til rundsendelse kan være afgørende. Her benytter de fleste systemer en søgning over hele den samlede mængde af konsistensinformation, men en anden mekanisme er med fordel benyttet i Midway [5]. Her opretholdes en kø af opdateringer for hver af de andre knuder i systemet, og lokale skrivninger samt indsamlede opdateringer ved låse distribueres til disse under hensyntagen til et copyset. Således er det ikke nødvendigt at udføre en tidskrævende gennemsøgning af konsistensinformationen — man overfører blot opdateringskøren, og tømmer den derefter.

3.3 TreadMarks

TreadMarks (TM) [2] er et eksempel på et moderne sidebaseret SMLB-system. TM benytter Lazy Release Consistency (LRC), og implementerer dette gennem en multiple-writers invalidate protokol. For at forøge ydelsen benytter TM en skræddersyet letvægtsprotokol baseret på UDP til datatransport, snarere end en eksisterende standardprotokol såsom TCP.

Brugen af multiple writers protokol modvirker de ydelsesmæssige problemer som sidebaserede DSM-systemer ofte lider af i forbindelse med *falsk deling*. Forsøg indikerer dog at denne mekanisme er så ydelsesmæssigt krævende, at effekten er neglicherbar [11].

TM udemærker sig ved at opnå en god ydelse i forhold til tidligere sidebaserede systemer som IVY [15]. Dette menes i høj grad at skyldes brugen af LRC, der kraftigt begrænser mængden af datakommunikation i forhold til mere restriktive lagermodeller som Sequential Consistency.

TM må også betegnes som et ganske brugervenligt (eller måske snarere programmørvenligt) system. Alle delte data allokeres ved brug af en funktion tilsvarende *malloc*, og tilgås gennem pointere på linje med alle andre dynamisk allokerede data i C.

Kritik Det faktum at alle delte data erklæres dynamisk, medfører to ulemper for programmører: Allokering er hverken automatisk eller typesikker, og pointertilgang til data er en notorisk fejlkilde — enkelte moderne sprog som Java elliminerer af denne grund helt pointere fra sproget.

Ydelsesmæssigt har alle kendte målinger målinger vist at TM leverer en glimrende ydelse, som må siges at være blandt de bedste der kendes fra SMLB-systemer. Det faktum at TM baserer administrationen af hukommelseshierakiet på MMU funktioner, som må forventes at blive relativt dyrere i forhold til mere almindelige ordre i moderne maskiners instrukssionssæt, indikerer at TM på moderne maskiner vil være ude af stand til at opretholde sit høje ydelsesniveau.

3.4 Midway

Midway [5] benytter som sin primære konsistensmodel Entry Consistency (EC). Dette gennemføres i praksis gennem en kombination af compilergenererede og programmørgenererede annotationer. Mere konkret associerer programmøren data til en kritisk region, og systemet sørger på den baggrund for at opdatere de relevante dele af lageret ved indgang til en kritisk region.

Midway tilbyder andre, og mere brugervenlige, konsistensmodeller, som frit kan blandes med EC i samme program. Formålet er at lette konvertering af programmer fra f.eks. Release Consistency til EC, ved at gøre det muligt at gennemføre denne konvertering trinvist.

Midway leverer en fortræfelig ydelse med benchmarkprogrammer, men denne ydelse opnås kun hvis programmerne er konverteret til EC modellen. Dette konverteringsarbejde er typisk meget tidskrævende, og kan kun udføres af en programmør der kender brugerprogrammets tilgangsmønstre.

Kritik To faktorer har medvirket til at hæmme Midways anvendelse: Kravet om at en specialiseret oversætter skal benyttes, og det faktum at konvertering af programmer til EC modellen er ganske arbejdskrævende.

3.5 ORDER

ORDER [18] er ikke tænkt som et udviklingmiljø for programmører, men snarere som en platform en oversætter kan generere programmer til. Således er det nødvendigt — om muligt – explicit at angive hvilke hukommelsesregioner i det delte lager der skal opdateres ved synkronisering for at opnå god ydelse. Systemet er udviklet under antagelse af, at det er muligt at konstruere en oversætter der kan afdække delingsmønstre i en grad, hvor der kan opnås god ydelse med de fleste brugerprogrammer.

I lighed med Midway implementeres hukommelseshierakiet 100% i software, og der gives adgang til andre konsistensmodeller. Snarere end at tilbyde Release Consistency, kan et ORDER program benytte Message Passing i samme program som SMLB-paradigmet. ORDER er implementeret som en udvidelse af MP DSM-systemet CARLOS, og de to programmeringsmodeller kan således blandes frit i samme program. Denne konsistensmodel kaldes Partial Message-Driven Relaxed Consistency.

En markant detalje ved ORDER er, at det anvender en ren update protokol i et forsøg på at nedbringe latenstid og mængden af meddelelser der udveksles over netværket.

Den måske vigtigste designmæssige detalje ved ORDER er, at der benyttes en dynamisk delingsenhed. Dette medfører at falsk deling fuldstændigt undgås. Denne dynamiske delingsenhed er implementeret ved brug af en såkaldt Low Level Objects mekanisme.

Ydelsesmæssigt udviser ORDER fremragende ydelse med simple testprogrammer, men har problemer med at opretholde denne høje ydelse ved kørsler med større benchmarkprogrammer. Dette henføres til at synkronisering i ORDER er en tidsmæssigt dyr operation, hvilket medfører at ORDER er meget sårbart overfor programmer der synkroniserer kun for at synkronisere, snarer end for at udveksle data.

Kritik Som nævnt i afsnit 2.4.3 er det bla. på grund af aliasproblemer umuligt at udvikle en oversætter, der statisk kan afdække delingsmønstre ved oversættelsen. I praksis medfører dette at en programmør der vil konvertere programmer til ORDER skal udføre samme arbejde som ved konvertering af programmer til Midway.

En af årsagerne til at især barrierer er en ydelsesmæssigt dyr operation i ORDER er, at der aktivt forsøges at nedbringe mængden af konsistensinformation ved "sammensmeltning" af Low Level Objects der ligger i forlængelse af hinanden, og har samme Lamport-tid. Det er ikke klart at dete har en ydelsesmæssigt positiv effekt, udover at nedbringe hukommelsesforbruget for systemet. Hvis opdelingen af lageret i Low Level Objects er et udtryk for det aktuelle brugerprograms tilgangsmønster, vil dette arbejde jo være spildt, i og med at de sammensmeltede objekter i den nærmeste fremtid kan forventes at skulle deles igen, hvis bruegrprogrammet vedbliver med at udvise samme tilgangsmønster.

Da der er meget store forskelle på strukturen af MP og SLMB programmer, betragter jeg det som tvivlsomt at der kan udføres trinvis raffinering af programmer ved overgang fra SMLB til MP som programmeringsparadigme.

3.6 Samlet diskussion og kritik af de beskrevne DSM-systemer

For de tre beskrevede systemer gælder at kun et af dem — TreadMarks — er i stand til at levere en god ydelse med eksisterende programmer, uden der gennemføres en arbejdskrævende konvertering og tilpasning.

For ORDER og Midway gælder endvidere at de er meget afhængige af specialiserede oversættere. En sådan oversætter findes overhovedet ikke til ORDER, og portering af en oversætter anses generelt for at være meget arbejdskrævende.

Hvad angår programmeringsmodeller, er der alvorlige mangler ved alle tre systemer. TreadMarks kræver at delte data erklæres dynamisk og tilgås ved dereferering, hvilket er en alvorlig fejlkilde. Konvertering af eksisterende programmer til ORDER og Midway kan kun udføres af en erfaren programmør, der på forhånd kender programmet indgående. Nyudvikling til disse to systemer må også forventes at være besværligt, da der kræves et godt overblik over programmets tilgangsmønstre før der garanteres korrekt eksekvering.

På den baggrund må TreadMarks nok siges at være det bedste "all round" system, da det både ved konvertering og nyudvikling stiller færrest krav til programmøren, og generelt leverer en god ydelse uden at kræve en gennemgribende tuning af programmet til den specifikke platform.

3.7 Forslag til designvalg i fremtidens SMLB-systemer

På baggrund af den forudgående fremlæggelse af problemstillinger og diskussion af mulige løsninger, giver jeg her et bud på hvilke metoder jeg mener med fordel kan implementeres for at konstruere en brugervenligt SMLB-system der kan levere en høj ydelse. Forslagene er opdelt i ydelsesmæssige tiltag og betjeningskomfortmæssige tiltag.

3.7.1 Overordnet designfilisofi

Først og fremmest er det vigtigt at have den rigtige tilgang til design, når man udvikler software for at opnå høj ydelse. Der findes ikke nogen enkelt forbedring, der alene sikrer god ydelse — der er snarere tale om en synergieffekt mellem arkitektoniske og designmæssige tiltag, der supplerer hinanden.

Der er dog visse grundlæggende antagelser man som udvikler kan gøre sig, der vil fremme en god ydelse på det resulterende system. Disse er som følger:

- Brugen af "fast paths", dvs. optimering efter de hyppigst optrædende situationer. Dette følger af grundsætningen "Make the common case fast" fra [6].
- Antag at der udvises spatiel og temporær lokalitet i brugerprogrammet.
- Udnyt parallellitet ved at overlappe kommunikation og beregning.
- Benyt velkendte benchmarkprogrammer og profileringsværktøjer til ydelsesmåling, og derpå følgende tuning. Sammenlign gerne med eksisterende SMLB-systemer, da dette kan hjælpe med at afdække årsagerne til evt. dårlig ydelse.

3.7.2 Ydelsesfremmende tiltag

Følgende arkitektoniske tiltag kan forventes at fremme en god ydelse i et SMLB system.

- Dynamisk delingsenhed.
- Dynamisk afdækkelse af lagerblokkes mobilitet.
- Brug af opdateringskøer i stedet for søgning i al konsistensinformation til at afgøre hvilke opdateringer skal udveksles ved synkronisering.
- Benyt et hukommelseshieraki der minder om det, der kendes fra moderne arbejdsstationer. Dette vides at have en positiv effekt på de fleste programmer, og det er en rimelig antagelse at dette også gælder for distribuerede programmer.
- Giv programmøren mulighed for at gennembryde "den semantiske mur" i SMLB-paradigmet. Tilbyd muligheder for direkte at kontrollere udlægning af data i lageret og rekvirering af opdateringer ved indgang i kritisk region.

3.7.3 Betjeningskomfort og brugervenlighed

Følgende designpunkter bør tages i betragtning, da de vil medvirke til at gøre konvertering og nyudvikling på systemet nemmere.

- Tilbyd en ren SMLB programmeringsmodel, der ikke kræver at programmøren har et totalt og korrekt overblik over tilgangsmønstre og dataflow for at opnå acceptabel ydelse.

- Gør flytning fra andre SMLB systemer nemmere, ved som minimum at tilbyde semantisk ekvivalente funktioner til alle funktioner der er til rådighed i andre systemer.
- Tilbyd om muligt en model for allokering og tilgang til delte data, som er funktionelt ekvivalent til brugen af normale variable.

4 SENSE - design, brug og implementation

På baggrund af argumentationen og diskussionen i det foregående, designede og implementerede jeg et SMLB DSM-system — SENSE. Systemet kan kort klassificeres som et SMLB-system med dynamisk delingsenhed, lazy-release protokol, valg mellem tre protokoller (invalidate, update, hybrid) ved låse og to protokoller (invalidate, update) ved barrierer. Systemet er implementeret i C++ uden brug af MMU-hardware, og benytter TCP som transportprotokol. Endvidere kræver det adgang til Standard Template Library (STL) og Posix-tråde på kerneniveau. Systemet er udviklet under et UNIX-lignende operativsystem (Linux), men er ikke bundet til dette, eller for den sags skyld nogen andre UNIX-faciliteter end de nævnte.

Navnet SENSE er – ukarakteristisk i UNIX-verden – *ikke* et akronym, men primært valgt ud fra den glæde det ville give mig at kunne skrive:

```
make sense
```

for at oversætte. En anden, og mere alvorlig, grund til navnet var, at jeg som en overordnet målsætning ønskede at demonstrere, at man ved anvendelse af almindelig fornuft (common sense) kunne nå frem til en implementation af SMLB, som var både effektiv og overskuelig.

4.1 Målsætninger

Som grundlag for designet af SENSE, fastlagde jeg en række målsætninger, som blev lagt til grund for de konkrete designvalg. Disse var i prioriteret rækkefølge:

1. Ydelse — programmer skulle kunne udføres med en ydelse der var sammenlignelig med eksisterende SMLB-systemer på moderne netværk og datamater.
2. Skalérbarhed — programer eksekveret under SENSE skulle opnå speed-up der var sammenlignelig med eksisterende SMLB-systemer.
3. Kort udviklingstid — jeg ønskede at demonstrere, at det er muligt at udvikle et SMLB-system på relativt kort tid.
4. Brugervenlighed — programmørens arbejde skulle gøres så let som muligt.
5. Kompatibilitet — systemet skulle så vidt muligt kunne køre eksisterende programmer skrevet til SMLB-paradigmet, uden det var nødvendigt at foretage andet end trivielle ændringer.

6. Portabillitet — systemet skulle være let at flytte til andre operativsystemer, maskinarkitekturer eller netværksprotokoller.

Selvom ovenstående liste var prioriteret, skal det ikke forstås som om jeg valgte at prioritere ydelse på *bekostning* af de andre mål, men snarere som at jeg valgte at implementere tiltag der forøgede ydelsen under hensyntagen til de lavere prioriterede punkter.

4.2 Designfasen

I det følgende beskrives hvilke konkrete designbeslutninger der blev taget for at leve op til de nævnte målsætninger. De kriterier jeg satte for at have opfyldt målsætningerne - successkriterier - bliver også udspecificeret her.

4.2.1 Ydelse

For at opnå høj ydelse kan man – groft sagt – benytte arkitektoniske og implementationsmæssige midler. Til de arkitektoniske regner jeg f.eks. valg af protokol, hvorimod programmeringsmæssige tiltag såsom eliminering af kopiering hører under implementation.

Med henblik på at holde udviklingstiden nede, valgte jeg at begrænse de arkitektoniske tiltag til dem, jeg forventede ville give den største forbedring af ydelsen. Disse blev på baggrund af argumentationen i de foregående kapitler som følger:

- Anvendelse af dynamisk delingsenhed for at undgå falsk deling.
- Update og hybridiseret update/invalidate protokol med henblik på at reducere mængden af beskedudvekslinger mellem knuderne.
- Undlad at konsolidere konsistensinformationen ved synkronisering.
- Oprethold køer af ventende opdateringer på en pr. knude basis.
- Undgå brug af broadcastkommunikation ved at benytte kendte mellemmænd til vidregivelse af ejerskab af låse.
- Barriere implementeres seperat, og ikke ved brug af låse.
- Brug af Lazy Release Consistency som konsistensmodel.

Med hensyn til implementationen, valgte jeg at benytte følgende teknikker:

- Bogføring af læsning og skrivning til delt lager udføres uden brug af MMU-faciliteter.
- Brug af "compilervenlige" programmeringsteknikker såsom udstrakt anvendelse af konstantdefinitioner og inlining af funktioner.
- Etablering af "fast paths" ved brug af specialicerede funktioner til håndtering af hyppigt forekommende situationer.
- Begrænsning af opslag i konsistensinformation ved detektering af sekventielle (eller gentagne) læsninger og skrivninger.

Det eneste af disse punkter der ikke er dækket i de forgående kapitler er det sidste. Denne implementationsdetalje er inspireret af opbygningen af hukommelseshierakiet i moderne datamater, og kan betragtes som en minimal cache.

Brugen af "fast paths" er den eneste af de nævnte designmetoder der er i modstrid med nogen af de andre målsætninger - kort udviklingstid - da det medfører at kildekoden bliver mindre generel, og der indføres flere mulige fejlpunkter. På den anden side er den ydelsesforbedring man kan opnå med denne metode ofte af anseelig størrelse, og teknikken er derfor ofte anvendt i kritiske dele af operativsystemer, såsom kernen og protokolstakken.

Af hensyn til ønsket om kort udviklingstid, valgte jeg at undlade at implementere flertrådethed i SENSE, udover den interne parallellitet der ville være nødvendig for at kunne håndtere flere kommunikationskanaler gennem sockets. Dette skete ud fra den betragtning at flertrådede programmer er væsentligt sværere at finde fejl i, da en hel klasse af fejl - væddeløbstilstande - optræder i denne sammenhæng. I stedet besluttede jeg mig for at holde døren åben for en fremtidig tilføjelse af flertrådethed under implementationen.

Succeskriterier Som succeskriterie for ydelsen valgte jeg, at opnå en ydelse der var stort set tilsvarende til TreadMarks med et eller flere (helst alle) de alment brugte benchmarkprogrammer (se afsnit 2.3), hvor "stort set" er defineret som indenfor en størrelsesorden. Dette mål kan måske virke uambitiøst, men er i virkeligheden ganske realistisk. SENSE er første version af et eksperimentelt system, og det kan derfor ikke forventes at det kører optimalt - i første omgang skal det bringes til at køre i det hele taget. Det er ikke unormalt at kunne forbedre ydelsen af et program med en størrelsesorden ved kritisk gennemgang af måleresultater og brug af profileringsværktøjer til identificering af flaskehalse. På baggrund af min argumentation i de foregående kapitler må målet siges at være realistisk - om det i praksis viser sig at være tilfældet, kan kun afklares ved konkrete målinger på ydelsen. Målet kan heller ikke siges at være uambitiøst, da selv en kneben opfyldelse af det (dvs. ydelse der konsekvent er en faktor 10 lavere end eksisterende systemer) peger i retning af, at en tilsvarende ydelse kan opnås i fremtiden.

4.2.2 Skalerbarhed

Da de designpunkter der blev blev valgt for at opnå høj ydelse, samtidigt kan antages at fremme skalérbarheden i systemet, var der ikke grund til at tilføje designpunkter her.

Succeskriterier En skalérbarhed på linje med TreadMarks for benchmarkprogrammerne ville være et tilfredsstillende resultat.

4.2.3 Kort udviklingstid

Udvikling af er DSM-system betragtes traditionelt som en meget stor programmeringsopgave. Estimeringer af tidsforbruget på udviklingen ligger sjældent på under et mandår for en erfaren programmør. På den anden side kan dette store tidsforbrug i de fleste tilfælde henføres til udvikling af skræddersyede transportprotokoller [18] [19, sektion 2.4.5]. For de sidebaserede systemer gælder også at man her anvender programmeringsteknikker der absolut ikke er i alment brug.

På grundlag af disse observationer besluttede jeg mig for at gøre det til en af de gundlæggende målsætninger for SENSE projektet at bevise, at et SMLB-system kan implementeres med et mere overskueligt tidsforbrug.

For at opnå dette mål, valgte jeg at benytte følgende teknikker:

- Brug af eksisterende netværksprotokoller til datatransport.
- Brug af standardbiblioteker og generiske algoritmer.
- Objektorienteret og modulariseret udvikling

Jeg valgte at bruge TCP-protokollen og Berkley Sockets (normalt blot kaldt sockets) til datatransport. TCP giver som beskrevet i afsnit 2.4.2 på moderne datamater og datanet tæt på den teoretisk maksimale hastighed for en protokol der er stabil, forbindelsesorienteret, punkt-til-punkt og bidirektionel. Sockets er nok den mest udbredte programmeringsmodel til kommunikation over datanet, især i forbindelse med TCP. Brugen af sockets i sammenhæng hvor et program skal være i stand til både at modtage og sende data, medfører i praksis at man benytter multiprogrammering (multitasking). Dette besluttede jeg mig for at gøre gennem brug af POSIX letvægtstråde, da denne facilitet for det første findes på en lang række operativsystemer og datamater, og for det andet kan forventes at være en meget effektiv metode til håndtering af opgaven.

Med hensyn til standardbiblioteker, stod det på et tidligt tidspunkt klart, at der ville blive behov for en opslagsmekanisme - et såkaldt directory - til implementation af bogføring af skrivninger og læsninger til delt lager. Dette directory skulle give mulighed for at finde lagerblokke der indeholdt en given adresse. Det to klassiske metoder til implementation af dette er hash-tabeller og træer. Hashing giver mulighed for opslag i konstant tid, men kun under forudsætning af, at man præsenterer det for en nøgle der er i tabellen. Da dette ikke altid vil være tilfældet [18] i et DSM-system, valgte jeg i stedet at benytte et træ, da dette giver mulighed for at udføre denne operation med tidskomplektitet log2(n). Envidere fremgår det klart af beskrivelsen af ORDER [18] at netop implementationen af opslagsmekanismen med brug af hash-tabeller var meget arbejdskrævende.

På grundlag af dette, besluttede jeg mig for at benytte datatypen *map* fra Standard Template Library (STL) [17, sektion 19.8] — et bibliotek af datatyper, objekter og generiske objekter, som er en del af ANSI C++ standarden. Denne datatype er i STL implementeret ved brug af ballancerede binære træer, og vides at være yderst effektiv. Som det fremgår af [4, column 4] er netop implementation af denne slags fundamentale algoritmer notorisk arbejdskrævende, og de resulterende datatyper og funktioner er ofte fejlbehæftede. Det var derfor oplagt at benytte STL's faciliteter til opgaven.

Som et kompromis mellem høj ydelse og hurtig udvikling, valgte jeg at benytte C++ som programmeringssprog. Sproget kan, hvis det anvendes korrekt, give en ydelse der er på højde med C, og har desuden indbygget en lang række faciliteter der fremmer modularisering, kodegenbrug og overskuelighed. At STL giver standardiseret og platformuafhængig adgang til generiske algoritmer og datatyper er også et stort plus i denne sammenhæng.

Succeskriterier Succeskriteriet definerede jeg her som en udviklings- og afprøvningsfase på ca. en måned. Da dette vil være ca. en størrelsesorden lavere end det tidsforbrug der normalt forventes på implementation, vil det være et signifikant resultat, hvis det opnås.

4.2.4 Brugervenlighed

Som nævnt i afsnit 3.2.7 medfører valget af en ren softwarebaseret bogføring af tilgang til det delte lager, at dette skal ske gennem brug af primitiver, eller anden eksplicit erklæring af, hvilke dele af lageret det tilgås. Dette harmonerer dårlig med en målsætning om, at gøre systemet let at bruge for en programmør. På den anden side giver netop C++ mulighed for at indkapsle dette arbejde i et objekt, og siden definere de almindelige operatorer (+-*/= osv.) samt automatisk konvertering til og fra grundtyper som heltal og sandhedsværdier. Disse definerede operatorer bliver på oversættelsestidspunktet omsat til de relevante metoder, og brugen af dem medfører således ikke nogen ydelsesmæssige omkostninger.

Succeskriterier Som designmål og successkriterie valgte jeg, at det skulle være muligt at benytte en syntaks der svarer til den, der ses i programeksemplerne i appendix A.

4.2.5 Kompatibillitet

En stor del af kompatibiliteten med eksisterende programmer skrevet til andre SMLB-systemer opnås alene ved at stille de grundlæggende synkroniseringsprimitiver (låse og barrierer) til rådighed, og sikre at de er semantisk ekvivalente. Tilbage er allokering af variable i delt lager, og tilgang til disse.

Allokering udføres i de mest udbredte SMLB-systemer med en mekanisme, der svarer til dynamisk allokering af variable i C, dvs. med en funktion tilsvarende *malloc*, og tilgang til variable gennem pointere. Denne fremgangsmåde er ganske besværlig i forhold til brug af almindelige variable, men det er nødvendigt at give adgang til en tilsvarende mekanisme, da det ellers er nødvendigt at omskrive store dele af eksisterende programmer for at bringe dem til at fungere med SENSE.

Jeg valgte ikke at indføre en analog til Treadmark's *Tmk-distribute* funktion, da jeg for det første betragtede den som redundant - den synkroniserer indholdet af variable eksplicit mellem knuder, hvilket jo ellers er hele formålet med shared memory - og den for det andet er stort set umuligt at implementere, uden at skulle tildele en knude en speciel position som hovedknude. Jeg betragtede denne specialisering som værende i modstrid med målsætningen om hurtig udvikling.

Succeskriterier Hvad angår succeskriterier, valgte jeg at anse det som en success at få et brugbart program, såsom et af benchmarkprogrammerne, flyttet fra et andet SMLB system til SENSE på en arbejdsdag.

4.2.6 Portabillitet

Grundlæggende sikrer man at et program er portabelt (også kaldet *flytbart*) ved at undgå at basere det på specifikke faciliteter eller omstændigheder, der ikke er til rådighed på andre datamater, operativsystemer eller datanet (under et: platforme). Da de specifikke faciliteter jeg havde besluttet mig for at benytte (TCP, Berkley Sockets og POSIX-tråde) enten er til rådighed på langt fleste platforme, eller kan erstates med funktionelt tilsvarende faciliteter, stod det klart, at jeg kunne sikre en høj grad af portabillitet ved at indkapsle brugen af disse i objekter, og nøjes med at bruge den mest basale funktionalitet disse faciliteter stiller til rådighed. Hvad gælder ANSI C++ er dette sprog efterhånden tilgængeligt på alle moderne platforme (Linux, Windows, QNX, BeOS etc.).

Udover dette er det blot nødvendigt at sikre, at man ikke benytter platformspecifikke antagelser om størrelsen og repræsentationen af de grundlæggende datatyper, såsom at antage at datatypen *int* repræsenteres med 32 bit. I de tilfælde hvor dette ikke kan undgås, kan arbejdet med at flytte programmet til en anden platform gøres lettere ved at definere datatyper (eg. *byte*) og samle disse definitioner i et enkelt modul af kildekoden, så de er let tilgængelige.

Da disse betragtninger i det store og hele er i overensstemmelse med almindelig god programmeringspraksis, forventede jeg ikke at dette ville medføre en væsentlig forøgelse af arbejdsbyrden, og dermed være i modstrid med målsætningen om at nedbringe udviklingstiden.

Succeskriterier Portabilitet er den lavest prioriterede målsætning, da det ikke er en del af rationalet for denne opgave. Jeg valgte på trods af dette at tage det med som en målsætning, da det i forskningsøjemed er væsentligt at sikre at et program kan bringes til at køre på så mange platforme som muligt. På den baggrund forventede jeg ikke at bruge tid på at forsøge at leve op til denne målsætning - altså faktisk at gennemføre en flytning af SENSE til en anden platform - da det ville overskride rammerne for opgaven. Det skulle dog helst være muligt at flytte SENSE til en anden UNIX-lignende platform med mindre end en arbejdsuges indsats.

4.3 Implementations- og verificeringsfasen

Her beskriver jeg det arbejde jeg udførte og de erfaringer jeg gjorde under denne del af projektet. Beskrivelsen er opdelt i tre afsnit: programmeringsfasen, verificeringsfasen og tuningsfasen. Programmeringsfasen beskæftiger sig med implementeringen af det i de forudgående afsnit beskrevede design for SENSE. Verificering er den del af arbejdet der blev brugt på at verificere, at designet levede op til de funktionelle krav. Tuningen er den del af arbejdet, hvor systemet modificeres på grundlag af de indledende målinger, med henblik på at opnå en højere ydelse. Den endelige verificering af ydelsen er beskrevet seperat i det følgende kapitel.

4.3.1 Programmeringsfasen

Ud fra de nævnte designbeslutninger stod det klart, at programmet kunne anskues som et directory, der blev tilgået og vedligeholdt af brugerprogrammet og synkroniseringsprimitiverne, suppleret af et seperat program, der udfyldte funktionen som velkendt mellemmand ved overgivelse af kontrol over kritiske regioner - i det følgende kaldt *lockmanager*. Dette åbnede mulighed for at holde projektet overskueligt ved at opdele implementationen i flg. faser, som alle blev afsluttet med en kort testfase, der skulle sikre at alle implementerede komponenter fungerede korrekt. Faserne blev som følger:

1. Implementation af directory-mekanismen, herunder læsning og skrivning til det delte lager samt opretholdelse af køer med udestående opdateringer til de andre knuder.

2. Oprettelse af den basale kommunikationsinfrastruktur ved brug af sockets og POSIX-tråde, implementation af lockmanager og synkronisering af opstart og nedlukning af knuder.

3. Implementation af barrierer med invalidate-protokol.

4. Implementation af låse med invalidate-protokol.

5. Implementation af update og hybridiseret update/invalidate protokol for låse, samt update-protokol for barrierer.

Verificeringen beskrives seperat i følgende afsnit.

Implementationen gik nærmest forbavsende glat. Her var det en klar fordel af benytte standardkomponenter og faciliteter. Funktionalitet og anvendelse er for det første er meget veldokumenteret og generel, og for det andet gennnemprøvet i en grad, hvor jeg ikke én eneste gang oplevede at finde fejl og mangler i dem. Den praktiske værdi af dette kan nærmest ikke overdrives - ved udvikling med mindre modne værktøjer er det desværre min erfaring at det er reglen, snarere end undtagelsen, at størstedelen af tidsforbruget optages af lange, uproduktive søgninger i support-databaser. Fra et systemudviklingsmæssigt perspektiv er dette bemærkelsesværdigt, omend næppe overraskende.

Første fase - implementation af de basale datastrukturer - blev udført under Windows NT ved brug af Borland C++ Builder. Dette udviklingsmiljø egner sig i langt højere grad end den sædvanlige UNIX-kombination af emacs og gcc til en indledende designfase, da det indeholder faciliteter såsom en interaktiv debugger og kontekst-sensetiv hjælp. At starte implementationen på denne platform havde yderligere den fordel, at det fokuserede min opmærksomhed på portabillitet.

Resten af faserne blev udført med emacs og gcc under Linux, og bød igen ikke på nogen overraskelser. Jeg valgte dog at afvige fra planen, og implementere mulighed for en ren update protokol i forbindelse med barrierer før låsene blev implementeret, da det for det første viste sig at være yderst simpelt, og for det andet gav en markant højere ydelse i det valgte testprogram (3DFFT).

Opfyldelsen af målsætningen om brugervenlighed skete gradvist, og var ikke begrænset til nogen af faserne. Brugerprogrammer kan i princippet tilgå det delte lager ved brug af primitiverne *sense_write* og *sense_read*, samt allokere dele af lageret med funktionen *sense_malloc*, men da dette langt fra er praktisk, begyndte jeg allerede tidligt i fase 1 at indkapsle dette i et objekthieraki til håndtering af grundtyper (int, float, bool etc.) i delt lager. Oprindeligt blev disse delte variable tilgået genem metoderne *read* og *write*, hvilket fremgår af testprogrammerne i appendix B. Ved udgangen af fase 5 var dette udviklet i en grad, hvor delte variable kunne bruges fuldstændigt transperant på linje med almindelige variable.

Undervejs i udviklingen viste det sig, at det var relativt let at tilføje en mulighed for at spore alle tilgange til lageret. SENSE kan altså bruges i en rolle som profileringssystem for SMLB-programmer - noget der i høj grad kan være nyttigt, selv hvis man i sidste ende ønsker at eksekvere programmet på et andet system. Denne funktionalitet var ikke planlagt fra starten, men var så nem at tilføje, og af så stor potientiel nytte, at jeg valgte at implementere den.

Opsummering Programmeringsfasen blev gennemført uden problemer, i og med at det oprindelige design for opbygningen af SENSE blev implementeret uden ændringer. Der blev altså ikke afdækket nogle strukturelle mangler. Den praktiske del - selve programmeringen - blev også gennemført uden problemer, og illustrerede værdien af gode, gennemprøvede udviklingsværktøjer og standardbiblioteker.

Tidsforbrug Det totale tidsforbrug på programmeringsfasen var to almindelige arbejdsuger, dvs. ca. 10 arbejdsdage af hver 7 timer.

4.3.2 Verificering og fejlfinding

I det følgende beskriver jeg forløbet af test- og verifikationsfaserne mellem - og i nogen tilfælde under - faserne i implementationen.

Fase 1 Som nævnt i foregående afsnit benyttede jeg mig i fase 1 af meget simple testprogrammer, som sikrede at skrivning og læsning til det delte lager, der teknisk set på dette tidspunkt endnu ikke var delt, gav korrekte resultater, og at directory-strukturen blev oprettet og modificeret korrekt. I denne del af udviklingen var der ikke implementeret kommunikation med andre knuder, og kald til låsefunktionen udførte blot en opskrivning af lamport-tiden. Den endelige udgave af det simple testprogram findes i afsnit B.1.

Det andet testprogram jeg udviklede i denne fase, havde til formål at give en indikation af, om den valgte løsning til sporing af tilgang til det delte lager var tilstrækkeligt hurtig til at kunne leve op til de ønskede mål for ydelse. Uformelle målinger med dette program indikerede, at sekventiel tilgang til det delte lager gennem de definerede datatyper var ca. 3 gange langsommere end almindelig tilgang til lager gennem grundtyper. Dette var et yderst opmuntrende resultat, da jeg på grundlag af mekanismens kompleksitet i bedste fald havde håbet på en ydelse der var indenfor samme størrelsesorden som normal, lokal lagertilgang. Resultatet indikerede at sekventiel tilgang til det delte lager i den endelige udgave af SENSE ville kunne udføres med næsten samme hastighed som normal lagertilgang.

At hastigheden for lagertilgang var så meget hurtigere end forventet, skyldes først og fremmest brugen af en minimal cache (se afsnit 4.2.1), der sikrede at sekventiel tilgang til det delte lager kan udføres uden opslag i directory. Denne mekanisme bliver beskrevet udførligt under programbeskrivelsen.

Fase 1 forløb ganske gnidningsfrit, og bød ikke på nogen uheldige overraskelser. Alle tests indikerede at den grundlæggende infrastruktur - directory og delt lager - fungerede tilfredsstillende, og var i stand til at levere en acceptabel ydelse.

Fase 2 I forbindelse med fase 2 blev der kun udviklet et enkelt testprogram, som ikke var relateret til SENSE som sådan. Da der blot var tale om et simpelt program på 20 linjer der oprettede og nedlagde en POSIX tråd, har jeg valgt ikke at tilføje det blandt testprogrammerne. Det egentlige testprogram i denne fase var udviklingen af programmet *lockman* (kildetekst i afsnit C.1), der i samarbejde med selve SENSE varetager opsætning og nedlukning af de nødvendige kommunikationskanaler. I denne fase blev tilføjet funktionerne til initialisering af SENSE, samt nedlukning af knuderne (sense_init og sense_finish).

I lighed med fase 1, forløb også fase 2 uden nogen større problemer eller overraskelser. Største udfordring var at beslutte sig for en passende organisation af infrastrukturen, primært at fordele ansvaret i client/server forholdene mellem de enkelte knuder.

Fase 3 Det primære testprogram i første del af fase tre, var også et ganske simpelt program, der blot undersøgte at data blev rekvireret korrekt under antagelse af happens-before relationen. Den endelige udgave af dette program er bragt i afsnit B.2. Da først dette program var bragt til at eksekvere korrekt, hvilket skete uden jeg konstaterede andet end trivielle fejl i barrieremekanismen, valgte jeg at konvertere benchmarkprogrammet 3DFFT til SENSE. Dette program er af overskuelig størrelse, og benytter kun barrierer til synkronisering. Kildekoden var blevet stillet til rådighed af Peter Keleher, og var en variant af testprogrammet der blev benyttet i [16]. Eneste forskel i forhold til den oprindelige udgave var, at det var blevet tilpasset CVM [9] - et sidebaseret SMLB-system, der er udviklet med henblik på at undersøge effekten af konsistensmodeller og en mere finkornet procesmodel end SENSE og TreadMarks.

Her stødte jeg på de første alvorlige udfordringer. Programmet blev leveret uden dokumentation, og indeholder kun et minimum af kommentarer. Hvad værre er: det er skrevet i såkaldt "old-style C", dvs. den oprindelige præ-ANSI dialekt af C. Som mange andre tidlige C programmer forlader det sig i høj grad på, at et array i C er typemæssigt og funktionelt ekvivalent til en pointer til grundtypen. Da både CVM og TreadMarks bruger almindelige C datatyper til delt lager - disse allokeres blot i en bestemt region af lageret, som systemet varetager - er der i disse systemer heller ikke forskel på en array af (eller en pointer til) en grundtype som f.eks. *int* i delt og lokalt lager. Dette er ikke tilfældet i SENSE, hvor delte datatyper nok er funktionelt ekvivalente til lokale, men typemæssigt forskellige!

Da en lang række mellemregninger i 3DFFT udføres både i lokalt og delt lager, blev jeg altså nød til at duplikere en lang række funktioner i programmet, der i praksis kun adskilte sig ved, hvilken type parametre de krævede. Dette må dog snarere betragtes som et irritationsmoment, end en decideret udfordring, omend det ikke faldt i tråd med målsætningen om kompatibillitet med eksisterende SMLB-programmer. Disse problemer kan i praksis løses i langt de fleste tilfælde ved at definere passende konverteringsmetoder til de delte datatyper, men skyldes altså til en vis grad at forældede programmeringsteknikker blev anvendt i programmet.

Et andet, og mere alvorligt, problem var, at lagerforbruget viste sig at være uventet højt. Dette skyldtes at jeg havde valgt at erklære arrays som *shared-type a[størrelse]*, snarere end at oprette en decideret array-type rettet mod delt lager. Denne type erklæring medfører i SENSE at der allokeres lager til registrering af hvert enkelt element i arrayet (størrelse, samt placering i delt lager), snarere end blot en enkelt informationsblok for hele arrayet.

Der var to grunde til, at jeg valgte denne form for erklæringer. Først og fremmest eliminerede det behovet for at modificere programmet til at håndtere arrays af delte datatyper, da disse så kunne tilgås med pointere på vanlig C facon. For det andet er det muligt at definere metoder og operatorer i C++, der ville gøre det unødvendigt at modificere programmet yderligere for at det skulle kunne håndtere en speciel array-type til delt lager, men dette ville indføre en yderligere fejlkilde i SENSE. Jeg valgte på den baggrund at affinde mig med situationen, og notere mig at en fremtidig udvidelse af de delte datatyper ville eliminere dette problem.

Det sidste, og mest alvorlige, problem var, at jeg var ude af stand til at vurdere, om programmet leverede et korrekt resultat. Det blev som nævnt leveret uden dokumentation af brug og forventede uddata, og jeg kunne altså blot konstatere, at det terminerede korrekt. Ved nærmere overvejelse besluttede jeg, at det i og for sig var *uvæsentligt*, om jeg kunne afgøre om uddata var korrekte eller ej!

Et nærmere studie af kildekoden viste, at programmet vil udføre det samme arbejde, uanset om data beregnes korrekt eller ej - eksekveringen er ikke afhængig af hverken inddata eller mellemregninger, men består blot af en række transformationer og beregninger. Da alle mine testprogrammer hidtil havde leveret korrekte resultater, og da en måling af data der blev udvekslet ved barrierer var konsistent med resultater fra de kørsler af programmet der var dokumenteret i [16], vurderede jeg at programmet under alle omstændigheder kunne anvendes til det planlagte formål: en evaluering af SENSE's effektivitet.

Fase 4 Det indledende arbejde med implementation af låse forløb problemfrit, da store dele af programkoden og logikke kunne tages direkte fra barriere-implemenattionen. Testprogrammer afdækkede dog et strukturelt problem, da jeg ikke havde taget højde for at det var nødvendigt at sikre sig en knude var ankommet til barrierer i brugerprogrammet, før dens udestående opdateringer i opdateringskøen blev fjernet, og klargjort til transmission. Dette medførte at happens-before relationen kunne brydes. En mindre modificering af programkoden for barrierer (ca. 5 linjer) løste problemet.

I denne fase havde jeg til gengæld store problemer med konvertering af benchmarkprogrammerne, som ikke skyldtes de nævnte problemer med "old-style C". Som kandidater til konvertering havde jeg valgt quicksort (QS) og travelling sales person (TSP), med QS som primær kandidat - dette på grundlag af programmernes størrelse. Uheldigvis viste det sig, at QS både benyttede C-makroer på uigennemskuelig vis (for mig), systemkald hvis semantik jeg var ubekendt med (*sigpause*) og en DSM-facilitet jeg havde undladt at implementere, da jeg betragtede den som redundant (Tmk_distribute/CVM_distribute). Dette medførte at jeg koncentrerede arbejdet om TSP.

Undervejs i konverteringen af TSP, implementerede jeg de mest almindelige operatorer og konverteringsmekanismer for min shared-int datatype, hvilket i høj grad lettede arbejdet. Ydermere definerede jeg en række kompatibillitets-funktioner, som oversatte TreadMarks-funktioner til deres tilsvarende SENSE funktion, så jeg ikke var nødt til at udføre en "søg og erstat" på kildekoden.

Indledende arbejde med låse. Relativt problemfrit med testprogrammer - struktur kunne tages fra barrierer i det store og hele. Afdækkede et enkelt problem, der skyldtes at jeg var for ivrig i at indlede barrierer, hvilket brød emd happens-before relationen.

Fase 5 Ganske som det var tilfældet med invalidate-protokollen for låse, kunne jeg i det store og hele genbruge kildekoden fra barriererne til at implementere update og hybrid-protokoller ved låse.

Kørsler med TSP afdækkede et par trivielle fejl i implementationen af update- og hybrid protokollerne, som ikke var strukturelle, men simple overseelser fra min side. Implementation af update/invalidate problemfri og simpel.

Opsummering Den mest krævende del af verificeringen, viste sig at være konverteringen af benchmarkprogrammerne. Dette krævede en del indsigt i deres funktion, hvilket jeg ikke havde taget højde for. Det ville have lettet arbejdet en hel del, hvis jeg havde implementeret en analog til *Tmk_distribute* funktionen, men da jeg som nævnt i afsnit 4.2.5 betragtede den som redundant, havde jeg ikke under designet af SENSE taget hensyn til denne funktionalitet. I et velskrevet og veldokumenteret program, ville det være en smal sag at foretage de nødvendige ændringer, men disse programmer var ingen af delene!

Selve verificeringen af SENSE som sådan gik genske problemfrit. Der blev kun afdækket en enkelt strukturel fejl, som kunne løses med et minimum af indsats. Resten af de fejl der blev fundet kan kategoriseres som overseelsesfejl, og hører retteligt til i småtingsafdelingen. Fejl af denne type - eksempelvis manglende initialisering af pointere eller "off by one" - forekommer i alle programmer af en vis størrelse.

Tidsforbrug: ca. 14 dage.

4.3.3 Tuningsfasen

Da fintuning af ydelse ofte medfører at kildekoden gøres mindre generel, havde jeg udsat denne del af arbejde til jeg var overbevist om, at SENSE fungerede korrekt. Tuningen blev udført ved brug af profileringsværktøjet *gprof*, som giver et overblik over hvilke dele af programmet der dominerer eksekveringstiden. Da dette kun blive opgjort for CPU-forbrug, er det ingen erstatning for tidsmålinger i et distribueret system, så jeg valgte at indføre en statistikfunktion, som kan slås til ved definering af en makro i en headerfil.

Brug af profilering identificerede enkelte funktioner som gode kandidater til inlining. Udførslen af denne inlining alene medførte en reduktion på køretiden for 3DFFT på ca. 20%! Effekten var så dramatisk for netop dette program, da eksekveringen er kraftigt domineret af tilgangen til det delte lager.

Profilering af 3DFFT identificerede desuden visse af STL's map-funktioner som ydelsesmæssigt kritiske regioner. Det er ikke ensbetydende med at de kan forbedres, men indikerer blot at de bliver brugt meget under eksekveringen af dette program. Da jeg ikke har gennemgået kildekoden til dette komponent, er jeg ikke i stand til at afgøre om en implementation der er specialiceret til objektets funktion i SENSE ville medføre en forbedring af ydelsen. Det er dog normalt tilfældet at man kan forbedre ydelsen af programkomponenter drastisk, hvis man optimerer til det hyppigste tilfælde, som altså her er identificeret gennem profilering.

Tidsmålinger på kørsler af FFT afslørede et andet problem: barrierer blev udført uventet langsomt. En nærmere undersøgelse viste, at hastigheden for datatransmission lå omkring en størrelsesorden lavere end forventet i denne del af programmet. Dette viste sig at skyldes Ethernets karakteristik. Da Ethernet er et delt medie, kan kun én adaptor tilsluttet nettet transmittere af gangen. Jeg havde antaget at TCP og Ethernets indbyggede faciliteter til håndtering af dette ville være tilstrækkeligt, men i praksis var det ikke tilfældet. Jeg valgte derfor at koordinere kommunikationen ved barrierer således at to kunder ville afgøre hvilken der skulle transmittere først på grundlag af deres knudenummer. Jeg udførte kun denne optimering for update-protokollen, da det allerede på dette tidspunkt stod klart, at FFT ikke ville levere hverken tilfredsstillende ydelse eller skalérbarhed med invalidate eller hybrid-protokol på målsystemet. Optimeringen må forventes at være unødvendig og decideret ydelseshæmmende ved brug af mere moderne netværk som f.eks. ATM, der både har indbygget mere effektive transmissionskontrolmekanismer og er fuld duplex.

Opsummering Et profileringsprogram må betragtes som uundværligt til en opgave af denne type. Effekten af de optimeringer der blev indført på grundlag af brugen af dette var meget markant, omend der kun var tale om konstante faktorer. Et profileringsværktøj kan i mange tilfælde også hjælpe med at afdække muligt algoritmiske optimeringer (store O forbedringer), ved at identificere ydelsesmæssigt kritiske regioner i programmet.

Tidsmålinger påpegede behovet for strukturel ændring i dele af SENSE, som dog må betragtes som værende yderste specifikke for den valgte netværkstype (Ethernet).

Dele af standardbiblioteket STL's implementation af binære træer er ydelseskritiske ved eksekvering af 3DFFT, hvilket indikerer at ydelsen kunne forbedres ved at benytte en specialiceret implementation i stedet.

Tidsforbruget var ca. 3 arbejdsdage

4.3.4 Opsummering og konklusion

Alt i alt må implementationen siges at være udført efter planen, og relativt smertefrit. Implementationen blev gennemført på hvad der svarer til 5 normale arbejdsuger, og jeg opfyldte således min målsætning om hurtig udvikling til fulde.

De største problemer skyldes ikke SENSE som sådan, men de valgte testprogrammer (3DFFT, QS og FFT), og min manglende erfaring med disse. Havde jeg på forhånd haft mere erfaring med dem, kunne tidsforbruget på implementationen sagtens have været en uge mindre.

På den anden side havde jeg en stor fordel i denne del af projektet, da jeg på forhånd var bekendt med flertrådet programmering , netværkskommunikation, C++, UNIX samt netværksprotokollers opbygning og karakteristika. En programmør uden denne form for erfaring, ville højest sandsynligt ikke have været i stand til at gennemføre implementationen med samme lave tidsforbrug.

Den endelige implementation af SENSE endte med at følge det oprindelige design til punkt og prikke, med undtagelse af to mindre områder. Den ene designændring skyldtes som nævnt Ethernets specifikke karakteristika, og den anden må siges at være triviel. Grundig planlægning viste sig således her, som forventet, at være et godt grundlag for hurtig udvikling af stabile systemer.

Grundig og velplanlagt verifikation viste sig også at være af stor vigtighed. Uden denne, ville det ikke have været muligt at udføre en fase-baseret udvikling, da jeg jo ellers ikke ville have vidst hvor jeg skulle lede, hvis og når der blev afdækket en fejl. Brugen af verifikation mellem (og under) faserne gjorde, at jeg til en hver tid med rimelig sikkerhed kunne udpege den del af kildekoden, som var årsag til en fejl eller uregelmæssighed.

Brugen af gennemprøvede standardværktøjer og standardbiblioteker gjorde udviklingen klart lettere og hurtigere. Det er nærmest uvurderligt at kunne have tillid til, at basale funktioner er implementeret effektivt og fejlfrit, hvilket er tilfældet med STL. Der er ingen grund til at genopfinde den dybe tallerken eller det varme vand. Tiden kan i stedet bruges fornuftigt og produktivt andetsteds, og der jo intet der forhindrer en i senere at implementere mere målrettede funktioner, hvis behovet skulle vise sig.

Det gav også klare fordele at implementere i C++. Sproget gør det nemt at modularisere programmet, og brugen af objekter medfører at fejl typisk kun skal rettes et enkelt sted i kildekoden. Jeg benyttede mig af ydelsesmæssige grunde ikke af mulighederne for nedarvning, og dermed genbrug af kode, men vurderer i retrospekt at det ville have forkortet udviklingstiden uden at påvirke ydelsen mærkbart.

Summa summarum: De klassiske systemarbejdsdyder viste deres værd! Havde jeg blot hovedløst kastet mig ud i programmeringen, var der sikkert aldrig kommet et fungerende DSM-system ud af arbejdet. I stedet blev resultatet en velstruktureret system, som ikke havde nogen kendte fejl eller uregelmæssigheder da jeg gik ind i tuningfasen. Der blev under tuningen ikke afdækket nye fejl. Dette alene må siges at være et bemærkelsesværdigt resultat, og er - baseret på mine personlige erfaringer - langt fra sædvanligt.

4.4 Programbeskrivelse

I det følgende beskrives kort den overordnede opbygning af SENSE. Enkelte centrale dele af programmet fremhæves og beskrives mere detaljeret. Programbeskrivelsen er skrevet så den principielt kan læses løsrevet fra resten af opgaven, men forudsætter kendskab til generelle koncepter i SMLB-programmering og datamatarkitektur.

4.4.1 Opbygning

Formålet med et DSM system er at stille en programmeringsgrænseflade - et API - til rådighed for programmøren, der minder om almindelig multiprogrammering med *fork()* eller letvægtstråde. Bag kulisserne sikrer systemet at synkronisering mellem brugerprogrammets processer udføres korrekt, og at der opretholdes et tilstrækkeligt opdateret billede af det delte lager til at programmer der ikke indeholder væddeløbstilstande eksekverer korrekt. Dette gøres ved at bogføre *Lamport-tiden* for alle skrivninger til lageret, samt evt. at registrere hvilken af brugerprogrammets processer der har udført skrivningen, i tilfælde af at man ønsker at benytte en hel eller delvis *invalidate protokol*.

På den baggrund er den mest frugtbare måde at betragte SENSE's opbygning at anskue systemet som en datastruktur - det såkaldte directory - der håndterer lagring og fremskaffelse af konsistensinformation, dvs. information om hvilke dele af lageret der er ændret, hvor og hvornår, og en kommunikationsinfrastruktur der giver mulighed for synkronisering af knuders opstart og nedukning, rekvirering af lagerændringer samt brug af synkroniseringsprimitiver for låse og barrierer. Et seperat program - *lockmanager* - varetager rollen som velkendt mellemmand ved rekvirering af låse. Dette program er til enhver tid i besiddelse af korrekt viden om, hvilke låse er ejet af hvilke knuder.

Directoryet tilgås af brugerprogrammet gennem allokering med funktionen *sense_malloc*, læsning og skrivning til dele af det delte lager med funktionerne *sense_read* og *sense_write*, samt ved brug af synkroniseringsprimitiverne *sense_acq*, *sense_rel* og *sense_barrier*. Alle skrivninger til lageret registres i det lokale directory, samt i en kø af udestående lagerændringer for hver af de andre knuder involveret i eksekveringen af brugerprogrammet.

En stor del af dette - allokering, læsning og skrivning - er skjult for programmøren, idet denne del af kommunikationen med directory er indkapslet i en række objekter, således at delte datatyper kan erklæres på lige med almindelige variable. Pt. er implementeret en delt heltalstype (*Tsense_sint*) og dobbeltpræcission flydende kommatal (*Tsense_sdouble*).

Kommunikationsinfrastrukturen består af tovejs kommunikationskanaler til følgende opgaver:

1. Rekvirering af en lås hos lockmanager.
2. Modtagelse af rekvireringsønske for låse fra lockmanager.
3. Rekvirering af indholdet af et lagerområde hos en anden knude.
4. Modtagelse og håndtering af ovenstående.
5. Signallering af ankomst ved barrierer samt udveksling af lagerændringer i den forbindelse.
6. Overgivelse af ejerskab af en lås til en anden knude, samt udveksling af directory- og lagerændringer.

Kanalerne under punkt 2, 4 og 6 overvåges ved brug af Posix-letvægtstråde. Alle kanaler er implementeret ved brug af et objekt, der benytter berkley-sockets og TCP som transportmekanisme.

Ved synkronisering med låse udveksles køer af udestående directoryændringer mellem de to involverede knuder. Et *copyset* benyttes til at afgøre om de modtagede ændringer skal sættes i kø til eventuelle andre knuder. Copysettet består af en boolsk variabel for hver af knuderne i systemet, og sættes til sand for knuden skrivningen er udført på, samt modtagerknuden. Således er copysettet en dæmpermekanisme, der begrænser mængden af information der rundsendes i systemet. Uden den ville det være umuligt at afgøre om en modtaget directoryændring skulle rundsendes eller ej.

Ved synkronisering med barrierer udveksles ændringer uden copyset, da alle knuders directory opdateres ved denne operation. Det er altså ikke nødvendigt at rundsende denne information.

4.4.2 Protokoller

SENSE giver mulighed for at benytte tre varianter af Lazy Release Consistency protokollen (LRC) ved låse. Disse er:

- Ren invalidate. Ved denne protokol rundsendes kun directoryændringer, men ikke indholdet af de lagerområder der er opdateret.
- Ren update. Her rundsendes både directory- og lagerændringer.
- Hybridiseret update/invalidate. Her rundsendes både directory- og lagerændringer, hvis directory indikerer at et lagerområde har en tendens til at blive rekvireret.

Den hybridiserede protokol er ikke til rådighed ved barrierer. Her kan kun vælges mellem update og invalidate.

Hybridprotokollen forsøger at afdække lagerblokkes mobilitet ved at registrere hvor mange gange indholdet af et lagrområde er blevet rekvireret fra en anden knude. Dersom dette antal rekvireringer overstiger en grænseværdi, vil indholdet af denne lagerblok blive rundsendt sammen med directory-ændringer ved synkronisering med låse. Der er pt. ikke implementeret en dæmpermekanisme, så brug af hybridprotokol med en for lav grænseværdi medfører risiko for rundsendelse af irelevant information.

Valg af protokol for låse og barrierer, fastsættelse af grænseværdi for hybrisprotokollen samt en lang række andre parametre for SENSE, håndteres ved erklæring af makroer i filen *Defaults.h*. Kommentarer i filen indikerer brug af makroerne.

4.4.3 Bogføring af lagertilgang

I directory opretholdes et billede af det delte lager som en mængde af disjunkte, fortløbende lagerblokke. Directory bogfører følgende information for hver blok:

- Startadresse.
- Slutadresse.
- Lamporttid for sidste ændring.
- Hvilken knude der er i besiddelse af opdateret indhold af lagerområdet.
- Hvor mange gange indholdet af denne lagerblok er blevet rekvireret. Dette registreres kun ved brug af invalidateprotokol.

Disse informationsblokke registreres i et ballanceret binært træ, der er implementeret ved brug af STL's *map* type. Slutadressen bruges som indexværdi, da en informationsblok så kan fremskaffes ved at benytte funktionen *lower_bound* til søgning. Skrivninger og læsning til directory udføres ikke direkte, men gennem en mikrocache bestående af en informationsblok med ovenstående information for skrivning og læsning. Disse initialiseres til værdier der vides at være udenfor det delte lager ved opstart af SENSE, samt efter synkronisering med låse eller barrierer. Første læsning eller skrivning til det delte lager udløser således en cachefejl, der medfører at den relevante cacheblok initialiseres med den relevante information for lagerområdet der indeholder den ønskede adresse.

Skrivecachen bliver skrevet til directory hvis en skrivning udenfor informationsblokken udføres. Ved samme lejlighed tilføjes informationen også til opdateringskøerne for de andre knuder. Endvidere bliver skrivecachen skrevet til directory før udførslen af synkronisering med låse eller barrierer.

Ved læsning fra en given adresse, undersøges først om adressen ligger indenfor de områder der dækkes af læse- eller skrivecachen. Er dette ikke tilfældet, undersøges om der er tale om en adresse der ligger i umiddelbar forlængelse af den nuværende læsecache. Hvis det er tilfældet, kan cachen opdateres på konstant tid ved brug af *next* metoden fra STL map. Ellers må informationen fremskaffes i log2(n) tid ved søgning i directory med *lower_bound* funktionen. Tilbage er kun at undersøge om indholdet af lagerblokken forefindes lokalt, og evt. rekvirere indholdet fra ejerknuden. Bruges hybridprotokol opskrives rekvireringstallet for informationsblokken i directory ved denne lejlighed.

En metode tilsvarende integreringsfunktionen for skrivecachen benyttes til integrering af directoryændringer ved synkronisering, men integreringen foretages her under hensyntagen til informationsblokkenes Lamporttid. Dette er ikke nødvendigt ved integrering af skrivecachen, da alle lokale skrivninger vides at have en højere Lamporttid end informationen i directory.

Ved brug af updateprotokol bogføres læsninger ikke til directory, da lageret vides at være fuldt opdateret.

Funktioner vedrørende tilgang til directory og det delte lager er implementeret i filerne *sensememory.h*, *sensememory.cpp*, *directory.h* og *directory.cpp*. Håndtering af synkronisering med låse og barrierer er implementeret i filerne *Lock.h*, *Lock.cpp*, *sense_barrier.h* og *sense_barrier.cpp*.

4.4.4 Netværkskommunikation

Netværkskommunikation er implementeret ved brug af berkley sockets og TCP. Opsæningen af endepunkter og transmission og modtagelse af data er indkapslet i to objekter *Tsense_servsock* og *Tsense_clisock*. Det vil således være ganske simpelt at ændre SENSE til at benytte en anden netværksprotokol, da ændringerne vil være lokaliseret her.

Objekterne er implementeret i filerne *sense_connections.h* og *sense_connections.cpp*.

4.4.5 Datatyper i delt lager

Som en del af API'et er implementeret to basale delte datatyper: heltal (*Tsense_sint*) og dobbeltpræcissions flydende kommatal (*Tsense_sdouble*). Allokering af delt lager håndteres automatisk ved erklæring gennem konstruktører defineret for objekttyperne. Endvidere er operatorer for konvertering til den tilsvarende grundtype, præfixvarianter af op- og nedtælling (++ og –) samt operatoren "=" defineret. Som et eksempel er også implementeret en vektor af dobbeltpræcissions flydende kommatal.

Typerne er implementeret i filen *sensememory.h*.

4.5 Konfigurering

Det er muligt, og for nogen brugerprogrammer nødvendigt, at konfigurere visse dele af SENSE. Dette gøres ved at ændre i filen *Defaults.h*. De vigtigste parametre er som følger:

Hukommelsesstørrelse justeres ved at ændre størrelsen af den globale konstant *sense_memsize*.

Længen af opdateringskøer justeres ved at ændre størrelsen af den globale konstant *sense_updqlen*.

Antal låse til rådighed afgøres af den globale konstant *sense_maxlocks*.

Protokol ved låse vælges ved at definere en af makroerne *SENSE_pure_update*, *SENSE_pure_invalidate* eller *SENSE_hybrid*.

Update protokol ved barrierer vælges ved at definere makroen *SENSE_barriers_update*. Hvis makroen ikke er defineret, benyttes invalidate protokol. Makroen defineres automatisk, hvis der er valgt ren updateprotokol ved låse.

Statistikfunktion slås til ved at definere makroen *SENSE_stats*.

Debugfunktion slås til ved at definere makroen *SENSE_debug*. Dette giver bla. et klart overblik over alle tilgange til det delte lager, men da mængden af oplysninger der udskrivet er genske voldsom, bør denne funktion bruges med varsomhed.

Netadresserne for de involverede maskiner defineres i filen *nodeaddr.cpp*.

4.6 Programmering med SENSE

Programmering med SENSE er ganske simpelt. Der benyttes C++ som værtssprog, og programmerne opbygges ganske som alle andre programmer i SMLB paradigmet, dvs. erklæring og initialisering af delte datastrukturer udføres under opstartsfasen. Alle delte variable skal erklæres i samme rækkefølge på alle knuder, da det ellers ikke garanteres at de allokeres på samme position i det delte lager. Følgende faciliteter er til rådighed:

Allokering udføres gennem brug af de delte datatyper. Delte variable erklæres ganske som lokale.

Tilgang til delt lager udføres med samme semantik som tilgang til lokalt lager. Variable erklæret med de delte lagertyper kan bruges ganske som normale lokale variable.

Synkronisering udføres med funktionerne *sense_acq()* og *sense_rel()* for låse, samt med funktionen *sense_barrier()* for barrierer.

Knude ID kan læses af den globale konstant *sense_thisnode*.

Antal knuder kan læses af den globale konstant *sense_nodes*.

Initialisering af SENSE udføres med funktionen *sense_init()*.

Nedlukning af SENSE udføres med funktionen *sense_finish()*.

Følgende korte eksempelprogram illustrerer brugen af de basale funktioner.

```
// Kort eksempelprogram i SENSE

// headerfilen sense_unit.h indeholder de nødvendige
// definitioner
#include ``sense_unit.h''

// Erklæring af et 2D array af heltal i delt lager med
// 1 kolonne pr. knude
const int veclen=100;
const int arrlock=1;
Tsense_sint arr [sense_nodes][veclen];

main() {
  sense_init();
  sense_acq(arrlock);
  for (int n=0 ; n<veclen ; n++)
    arr[sense_thisnode][n]=sense_thisnode;
  sense_rel(arrlock);
  sense_barrier();
  sense_finish();
}
```

Af kompatibillitetshensyn er defineret funktioner til oversættelse af de mest almindelige CVM og TreadMarks-funktioner, dvs. opstart, nedlukning og synkronisering.

4.7 Oversættelse og eksekvering

For at kunne eksekvere et brugerprogram under SENSE; skal der oversættes en eksekverbar fil til hver knude, samt en version af *lockman*. Makefilen er konstrueret med henblik på at lette denne opgave. Under antagelse af, at eksempelprogrammet fra forrige afsnit er gemt under navnet *ex.cpp*, skal følgende tilføjes makefilen:

```
ex: ex.o $(SENSEOBJS)
$(CHOST) $(CFLAGS) $(CLIBS) -o ex ex.o $(SENSEOBJS)

ex.o: ex.cpp
$(CHOST) $(CFLAGS) -c ex.cpp
```

Herefter kan brugerprogrammet oversættes for hver knude ved brug af følgende kommando:

```
make ex NODE=0 NODES=2
```

Som det fremgår vil dette producere et program, der kan eksekveres på den maskine der er registreret som knude 0 i filen *nodeaddr.cpp*.

For at oversætte *lockman*, benyttes følgende kommando:

```
make lockman NODE=0 NODES=2
```

Her er det underordnet hvilken værdi der angives for NODE.

For at starte eksekveringen startes først lockman på den maskine, der er angivet som lockmanager i filen *nodeaddr.cpp*. Herefter startes brugerprogrammerne på deres respektive maskiner. Når alle brugerprogrammer er startet, vil lockman tillade at eksekveringen udføres, samt synkronisere nedlukningen af knuderne.

4.8 Kompatibillitet med andre SMLB systemer

SENSE har en semantik der er subtilt anderledes end TreadMarks, idet der ikke udarbejdes en *diff* ved synkronisering. Dette medfører at man i visse tilfælde under TreadMarks kan tillade sig at undlade at beskytte tilgang til delte variable med låse eller barrierer. I SENSE vil et program skrevet under denne antagelse *ikke* være DRF! Disse effekter er ganske subtile, men optræder så vidt jeg ved ikke hyppigt i eksisterende programmer.

4.9 Forslag til vidreudvikling

SENSE lider i mage tilfælde under et stort hukommelsesforbrug, da der endnu ikke er defineret en specialiceret delt arraytype, der er kompatibel med normale C og C++ arrays. Dette burde være muligt at gøre for en erfaren C++ programmør. Af andre hukommelsesforbrugende elementer kan nævnes at der erklæres en opdateringskø for alle knuder, inklusiv den knude programmet eksekveres på. Dette er unødvendigt, og vil være trivielt at rette. Endelig registreres copysets med en vektor af booleans, hvilket de fleste C++ compilere vil allokere i form af maskinord. Det burde være muligt at registrere dette i form af enkelte bits i en variabel, ved brug at bitvis *and* og *or*. Da dette også vil nedbringe mængden af data der transmitteres ved synkronisering drastisk, bør det i høj grad prioriteres ved fremtidig udvikling.

Rent API mæssigt så jeg gerne at det var muligt at erklære flere regioner af delt lager, så en maskine kunne deltage i en *eller flere* beregningsklynger. På denne måde kunne man udarbejde en mere avanceret fordeling af beregningen på knuder, ved at indføre et hieraki, eller for den sags skyld simulere en hyperkube.

Opsamlingen af lageropdateringer ved barrierer udføres i sin nuværende form ved kopiering. Dette kunne formegentlig med fordel i stedet udføres med *scatter-gather write* til netforbindelsen med UNIX-systemkaldet *writev*.

Det burde være muligt at udvikle en mekanisme, der gjorde allokering af globale delte data synligt for compileren, således at der kunne gennemføres yderligere optimeringer ved oversættelse. På denne måde kunne der opnås en ydelsesmæssig forbedring af læsning til delt lager med en konstant faktor, da variables placering i lageret ville være kendt for oversætteren.

For yderligere kompatibillitet med TreadMarks og CVM ville det være en fordel at implementere en analog til funktionen *Tmk_distribute*.

Oversættelse og opstart af brugerprogrammer kan i pricippet gøres meget mere brugervenligt.

4.10 Opsummering og konklusion

Ved afslutningen af implementationsfasen mener jeg at have opfyldt målsætningerne om hurtig udvikling, brugervenlighed, kompatibillitet, portabillitet og brugervenlighed. Systemet blev udviklet på ca. 5 uger, programmeringsmodellen må siges at være næsten tilsvarende den der vises i pseudokodeeksemplerne i kapitel A og de problemer der opstod med flytning af testprogrammer fra TreadMarks skyldtes primært at de var dårligt dokumenteret og min manglende kendskab til dem. Med hensyn til portabillitet benytter SENSE meget få faciliteter der må betragtes som platform-specifikke, og størstedelen af dem er indkapslet i objekter. Tilbage er kun at afgøre om SENSE lever op til mine målsætninger om ydelse og skalérbarhed, hvilket vil blive undersøgt i de følgende kapitler.

5 SENSE — måling af ydelse

I dette kapitel beskrives systemet der blev benyttet til ydelsesmåling og de benyttede benchmarkprogrammer. Desuden bringes en oversigt over måleresultaterne.

5.1 Forsøgsopstilling

Som forsøgssystem blev brugt DIKU's Linux-cluster. Denne består af 8 arbejdsstationer baseret på Intel Pentium II processorer med 450MHz clockfrekvens. Fire af disse er enkelt-CPU maskiner med 256MB RAM, og fire er dual-CPU maskiner med 512MB RAM. Den benyttede version af Linux giver ikke adgang til brug af den anden processor i dual-CPU systemerne, og disse leveres således den samme ydelse som single-CPU maskinerne. Maskinerne er forbundet med et 100Mbis/sek Ethernet gennem en switch.

Til sammenligning blev i [16] benyttet 133MHz HP PA-RISC baserede arbejdsstationer med 96MB RAM, der var forbundet med et 100Mbit/sek FDDI netværk.

Det er værd at bemærke sig, at CPU/båndbredde forholdet på disse to testsystemer er meget forskelligt. Det kan forventes at DIKU clusterens CPU/båndbredde forhold er mindst 4 gange højere, da Ethernet ikke udnytter båndbredden nær så effektivt som FDDI.

Af andre forskelle kan nævnes, at der ikke benyttes samme operativsystem og oversætter.

En direkte sammenligning mellem ydelse på de to testsystemer er således ikke mulig at udføre med et meningsfuldt resultat til følge.

5.2 Benchmarkprogrammerne

Til brug under ydelsesmålingen havde jeg konverteret to af de programmer der blev benyttet til ydelsesmåling på TreadMarks i [16] til SENSE systemet. Disse programmer, 3DFFT og TSP, er faktiske eksempler på brugerprogrammer, med alt hvad dette indebærer. En konsekvens af dette er, at de ikke kan forventes at opnå optimal ydelse eller speedup, da de er udviklet til at løse et praktisk problem, snarere end for at stille et DSM system i et positivt lys.

5.2.1 3DFFT

Dette program udfører fast fourier transformation på et 3D array af tilfældigt genererede tal. Programmet tilhører den kategori, der i afsnit 2.6 karakteriseres som stratificerede. Arbejdet der udføres på knuderne er meget symmetrisk, og det kunne i princippet udføres på en SIMD datamat - en arkitektur hvor alle knuder udfører samme eksekveringsstrøm på hver deres del af inddatasættet. Programmets tilgangsmønstre er yderst fjendtlige mod et moderne hukommelseshieraki, i og med at data ikke læses eller skrives sekventielt. Endvidere har 3DFFT en relativt lav *computation/communication* ratio. Ideelle omstændigheder for 3DFFT er altså et system med en hurtig kommunikation og et meget hurtigt hukommelseshieraki. På denne baggrund kan man ikke forvente at opnå en høj ydelse med dette program under SENSE, da systemets hukommelseshieraki er meget asymetrisk. Læsninger og skrivninger der ikke er sekventielle udføres ikke i konstant tid, men har en tidskompleksitet log2(n). Endvidere vil ikke-sekventielle skrivninger opdele det delte lager i mange små lagerområder, hvilket både medfører en forøget søgetid i directory og et stort lagerforbrug til registrering af konsistensinformation.

5.2.2 TSP

TSP implementerer travelling salesperson algoritmen ved brug af en arbejdskø. De involverede knuder optræder både som producenter og konsumenter af delvist færdiggjorte ture i arbejdskøen. I denne version af programmet er læsning af den delte variabel der indeholder længden af den hidtil korteste tur ikke beskyttet af en lås, da denne værdi vides at være konstant faldende. TSP har en relativt høj computation/communication ratio, og udveksler relativt få data mellem knuderne. Arbejdssættet ved brug af inddatafil med 19 byer er ca. to størrelsesordener mindre end det 3DFFT benytter ved transformation af en 64x64x64 datastruktur. Et problem med TSP er, at eksekveringsprofilen både er stærkt afhængig af inddata og den rækkefølge knuder aftager og indsætter ture i arbejdskøen i. Det er altså svært at opnå konsistente resultater ved ydelsesmåling med dette program. Til gengæld er det nemt at opnå god skalérbarhed, så længe programmet bliver givet et tilstrækkeligt stort problem at beregne. Ved brug af for små problemer vil man opleve at størstedelen af knuderne tilbringer en stor del af tiden med at vente på at der bliver ture til rådighed i arbejdskøen.

5.3 Målsætning

Den oprindelige målsætning med ydelsesmålingen var at undersøge om systemet levede op til designmålene med hensyn til ydelse og skalérbarhed. Disse mål var at opnå et eksekveringstid der var indenfor samme størrelsesorden som den der kan opnås med de samme programmer eksekveret under TreadMarks eller et tilsvarende sidebaseret SMLB system. Ideelt skulle dette undersøges gennem en direkte sammenligning på samme testsystem, da selv små variationer i datamat- eller netværksarkitektur kan udmønte sig i store forskelle i eksekveringstid eller skalérbarhed.

Dette viste sig desværre at være umuligt. TreadMarks er i nyere versioner ikke umiddelbart tilgængeligt, da det i dag er et kommercielt produkt, og den version DIKU er i besiddelse af er ikke porteret til Linux, men kørte kun på Amigos systemet - et netværk af Alpha-baserede arbejdsstationer tilsluttet et ATM netværk. Da Amigos i forsøgsperioden var udsat for alvorlige driftsproblemer, vurderede jeg at resultater indsamlet på denne platform ville være ubrugelige.

Med hensyn til CVM, der ville have været et brugbart alternativ til TreadMarks, havde forsøgsgruppen bag dette system under ledelse af Peter Keleher annonceret at de ville frigive en ny version der bla. kunne eksekveres under Linux den 31/7, men de har tilsyneladende ikke været i stand til at holde denne deadline.

Jeg var således ude af stand til at udføre en direkte sammenligning, og måtte i stedet nøjes med at sammenligne eksekveringstider ud fra en vurdering af de målte platformes karakteristika.

Hvad angår skalérbarhed, gjorde manglen på en fælles referenceplatform det også umuligt at udføre en direkte sammenligning. For 3DFFT's vedkommende er det rimeligt at antage at netværkskommunikationen vil dominere eksekveringstiden på forsøgssystemet, og da forsøgsplatformen har et markant dårligere CPU/båndbredde forhold end det system der blev benyttet til ydelsesmåling på TreadMarks i [16], risikerer man at en ydelsesmåling i praksis kun måler netværkets hastighed, snarere end DSM-platformens evner.

På denne baggrund måtte jeg acceptere at benytte en mindre ambitiøs forsøgsstrategi end jeg oprindeligt havde håbet. Sigtet med målingerne blev som følger:

1. Kørsler med en enkelt beregningsknude med samme problemstørrelser for begge programmer som blev benyttet i [16]. Tidsmålinger fra disse kørsler skal ligge til grund for en vurdering af ydelsen i forhold til den for TreadMarks rapporterede under hensyntagen til en vurdering af platformenes indvirkning.

2. Kørsler med 1–8 beregningsknuder og TSP med en passende stor beregningsopgave til at afdække skalérbarheden. Disse kørsler udføres med alle tre former for protokol ved låse — update, invalidate og hybrid.

3. Kørsler med 3DFFT og begge former for barriereprotokol — update og invalidate. Dette for at belyse ydelsesmæssig forskel på protokollerne.

5.3.1 Forsøgsbeskrivelse

På baggrund af overvejelserne beskrevet i forrige afsnit, valgte jeg at gennemføre følgende testkørsler:

Update protokol TSP på 1–8 beregningsknuder med parametrene *-f19b -r15*. Dette er samme inddata — 19 byer — som blev benyttet i [16], men der afsøges en større mængde mulige løsninger. Disse testkørsler havde til formål af vise at en acceptabel grad af skalérbarhed kunne opnås, samt at danne grundlag for en sammenligning med de andre protokoller.

TSP på 1 beregningsknude med parametrene *-f19b -r12*. Dette er præcis de samme parametre og inddata som blev benyttet i [16]. Denne kørsel skulle danne grundlag for en ydelsesmæssig sammenligning med TreadMarks.

TSP på 1 og 8 beregningsknuder med parametrene *-f19b -r16*. Formålet med at benytte en større rekursuv dybde var at undersøge, om en større problemstørrelse ville give udslag i en højere speedup.

3DFFT på 1–8 knuder med beregning af et inddatasæt på 64x64x32. Systemets store hukommelsesforbrug gjorde det umuligt at udføre denne kørsel med et inddatasæt på 64x64x64 som blev benyttet i [16] på mere end en beregningsknude. Formålet var at undersøge skalerbarhed og sammenligne ydelse med de andre protokoller.

3DFFT på 1 knude med inddatasæt på 64x64x64. Dette for sammenligning med TreadMarks.

Hybrid protokol TSP på 1–8 beregningsknuder med parametrene *-f19b -r12*. Uformelle testkørsler havde vist at eksekveringstiden med en større problemstørelse ville være endog meget tidskrævende på målsystemet. Disse kørsler havde til formål at danne grundlag for en sammenligning med TreadMarks, samt belyse ydelsesmæssige forskelle mellem låseprotokollerne.

Der blev ikke udført målinger med hybridprotokol og 3DFFT, da der ikke var implementeret hybridprotokol for barrierer — den eneste form for synkronisering programmet benytter.

Invalidate protokol TSP med samme parametre og formål som under hybrid protokollen.

3DFFT på 1..8 knuder med inddatasæt på 64x64x32. Dette med samme formål som under update protokol.

5.3.2 Forventninger

På baggrund af 3DFFT's tilgangsmønstre og computation/communication forhold, forventede jeg ikke at kunne måle nogen markant speedup. Det højeste opnåede speedup ved kørsler med 3DFFT og TreadMarks på et testsystem med et markant bedre CPU/båndbredde forhold og et dobbelt så stort inddataset var ca. 4,5. Til gengæld ville kørsler med 3DFFT give en indikation af SENSE's directory ydelse. Da 3DFFT udfører en total distribuering af lagerændringer for hver iteration, var det at forvente at update protokollen ville give en bedre ydelse end invalidate protokollen. Hvad angår TSP forventede jeg at kunne påvise et acceptabelt speedup for alle protokoller ved brug af 8 beregningsknuder. I forhold til TreadMarks betragtede jeg et speedup på 6 +-1 som acceptabelt, da det højeste opnåede speedup der blev rapporteret for dette program var ca. 6,2. I den direkte sammenligning med TreadMarks ved kørsler med en enkelt beregningsknude, forventede jeg at opnå en eksekveringstid der, justeret for forskellen i CPU ydelse mellem testsystemerne, var stort set den samme. Under antagelse af at moderne CPU'er generelt eksekverer flere instruktioner pr. clockfrekvens, anslog jeg justeringsfaktoren til 4. I sammenligningen mellem hybrid og invalidate protokol forventede jeg at kunne påvise en markant positiv effekt af hybridprotokollen.

5.4 Resultater

For alle de viste resultater gælder at der er tale om målinger på en enkelt knude.

5.4.1 TSP, update, 19 byer, rekursiv dybde=15

Antal knuder	1	2	3	4	5	6	7	8
Eksekveringstid (sek)	364,01	189,01	131,77	105,70	87,36	80.18	83,12	78,78
Speedup	1,00	1,93	2,76	3,44	4,17	4m54	4,38	4,62
Beskedder udvekslet	0	664	564	486	446	388	376	294
Data udvekslet (KB)	0	1146	2747	4727	8069	12743	14915	17091
Mem. læsninger (10**6)	1975	####	2146	1659	1357	1137	1023	930
Mem. skrivninger	55103	40747	36683	35682	32061	27811	27387	27677
Skrivecache hit%	36	32	31	30	29	28	27	27
CPU %	100	96	94	91	91	85	74	72

Kommentar Ved kørsel med to knuder opstod der overflow i den variabel der optalte antallet af læsninger til det delte lager.

5.4.2 TSP, update, 19 byer, rekursiv dybde=16

Antal knuder	1	8
Eksekveringstid (sek)	551,03	99,63
Speedup	1,00	5,53
Beskedder udvekslet	0	242
Data udvekslet (KB)	0	6770
Mem. læsninger (10**6)	1137	1617
Mem. skrivninger	22397	14820
Skrivecache hit%	27	18
CPU %	100	94

Kommentar Ved kørsel med 8 beregningsknuder blev der registreret stærkt afvigende eksekverinstider over flere kørsler. Tiderne varierede fra 87 sek. til 110 sek. Det valgte resultat er det som lå tættest på gennemsnittet af 10 kørsler.

5.4.3 TSP, invalidate, 19 byer, rekursiv dybde=12

Antal knuder	1	2	3
Eksekveringstid (sek)	157,33	498,77	879,31
Speedup	1,00	0,32	0,18
Beskedder udvekslet	0	50564	50144
Data udvekslet (KB)	0	4413	279103
Mem. læsninger (10**6)	401	212	143
Læsecache hit%	9	9	9
Læsefejl	0	9288	9187
Mem. skrivninger	375223	237105	191058
Skrivecache hit%	42	39	38
CPU %	100	18	14

Kommentar Læsefejl er antallet af opdateringer af det delte lager der blev rekvireret ved andre knuder. Testkørsler blev kun udført for 1–3 beregningsknuder, da det stod klart at der ikke kunne opnås speedup.

5.4.4 TSP, hybrid, 19 byer, rekursiv dybde=12

Antal knuder	1	2	3
Eksekveringstid (sek)	191,05	355,60	410,30
Speedup	1,00	0,54	0,47
Beskedder udvekslet	0	31999	32988
Data udvekslet (KB)	0	5226	18391
Mem. læsninger (10**6)	401	209	144
Læsecache hit%	9	9	9
Læsefejl	0	5619	4636
Mem. skrivninger	375223	253506	214052
Skrivecache hit%	42	41	39
CPU %	100	27	16

Kommentar Læsefejl er antallet af opdateringer af det delte lager der blev rekvireret ved andre knuder. Testkørsler blev kun udført for 1–3 beregningsknuder, da det stod klart at der ikke kunne opnås speedup.

5.4.5 TSP, alle protokoller og TreadMarks, 19 byer, rekursiv dybde=12

Protokol	Update	Invalidate	Hybrid	Tmk/4
Eksekveringstid (sek)	26,73	157,33	191,05	23,75

Kommentar Som tidligere nævnt er eksekveringstiden for TreadMarks taget fra [16]. Til grundlag for sammenligningen ligger en antagelse om, at ydelsesforholdet mellem den platform TreadMarks blev målt på og arbejdsstationerne i DIKU's Linux-cluster er ca. 4.

5.4.6 3DFFT, update, 64x64x32

Antal knuder	1	2	3
Eksekveringstid (sek)	14,71	49,93	61,75
Speedup	1,00	0,29	0,24
Beskedder udvekslet	0	46	91
Data udvekslet (KB)	0	261595	351519
Mem. læsninger (10**6)	24	12	8
Mem. skrivninger (10**6)	24	12	8
Skrivecache hit%	56	56	56
Barrieretid	0	41,17	54,51
Heraf netværk	0	30,98	#
CPU %	100	27	16

Kommentar Det var ikke muligt at opgøre netværkstiden ved barrierer ved 3 knuder, da kommunikationen bliver varetaget af letvægtstråde. Skrivninger til det delte lager er for overskuelighedens skyld her opgjort i millioner. Da resultaterne for 3 beregningsknuder tydeligt indikerede at speedup ikke kunne opnås, blev forsøget afbrudt.

5.4.7 3DFFT, invalidate, 64x64x32

Antal knuder	1
Eksekveringstid (sek)	25.27
Mem. læsninger (10**6)	24
Læsecache hit%	50
Mem. skrivninger (10**6)	24
Skrivecache hit%	56

Kommentar Eksekveringstiden ved brug af mere end 1 beregningsknude var så ekstremt høje, at det ikke var praktisk muligt at gennemføre tidsmålinger på disse.

5.4.8 3DFFT, update og TreadMarks, 64x64x64

System	SENSE update	TreadMarks/4
Eksekveringstid	33,75	10,25

5.4.9 Latenstid på testsystemets netværk

Følgende data blev indsamlet ved brug af programmet *ping*, som måler latenstiden for kommunikation over netværk.

```
[dduck@sci0 dduck]$ ping sci1
PING sci1 (192.68.8.2): 56 data bytes
64 bytes from 192.68.8.2: icmp_seq=0 ttl=64 time=0.2 ms
64 bytes from 192.68.8.2: icmp_seq=1 ttl=64 time=0.1 ms
64 bytes from 192.68.8.2: icmp_seq=2 ttl=64 time=0.1 ms
64 bytes from 192.68.8.2: icmp_seq=3 ttl=64 time=0.1 ms
64 bytes from 192.68.8.2: icmp_seq=4 ttl=64 time=0.1 ms
```

```
64 bytes from 192.68.8.2: icmp_seq=5 ttl=64 time=0.1 ms
64 bytes from 192.68.8.2: icmp_seq=6 ttl=64 time=0.1 ms
64 bytes from 192.68.8.2: icmp_seq=7 ttl=64 time=0.1 ms
64 bytes from 192.68.8.2: icmp_seq=8 ttl=64 time=0.1 ms
64 bytes from 192.68.8.2: icmp_seq=9 ttl=64 time=0.1 ms
64 bytes from 192.68.8.2: icmp_seq=10 ttl=64 time=0.1 ms
64 bytes from 192.68.8.2: icmp_seq=11 ttl=64 time=0.1 ms
64 bytes from 192.68.8.2: icmp_seq=12 ttl=64 time=0.1 ms
64 bytes from 192.68.8.2: icmp_seq=13 ttl=64 time=0.1 ms
64 bytes from 192.68.8.2: icmp_seq=14 ttl=64 time=0.1 ms
64 bytes from 192.68.8.2: icmp_seq=15 ttl=64 time=0.1 ms

--- sci1 ping statistics ---
16 packets transmitted, 16 packets received, 0% packet loss
round-trip min/avg/max = 0.1/0.1/0.2 ms
```

5.4.10 Profilering af TSP, 1 knude, update, 19 byer, rekursiv dybde=12

Funktion	% af eksekveringstid	Antal kald
TSP visit_node	98,57	616
STL map lower_bound	0,41	224712
TSP calc_bound	0,23	3754
STL map insert_unique	0,19	66820
SENSE wflush	0,10	218440
STL map ???	0,10	66820
TSP split_tour	0,10	1458

Kommentar Her er kun medtaget funktioner der udgør mere end 0,1% af eksekveringstiden.

5.4.11 Opsummering af profilering for TSP

Total TSP	Total SENSE og STL
98,9%	0,8%

5.4.12 Profilering af 3DFFT, 1 knude, update, 64x64x64

Funktion	% af eksekveringstid	Antal kald
STL map lower_bound	46,72	24628836
SENSE wflush	15,27	21437927
FFT FFTZ2	6,92	147456
FFT slave	3,92	1
FFT TRANSX	3,77	5
FFT CFFT3_2	3,68	4
STL map insert	3,02	1589126
STL map insert_unique	2,88	1589126
FFT FFTZ1	2,49	73728
FFT FFTZ1	2,16	24576
FFT FFTZ2	1,35	49152
FFT VRANLC	1,29	2
FFT FFTZ2	1,23	49152
STL map erase	1,20	1

Kommentar Her er kun medtaget funktioner der står for mere end 1% af eksekveringstiden.

5.4.13 Opsummering af profilering for FFT

Total FFT	Total SENSE og STL
26,81%	69,09%

6 Diskussion og evaluering

I dette kapitel perspektiveres resultaterne i forhold til forventningerne.

6.1 Resultater fra TSP, generelt

Resultaterne der blev opnået med update protokollen må siges at være tilfredsstillende, både med hensyn til eksekveringstid og skalérbarhed. Her opnås en ydelse der må siges at være sammenlignelig med TreadMarks, i den grad det er muligt at afgøre det, forskellene i testplatformene taget i betragtning.

Det samme kan ikke siges om update og invalidate protokollerne. Det kan ikke afgøres med sikkerhed hvad årsagen er, uden en direkte sammenligning mellem systemerne på identiske platforme, men i forhold til update protokollen er det markant at der udveksles langt flere beskedder (3 størrelsesordener flere) og langt mere data (igen 3 størrelsesordener mere). Ved tre knuder under ren invalidate protokol er det optimale tidsforbrug på den registrerede dataoverførsel på 100Mbit Ethernet med den målte latenstid henholdsvis 4 sekunder for selve transmissionen og 50 sekunder i latenstid. De tilsvarende tal for hybrid protokollen er 3 sekunder og 32 sekunder. For update protokollen er tidsforbruget kun 0,3 sekund for transmission og 5 sekunder i latenstid.

Envidere er det klart at de mange læsninger — adskillige millioner — der med update protokollen udføres helt uden at konsultere directory, må have en ydelsesnedsættende effekt for invalidate protokollerne. Dette fremgår klart af resultaterne ved kørsel med en enkelt beregningsknude, hvor invalidate protokollerne er 3–4 gange langsommere end update protokollen.

Det fremgår klart af resultaterne, at programmets tilgangsmønstre har givet SENSE en mulighed for at nyde godt af mikrocachen ved skrivninger. Næsten en tredjedel af skrivningerne er blevet håndteret gennem cachen, og har således ikke medført tilgang til træstrukturen i selve directoryet.

Med hensyn til læsninger, er effekten mere begrænset. Her er kun omkring 10% af læsninger blevet tilfredsstillet fra mikrocachen. Dette resultat antyder at TSP ikke udfører læsetilgang til lageret sekventielt. Optimeringer i programmet ville formegentlig kunne rette op på dette, og resultere i en implementation af algoritmen der kunne eksekveres mere effektivt med invalidate protokollerne.

Hybrid protokollens metode til afdækkelse af mobilitet må siges at have haft en gavnlig effekt, idet denne protokol ved tre beregningsknuder har en eksekveringstid der er under halvdelen af den rene invalidate protokols. På trods af dette opnås ikke noget speedup.

Opsummering Med update protokol blev der opnået tilfredsstillende resultater. Hybridprotokollen var markant mere effektiv end den rene invalidate protokol, men fremviste alligevel et uventet dårligt resultat. Den rene invalidate protokol leverede denne måleseries dårligste resultater, både med hensyn til eksekveringstid og skalérbarhed. Årsagerne til invalidate protokollernes dårlige ydelse er en langsommere læsetilgang til det delte lager, samt et markant højere (3 størrelsesordener!) niveau af beskedudveksling og datatransmission.

6.2 Resultater fra 3DFFT, generelt

SENSE er tydeligvis ude af stand til at opnå et speedup på den benyttede testplatform med dette program. Selv hvis netværket havde været en faktor 10 hurtigere, ville der ikke være opnået et speedup. Faktisk ville speedup kun have været markant, hvis barrierer var blevet udført helt "gratis" — her ville speedup have været 1,68.

Dette skyldes uden tvivl, at 3DFFT's tilgangsmønster er noget nær det værste man kan udsætte SENSE's hukommelsesssytem for. En stor del af tilgangene er *strided*, dvs. fremadskridende med fast afstand > 1. I SENSE medfører denne form for skrivninger at lageret opdeles i mange små lagerblokke, som alle har tilknyttet en informationsblok der skal registreres, og ved synkronisering transmitteres. Størstedelen af lagerforbruget of datatransmissionen består således af denne konsistensinformation. Endvidere er netop mængden af disse informationsblokke det *n* som store-O karakteristikken for søgninger i directory afhænger af.

Da disse effekter er en direkte konsekvens af designet af SENSE's hukommelseshieraki og programmets tilgangsmønstre, er der ikke tale om effekter som kan neutraliseres eller undgås uden en så gennemgribende designændring af SENSE, at der ville være tale om et helt andet system.

Opsummering 3DFFT programmets tilgangsmønstre er stærkt ydelseshæmmende for et moderne hukommelseshieraki, som beskrevet i afsnittet 2.4.1. Da SENSE's hukommelseshieraki er opbygget ud fra samme model, og således er meget følsomt overfor lagertilgang der ikke udviser temporær eller spatiel lokalitet, er det ikke overraskende at både ydelse og skalébarhed er dårlig. Der er ikke umiddelbart nogen måde dette resultat kan forbedres på, bortset fra en total ombygning af SENSE, eller en revision af 3DFFT der indfører et mere sekventielt tilgangsmønster.

6.3 Sammenligning mellem protokoller

På baggrund af de opnåede resultater, kan der ikke herske nogen tvivl om, at den rene update protokol er den mest effektive af de tre protokoltyper der er til rådighed i SENSE. Dette skyldes ikke alene at mængden af meddelelser og andre data der transmitteres er markant lavere, men også at der kan benyttes en læsninger til delt lager der ikke konsulterer en bogføringsmekanisme — alle relevante opdateringer i henhold til happens-before relationen vides at være integreret i lageret når ejerskab af en kritisk region er etableret.

Den negative effekt ved overførsel af store mængder data kan forventes at blive mindre i fremtiden, da der stort set ikke er nogen begrænsning på, hvor høj en båndbredde der kan opnås på datanet. Derimod kan der ikke forventes markante forbedringer af latenstiden for overførsel af data med meget mere end en størrelsesorden i forhold til det benyttede testsystem, da denne i sidste ende er afhængig af lyshastigheden i det benyttede medie. På den baggrund kan det forventes at update protokollen også vil være invalidate protokollerne ydelsmæssigt overlegen på fremtidens datanet.

Der er ingen tvivl om, at der eksisterer — eller kan konstrueres — programmer, hvor invalidate protokollerne ville være update protokollen ydelsesmæssigt overlegen. Disse ville være karakteriseret ved tilgangsmønstre, hvor de enkelte knuder tilbragte størstedelen af deres eksekvering med at uddrage relativt få oplysninger fra et stort datasæt, der ikke var delt med andre knuder. Dette er ikke tilfældet for de brugte testprogrammer, Her produceres ikke nok irelevante opdateringer til at svinge ballancen i invalidate protokollernes favør. Specifikt produceres der i 3DFFT *ingen* irelevante opdateringer.

I sammenligning mellem de to invalidate protokoller, er det tydeligt at den mængde ekstraadministration som indføres af hybrid protokollen (hvilket fremgår af kørsler med en enkelt beregningsknude) i højeste grad bliver opvejet af bedre ydelse for flere knuder ved kørsler med TSP. Det er et markant og interessant resultat, at en så simpel mekanisme til afdækning af lagerblokkes mobilitet, har en så kraftig effekt. Ved tre beregningsknuder eksekverer hybrid protokollen i mindre end den halve tid af den rene invalidate protokol.

Opsummering Update protokollen er for de brugte testprogrammer og problemstørrelser klart den mest effektive, hvilket skyldes den bedre økonomisering med netværkskommunikation, og den mere simple læsemekanisme til det delte lager. Hybridprotokollen er næsten dobbelt så effektiv som den rene invalidate protokol, hvilket skydes dens evne til at nedbringe netværkskommunikationen med ca. 50%.

6.4 Sammenligning med TreadMarks

Som nævnt er en sammenligning mellem DSM-systemer på forskellige testplatforme og med forskellige problemstørrelser for testprogrammerne næsten meningsløs. CPU/båndbredde forholdet på de to systemer er meget forskellig, og det FDDI netværk der blev benyttet til testkørslerne med TreadMarks implementerer i modsætning til Ethernet fairness, og vil derfor udvise en meget forskellig opførsel ved høj belastning. Det er dog muligt at lave en grov sammenligning mellem de kendte resultater for TreadMarks fra [16] og kørslerne med en enkelt beregningsknude for SENSE.

Under antagelse af at arbejdsstationen SENSE blev testet på er ca. 4 gange hurtigere end den HP PA-RISC 133MHz maskine TreadMarks blev testet på, kan det udledes at ydelsen for TSP med update protokol var sammenlignelig. Dette må siges at være et glimrende resultat, når det tages i betragtning at SENSE er et nyudviklet system, der ikke har gennemgået en grundig tuning.

Resultaterne for 3DFFT er mindre opmuntrende, men dog stadig indenfor samme størrelsesorden, og lever derfor op til det sycceskriterie jeg satte for ydelsen under planlægningsfasen. TreadMarks er dog så meget hurtigere — en faktor 3 — at det er tvivlsomt om SENSE kan optimeres til at opnå en tilsvarende ydelse.

Et mere interessant spørgsmål er, om det er *ønskværdigt* at optimere SENSE til at opnå en bedre ydelse med 3DFFT. Programmets tilgangsmønster er ikke bare hæmmende for ydelsen for SENSE, men vil også give dårlig ydelse i en ikke-distribueret implementation. De mange *strided* lagertilgange er ikke i overensstemmelse med den antagelse om spatiel og temporær lokalitet i lagertilgange som moderne datamater udvikles efter. De fleste oversættere forsøger at undgå at generere programmer der udviser denne effekt, og en væsentlig del af programtuning i nutidens programmeringsmiljøer består i at ændre algoritmer til at benytte en mere cache-venlig lagertilgang. Fremtidens programmer vil således alt andet lige have en tendens til *ikke* at udvise tilgangsmønstre af denne karakter, især i forbindelse med højhastigheds talknusning.

Uden en mere omfattende undersøgelse af de eksisterende (og projekterede) brugerprogrammer, vil det altså være halsløs gerning at optimere efter et program, der med rimelighed kan antages at udvise atypisk adfærd.

Opsummering Da det har været umuligt at gennemføre en direkte sammenligning mellem SENSE og TreadMarks, kan der kun tentativt udføres en grov ydelsessammenligning på baggrund af måleresultater fra kørsler med en enkelt beregningsknude. Disse indikerer at SENSE lever op til succeskriteriet for ydelse, dvs. en ydelse der er i samme størrelsesorden som TreadMarks med de samme programmer, på den samme testplatform. En tilsvarende sammenligning for skalérbarhed kan ikke gennemføres, på grund af markante forskelle i testplatformenes CPU/båndbredde forhold og de benyttede datanets ydelseskarakteristik.

7 Konklusion

Alt i alt mener jeg at have levet op til opgavens målsætning: Jeg har gennemført en undersøgelse af de grundlæggende problemstillinger indenfor SMLB

DSM-systemer, og mener at have underbygget størstedelen af mine arkitektoniske argumenter i kapitel 3 med resultaterne fra implementations- og testfasen. En markant undtagelse er hybridprotokollen, som — på trods af at den leverede en klart bedre ydelse end den rene invalidate protokol — ikke levede op til de resultater der blev opnået med update protokollen. Det er dog værd at bemærke sig at en hybridisering af invalidate protokoller i eksisterende sidebaserede systemer, efter samme metode som benyttes i SENSE, må antages at medføre forbedret ydelse og nedsat netværkskommunikation. En sådanne modifikation må antages at være ganske simpel at indføre.

Jeg mener at SENSE har vist sig at leve op til forventningerne. Jeg betragter målsætningerne om brugervenlighed, kort udviklingstid og portabillitet som fuldt opfyldt. Hvad angår ydelse og skalérbarhed er resultaterne opmuntrende, men ikke endegyldigt positive. Et klart svar på, om successkriteriet — en ydelse tilsvarende TreadMarks — blev opnået kan ikke gives på grundlag af testkørsler med 2 testprogrammer uden direkte sammenligning på ens platforme. Således fik jeg ikke besvaret et af de spørgsmål som lå mig mest på sinde: om et rent softwarebaseret hukommelseshieraki ville være et delvist hardwarebaseret (MMU) overlegent uden tuning af testprogrammerne.

Jeg er dog på baggrund af profileringsresultaterne overbevist om, at der kan opnås markante konstant-faktor forbedringer af ydelsen ved indførelse af flere "fast paths" i den kritiske eksekveringssti. Ydermere er det min vurdering at ydelsen kunne forbedres yderligere ved indførelse af flere arkitektoniske finesser fra moderne arbejdsstationers hukommelseshierakiopbygning. Et ekstra niveau af cache mellem mikrocachen og directory — tilsvarende strukturen med 2 niveauer af cache i moderne processorer — er et oplagt emne for udvidelse.

Med hensyn til protokoller har testkørslerne tydeligt vist at den rene update protokol er invalidate protokollerne langt overlegne. Det fremgår også klart at antallet af udvekslede beskedder ved brug af update protokol er langt mindre end det der ses for TreadMarks (2–3 størrelsesordener) ved sammenlignelige problemløsninger med TSP programmet. Dette resultat er i overenstemmelse med de resultater der er rapporteret for ORDER i [18]. Da ORDER og SENSE deler en lang række grundlæggende designpunkter er dette resultat ikke overraskende, men det er opmuntrende at en ydelse i samme klasse kan opnås for programmer der stort set ikke er modificeret eller annoteret i forhold til deres oprindelige udformning.

Jeg mener godt jeg kan tillade mig at være godt tilfreds med SENSE. Der er tale om et rent softwarebaseret system, der leverer en ydelse som i de fleste andre systemer ville kræve et stort arbejde fra programmørens side med indførelse af annoteringer, eller en kraftigt specialiseret oversætter. SENSE er hurtigt, overskueligt, brugervenligt og *meget* portabelt. Da der er mange oplagte optimeringer der endnu ikke er indført i systemet, forventer jeg at vidreudvikling vil gøre SENSE fuldt konkurrencedygtigt med andre SMLB systemer på alle områder.

Personligt mener jeg det mest interessante resultat af den praktiske del af opgaven er, at det lykkedes at udvikle SENSE på meget kort tid. Metoderne der blev anvendt er ikke ukendte, men desværre alt for sjældent anvendt i praksis. Brugen af grundig planlægning og stabile standardværktøjer medførte en klar forbedring af produktiviteten.

A Programeksempler i pseudokode

Her er samlet de programeksempler der var for omfangsrige til at bringe direkte i teksten. De fleste af programmerne er lavet i pseudokode, der enten minder om Pascal eller C, afhængig af programmets karakter. Variable, arrays eller objekter der deles mellem knuder erklæres med tilføjelsen *shared*. Normalt eksekveres samme program på alle knuder, og en knude kan spørge til sit nummer med funktionen *nodeID*. Dette er tilsvarende til metoden man bruger i UNIX til at afgøre om man eksekverer en moderprocess eller et barn.

A.1 Synkron og asynkron kommunikation

Her illustreres i pseudokode forskellen på et program der udfører synkron og asynkron kommunikation. Muligheden for overlap af kommunikation og beregning ved asynkron kommunikation fremgår af eksemplet. Programmet fylder ti vektorer med tilfældige tal, og transmitterer dem til modtageren, der udskriver dem på skærmen.

A.1.1 Synkron kommunikation

```
Program Producent;

Var
  V : Array[1..100] of Integer;
  N, M : Integer;
  C : Connection;

Begin
  C:=open_connection(konsument);
  for M:=1 to 10 do
    for N:=1 to 100
      V[N]:=random();
    transmit(V);
  close_connection(C);
end.

program Konsument;
Var
  V : Array[1..100] of Integer;
  N : Integer;
  C : Connection;

Begin
  C:=accept_connection(producent);
  for N:=1 to 10 do
    Begin
      recieve(V);
      print(V);
    End;
  close_connection(C)
```

```
End.
```

A.1.2 Asynkron kommunikation

For overskuelighedens skyld benyttes der kun overlap af kommunikation og beregning i producentprogrammet.

```
Program Producent;

Var
  V1, V2 : Array[1..100] of Integer;
  M : Integer;
  C : Connection;
  First : Boolean;

Procedure fill_array(Var A : Array[1..100] of Integer)
Var
  N : Integer;
Begin
  For N:=1 to 100 do
    A[N]:=random()
End;

Begin
  First:=true;
  C:=open_connection(konsument);
  for M:=1 to 5 do
    Begin
      if First then
         First:=false;
      else
         while C.notfinished do
           pause(5);
      fill_array(V1);
      C.transmit(V1);
      fill_array(V2);
      C.transmit(V2);
    End;
end.

Program Konsument;
Var
  V : Array[1..100] of Integer;
  N : Integer;
  C : Connection;

Begin
  C:=accept_connection(producent);
  for N:=1 to 10 do
    Begin
```

```
      recieve(V);
      while C.notfinished do
        pause(5);
      print(V);
    End;
  close_connection(C)
End.
```

A.2 Tre programmeringsmodeller for parallellisme

Her illustreres de tre fremherskende programmeringsmodeller indenfor DSM-systemer: Beskedbaseret (Message Passing, MP), delt lager (Shared Memory, SM) og objektbaseret (Mobile Objects, MO). Programmet fordeler arbejdet med at fylde en 2x100 matrice med tilfældige tal på to knuder. SM og MO programmet benytter *barrierrer* til synkronisering, som sikrer at alle knuder er nået til samme sted i eksekveringen, før de får lov til at fortsætte. I MP implementationen benyttes i stedet synkron kommunikation.

A.2.1 Arbejdsdeling, beskedbaseret

Bemærk at det ville være relativt simpelt at opnå bedre overlap af beregning og kommunikation ved at bruge asynkron kommunikation. Dette ville give mulighed for at udskrive knude 0's vektor samtidigt med at knude 1's vektor blev modtaget.

```
Program SimpelMP;

Var
  M : Array[0..1, 1..100] of Integer;

Procedure fill_array(Var V : Array[1..100] of Integer;
Var
  N : Integer;
Begin
  for N:=1 to 100 do
    V[N]:=random();
End;

Begin
  fill_array(M[NodeID]);
  if NodeID=1 then
    transmit_sync(M[1], 0);
  else
    Begin
      recieve_sync(M[1], 1);
      WriteLn(M);
    End;
End.
```

A.2.2 Arbejdsdeling, delt lager

Denne implementation er mere overskuelig end MP programmet, da kommunikation er implicit. Det er altså ikke programmørens opgave at håndtere transmission af data fra knude 1 til knude 0.

```
Program SimpelSM;

Var
  M : Array[0..1, 1..100] of shared Integer;

Procedure fill_array(Var V : Array[1..100] of Integer;
Var
  N : Integer;
Begin
  for N:=1 to 100 do
    V[N]:=random();
End;

Begin
  fill_array(M[NodeID]);
  barrier();
  WriteLn(M);
End.
```

A.2.3 Arbejdsdeling, mobile objekter

Dette program minder en del om SM programmet, omend den objektbaserede syntaks er en kende mere omfangsrig. Hvad *ikke* umiddelbart fremgår af programmet er, at skrivningerne til objektet i et system med single-writer protokol på en pr. objekt-basis vil blive serialiseret! Dette program vil altså ikke skalere på en stor del af MO DSM-systemerne.

```
Program SimpelMO;

Class
  tMatrix shared
    Data:
      M : Array[0..1, 1..100] of Integer;

    Methods:
      Procedure FyldVec(N : Integer);
  End; {tMatrix}

Var
  M : tMatrix;

Procedure tMatrix.fill_array(Index : Integer)
Var
  N : Integer;
Begin
```

```
  for N:=1 to 100 do
    M[N]:=random();
End;

Begin
  M.fill_array(NodeID);
  barrier();
  WriteLn(M);
End.
```

A.3 Producent/konsument programmet

Producent/konsumment programmet illustrerer nogle af de mest grundlæggende faciliteter et parallelt miljø skal stille til rådighed: Deling af data og synkronisering mellem processer der tilgår de delte data. Det vises her i to udgaver: En ukorrekt uden synkronisering, der giver mulighed for væddeløbstilstand (data races) og en korrekt med synkronisering.

A.3.1 Producent/konsument usynkroniseret

Dette program indeholder en væddeløbstilstand, da der er risiko for at producenten vil tilgå køen af data samtidigt med konsumenten. Følgende sekvens af ordrer kan blive resultatet:

```
(producent) Tail := Tail +1
(konsument) RemoveQ := Q[Tail]
(konsument) Tail := Tail -1
(producent) Q[Tail] := x;
```

Dette er tydeligvis ikke det ønskede resultat, da konsumenten modtager en værdi der i værste fald er udefineret, og i bedste fald er forældet. Ydermere overskriver producenten en værdi, der endnu ikke er behandlet af konsumenten.

```
Program ProdKons;

Const
    Qsize = 100;

Var
  Q : Array[0..Qsize] of Shared Integer;
  Tail : Shared Integer;

procedure InsertQ(x : Integer);
begin
  Tail := Tail+1;
  Q[Tail] := x;
end;

function RemoveQ : Integer;
begin
```

```
  RemoveQ := Q[Tail];
  Tail := Tail -1;
end

procedure Producent;
begin
  repeat
    if Tail<Qsize then
      InsertQ(Random());
  until false;
end;

procedure Konsument;
begin
  repeat
    if Tail<>-1 then
      WriteLn(RemoveQ);
  until false;
end;

begin
  Tail=-1;
  start Producent on node 0
  start Konsument on node 1
end.
```

A.3.2 Producent/konsument synkroniseret

I denne udgave af programmet er der tilføjet en lås, som sikrer udelelig adgang til den delte datastruktur. På den måde undgås væddeløbstilstandene.

```
Program ProdKons;

Const
    Qsize = 100;

Var
  Q : Array[0..Qsize] of Shared Integer;
  Tail : Shared Integer;

procedure InsertQ(x : Integer);
begin
  Tail := Tail+1;
  Q[Tail] := x;
end;

function RemoveQ : Integer;
begin
  RemoveQ := Q[Tail];
  Tail := Tail -1;
```

```
end

procedure Producent;
begin
  repeat
    Acquire(1);
    if Tail<Qsize then
      InsertQ(Random());
    Release(1);
  until false;
end;

procedure Konsument;
begin
  repeat
    Acquire(1);
    if Tail<>-1 then
      WriteLn(RemoveQ);
    Release(1);
  until false;
end;

begin
  Tail=-1;
  start Producent on node 0
  start Konsument on node 1
end.
```

A.4 Samtidighed eller mangel på samme

Dette programmeringseksempel illustrerer behovet for en kausal ordning af skrivninger i et SMLB-system. Uddata bør være 2 0 og 0. Eksemplet svarer ikke fuldstændigt til det, der diskuteres i teksten, da der for iverskuelighedens skyld er brugt en barriere til synkronisering.

```
Program samtidighed;

Var
  V : Array [1..3] of Shared integer;
  N : Integer;

Begin
  if NodeID=0 then
    For N:=1 to 3 do
      V[N]:=0;
  barrier();
  if NodeID=2 then
    V[1]:=2;
  barrier();
  print(V);
```

```
End.
```

A.5 Irelevante opdateringer

I dette eksempel opdateres en stor del af det delte lager, men kun en lille del af det delte lager er af interesse for hovedknuden (knude 0).

```
Program Irelevant;

Var
  A : Array [0..1, 1..10000] of Shared integer;
  Count : Array [0..1] of Shared integer;
  N, Total : Integer;


Begin
  for N:=1 to 10000 do
    A[NodeID,N]:=random();
  Count[NodeID]:=0;
  for N:=1 to 10000 do
    Count[NodeID]:=Count[NodeID]+A[NodeID, N];
  barrier();
  if NodeID=0 then
    begin
      Total:=0;
      for N:=0 to 1 do
        Total:=Total+Count[N];
      print(Total);
    End;
end.
```

B SENSE test- og eksempelprogrammer

Her vises eksempler på simple SENSE programmer, samt nogle af de testprogrammer jeg benyttede i den indledende test- og verificeringsfase.

B.1 Testprogram 1

Dette var det første testprogram jeg udviklede til at undersøge de grundlæggende funktioner i directory.

```
#include <sys/times.h>
#include<stdlib.h>

#include "sense_unit.h"

void printtime()
{
  struct tms tmp;
```

```
  times (&tmp);
  cout << "User time is   " << (float) tmp.tms_utime/100 << "\n";
  cout << "System time is " << (float) tmp.tms_stime/100 << "\n";
}

const int limit = 500000;
const int ulim = 100;
Tsense_sint stest[limit];
int ntest[limit];

void main()
{
  sense_init();
  /*
  Tsense_sint i, pad1, test2, pad2, test3;
//  Tsense_sint atest[20000];
  int i2;
  i.write(1);
  i2=i.read();
  i.write(-1);
  test2.write(1); // Should provoke a flush
  test3.write(3); // Should provoke another flush - check if dirunit start is revi
  i2=i.read();
  sense_acq(2);
  Tsense_sdouble d;
  d.write(1.0/3.0);
  double d2;
  d2=d.read();
  test2.write(2); // Flush the double
  char c;
  cin >> c;
  */

  cout << "Starting " << ulim<< "Xwrite to " << limit << " nor-
mal ints now\n";
  printtime();
  long int total=0;
  for (int n=0; n<ulim; n++) {
    for (int i=0; i<limit; i++) {
      ntest[i]=rand();
      total=total+ntest[i];
      }
    }
  cout << total << "\n";
  cout << "Test finished.\n";
  printtime();
  cout << "Starting " << ulim << "Xwrite to " << limit << " shared ints now\n";
  total=0;
  for (int n=0; n<ulim ; n++ ) {
    for (int i=0; i<limit; i++) {
```

```
      stest[i].write(rand());
      total+=stest[i].read();
      }
    }
  cout << total;
  cout << "Test finished.\n";
  printtime();
}
```

B.2 Testprogram 2

Dette testprogram undersøger om lageropdateringer rekvireres korrekt i henhold til happens-before relationen.

```
#include <sys/times.h>
#include<stdlib.h>

#include "sense_unit.h"

void printtime()
{
  struct tms tmp;
  times (&tmp);
  cout << "User time is   " << (float) tmp.tms_utime/100 << "\n";
  cout << "System time is " << (float) tmp.tms_stime/100 << "\n";
}

void main()
{
  Tsense_sint test[1000];
  sense_init();
  cout << "**** User program starts\n";
  if (sense_thisnode==0) {
    cout << "Writing to first part of array\n";
    for (int n=0; n<500 ; n++) {
      test[n].write(1);
    }
    cout << "Done\n";
  }
  else {
    cout << "Taking a nap\n";
    sleep(10);
    cout << "Done\n";
  }
  cout << "Waiting at barrier, ltime is " << sense_ltime << "\n";
  sense_barrier();
  cout << "Past barrier, ltime is " << sense_ltime << "\n";
  cout << "Trying to make node 1 request range 0..499\n";
  if (sense_thisnode==1) {
    int sum=0;
```

```
    for (int n=0; n<500; n++) {
      sum+=test[n].read();
      test[n].write(2);
    }
    cout << "Sum is " << sum << "\n";
  }
  else {
    cout << "Taking a nap\n";
    sleep(10);
    cout << "Done\n";
  }
  cout << "Waiting at barrier, ltime is " << sense_ltime << "\n";
  sense_barrier();
  cout << "Past barrier, ltime is " << sense_ltime << "\n";
  cout << "Trying to make node 0 re-request range 0..499\n";
  if (sense_thisnode==0) {
    int sum=0;
    for (int n=0; n<500; n++) {
      sum+=test[n].read();
    }
    cout << "Sum is " << sum << "\n";
  }
  /*  else {
    cout << "Taking a nap\n";
    sleep(10);
    cout << "Done\n";
  }
  */
  sense_finish();
}
```

C Kildekode for SENSE

Her bringes kildekoden for SENSE.

C.1 lockman

```
/*
This is the lock manager. It forwards lock acquire requests to current
owner of the lock. It is also responsible for coordinating the connections
on the computational nodes. That job ought to be done by a seper-
ate program,
but as it fits right into the init-state here, we make this program
do two jobs, which usually is a Bad Thing

Code is ugly, but we want to preserve the connections in or-
der to save
on startup time for establishing them.
```

```
Message format:
Ingoing request is 2*int, first is node ID of sender, sec-
ond is lock ID. These
sockets must be set with "expedite" flag in order to keep TCP from delaying
the small packets in the hope of additional traffic. This will keep latency
to a minimum. Lamport equalization is performed by the com-
putational nodes
while exchanging consistency information, so no information is needed here.

Possible optimizations:
* Presently the server handles only one lock at a time! It's quite pos-
sible to
replace mainlock mutex with an array of mutexes protecting one lock at a time
* Share the load of managing locks over n servers

Possible enhancements:
* Make it posible to close the lockmanager as a part of sense_end()
*/

#define _REENTRANT
//#define silent
#include <pthread.h>
#include "nodeaddr.h"
#include "sense_connections.h"
#include "Defaults.h"
#include "sense_control.h"
#include "thread_support.h"

// Who owns which lock?
int owner[sense_maxlocks];
// Secures atomic access to owner
pthread_mutex_t lockman_main;
// Waits for the computational nodes to signal that their server connections
// are up
//pthread_mutex_t servers_up;
// Sockets for ingoing connections - request lock and ack handoff
Tsense_servsock ingoing[sense_nodes];
// Sockets for syncing control flow during initialization
Tsense_servsock initcontrol[sense_nodes];
// Sockets for forwarding recquest lock msg. - these are client sockets
// as we would like to know where they go!
Tsense_clisock outgoing[sense_nodes];
// Array of thread ID's used for initialization
pthread_t LMtids[sense_nodes];
// New infrastructure - one mutex pr. node
pthread_mutex_t lock_mutex[sense_maxlocks];

#ifdef SENSE_stats
double maintime=0.0;
double forwardtime=0.0;
```

```
double acqtime=0.0;
int numrequests=0;
Tsense_timer maintimers[sense_nodes];
Tsense_timer forwardtimers[sense_nodes];
Tsense_timer acqtimers[sense_nodes];
#endif


/*
Function for handeling the lock handoff
This is actually an iterative server. All it does is forward-
ing the request
to acquire a lock, and then waiting for the ack on completed hand-
off from
the original requestor
*/
void * threadfun(void * p) {
  cout << "Thread function for recieving lock request is up.\n";
  Tsense_servsock * thissock;
  int nodeID; // The node requesting the lock
  int lockID; // The lock it requests
  thissock = (Tsense_servsock * ) p;
  thissock->accept();
  do {
    nodeID=thissock->recieve();
    if (nodeID==-1) {
      cout << "LockThread: Killing myself now...\n";
      pthread_exit(p);
    }
    #ifdef SENSE_stats
    maintimers[nodeID].start();
    forwardtimers[nodeID].start();
    numrequests++;
    #endif
    lockID=thissock->recieve();
    #ifndef silent
    cout << "LockThread: Got request for lock " << lockID << " from node ";
    cout << nodeID << "\n";
    #endif
    // Secure atomic access to owner, and insure that we do not start handing
    // off the lock to two nodes at the same time
    //pthread_mutex_lock(&lockman_main);
    #ifndef silent
    cout << "Locking mutex " << lockID << "\n";
    #endif
    pthread_mutex_lock(&lock_mutex[lockID]);
    int dest_nodeID=owner[lockID];
    // Tell current owner who wants the lock
    #ifndef silent
    cout << "Forwarding request to node " << dest_nodeID << "\n";
```

```
    #endif
    outgoing[dest_nodeID].send(nodeID);
    outgoing[dest_nodeID].send(lockID);
    #ifdef SENSE_stats
    forwardtimers[nodeID].end();
    acqtimers[nodeID].start();
    #endif
    // Wait for handoff from new owner
    int ack;
    #ifndef silent
    cout << "Waiting for handoff acq from original requestor\n";
    #endif
    ack=thissock->recieve();
    /*
    cout << "Got it - sending goahead to old owner\n";
    outgoing[dest_nodeID].send(ack);
    */
    #ifndef silent
    cout << "Done, back to listening\n";
    #endif
    // Now the inquiring node owns the lock
    owner[lockID]=nodeID;
    //pthread_mutex_unlock(&lockman_main);
    #ifndef silent
    cout << "Unlocking mutex " << lockID << "\n";
    #endif
    #ifdef SENSE_stats
    maintimers[nodeID].end();
    acqtimers[nodeID].end();
    maintime+=maintimers[nodeID].elapsed();
    forwardtime+=forwardtimers[nodeID].elapsed();
    acqtime+=acqtimers[nodeID].elapsed();
    #endif
    pthread_mutex_unlock(&lock_mutex[lockID]);
  } while (true);
}

void init(){
  cout << "Lockman, expecting " << sense_nodes << " nodes\n";
  #ifdef SENSE_debug
  cout << "Initializing main control sockets (for internal barrier).\n";
  #endif
  // Initialize the internal barrier manager
  master_init_control();
  #ifdef SENSE_debug
  cout << "done\n";
  #endif


  // Initialize the main mutex
```

```
  pthread_mutex_init(&lockman_main, NULL);

  for (int n=0; n<sense_maxlocks; n++) {
    owner[n]=0; // All locks start up owned by node 0
  }

  cout << "Preparing to init the connections for locks\n";
  spawn_server_connections(ingoing, sense_nodes,
  sense_lockman_port_start, LMtids, -1, threadfun);
  cout << "done\n";

  cout << "Waiting for computational nodes at internal barrier.\n";
  barrier();
  cout << "All nodes have arrived - setting up client sockets\n";

  // When all nodes are in, we can continue setting up
  // connections
  for (int n=0; n<sense_nodes; n++) {
    // And now it's safe to open the connection for forward-
ing lock req's
    outgoing[n].init(sense_nodes_IP[n], sense_lockman_forward_port, true);
  }

  cout << "All client sockets are up, waiting for nodes at barrier.\n";
  barrier();
  cout << "All nodes arrived, ready to do some *real* computation.\n";

}

void main(){
  init();
  cout << "Waiting for the computational nodes to finish at barrier.\n";
  barrier();
  cout << "Waiting for them to kill themselves\n";
  for (int n=0; n<sense_nodes; n++) {
    outgoing[n].send(-1);
  }
  barrier();
  cout << "Fine, now we can exit\n";
  barrier();
  #ifdef SENSE_stats
  cout << "Statistics:\n";
  cout << "-----------\n";
  cout << "Aver. time to handle lock request was " << maintime/numrequests;
  cout << "\n";
  cout << "Avrg. time to forward the request was " <<
   forwardtime/numrequests;
  cout << "\n";
  cout << "Avrg. time waiting for acq. from owner was " <<
acqtime/numrequests;
```

```
  cout << "\n";
  cout << "Number of locks handled was " << numrequests << "\n";
  #endif
  cout << "Done, bye!\n";
}
```

C.2 Defaults.h

```
/*
In this file you define the various constants pertaining to the program
you wish to execute using SENSE. These include size of shared mem-
ory,
network addresses of the involved computation nodes and lock/barrier
managers.

Other things defined here is #define's that decide wether the sys-
tem is
to be run in debug mode.

The type of the UC (minimal sharing unit - usually a word) and other
protability-related stuff is defined in unit "portability"

The IP addresses of the computers involved is defined in nodeaddr.h/cpp
*/

//-----------------------------------------------------------
-------------------
#ifndef DefaultsH
#define DefaultsH
//-----------------------------------------------------------
-------------------

#include "portability.h"

// Number of computational nodes
const int sense_nodes=NUMNODES;

// Which node is this one?
const int sense_thisnode=THISNODE;

// The trailing int must be equal to sense_nodes. Currently sup-
ports a max
// of 8 nodes
//#define numnodes2
#include "sense_copysets.h"

// Makes the units print all sorts of information
//#define SENSE_debug
```

```
// Collect stats on memory usage and messages transmitted
#define SENSE_stats

// Uncomment *only one* of following - this decides which pro-
tocol is used
// at locks!!!!
// When using pure update barriers will be pure update too
#define SENSE_pure_update
//#define SENSE_pure_invalidate
//#define SENSE_hybrid

// If we use hybrid, we need a treshold. This is how many times a block
// of memory has to be requested before it's automagically forwarded
// Good values should be in the range of 2..10, depending on sharing
// patterns and number of iterations. There is currently NO damper mech-
anism,
// so take care setting this too low!!! Study output of SENSE_stats!!!!
const int hybrid_treshold=2;

// At some point the following will decide if barriers will be up-
date or
// invalidate. As SENSE primarily is concerned with study-
ing the read/write
// patterns concerning locks, we may never know
//#define SENSE_barriers_update

#ifdef SENSE_pure_update
#define SENSE_barriers_update
#endif

// Size of shared memory in UC's
const Tsense_addr sense_memsize=250000;

// Maximum number of locks
const int sense_maxlocks=100;

// Do we get a warning if we lock a lock twice?
const bool locks_strict=true;

// Shall we check sanity regarding length of directory blocks?
// This is only necessary during debugging of SENSE!
const bool check_dirblock=false;

// Shall we test if the update protocols integrate updates correctly?
// Also only necessary when debugging SENSE
const bool check_integrate=false;

// Shall we do general sanity checks? Only when debugging SENSE
const bool sense_sanity = false;
```

```
// How many Tsense_updqunits in every queue? They are pretty small
// (~20 bytes),
// but we keep a queue for each node. At 8 nodes a updqlen of 100000 means
// 100000*20*8/1024=approx 15Kb. Conclusion: On a modern ma-
chine you might
// as well cranck it up to some 200.000 or a whole million
const int sense_updqlen = 75000;

// Check for overflow in memory, updq or dirq? (directory.h)
#define SENSE_check_q_overflow

#endif
```

C.3 sensememory.h, sensememory.cpp

```
/*
Functions in this file declares the shared memory, and the func-
tions for
reading and writing to shared memory.

Also defined here are specialized data types for stuff like ar-
rays etc.

Why no free()? Usually parallel programs do *not* free any mem-
ory during
execution, and it keeps the complexity down.
*/

//---------------------------------------------------------
-------------------
#ifndef sensememoryH
#define sensememoryH
//---------------------------------------------------------
-------------------

#define _REENTRANT
#include "portability.h"
#include "directory.h"
#include "Defaults.h"
#include "sense_connections.h"
#include "thread_support.h"

// The shared memory - exposed in order to make it possible to inline
// sense_read
extern Tsense_UC memory[sense_memsize];

// Initialize the connections
void mem_init_servsocks();
void mem_init_clisocks();
```

```
// Kill the connections
void mem_kill_sockets();

// sense_write writes a word to the shared memory
inline void sense_write(const Tsense_addr a, const Tsense_UC w)
{
  #ifdef SENSE_stats
  stats_writes++;
  #endif
  memory[a]=w; // Write to mem
  sense_dir.write(a); // Notify the directory
}


// sense_read reads a word from the shared memory
inline Tsense_UC sense_read(const Tsense_addr a)
{
  #ifdef SENSE_stats
  stats_reads++;
  #endif
  #ifndef SENSE_pure_update
  // If we use pure update ALL memory is up to date, and we do not need to
  // check the directory
  sense_dir.read(a); // Notify the directory - may cause a fault and fetch
  #endif
  return memory[a];
}

// Updates a block in memory from another node - called by di-
rectory read
void mem_retrieve_from(int, Tsense_addr, int);

// Writes a block directly to memory - called when using up-
date and hybrid
// protocol
inline void sense_write_raw(const Tsense_addr start, const int len,
Tsense_UC *buf)
{
  for (int n=0; n<len; n++)
    memory[n+start]=buf[n];
}

// Reads a block directly from memory - called when using up-
date and hybrid
// protocols
inline void sense_read_raw(const Tsense_addr start, const int len,
Tsense_UC *buf)
{
  for (int n=0; n<len; n++)
```

```
    buf[n]=memory[n+start];
}


// Only for debugging TSP
int sense_read_raw(const Tsense_addr);
/*
The base type Tsense_shared defines objects that can be al-
located in shared
memory. Using this you can define types like shared_array or shared_word.
*/

class Tsense_shared {
    private:
        Tsense_addr size;  // Size of object - may not be necessary
        Tsense_addr start; // Where is it allocated in shared mem
    public:
//        Tsense_shared() {/*setsize(1)*/;}
        // Getters and setters - may be overridden
        Tsense_addr getsize() const {return size;}
        void setsize(const Tsense_addr n) {size=n;};
        Tsense_addr getstart() const {return start;}
        void setstart(const Tsense_addr n) {start=n;}
        // Read and write to shared var
        Tsense_UC read() const {return sense_read(getstart());}
        void write(const Tsense_UC val) const {sense_write(getstart(), val);}
Tsense_shared& Tsense_shared::operator= (const Tsense_shared& s) {
  this->write(s.read());
  return *this;
}
    };



// sense_malloc allocates the necessary number of UC's
void sense_malloc(Tsense_shared &);

// This is a shared int, assumes that sizeof(sense_UC)== sizeof(int)
class Tsense_sint : public Tsense_shared {
 public:
  int read() const {
    return (int) sense_read(getstart());
  }
  void write(const int i) const {
    sense_write(getstart(), (Tsense_UC) i);
  }
  Tsense_sint() {setsize(1); sense_malloc(*this);}
  operator int() const {return read();}
  Tsense_sint(const int i) {write(i);}
  Tsense_sint& operator= (const int i) {write(i); return *this;}
```

```
  Tsense_sint& operator+= (const int i) {
    int tmp=this->read();
    tmp+=i;
    this->write(tmp);
    return *this;
  }
  Tsense_sint& operator++() {
    int tmp=this->read();
    this->write(tmp+1);
    return *this;
  }
  Tsense_sint& operator--() {
    int tmp=this->read();
    this->write(tmp-1);
    return *this;
  }
};

typedef union {
  struct {
    unsigned int a, b;
    } theInts;
    double d;
  } Tsense_convert2UC ;

// This is a shared double, assumes that sizeof(double) == 2*sizeof(sense_UC)
class Tsense_sdouble : public Tsense_shared {
    public:
        Tsense_sdouble() {setsize(2); sense_malloc(*this);}
        double read() const { Tsense_convert2UC tmp;
                        tmp.theInts.a = sense_read(getstart());
                        tmp.theInts.b = sense_read(getstart()+1);
                        return tmp.d;}
        void write(const double val) {
                        Tsense_convert2UC tmp;
                        sense_write(getstart(), tmp.theInts.a);
                        sense_write(getstart()+1, tmp.theInts.b);
                        }
  operator double() const {return read();}
  Tsense_sdouble(const double i) {write(i);}
  Tsense_sdouble& operator= (const double i) {write(i); re-
turn *this;}
  Tsense_sdouble& operator+= (const double i) {
    double tmp=this->read();
    tmp+=i;
    this->write(tmp);
    return *this;
  }
  Tsense_sdouble& operator++() {
    double tmp=this->read();
```

```
    this->write(tmp+1);
    return *this;
  }
  Tsense_sdouble& operator--() {
    double tmp=this->read();
    this->write(tmp-1);
    return *this;
  }
    };

// A shared vector of doubles
class Tsense_svecdouble : Tsense_shared {
 private:
  Tsense_convert2UC tmp;
 public:
  Tsense_svecdouble(int len) {setsize(2*len); sense_malloc(*this);}
  double read(const int pos) {
    tmp.theInts.a=sense_read(getstart()+pos);
    tmp.theInts.b = sense_read(getstart()+pos+1);
    return tmp.d;
  }
  void write(const int pos, const double val) {
    tmp.d=val;
    sense_write(getstart()+pos, tmp.theInts.a);
    sense_write(getstart()+pos+1, tmp.theInts.b);
  }
};

/*
Suggested extensions:
    * Array of (type) allocated by row or col
    * Cached vars, re-read after acquire and flushed at release
      (or read-request from other node)
*/

// Connections for memory requests
extern Tsense_servsock memreq_in[sense_nodes];
extern Tsense_clisock memreq_out[sense_nodes];
#endif

// Implementation of memory functions
#include "sensememory.h"
#include "nodeaddr.h"

#ifdef SENSE_debug
#include <iostream.h>
#endif

#ifdef SENSE_stats
#include "stats.h"
```

```
#endif

// The dir is external
extern Tsense_dir sense_dir;

// The connections defined in the header
Tsense_servsock memreq_in[sense_nodes];
Tsense_clisock memreq_out[sense_nodes];

// The shared memory
Tsense_UC memory[sense_memsize];

// Top of memory - starts at 0. Points to next available shared mem location
Tsense_addr memtop=0;

// We'll need some threads for accepting the connections
pthread_t Mtids[sense_nodes];

// The function used for the threads - very simple
void * memthread(void *ptr) {
  Tsense_servsock *sockptr = (Tsense_servsock *) ptr;
  sockptr->set_expedite();
  sockptr->accept();
  #ifdef SENSE_debug
  cout << "Memory socket ready to accept requests.\n";
  #endif
  do {
    int start= sockptr->recieve();
    if (start==-1) {
      #ifdef SENSE_debug
      cout << "MemSock: Killing myself...\n";
      #endif
      return (ptr);
    }
    int len= sockptr->recieve();
    #ifdef SENSE_debug
    cout << "Recieved request for memory bock at " << start;
    cout << " length " << len << ", complying!\n";
    #endif
    #ifdef SENSE_stats
    stats_msg_in++;
    stats_msg_out++;
    #endif
    sockptr->send((unsigned char *) &memory[start],
(int) (len*sizeof(Tsense_UC)));
    #ifdef SENSE_debug
    cout << "done\n";
    #endif
    // If we are using the hybrid protocol we have to tell the directory
    // that this memory block was just requested
```

```
    #ifdef SENSE_hybrid
    // Tell the dir we did it
    #endif
  } while (true);
}

// Updates a block in memory from another node - called by di-
rectory read
void mem_retrieve_from(int nodeID, Tsense_addr start, int len) {
  #ifdef SENSE_stats
  stats_read_faults++;
  stats_msg_in++;
  stats_msg_out++;
  #endif
  if (check_dirblock) {
    if (len<=0) {
      cerr << "SENSE error: trying to request memory range <=0 (len=";
      cerr << len << ")\n";
      exit(-1);
    }
  }
  memreq_out[nodeID].send((int) start);
  memreq_out[nodeID].send(len);
  memreq_out[nodeID].recieve((unsigned char *) &memory[start],
len*sizeof(Tsense_UC));
}

// Initializes the server sockets and creates threads wait-
ing for
// connections from clientsockets
void mem_init_servsocks() {
  spawn_server_connections(memreq_in, sense_nodes,
sense_reqmem_port_start, Mtids, sense_thisnode, memthread);

}

// Initializes the client sockets and joins with the threads created
// by mem_init_servsocks
void mem_init_clisocks() {

  // Init client end of lock connection between nodes
  for (int i=0; i<sense_nodes; i++)
    // Don't make a socket for this node
    if (i!=sense_thisnode)
      // Connect to the other nodes incoming_lock sockets with
      // our outgoing_lock
      memreq_out[i].init(sense_nodes_IP[i],
sense_reqmem_port_start+sense_thisnode);

}
```

```
void mem_kill_sockets() {
  for (int i=0; i<sense_nodes; i++) {
    if (i!=sense_thisnode)
      memreq_out[i].send(-1);
  }
}

// Allocate an object o in shared memory
void sense_malloc(Tsense_shared &o)
{
  o.setstart(memtop);
  memtop+=o.getsize();
  #ifdef SENSE_stats
  stats_mem_maxsize=memtop+1;
  #endif
  if (check_dirblock) {
    if (o.getsize()<=0) {
      cerr << "SENSE error (sense_malloc): Trying to allo-
cate <1 UC's\n";
    }
  }

#ifdef SENSE_check_q_overflow
  if (memtop>sense_memsize) {
    cerr << "Shared memory exhausted! Increase size of sense_memsize\n" ;
    exit(EXIT_FAILURE);
    }
#endif
}

// Only for debugging TSP
int sense_read_raw(const Tsense_addr start) {
  return (int) memory[start];
}
```

C.4 Lock.h, Lock.cpp

```
/*
Here we define the functions regarding locks

*/
//----------------------------------------------------------
--
#ifndef LockH
#define LockH
//----------------------------------------------------------
--

#define _REENTRANT
```

```
#include "sense_connections.h"
#include "Lamport.h"
#include "Defaults.h"
#include "directory.h"
#include "nodeaddr.h"
#include "thread_support.h"

// Acquire a lock
void sense_acq(int);

// Release a lock
void sense_rel(int);

// Arrive at barrier
void sense_barrier(int);

// Init the serversockets
void lock_init_servsocks();

// Init the client sockets
void lock_init_clisocks();

// Kill connected server sockets
void lock_kill_sockets();

// Connections for requesting and satisfying lock requestss
// One to the lockmanager
extern Tsense_clisock lockman_out;
// One from the lockmanager
extern Tsense_servsock lockman_in;
// One for each of the other nodes to forward the lock to
extern Tsense_servsock incoming_lock[sense_nodes];
// My end of above - to forward the lock and consistency info to
extern Tsense_clisock outgoing_lock[sense_nodes];

#endif


#include "Lock.h"
#include "sense_typeconv.h"
#include "thread_support.h"
#include <pthread.h>
#include "sensememory.h"

// Connections for requesting and satisfying lock requestss
// One to the lockmanager
Tsense_clisock lockman_out;
// One from the lockmanager
Tsense_servsock lockman_in;
// One for each of the other nodes to forward the lock to -
```

```
these are
// run in threads
Tsense_servsock incoming_lock[sense_nodes];
// My end of above - to forward the lock and consistency info to
Tsense_clisock outgoing_lock[sense_nodes];
// Which locks do I own?
bool locks_owned[sense_maxlocks];
// Which locks are locked?
bool locks_locked[sense_maxlocks];
// Who, if anyone, is waiting for the lock
int locks_pending[sense_maxlocks];
// TID's for threads created using accept_server_connections
pthread_t Ltids[sense_nodes+1];
// Mutex for synchronizing lock request
pthread_mutex_t lockacqmutex;
// Who owns the lock we are trying to acquire?
int lock_owner_ID;
// What is it's Ltime?
int lock_owner_ltime;
// Pointer to memory updates - used by thread in update protocol
Tsense_UC * acqmembuf=0;
int acqmembufsize=0;
// Mutex ensuring that lock handoff and acquire won't
// be initiated concurrently
pthread_mutex_t lock_sync_mutex;

#ifdef SENSE_stats
Tsense_timer locktimers[sense_maxlocks];
#endif


#ifdef SENSE_pure_invalidate
// Integrates the updq units in the sockets buffer into the di-
rectory, and
// reads the relevant areas of memory from the socket
void sense_lock_integrate(byte * &bufptr, int nbytes) {
  // Number of Tsense_updqunit's in buffer
  int nunits=nbytes/sizeof(Tsense_updqunit);
  // Make a copy of the buffer
  Tsense_updqunit * buf = (Tsense_updqunit *) bufptr;
  bufptr=0;
  // For each unit
  #ifdef SENSE_debug
  cout << "Integrate: memory before integrate, Ltime " << sense_ltime << ":\n";
  sense_dir.print();
  cout << "\nIntegrating these updates:\n";
  #endif
  for (int n=0; n<nunits ; n++) {
    // Insert into queues for other nodes
    #ifdef SENSE_debug
```

```
    buf[n].print();
    cout << "\n";
    #endif
    sense_upd.insert(buf[n]);
    sense_general_tomap(sense_dir.dir, buf[n].info);
  }
  #ifdef SENSE_debug
  cout << "\nIntegrate: memory after integrate\n";
  sense_dir.print();
  #endif
  // Remove the buffer of updates from memory
  delete [] buf;
}
#endif

#ifdef SENSE_pure_update
// Integrates the updq units in the sockets buffer into the di-
rectory, and
// reads the relevant areas of memory from the socket
// This version is for the pure update protocol
void sense_lock_integrate(byte * &updbuf, int updbufs,
      byte * membuf, int membufs) {
  #ifdef SENSE_debug
  cout << "Integrate: entering\n";
  #endif
  // Number of Tsense_updqunit's in buffer
  int nunits=updbufs/sizeof(Tsense_updqunit);
  // Make a copy of the pointer to the buffer just in case some synch is
  // off somewhere - else we risk getting the buffer over-
written by
  // some communication... This is not necessary, but I'm paranoid
  Tsense_updqunit * buf = (Tsense_updqunit *) updbuf;
  updbuf=0;
  int membufpos=0;
  #ifdef SENSE_debug
  cout << "Integrate: before loop - about to integrate " << nunits;
  cout << " dirunits\n";
  #endif
  //cout << "(1026) er " << sense_read_raw(1026) << "\n";
  // For each unit
  for (int n=0; n<nunits ; n++) {
    // Insert into queues for other nodes
    // Setting atnode is not necessary as we are using pure update
    // protocol, but better safe than sorry
    buf[n].info.setatnode(sense_thisnode);
    sense_upd.insert(buf[n]);
    /*
    if ((buf[n].info.getstart()>=1026) && (buf[n].info.getend()<=1026))
      cout << "Integrate: Block containing 1026 (no. " << n << ") is ";
    else
```

```
      cout << "Integrate: Block no " << n << " is:";
    buf[n].print();
    cout << "\n";
    cout << "Top of membuf (pos " << membuftop << ") is ";
    cout << (Tsense_UC) membuf[membufpos] << "\n";
    sense_trace_integrate=true;
    */
    sense_general_tomap_upd(sense_dir.dir, buf[n].info,
    (Tsense_UC *) membuf, membufpos);
    /*
    if ((buf[n].info.getstart()>=1026) && (buf[n].info.getend()<=1026)) {
      cout << "stopping - now it has gone wrong\n";
      cout << "(1026) er " << sense_read_raw(1026) << "\n";
      exit(-1);
    }
    */
  }
  if (check_integrate) {
    if ((membufpos*sizeof(Tsense_UC))!=membufs) {
      cerr << "SENSE error (integrate): Mismatch in membuf-
pos (";
      cerr << membufpos*sizeof(Tsense_UC) << ") and membufs (";
      cerr << membufs << ")\n";
      exit(-1);
    }
  }
  #ifdef SENSE_debug
  cout << "Integrate: After loop\n";
  #endif
  // Remove the buffer of updates from memory
  delete [] buf;
  // Remove the membuf from memory
  delete [] membuf;
  #ifdef SENSE_debug
  cout << "Integrate:leaving\n";
  #endif
}
#endif

#ifdef SENSE_hybrid
// Integrates the updq units in the sockets buffer into the di-
rectory, and
// reads the relevant areas of memory from the socket
// This version is for the pure update protocol
void sense_lock_integrate(byte * &updbuf, int updbufs,
      byte * membuf, int membufs) {
  #ifdef SENSE_debug
  cout << "Integrate: entering\n";
  #endif
  // Number of Tsense_updqunit's in buffer
```

```
  int nunits=updbufs/sizeof(Tsense_updqunit);
  // Make a copy of the pointer to the buffer just in case some synch is
  // off somewhere - else we risk getting the buffer over-
written by
  // some communication... This is not necessary, but I'm paranoid
  Tsense_updqunit * buf = (Tsense_updqunit *) updbuf;
  updbuf=0;
  int membufpos=0;
  // For each unit
  for (int n=0; n<nunits ; n++) {
    // Insert into queues for other nodes
    // Setting atnode is not necessary as we are using pure update
    // protocol, but better safe than sorry
    sense_upd.insert(buf[n]);
    if (buf[n].info.getrequested()>hybrid_treshold) {
      buf[n].info.setatnode(sense_thisnode);
      #ifdef SENSE_debug
      cout << "Integrate: integrating memory update\n";
      #endif
      sense_general_tomap_upd(sense_dir.dir, buf[n].info,
      (Tsense_UC *) membuf, membufpos);
    }
    else {
      #ifdef SENSE_debug
      cout << "Integrate: Integrating directory update\n";
      #endif
      sense_general_tomap(sense_dir.dir, buf[n].info);
    }
  }
  if (check_integrate) {
    if ((membufpos*sizeof(Tsense_UC))!=membufs) {
      cerr << "SENSE error (integrate): Mismatch in membuf-
pos (";
      cerr << membufpos*sizeof(Tsense_UC) << ") and membufs (";
      cerr << membufs << ")\n";
      exit(-1);
    }
  }

  // Remove the buffer of updates from memory
  delete [] buf;
  // Remove the membuf from memory
  delete [] membuf;
}
#endif

// The function for monitoring connections incoming_lock[]
// These updates show up after this node has done it's part of the
// lock handoff
void * lockthreadfun(void * ptr) {
```

```
  #ifdef SENSE_debug
  cout << "Lock server socket initializing.\n";
  #endif
  Tsense_servsock *sockptr = (Tsense_servsock *) ptr;
  sockptr->accept();
  #ifdef SENSE_debug
  cout << "Lock socket ready to accept requests.\n";
  #endif
  do {
    #ifdef SENSE_stats
    stats_msg_in++;
    #endif
    // Who is calling?
    lock_owner_ID=sockptr->recieve();
    if (lock_owner_ID==-1) {
      #ifdef SENSE_debug
      cout << "Killing myself now\n";
      #endif
      pthread_exit(ptr);
    }
    // Get the Ltime from the remote node...
    lock_owner_ltime=sockptr->recieve();
    #ifdef SENSE_debug
    cout << "LockReq: Got request to hand off lock from node ";
    cout << lock_owner_ID << ", who has Ltime ";
    cout << lock_owner_ltime << "\n";
    #endif
    // Then get the updates
    sockptr->getmsg();
    #ifdef SENSE_debug
    cout << "LockReq: Got " << sockptr->msgsize/sizeof(Tsense_updqunit);
    cout << " updates from node " << lock_owner_ID << " (that is ";
    cout << sockptr->msgsize << " bytes)\n";
    #endif
    // Integrate them in directory (and possibly memory, if up-
date protocol)
    #ifdef SENSE_pure_invalidate
    sense_lock_integrate(sockptr->bufptr, sockptr->msgsize);
    #endif
    #ifndef SENSE_pure_invalidate
    // If we use hybrid or update the memory updates mus be collected
    #ifdef SENSE_debug
    cout << "LockReq: Waiting for memory updates\n";
    #endif
    int nbytes=sockptr->recieve();
    #ifdef SENSE_debug
    cout << "LockReq: Expecting " << nbytes/sizeof(Tsense_UC)  <<
" updates\n";
    #endif
    byte * buf=0;
```

```
    if (nbytes>0) {
      buf = new byte[nbytes]; // memory is freed in integrate
      sockptr->recieve(buf, nbytes);
    }
    #ifdef SENSE_debug
    cout << "LockReq: Got the mem updates\n";
    #endif
    #endif
    #ifdef SENSE_pure_update
    sense_lock_integrate(sockptr->bufptr, sockptr->msgsize, buf, nbytes);
    #endif
    #ifdef SENSE_hybrid
    sense_lock_integrate(sockptr->bufptr, sockptr->msgsize, buf, nbytes);
    #endif
    // Now release the mutex so the main acquire lock func-
tion can continue
    pthread_mutex_unlock(&lockacqmutex);
  } while (true);
}

#ifdef SENSE_pure_invalidate
// Handles lock handoff to another node
// Mutex dirlock MUST be taken before this is done, as we mod-
ify the
// directory
void lock_handoff(int nodeID, int lockID){
  #ifdef SENSE_stats
  stats_msg_out++;
  stats_msg_in++;
  #endif
  #ifdef SENSE_debug
  cout << "Handoff: Preparing to handoff lock " << lockID
<< " to node " << nodeID;
  cout << "\n";
  #endif
  outgoing_lock[nodeID].send(sense_thisnode); // Tell other node who we are
  outgoing_lock[nodeID].send(sense_ltime); // This is out Ltime
  // Compress the updates queue
  sense_upd.compress(nodeID);
  int qlen=sense_upd.lists[nodeID].size();
  #ifdef SENSE_debug
  cout << "Handoff: Ready to send " << qlen << " updates to node ";
  cout << nodeID;
  cout << " (that is " << sizeof(Tsense_updqunit)*qlen << " bytes)\n";
  #endif
  outgoing_lock[nodeID].sendmsg((byte *) sense_upd.lists[nodeID].data,
sizeof(Tsense_updqunit)*qlen);
  #ifdef SENSE_debug
  cout << "Handoff: Lock and consistency info sent, waiting for new ";
  cout << "Ltime and updates\n";
```

```
  #endif
  sense_ltime=outgoing_lock[nodeID].recieve();
  #ifdef SENSE_debug
  cout << "Handoff: Got new Ltime (" << sense_ltime << ")\n";
  #endif
  outgoing_lock[nodeID].getmsg();
  int tmp=outgoing_lock[nodeID].msgsize/sizeof(Tsense_updqunit);
  #ifdef SENSE_debug
  cout << "Handoff: Got " << tmp << " updates from node " << nodeID;
  cout << " (that is " << outgoing_lock[nodeID].msgsize << " bytes)\n";
  #endif
  sense_lock_integrate(outgoing_lock[nodeID].bufptr,
outgoing_lock[nodeID].msgsize);
  // Now mark this lock as owned by the other node
  locks_owned[lockID]=false;
  #ifdef SENSE_debug
  cout << "Handoff: leaving\n";
  #endif
}
#endif

#ifdef SENSE_pure_update
// Handles lock handoff to another node
// Mutex dirlock MUST be taken before this is done, as we mod-
ify the
// directory
void lock_handoff(int nodeID, int lockID){
  #ifdef SENSE_stats
  stats_msg_out++;
  stats_msg_in++;
  #endif
  #ifdef SENSE_debug
  cout << "Handoff: Preparing to handoff lock " << lockID
<< " to node " << nodeID;
  cout << "\n";
  #endif
  outgoing_lock[nodeID].send(sense_thisnode); // Tell other node who we are
  outgoing_lock[nodeID].send(sense_ltime); // This is out Ltime
  // Compress the updates queue
  sense_upd.compress(nodeID);
  int qlen=sense_upd.lists[nodeID].size();
  int nUCs=0;
  for (int x=0; x<qlen; x++)
    nUCs+=sense_upd.lists[nodeID].data[x].info.getlen();
  #ifdef SENSE_debug
  cout << "Collecting " << nUCs << " memory updates in buffer\n";
  #endif
  Tsense_UC * membuf = new Tsense_UC[nUCs];
  int bufpos=0;
  for (int x=0; x<qlen; x++) {
```

```
    Tsense_dirunit tmp = sense_upd.lists[nodeID].data[x].info;
    sense_read_raw(tmp.getstart(), tmp.getlen(), &membuf[bufpos]);
    bufpos+=tmp.getlen();
  }
  #ifdef SENSE_debug
  cout << "Done collecting memory updates\n";
  cout << "Handoff: Ready to send " << qlen << " updates to node ";
  cout << nodeID;
  cout << " (that is " << sizeof(Tsense_updqunit)*qlen << " bytes)\n";
  #endif
  outgoing_lock[nodeID].sendmsg((byte *) sense_upd.lists[nodeID].data,
sizeof(Tsense_updqunit)*qlen);
  #ifdef SENSE_debug
  cout << "Handoff: done\n";
  cout << "Handoff: Sending memory updates\n";
  #endif
  int nbytes=nUCs*sizeof(Tsense_UC);
  outgoing_lock[nodeID].send(nbytes);
  if (nbytes>0)
    outgoing_lock[nodeID].send((byte *) membuf, nbytes);
  #ifdef SENSE_debug
  cout << "Handoff: done.\n";
  #endif
  delete [] membuf;
  #ifdef SENSE_debug
  cout << "Handoff: Lock, consistency info and mem updates sent, wait-
ing ";
  cout << "for new Ltime and dir+mem updates\n";
  #endif
  sense_ltime=outgoing_lock[nodeID].recieve();
  #ifdef SENSE_debug
  cout << "Handoff: Got new Ltime (" << sense_ltime << ")\n";
  #endif
  outgoing_lock[nodeID].getmsg();
  int tmp=outgoing_lock[nodeID].msgsize/sizeof(Tsense_updqunit);
  #ifdef SENSE_debug
  cout << "Handoff: Got " << tmp << " updates from node " << nodeID;
  cout << " (that is " << outgoing_lock[nodeID].msgsize << " bytes)\n";
  #endif

  // Now get the memory updates
  int nmembytes=outgoing_lock[nodeID].recieve();
  byte * meminbuf = 0;
  if (nmembytes>0) {
    meminbuf = new byte[nmembytes]; // This memory is freed by integrate
    outgoing_lock[nodeID].recieve(meminbuf, nmembytes);
  }
  #ifdef SENSE_debug
  cout << "Got " << nmembytes/sizeof(Tsense_UC) << " UC's of mem-
ory updates\n";
```

```
  #endif
  sense_lock_integrate(outgoing_lock[nodeID].bufptr,
       outgoing_lock[nodeID].msgsize,
       meminbuf, nmembytes);
  // Now mark this lock as owned by the other node
  locks_owned[lockID]=false;
}
#endif

#ifdef SENSE_hybrid
// Handles lock handoff to another node
// Mutex dirlock MUST be taken before this is done, as we mod-
ify the
// directory
void lock_handoff(int nodeID, int lockID){
  #ifdef SENSE_stats
  stats_msg_out++;
  stats_msg_in++;
  #endif
  #ifdef SENSE_debug
  cout << "Handoff: Preparing to handoff lock " << lockID
<< " to node " << nodeID;
  cout << "\n";
  #endif
  outgoing_lock[nodeID].send(sense_thisnode); // Tell other node who we are
  outgoing_lock[nodeID].send(sense_ltime); // This is out Ltime
  // Compress the updates queue
  sense_upd.compress(nodeID);
  int qlen=sense_upd.lists[nodeID].size();
  int nUCs=0;
  for (int x=0; x<qlen; x++) {
    Tsense_dirunit tmp=sense_upd.lists[nodeID].data[x].info;
    if (tmp.getrequested()>hybrid_treshold) {
      nUCs+=tmp.getlen();
    }
  }
  #ifdef SENSE_debug
  cout << "Collecting " << nUCs << " memory updates in buffer\n";
  #endif
  Tsense_UC * membuf = new Tsense_UC[nUCs];
  int bufpos=0;
  for (int x=0; x<qlen; x++) {
    Tsense_dirunit tmp = sense_upd.lists[nodeID].data[x].info;
    if (tmp.getrequested()>hybrid_treshold) {
      sense_read_raw(tmp.getstart(), tmp.getlen(), &membuf[bufpos]);
      bufpos+=tmp.getlen();
      // When they get it, let them know they've got it :)
      // NB - this is taken care of at reciever too
      sense_upd.lists[nodeID].data[x].info.setatnode(nodeID);
    }
```

```
  }
  #ifdef SENSE_debug
  cout << "Done collecting memory updates\n";

  cout << "Handoff: Ready to send " << qlen << " updates to node ";
  cout << nodeID;
  cout << " (that is " << sizeof(Tsense_updqunit)*qlen << " bytes)\n";
  #endif
  outgoing_lock[nodeID].sendmsg((byte *) sense_upd.lists[nodeID].data,
sizeof(Tsense_updqunit)*qlen);
  #ifdef SENSE_debug
  cout << "Handoff: done\n";
  cout << "Handoff: Sending " << nUCs << " UC's of memory updates\n";
  #endif
  int nbytes=nUCs*sizeof(Tsense_UC);
  outgoing_lock[nodeID].send(nbytes);
  if (nbytes>0)
    outgoing_lock[nodeID].send((byte *) membuf, nbytes);
  #ifdef SENSE_debug
  cout << "Handoff: done.\n";
  #endif
  delete [] membuf;
  #ifdef SENSE_debug
  cout << "Handoff: Lock, consistency info and mem updates sent, wait-
ing ";
  cout << "for new Ltime and dir+mem updates\n";
  #endif
  sense_ltime=outgoing_lock[nodeID].recieve();
  #ifdef SENSE_debug
  cout << "Handoff: Got new Ltime (" << sense_ltime << ")\n";
  #endif
  outgoing_lock[nodeID].getmsg();
  int tmp=outgoing_lock[nodeID].msgsize/sizeof(Tsense_updqunit);
  #ifdef SENSE_debug
  cout << "Handoff: Got " << tmp << " updates from node " << nodeID;
  cout << " (that is " << outgoing_lock[nodeID].msgsize << " bytes)\n";
  #endif

  // Now get the memory updates
  int nmembytes=outgoing_lock[nodeID].recieve();
  byte * meminbuf = 0;
  if (nmembytes>0) {
    meminbuf = new byte[nmembytes]; // This memory is freed by integrate
    outgoing_lock[nodeID].recieve(meminbuf, nmembytes);
  }
  #ifdef SENSE_debug
  cout << "Got " << nmembytes/sizeof(Tsense_UC) <<
" UC's of memory updates\n";
  #endif
  sense_lock_integrate(outgoing_lock[nodeID].bufptr,
```

```
        outgoing_lock[nodeID].msgsize,
        meminbuf, nmembytes);
  // Now mark this lock as owned by the other node
  locks_owned[lockID]=false;
}
#endif

// This thread waits for a forwarded lock request on the lock-
man_in socket
void * threadfunlockman_in(void * p) {
  // Get the connection up and running
  #ifdef SENSE_debug
  cout << "Thread monitoring forwarded lock requests up.\n";
  #endif
  Tsense_servsock * sockptr = (Tsense_servsock *) p;
  sockptr->accept();
  do {
    #ifdef SENSE_stats
    stats_msg_in++;
    #endif
    // We are blocked here, waiting for a lock request
    int who=sockptr->recieve();
    // Grab mutex ensuring no concurrent handoff and acq
    pthread_mutex_lock(&lock_sync_mutex);
    if (who==-1) {
      #ifdef SENSE_debug
      cout << "LockIncomingReq: Killing myself now.\n";
      #endif
      pthread_exit(p);
    }
    #ifdef SENSE_stats
    stats_lock_req_in++;
    #endif
    int which=sockptr->recieve();
    #ifdef SENSE_debug
    cout << "LockIncomingReq: lockman sent request for lock ";
    cout << which << " from node " << who << "\n";
    #endif
    // Avoid data race on lock handeling by using the main mutex
    pthread_mutex_lock(&dirlock);
    if (locks_locked[which]) {
      // Let this be handled later by release
      #ifdef SENSE_debug
      cout << "Lock is currently taken, marking a pending request.\n";
      #endif
      locks_pending[which]=who;
      #ifdef SENSE_stats
      stats_req_while_taken++;
      #endif
      // This is tricky! Here we block, while waiting for the main thread
```

```
      // of execution to reach the point where the lock is released
      /*
      cout << "LockIncomingReq: waiting for goahead\n";
      int dummy=sockptr.recieve();
      cout << "LockIncomingReq: got goahead, lock handeling done\n";
      */
    }
    else {
      sense_dir.wflush();
      #ifdef SENSE_debug
      cout << "LockIncomingReq: doing handoff now\n";
      #endif
      #ifdef SENSE_stats
      Tsense_timer handofftimer;
      handofftimer.start();
      #endif
      lock_handoff(who, which);
      // This is our new Ltime - decided by remote who now holds the lock
      #ifdef SENSE_debug
      cout << "LockIncomingReq: new Ltime computed by remote is ";
      cout << sense_ltime << "\n";
      #endif
      #ifdef SENSE_stats
      handofftimer.end();
      stats_lock_handofftime+=handofftimer.elapsed();
      #endif
      /*
      cout << "LockIncomingReq: waiting for goahead\n";
      int dummy=sockptr.recieve();
      cout << "LockIncomingReq: got goahead, lock handeling done\n";
      */
      sense_dir.wprime();
      sense_dir.rprime();
    }
    pthread_mutex_unlock(&dirlock);
    pthread_mutex_unlock(&lock_sync_mutex);
  } while (true);
}

// Initializes the server sockets and creates threads wait-
ing for
// connections from clientsockets
void lock_init_servsocks() {
  // Init the array of locks owned - might as well do it here
  for (int i=0; i<sense_maxlocks; i++) {
    locks_locked[i]=false; // No locks are locked at first
    locks_owned[i]=(sense_thisnode==0); // All locks start out owned by node 0
    locks_pending[i]=-1; // Noone is waiting to get at the locks now
  }
  #ifdef SENSE_debug
```

```
  cout << "Lock data structures initialized.\n";
  #endif
  spawn_server_connections(incoming_lock, sense_nodes,
sense_node_lock_port_start, Ltids, sense_thisnode, lockthreadfun);
  #ifdef SENSE_debug
  cout << "Server sockets for requesting locks are up.\n";
  #endif
  spawn_server_connections(&lockman_in, 1, sense_lockman_forward_port,
&Ltids[sense_nodes], -1, threadfunlockman_in);
  #ifdef SENSE_debug
  cout << "Servsock for recieving forwarded lock request from lock-
manager ";
  cout << "is up on port " << sense_lockman_forward_port << ".\n";
  #endif
}

// Initializes the client sockets and joins with the threads created
// by lock_init_servsocks
void lock_init_clisocks() {
  // Init client part of socket for contacting the lockmanager
  #ifdef SENSE_debug
  cout << "Connecting client socket to lockmanager at port " ;
  cout << sense_lockman_port_start+sense_thisnode << "...";
  #endif
  lockman_out.init(sense_lockman_IP, sense_lockman_port_start+sense_thisnode);
  #ifdef SENSE_debug
  cout << "Done\n";
  #endif
  // Init client end of lock connection between nodes
  #ifdef SENSE_debug
  cout << "Connecting client sockets for lock requests\n";
  #endif
  for (int i=0; i<sense_nodes; i++)
    // Don't make a socket for this node
    if (i!=sense_thisnode) {
      #ifdef SENSE_debug
      cout << "Trying at IP " << sense_nodes_IP[i] << ", port ";
      cout << sense_node_lock_port_start+sense_thisnode << "... ";
      #endif
      // Connect to other nodes incoming_lock sockets with our outgoing_lock
      outgoing_lock[i].init(sense_nodes_IP[i],
sense_node_lock_port_start+sense_thisnode);
      #ifdef SENSE_debug
      cout << "done\n";
      #endif
    }
}

// Closes all client sockets and tells the servers on the other end to die
void lock_kill_sockets() {
```

```
  lockman_out.send(-1);
  for (int i=0; i<sense_nodes; i++) {
    if (i!=sense_thisnode)
      outgoing_lock[i].send(-1);
  }
}

// Acquire a lock
void sense_acq_invhyb(const int l)
{
  pthread_mutex_lock(&lock_sync_mutex);
  #ifdef SENSE_stats
  stats_locks_taken++;
  locktimers[l].start(); // Init timer to collect time held
  Tsense_timer acqtime;
  acqtime.start();
  #endif
  // Make sure a lock request won't acces the directory while we han-
dle this
  #ifdef SENSE_debug
  cout << "Acq: getting dirlock mutex\n";
  #endif
  pthread_mutex_lock(&dirlock);
  #ifdef SENSE_debug
  cout << "Acq: got the mutex\n";
  #endif
  // Flush the current write unit
  #ifdef SENSE_debug
  cout << "Acq: doing wflush\n";
  #endif
  sense_dir.wflush();
  #ifdef SENSE_debug
  cout << "Acq: finished wflush\n";
  #endif
  // Do we own the lock?
  if (locks_owned[l]) {
    // In that case just mark it as locked
    #ifdef SENSE_debug
    cout << "Acq:Lock is owned by this node, just grabbing the lock\n";
    #endif
    #ifdef SENSE_stats
    stats_locks_local++;
    #endif
    if (!locks_locked[l])
      locks_locked[l]=true;
    else
      if (locks_strict)
cerr << "SENSE error: Trying to lock lock " << l << " twice!\n";
  }
  else {
```

```
    #ifdef SENSE_stats
    stats_msg_out+=3; // Lockman req and ack, updates to old owner
    #endif
    // Lock the lock-mutex in order to let the thread free it, when all info
    // caused by the following requests is integrated
    #ifdef SENSE_debug
    cout << "Acq: getting lockacqmutex\n";
    #endif
    pthread_mutex_lock(&lockacqmutex); // We lock it to let the thread free it
    // Tell lockman who we are, and which lock we want
    #ifdef SENSE_debug
    cout << "Acq: requesting lock " << l << " from lockman\n";
    #endif
    lockman_out.send(sense_thisnode);
    lockman_out.send(l);
    // Wait until the owner has contacted us on one of the wait-
ing threads
    // This is *not* a bug - we just have to wait until one of the threads
    // free the mutex again
    // This mutex makes sure we do not proceed before the owner has finished
    // sending its updates to this node
    pthread_mutex_lock(&lockacqmutex);
    // Compute new Ltime
    if (lock_owner_ltime>sense_ltime)
      sense_ltime=lock_owner_ltime+1;
    else
      sense_ltime++;
    #ifdef SENSE_debug
    cout << "Acq: returning new Ltime (" << sense_ltime << ")\n";
    #endif
    //    incoming_lock[lock_owner_ID].send(sense_thisnode);
    // Send it to the (now previous) owner
    incoming_lock[lock_owner_ID].send(sense_ltime);
    // Compress update queues
    sense_upd.compress(lock_owner_ID);
    // Send dir updates to (now previous) owner
    int qlen=sense_upd.lists[lock_owner_ID].size();
    incoming_lock[lock_owner_ID].sendmsg(
(byte*) sense_upd.lists[lock_owner_ID].data,
qlen*sizeof(Tsense_updqunit));
    // Remember to send the memory updates if we are using up-
date or
    // hybrid protocols

    #ifdef SENSE_pure_update
    // This is to be moved to a thread later for performance
    int nUCs=0;
    Tsense_UC * buf=0;
    for (int n=0; n<qlen ; n++)
      nUCs+=sense_upd.lists[lock_owner_ID].data[n].info.getlen(); // Phew...
```

```
    if (nUCs>0)
      buf = new Tsense_UC[nUCs];
    int bufpos=0;
    for (int n=0; n<qlen; n++) {
      Tsense_dirunit tmp=sense_upd.lists[lock_owner_ID].data[n].info;
      sense_read_raw(tmp.getstart(), tmp.getlen(), &buf[bufpos]);
      bufpos+=tmp.getlen();
    }
    if (check_integrate) {
      if (bufpos!=nUCs) {
cerr << "SENSE error (sense_acq): mismatch in size of buf-
pos (";
cerr << bufpos << ") and nUCs (" << nUCs << ")\n";
exit(-1);
      }
    }
    #ifdef SENSE_debug
    cout << "Done collecting memory updates\n";
    #endif
    int nbytes=nUCs*sizeof(Tsense_UC);
    incoming_lock[lock_owner_ID].send(nbytes);
    if (nbytes>0)
      incoming_lock[lock_owner_ID].send((byte *) buf, nbytes);
    delete [] buf;
    #endif
    #ifdef SENSE_hybrid
    // This is basicly the same as for update, but with a few more if's
    // This is to be moved to a thread later for performance
    int nUCs=0;
    Tsense_UC * buf=0;
    for (int n=0; n<qlen ; n++) {
      Tsense_dirunit tmp=sense_upd.lists[lock_owner_ID].data[n].info;
      if (tmp.getrequested()!=
sense_upd.lists[lock_owner_ID].data[n].info.getrequested()) {
cerr << "SENSE error: Wrong result of getrequested() in acq\n";
exit (-1);
      }
      if (tmp.getrequested()>hybrid_treshold)
nUCs+=tmp.getlen();
    }
    if (nUCs>0)
      buf = new Tsense_UC[nUCs];
    int bufpos=0;
    for (int n=0; n<qlen; n++) {
      Tsense_dirunit tmp=sense_upd.lists[lock_owner_ID].data[n].info;
      if (tmp.getrequested()>hybrid_treshold) {
sense_read_raw(tmp.getstart(), tmp.getlen(), &buf[bufpos]);
bufpos+=tmp.getlen();
      }
    }
```

```
    if (check_integrate) {
      if (bufpos!=nUCs) {
cerr << "SENSE error (sense_acq): mismatch in size of buf-
pos (";
cerr << bufpos << ") and nUCs (" << nUCs << ")\n";
exit(-1);
      }
    }
    #ifdef SENSE_debug
    cout << "Done collecting memory updates\n";
    #endif
    int nbytes=nUCs*sizeof(Tsense_UC);
    incoming_lock[lock_owner_ID].send(nbytes);
    if (nbytes>0)
      incoming_lock[lock_owner_ID].send((byte *) buf, nbytes);
    delete [] buf;
    #endif
    // Empty the queue of updates waiting for this node
    sense_upd.lists[lock_owner_ID].empty();
    // Note that this node has the lock, and that it's taken
    locks_owned[l]=true;
    locks_locked[l]=true;
    // send acqhandoff to lock manager
    lockman_out.send(0);
    // Free the locksync mutex
    pthread_mutex_unlock(&lockacqmutex);
  }
// after sync
sense_dir.wprime();
sense_dir.rprime();
// Free the main mutex
 #ifdef SENSE_debug
 cout << "Acq: Freeing dirlock\n";
 #endif
 pthread_mutex_unlock(&dirlock);
 pthread_mutex_unlock(&lock_sync_mutex);
 #ifdef SENSE_debug
 cout << "Acq: Done - leaving\n";
 #endif
 #ifdef SENSE_stats
 // Collect acqtime for statistics
 acqtime.end();
 stats_lock_acqtime+=acqtime.elapsed();
 #endif
}

// Used for constructing the buffer of memory updates while trans-
mitting the
// directory information. Sets global var acqmembuf to ad-
dress of buffer. This
```

```
// memory is freed later by sense_acq_upd. This can NOT be called before
// the updqueue for the target node has been compressed!!!!
void * acq_upd_threadfun(void * p) {
  int node=(*(int*)p);
  acqmembuf=0;
  int qlen=sense_upd.lists[node].size();
  for (int n=0; n<qlen ; n++)
    acqmembufsize+=sense_upd.lists[node].data[n].info.getlen(); // Phew...
  if (acqmembufsize>0)
    acqmembuf = new Tsense_UC[acqmembufsize];
  int bufpos=0;
  for (int n=0; n<qlen; n++) {
    Tsense_dirunit tmp=sense_upd.lists[node].data[n].info;
    sense_read_raw(tmp.getstart(), tmp.getlen(), &acqmembuf[bufpos]);
    bufpos+=tmp.getlen();
  }
  if (check_integrate) {
    if (bufpos!=acqmembufsize) {
      cerr << "SENSE error (sense_acq): mismatch in size of buf-
pos (";
      cerr << bufpos << ") and nUCs (" << acqmembufsize << ")\n";
      exit(-1);
    }
  }
}

// Acquire a lock
void sense_acq_upd(const int l)
{
  #ifdef SENSE_stats
  stats_locks_taken++;
  #endif
  // Make sure a lock request won't acces the directory while we han-
dle this
  pthread_mutex_lock(&dirlock);
  // Flush the current write unit
  sense_dir.wflush();
  // Do we own the lock?
  if (locks_owned[l]) {
    // In that case just mark it as locked
    #ifdef SENSE_stats
    stats_locks_local++;
    #endif
    if (!locks_locked[l])
      locks_locked[l]=true;
    else
      if (locks_strict)
cerr << "SENSE error: Trying to lock lock " << l << " twice!\n";
  }
  else {
```

```
    #ifdef SENSE_stats
    stats_msg_out+=3; // Lockman req and ack, updates to old owner
    #endif
    // Lock the lock-mutex in order to let the thread free it, when all info
    // caused by the following requests is integrated
    pthread_mutex_lock(&lockacqmutex); // We lock it to let the thread free it
    // Tell lockman who we are, and which lock we want
    lockman_out.send(sense_thisnode);
    lockman_out.send(l);
    // Wait until the owner has contacted us on one of the wait-
ing threads
    // This is *not* a bug - we just have to wait until one of the threads
    // free the mutex again
    // This mutex makes sure we do not proceed before the owner has finished
    // sending its updates to this node
    pthread_mutex_lock(&lockacqmutex);
    // Compute new Ltime
    if (lock_owner_ltime>sense_ltime)
      sense_ltime=lock_owner_ltime+1;
    else
      sense_ltime++;
    // Send it to the (now previous) owner
    incoming_lock[lock_owner_ID].send(sense_ltime);
    // Compress update queues
    sense_upd.compress(lock_owner_ID);
    // Spin off a thread for gathering the memory updates
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    pthread_t tid;
    pthread_create(&tid, &attr, acq_upd_threadfun, (void *) &lock_owner_ID);
    // While it's working, xmit the directory updates
    int qlen=sense_upd.lists[lock_owner_ID].size();
    // Send dir updates to (now previous) owner
    incoming_lock[lock_owner_ID].sendmsg(
(byte*) sense_upd.lists[lock_owner_ID].data,
qlen*sizeof(Tsense_updqunit));
    // Now join with the thread gathering the updates
    pthread_join(tid,0);
    cout << "acqupd: thread is done, nupdates is " << acqmem-
bufsize << "\n";
    // Cool, now we can xmit the updates
    int nbytes=acqmembufsize*sizeof(Tsense_UC);
    incoming_lock[lock_owner_ID].send(nbytes);
    if (nbytes>0)
      incoming_lock[lock_owner_ID].send((byte *) acqmembuf, nbytes);
    // Free the memory allocated for updates
    delete [] acqmembuf; acqmembuf=0;
    // Empty the queue of updates waiting for this node
    sense_upd.lists[lock_owner_ID].empty();
```

```
    // Note that this node has the lock, and that it's taken
    locks_owned[l]=true;
    locks_locked[l]=true;
    // send acqhandoff to lock manager
    lockman_out.send(0);
    // Free the locksync mutex
    pthread_mutex_unlock(&lockacqmutex);
  }
  // after sync
  sense_dir.wprime();
  sense_dir.rprime();
  // Free the main mutex
  pthread_mutex_unlock(&dirlock);
}

void sense_acq(const int l) {
  /*
#ifdef SENSE_pure_update
sense_acq_upd(l);
#endif
#ifndef SENSE_pure_update
  */
sense_acq_invhyb(l);
//#endif
}

// Release a lock
void sense_rel(int l)
{
  #ifdef SENSE_stats
  Tsense_timer reltimer;
  reltimer.start();
  #endif
  // Lock the dir mutex - we do NOT want to be interrupted here
  pthread_mutex_lock(&dirlock);
  // Flush the write unit
  sense_dir.wflush();
  // If locks_strict, test if we release a lock we own and have locked
  if (locks_strict) {
    if (!(locks_locked[l] && locks_owned[l])) {
      cerr << "SENSE error: Trying to free lock " << l << " which is not ";
      cerr << "locked or owned!\n";
    }
  }
  // Is there a pending request?
  if (locks_pending[l]!=-1) {
    pthread_mutex_lock(&lock_sync_mutex);
    #ifdef SENSE_debug
    cout << "Rel: node " << locks_pending[l] << " is wait-
ing for lock " << l;
```

```
    cout << ", handing it off\n";
    #endif
    #ifdef SENSE_stats
    Tsense_timer handofftimer;
    handofftimer.start();
    #endif
    lock_handoff(locks_pending[l], l); // Handoff calculates the stats
    locks_pending[l]=-1;
    #ifdef SENSE_stats
    // Collect timing data on handoff
    handofftimer.end();
    stats_lock_handofftime+=handofftimer.elapsed();
    #endif
    pthread_mutex_unlock(&lock_sync_mutex);
  }
  //    else
  //      sense_ltime++; // Is this necessary?
  locks_locked[l]=false;
  // after sync
  sense_dir.wprime();
  sense_dir.rprime();
  #ifdef SENSE_stats
  reltimer.end();
  locktimers[l].end();
  stats_lock_heldtime+=locktimers[l].elapsed();
  stats_lock_reltime+=reltimer.elapsed();
  #endif
  pthread_mutex_unlock(&dirlock);
}
```

C.5 directory.h, directory.cpp

```
/*
Defines the structures regarding the directory for the shared memory.
*/

//----------------------------------------------------------
-------------------
#ifndef directoryH
#define directoryH
//----------------------------------------------------------
-------------------

#define _REENTRANT
#include <map>
#include <queue>
#include "Defaults.h"
#include "portability.h"
#include <iostream.h>
#include <pthread.h>
```

```
#include "sense_timer.h"

#ifdef SENSE_stats
#include "stats.h"
#endif
extern void mem_retrieve_from(int, Tsense_addr, int);


/*
This is the info unit for a block in shared memory. It de-
fines the size of
the block, it's lamport time for last update, and - possi-
bly - heuristic
info regarding mobility.
*/

class Tsense_dirunit {
    private:
        Tsense_addr start; // Start of shared block
        Tsense_addr end;   // End of shared block
        int ltime; // Lamport time stamp of this block
        short int atnode; // If out of memory, where was it last?
    public:
#ifdef SENSE_hybrid
int requested; // Times the block was requested by other nodes
Tsense_dirunit() {requested=0;}
#endif
        // setters and getters
        void setstart(Tsense_addr n) {start=n;}
        Tsense_addr getstart() const {return start;}
        void setend(const Tsense_addr n) {end=n;}
        Tsense_addr getend() const {return end;}
        void setltime(const int n) {ltime=n;}
        int getltime() const {return ltime;}
int getlen() const {return (end-start)+1;}
        void print() const {
  cout << "Start:" << start << " End:" << end << " Ltime:" << ltime;
#ifdef SENSE_hybrid
  cout << " Requested: " << requested;
#endif
  cout << " At node:" << atnode;
}
bool islocal() {return (atnode==sense_thisnode);}
        void setatnode(const int n) {atnode=n;}
        int getatnode() const {return atnode;}
#ifdef SENSE_hybrid
int getrequested() const {return requested;}
// Special - we want to keep this high
void setrequested(const int n) {
  if (n>requested)
```

```
    requested=n;
}
#endif
// Copy other info (ltime, atnode etc.) from a dirunit
void dupID(const Tsense_dirunit src) {
  setltime(src.getltime());
  setatnode(src.getatnode());
#ifdef SENSE_hybrid
  setrequested(src.getrequested());
#endif
}
    };

/*
This is the info unit for the list of waiting updates for other nodes
*/
class Tsense_updqunit {
    public:
        Tsense_dirunit info; // Contains the dirblock
        bool copyset[sense_nodes]; // Who has seen this update?
        // Setters and getters
        void setinfo(const Tsense_dirunit u) {info=u;}
        Tsense_dirunit getinfo() const {return info;}
        void getcopyset(bool c[sense_nodes]) const
          {for (int n=0; n<sense_nodes; n++) {
             c[n]=copyset[n];
             }
          }
        void setcopyset(const bool c[sense_nodes])
          {for (int n=0; n<sense_nodes ; n++)
             copyset[n]=c[n];
          }
        // constructors
        // General constructor - just in case
        Tsense_updqunit() {
          for (int n=0; n<sense_nodes; n++)
            {copyset[n]=false;}
          copyset[sense_thisnode]=true;}
        // Specific constructor
        Tsense_updqunit(const Tsense_dirunit u) {
          for (int n=0; n<sense_nodes; n++)
            {copyset[n]=false;}
          copyset[sense_thisnode]=true;
          setinfo(u);}
        // Other functions;
        Tsense_updqunit(const Tsense_dirunit u,
const bool cset[sense_nodes]) {
            setinfo(u);
            for (int n=0; n<sense_nodes; n++) {
              copyset[n]=cset[n];
```

```
              }
            }
        void setseen_by(const int n) {copyset[n]=true;}
        bool getseen_by(const int n) const {return copyset[n];}

void print() const {
  cout << "copyset: ";
  for (int n=0; n<sense_nodes; n++) {
    if (copyset[n])
      cout << "T ";
    else
      cout << "F ";
  }
  info.print();
}

    };

// This is used for update queues instead of the STL queue, as the us-
age of
// this saves us the copying at sync points
class Tsense_fastupdq {
 public:
  Tsense_updqunit data[sense_updqlen];
  int tail;
  Tsense_fastupdq() { tail=-1;}
  void push(const Tsense_updqunit u) {
#ifdef SENSE_check_q_overflow
    if (tail==sense_updqlen-1) {
      cerr << "SENSE error: overflow in Tsense_fastupdq (" << tail+1;
      cerr << " elements) - increase size of sense_updqlen\n";
      exit(-1);
    }
#endif
    if (sense_sanity) {
      if (u.info.getlen()<1) {
cerr << "SENSE error: Attempt to insert unit with len<1 (";
cerr << u.info.getlen() << ") in fastupdq\n";
      }
    }
    tail++;
    data[tail]=u;
    #ifdef SENSE_stats
    if (tail>stats_updq_maxsize)
      stats_updq_maxsize=tail;
    #endif
  }
  void pop() {
#ifdef SENSE_check_q_overflow
    // OK, so it's underflow...
```

```
    if (tail<0) {
      cerr << "SENSE error: underflow in Tsense_fastupdq -
trying to pop an";
      cerr << " empty queue\n";
      exit(-1);
    }
#endif
    tail--;
  }
  Tsense_updqunit front() const {
#ifdef SENSE_check_q_underflow
    if (tail<0) {
      cerr << "SENSE error: attempt to get tail element of empty ";
      cerr << "Tsense_fastupdq\n";
      exit(-1);
    }
#endif
    return data[tail];
  }
  int size() const {return tail+1;};
  void empty() {tail=-1;};
};

class Tsense_fastdiruq {
 public:
  Tsense_dirunit * data;
  int tail;
  int qsize;
  void Tsense_fastdirq(const int n) {
    init(n);
  }
  void init(const int n) {
    data = new Tsense_dirunit[n];
    tail=-1;
    qsize=n;
  }
  ~Tsense_fastdiruq() {
    delete [] data;
  }
  void push(const Tsense_dirunit u) {
#ifdef SENSE_check_q_overflow
    if (tail==qsize-1) {
      cerr << "SENSE error: overflow in Tsense_fastupdq (" << tail+1;
      cerr << " elements) - increase size of sense_updqlen\n";
      exit(-1);
    }
#endif
    if (sense_sanity) {
      if (u.getlen()<1) {
cerr << "SENSE error: Attempt to insert unit with len<1 (";
```

```
cerr << u.getlen() << ") in fastupdq\n";
      }
    }
    tail++;
    data[tail]=u;
  }
  int size() {return tail+1;};
};


// Queue used for fast insert of Qunits while user program is running
typedef std::queue<Tsense_updqunit> Tsense_updq;

// Map used to shorten queue before sending updates at sync
typedef std::map<Tsense_addr, Tsense_updqunit> Tsense_updmap;

void sense_toqmap(Tsense_updmap &, Tsense_updqunit);

/*
This is the object responsible for handeling the queues of wait-
ing updates
to other nodes.
ToDo: We may want a dirtybit for the queues, so as to avoid short-
ening them
      when they are fixed - doubtful this is necessary, as in-
serts are
      frequent
*/
class Tsense_upd {
    public:
  //        Tsense_updq lists[sense_nodes]; // Array of lists, one pr. node
      Tsense_fastupdq lists[sense_nodes]; // Array of lists, one pr. node

        unsigned int last_seen[sense_nodes]; // When did we last see this node?
      // Inserts a dirunit into the queues with standard copyset
      // This method is used by wflush in dir
      void insert(const Tsense_dirunit d) {
        for (int n=0; n<sense_nodes; n++) {
          if (n!=sense_thisnode) {
            // Construct a queue-unit with default copyset
            //Tsense_updqunit tmp(d, sense_stdcpset[n]);
    Tsense_updqunit tmp(d, defcpset[sense_thisnode][n]);
    if (check_dirblock) {
      if (tmp.info.getlen()<=0) {
cerr << "SENSE error: Trying to insert block with ";
cerr << "an illegal length (" << tmp.info.getlen() << ")";
cerr << "in update queues\n";
exit(-1);
      }
    }
```

```
            // Insert it into queue
            lists[n].push(tmp);
#ifdef SENSE_debug
            cout << "Tsense_dir.insert: Inserted element into queue "
<< n << ", number of elements ";
            cout << "is now " << lists[n].size() << "\n";
#endif
            }
          }
        }
      // Insert a queue-unit in queues, if copyset permits it.
      // This method is used on synchronization to enqueue re-
cieved updates
      // from releaser of lock to the other nodes. There is no risk of
      // inserting it into lists[sense_thisnode], as sender sets copyset
      // to reflect that we already got it earlier on
      void insert(const Tsense_updqunit u) {
        for (int q=0; q<sense_nodes; q++) {
  Tsense_updqunit tmpu = u;
          if (!(u.getseen_by(q))) {
    tmpu.setseen_by(q);
    tmpu.setseen_by(sense_thisnode);
            lists[q].push(tmpu);
          }
        }
      }
      // Compress a list of updates
      void compress(int who) {
int llen=lists[who].size();
Tsense_updmap map;
// Init the map with one big unit at node -1, ltime -1
Tsense_updqunit u;
u.info.setstart(0);
u.info.setend(sense_memsize-1);
u.info.setltime(-1);
u.info.setatnode(-1);
map[sense_memsize-1]=u;
// Insert the elements into a map of updqunits
for (int n=0; n<llen; n++) {
  sense_toqmap(map, lists[who].front());
  lists[who].pop();
}
// Run through the map, and insert any block with Ltime!=-
1 into
// list
Tsense_updmap::iterator iter=map.begin();
int mlen=map.size();
#ifdef SENSE_debug
cout << llen << " queue elements became " << mlen << " map elements\n";
#endif
```

```
for (int n=0; n<mlen; n++) {
  if ((*iter).second.info.getltime()!=-1)
    lists[who].push((*iter).second);
  iter++;
}
        #ifdef SENSE_debug
cout << mlen << " map elements became " << lists[who].size();
cout << " list elements\n";
#endif
      }
      void compressto(int who, Tsense_fastdiruq &q) {
#ifdef SENSE_debug
cout << "compressto: entering\n";
#endif
int llen=lists[who].size();
Tsense_updmap map;
// Init the map with one big unit at node -1, ltime -1
Tsense_updqunit u;
u.info.setstart(0);
u.info.setend(sense_memsize-1);
u.info.setltime(-1);
u.info.setatnode(-1);
map[sense_memsize-1]=u;
// Insert the elements into a map of updqunits
#ifdef SENSE_debug
cout << "compressto: about to move list elements to map\n";
#endif
struct timeval tomapstart, tomapend;
gettimeofday(&tomapstart, 0);
for (int n=0; n<llen; n++) {
  sense_toqmap(map, lists[who].front());
  lists[who].pop();
}
gettimeofday(&tomapend, 0);
cout << "Compressto: time used on moving update list to map is :";
printtime(tomapstart, tomapend);
#ifdef SENSE_debug
cout << "compressto: done moving to map\n";
#endif
// Run through the map, and insert any block with Ltime!=-
1 into
// list
struct timeval toliststart, tolistend;
gettimeofday(&toliststart, 0);
Tsense_updmap::iterator iter=map.begin();
int mlen=map.size();
q.init(mlen);
#ifdef SENSE_debug
cout << llen << " queue elements became " << mlen << " map elements\n";
#endif
```

```
for (int n=0; n<mlen; n++) {
  if ((*iter).second.info.getltime()!=-1)
    q.push((*iter).second.info);
  iter++;
}
gettimeofday(&tolistend, 0);
cout << "Compressto: Time used on copying elements to list was: ";
printtime(toliststart, tolistend);
#ifdef SENSE_debug
cout << mlen << " map elements became " << q.size();
cout << " list elements\n";
cout << "compressto: exiting\n";
#endif
      }
      void moveto(int who, Tsense_fastdiruq &q) {
int slen=lists[who].size();
q.init(slen);
for (int n=0; n<slen; n++)
  q.data[n]=lists[who].data[n].info;
q.tail=slen-1;
lists[who].empty();
      }
      // Needs methods for streaming list over connection etc.
    };

extern Tsense_upd sense_upd;

typedef std::map<Tsense_addr, Tsense_dirunit> Tsense_dirmap;

/*
The class for the directory. Includes a map, methods for registering
reads and writes to shared mem, flushing writes and methods for explicitly
breaking the dir in chuncks.
*/

class Tsense_dir {
    public:
        Tsense_dirmap dir;
        Tsense_dirunit curwrite; // Unit currently being written
        Tsense_dirunit curread;  // Unit currently being read
Tsense_dirmap::iterator curreaditer;
        // Setters and getters
        void setcurwrite(const Tsense_dirunit n) {curwrite=n;}
        Tsense_dirunit getcurwrite() const {return curwrite;}
        void setcurread(const Tsense_dirunit n) {curread=n;}
        Tsense_dirunit getcurread() const {return curread;}
        // Constructor
        Tsense_dir(); // Init the directory
        // Other functions
        void read(const Tsense_addr a) {   // Read notifi-
```

```
cation to dir
  // If a is inside current readblock, there is nothing to do
  if ((a>=curread.getstart()) && (a<=curread.getend())) {
    #ifdef SENSE_stats
    stats_reads_cached++;
    #endif
    return;
  }
  // If a is inside current writeblock, there is nothing to do
  if ((a>=curwrite.getstart()) && (a<=curwrite.getend())) {
    #ifdef SENSE_stats
    stats_reads_cached++;
    #endif
    return;
  }
  // If a if in readblock following this
  #ifdef SENSE_debug
  cout << "Readbock is ";
  curread.print();
  cout << "\n";
  #endif
  if ((a-1)==curread.getend()) {
    #ifdef SENSE_debug
    cout << "Sequential read of memory detected on read of addr. " << a;
    cout << "\n";
    #endif
    #ifdef SENSE_stats
    stats_reads_sequential++;
    #endif
    curreaditer++;
    curread=(*curreaditer).second;
  }
  else {
    #ifdef SENSE_debug
    cout << "Read of addr " << a << " is outside current read/writeblock -
";
    cout << "Fetching from dir...\n";
    #endif
    // (flush current readblock w. updated data?
    //  Only if we do heuristic on read)
    // OK, if we are here, we need to get a new readblock
    curreaditer=dir.lower_bound(a);
    curread=(*curreaditer).second;
    #ifdef SENSE_debug
    cout << "Done!\n";
    #endif
  }
  // Is the memory up to date?
  if (!curread.islocal()) {
    // It isn't? Then we request an update
```

```
    #ifdef SENSE_debug
    cout << "Requesting this memory range : ";
    curread.print();
    cout << "\n";
    #endif
    if (check_dirblock) {
      if (curread.getlen()<=0) {
cerr << "SENSE error (Tsense_dir::read): ``;
cerr << somehow we got a block in the directory with ";
cerr << "an illegal length (" << curread.getlen() << ")\n";
cout << "Readblock data is: ";
curread.print();
exit(-1);
      }
    }
    mem_retrieve_from(curread.getatnode(),
curread.getstart(), (curread.getend()-curread.getstart())+1);
    // Now we have the data - mark it!
    curread.setatnode(sense_thisnode);
    // While we are at it - if using hybrid protocol we up requested
#ifdef SENSE_hybrid
    (*curreaditer).second.setrequested(curread.getrequested()+1);
#endif
    (*curreaditer).second.setatnode(sense_thisnode);
  }
}
        void write(const Tsense_addr a)
// Handles a write to address a - inlined for performance
// Order of checks based on assumption that most common sit-
uation is
// either extending writeblock top (sequential write) or miss
{
// if a extends current writeblock top, mark it, return
  if (a==curwrite.getend()+1) {
    curwrite.setend(a);
    #ifdef SENSE_stats
    stats_writes_cached++;
    #endif
    return;
  }
else
// if outside current writeblock flush it, create new con-
sisting only of a
  if ((a<curwrite.getstart()) || (a>curwrite.getend())) {
    wflush();
    curwrite.setstart(a);
    curwrite.setend(a);
#ifdef SENSE_hybrid
    // As a crude guess we set the writeblocks requested to the value of
    // the block containing a
```

```
    Tsense_dirmap::iterator i=dir.lower_bound(a);
    curwrite.requested=(*i).second.getrequested();
#endif
    }
  else
// if a extends current writeblock bottom, mark it, return
    if (a==curwrite.getstart()-1) {
      #ifdef SENSE_stats
      stats_writes_cached++;
      #endif
      curwrite.setstart(a);
    }
  #ifdef SENSE_stats
  else
    stats_writes_cached++;
  #endif
}
        void wflush(); // Flushes write unit at sync points
        void rflush(); // Flushes read unit at sync points
        void wprime(); // Inits write unit to harmless values
        void rprime(); // Re-reads read unit after sync
// Print contents of directory
void print() {
  for (Tsense_dirmap::iterator i=dir.begin(); i!=dir.end(); i++) {
    (*i).second.print();
    cout << "\n";
  }
}
        // Lacks methods for breaking up dir in chuncks
    };

// Function needed by barrier
void sense_general_tomap(Tsense_dirmap &, Tsense_dirunit );

// Function needed by update protocols barr and lock
void sense_general_tomap_upd(Tsense_dirmap &, Tsense_dirunit ,
                             Tsense_UC [], int &);

// Global instance of the directory
extern Tsense_dir sense_dir;

// The main mutex securing against cuncurrent access to the directory
extern pthread_mutex_t dirlock;

#endif


// Implements the directory functions
#include "directory.h"
#include "Lamport.h"
```

```
#include "sensememory.h"
#ifdef SENSE_stats
#include "stats.h"
#endif
#include <string.h>

// Allocate the undate queue
Tsense_upd sense_upd;

// Allocate the dir
Tsense_dir sense_dir;

// Allocate the main mutex
pthread_mutex_t dirlock;

// We might as well declare them now, as we know we'll need them and
// dynamic creation takes time
Tsense_updmap updmaps[sense_nodes];

// Init the directory
Tsense_dir::Tsense_dir()
{
  Tsense_dirunit t;

  // Init the current read block to all of memory, enter into map
  t.setstart(0);
  t.setend(sense_memsize-1);
  t.setltime(0);
  t.setatnode(sense_thisnode);
  setcurread(t);
  Tsense_addr t2=t.getend();
  dir[t2]=t; // Enter into map - it's the only block in the map now
  curreaditer=dir.begin(); // This will - in time - help with se-
quential reads


  // Init the current write block to one UC at end of mem-
ory. Saves repeated
  // checks for curwrite_loaded
  t.setstart(sense_memsize-1);
  t.setend(sense_memsize-1);
  t.setltime(0);
  setcurwrite(t);
}

// Called after sync to reset write unit to safe values
void Tsense_dir::wprime()
{
Tsense_dirunit t;
  t.setstart(sense_memsize-1);
```

```
  t.setend(sense_memsize-1);
  t.setltime(sense_ltime);
#ifdef SENSE_hybrid
  t.requested=0;
#endif
  setcurwrite(t);
}

// Called after sync to reset read unit to safe values
void Tsense_dir::rprime()
{
  // find block in dir containing last addr in readunit, as-
sumes seq. read
  #ifdef SENSE_debug
  cout << "Rprime: Current readblock is:\n";
  curread.print();
  cout << "\n";
  #endif
  curreaditer=dir.lower_bound(curread.getend());
  Tsense_dirunit t=(*curreaditer).second;
  setcurread(t);
  if (!curread.islocal()) {
    // It isn't? Then we request an update
    #ifdef SENSE_debug
    cout << "Requesting this memory range : ";
    curread.print();
    cout << "\n";
    #endif
    if (check_dirblock) {
      if (curread.getlen()<=0) {
cerr <<
"SENSE error (rprime): somehow we got a block in the direc-
tory with ";
cerr << "an illegal length (" << curread.getlen() << ")\n";
cout << "Readblock data is: ";
curread.print();
cout << "\n";
exit(-1);
      }
    }
    mem_retrieve_from(curread.getatnode(),
curread.getstart(), (curread.getend()-curread.getstart())+1);
    // Now we have the data - mark it!
    curread.setatnode(sense_thisnode);
    // While we are at it - if using hybrid protocol we up requested
#ifdef SENSE_hybrid
    (*curreaditer).second.setrequested(curread.getrequested()+1);
#endif
    (*curreaditer).second.setatnode(sense_thisnode);
  }
```

```
  #ifdef SENSE_debug
  cout << "Rprime: New readblock is:\n";
  curread.print();
  cout << "\n";
  #endif
}

// checks if a block has length <1
void checkblock(const Tsense_dirunit u, char * place) {
  if (check_dirblock) {
    if (u.getlen()<1) {
      cerr << "SENSE error (" << place << "), dirunit has size <1\n";
      cerr << "data folows: ";
      u.print();
      cout << "\n";
    }
  }
}

/*
Flushes current writeblock to directory. Sets ltime correctly
Special case of general insert of block, as current write-
block is always
newer or same ltime as blocks in directory by default
There are the following cases (N=new, E = Existing:

1) Total match
  NNNNN
  EEEEE

2) Partial match from start
  NNN
  EEEEE

3) Extends beyond from start
  NNNNN
  EEEee

4) Partial match to end
    NNN
  EEEEE

5) Included
   NNN
  EEEEE

6) Past edge
    NNN
  EEEeeee
*/
```

```
void Tsense_dir::wflush()
{
  if (check_dirblock) {
    if (curwrite.getlen()<=0) {
      cerr << "SENSE error (wflush): trying to enter a block with a length ";
      cerr << "<=0 (" << curwrite.getlen() << "), data follows\n";
      curwrite.print();
      cout << "\n";
      exit(-1);
    }
  }
#ifdef SENSE_stats
  stats_write_flushes++;
#endif
  // Set ltime for curwrite
  curwrite.setltime(sense_ltime);
  // Set this node as producer of this data
  curwrite.setatnode(sense_thisnode);
#ifdef SENSE_debug
  // Log to screen
  cout << "Wflush: Entering block w. ";
  curwrite.print();
  cout << " in shared memory\n";
#endif
  // Insert into lists of updates for other nodes
  sense_upd.insert(curwrite);
  // find block in dir containing .start
  Tsense_dirmap::iterator icurblock=dir.lower_bound(curwrite.getstart());
  // Make a copy for convinience
  Tsense_dirunit curblock=(*icurblock).second;
  // if total match (common case after few iter. of user prog. -
hopefully)
  if ((curblock.getstart()==curwrite.getstart()) &&
(curblock.getend()==curwrite.getend())) {
    //   This is case 1
    //   Replace ltime stamp etc., done
    (*icurblock).second.dupID(curwrite);
    return;
  }
  // If we are here, we need to do general block insert
  do {
    // Yes, we need to check again, if we are on iteration>1
    if ((curblock.getstart()==curwrite.getstart()) &&
(curblock.getend()==curwrite.getend())) {
      //   This is case 1
      //   Replace ltime stamp etc., done
      (*icurblock).second.dupID(curwrite);
      return;
    }
```

```
    if (curwrite.getstart()==curblock.getstart()) {
      if (curwrite.getend()<curblock.getend()) {
// This is case 2 - Partial overlap from start
// Adjust start of curblock
        (*icurblock).second.setstart(curwrite.getend()+1);
// insert new - done
        dir[curwrite.getend()]=curwrite;
        return;
      }
      else {
// This is case 3 - Extends beyond from start
// We know that it extends beyond curblock - we totally up-
date curblock
// Update ltime of curblock,
        (*icurblock).second.dupID(curwrite);
        curwrite.setstart(curblock.getend()+1);
checkblock(curwrite, "wflush, case 3");
      }
    }
    else
      // If we are here, we know that current.start<new.start
      if (curblock.getend()==curwrite.getend()) {
// Case 4 - partial overlap to end
        // Remove curblock from map
        dir.erase(icurblock);
        curblock.setend(curwrite.getstart()-1);
        dir[curblock.getend()]=curblock;
        dir[curwrite.getend()]=curwrite;
        return;
      }
      else {
        if (curblock.getend()>curwrite.getend()) {
  // Case 5 - new is totally enclosed in current
          dir.erase(icurblock);
          Tsense_dirunit frag1, frag2;
          frag1.setstart(curblock.getstart());
          frag1.setend(curwrite.getstart()-1);
  frag1.dupID(curblock);
          frag2.setstart(curwrite.getend()+1);
          frag2.setend(curblock.getend());
  frag2.dupID(curblock);
          dir[frag1.getend()]=frag1;
          dir[curwrite.getend()]=curwrite;
          dir[frag2.getend()]=frag2;
          return;
}
        else {
  // case 6 - new extends beyond current
          dir.erase(icurblock);
          Tsense_dirunit fragold, fragnew;
```

```
          // One fragment of old from start to start of new block
          fragold.setstart(curblock.getstart());
          fragold.setend(curwrite.getstart()-1);
  fragold.dupID(curblock);
          dir[fragold.getend()]=fragold;
          // One frament of new from start of new to end of old
          fragnew.setstart(curwrite.getstart());
          fragnew.setend(curblock.getend());
  fragnew.dupID(curwrite);
          dir[fragnew.getend()]=fragnew;
          // Adjust size of new
          curwrite.setstart(curblock.getend()+1);
  checkblock(curwrite, "wflush, case 6");
}
      }
    icurblock=dir.lower_bound(curwrite.getstart());
    curblock=(*icurblock).second;
  }
  while (true);
}

// Inserts a block in a map, considering L-time in existing blocks
// This is for inserting a list of updates recieved at synch
// points (lock or barr) when using invalidate protocol. Barrier
// *always* uses invalidate protocol!
// Presently assumes that map is prepared with one big block, so as
// to reuse algorithm from wflush
// ToDo: We may want to break up the blocks, even if ltime is <=
// in order to further the breaking up of memory into LLO's
void sense_general_tomap(Tsense_dirmap &m, Tsense_dirunit u)
{
  if (check_dirblock) {
    if (u.getlen()<=0) {
      cerr << "SENSE error (sense_general_tomap): ``;
      cerr << trying to enter a block with a length ";
      cerr << "<=0 (" << u.getlen() << "), data follows\n";
      u.print();
      cout << "\n";
    }
  }
  // logging to screen
  #ifdef SENSE_debug
  cout << "sense_general_tomap: Entering block w. ";
  u.print();
  cout << " in shared memory\n";
  #endif
// find block in dir containing .start
Tsense_dirmap::iterator iold=m.lower_bound(u.getstart());
// Make a copy for convinience
Tsense_dirunit old=(*iold).second;
```

```
// if total match (common case after few iters of user prog. -
hopefully)
if ((old.getstart()==u.getstart()) && (old.getend()==u.getend())) {
  //   This is case 1
  //   Replace ltime stamp etc., done
  if (old.getltime()<u.getltime()) {
    (*iold).second.dupID(u);
  }
  #ifdef SENSE_debug
  cout << "Sense_general_tomap: Leaving\n";
  #endif
  return;
}
// If we are here, we need to do general block insert
do {
  if ((old.getstart()==u.getstart()) && (old.getend()==u.getend())) {
    //   This is case 1
    //   Replace ltime stamp etc., done
    if (old.getltime()<u.getltime()) {
      (*iold).second.dupID(u);
    }
    return;
  }
  if (u.getstart()==old.getstart()) {
    if (u.getend()<old.getend()) {
//       This is case 2 - Partial overlap from start
      if (old.getltime()<u.getltime()) {
//      Adjust start of old
        (*iold).second.setstart(u.getend()+1);
//       insert new - done
        m[u.getend()]=u;
        }
      #ifdef SENSE_debug
      cout << "Sense_general_tomap: Leaving\n";
      #endif
      return;
    }
    else {
//    This is case 3 - Extends beyond from start
//    We know that it extends beyond old - we totally update old
      if (old.getltime()<u.getltime()) {
//      Update ltime of old,
        (*iold).second.setltime(u.getltime());
// Update atnode on old
        (*iold).second.setatnode(u.getatnode());
//      new.start=old.end+1 ' do another iteration
        u.setstart(old.getend()+1);
      }
      else
u.setstart(old.getend()+1);
```

```
    }
  }
  else {
//  If we are here, we know that current.start<new.start
    if (old.getend()==u.getend()) {
//    Case 4 - partial overlap to end
      // Remove old from map
      if (old.getltime()<u.getltime()) {
        m.erase(iold);
        old.setend(u.getstart()-1);
        m[old.getend()]=old;
        m[u.getend()]=u;
      }
      #ifdef SENSE_debug
      cout << "Sense_general_tomap: Leaving\n";
      #endif
      return;
    }
    else
      if (old.getend()>u.getend()) {
//         Case 5 - new is totally enclosed in current
        if (old.getltime()<u.getltime()) {
          m.erase(iold);
          Tsense_dirunit frag1, frag2;
          frag1.setstart(old.getstart());
          frag1.setend(u.getstart()-1);
  frag1.dupID(old);
          frag2.setstart(u.getend()+1);
          frag2.setend(old.getend());
  frag2.dupID(old);
          m[frag1.getend()]=frag1;
          m[u.getend()]=u;
          m[frag2.getend()]=frag2;
}
        #ifdef SENSE_debug
cout << "Sense_general_tomap: Leaving\n";
#endif
        return;
      }
      else {
//      case 6 - new extends beyond current
        if (old.getltime()<u.getltime()) {
          m.erase(iold);
          Tsense_dirunit fragold, fragnew;
          // One fragment of old from start to start of new block
          fragold.setstart(old.getstart());
          fragold.setend(u.getstart()-1);
  fragold.dupID(old);
          m[fragold.getend()]=fragold;
          // One frament of new from start of new to end of old
```

```
          fragnew.setstart(u.getstart());
          fragnew.setend(old.getend());
  fragnew.dupID(u);
          m[fragnew.getend()]=fragnew;
          // Adjust size of new
          u.setstart(old.getend()+1);
}
else
  u.setstart(old.getend()+1);
      }
  }
  iold=m.lower_bound(u.getstart());
  old=(*iold).second;
}
 while (true);
 cerr << "SENSE error: Unreachable branch reached in sense_general_tomap\n";
}

// Inserts a block in a map and relevant updates from buffer in mem-
ory,
// considering L-time in existing blocks
// This is for inserting a list of updates recieved at synch
// points (lock or barr) when using invalidate protocol. Barrier
// *always* uses invalidate protocol!
// Presently assumes that map is prepared with one big block, so as
// to reuse algorithm from wflush
// ToDo: We may want to break up the blocks, even if ltime is <=
// in order to further the breaking up of memory into LLO's
void sense_general_tomap_upd(Tsense_dirmap &m, Tsense_dirunit u,
                             Tsense_UC buf[], int &bufpos)
{
  if (check_dirblock) {
    if (u.getlen()<=0) {
      cerr << "SENSE error (sense_general_tomap): ``;
      cerr << trying to enter a block with a length ";
      cerr << "<=0 (" << u.getlen() << "), data follows\n";
      u.print();
      cout << "\n";
    }
  }
  // logging to screen
  #ifdef SENSE_debug
  cout << "sense_general_tomap: Entering block w. ";
  u.print();
  cout << " in shared memory\n";
  #endif
// find block in dir containing .start
Tsense_dirmap::iterator iold=m.lower_bound(u.getstart());
// Make a copy for convinience
Tsense_dirunit old=(*iold).second;
```

```
// if total match (common case after few iter. of user prog. -
hopefully)
if ((old.getstart()==u.getstart()) && (old.getend()==u.getend())) {
//   This is case 1
//   Replace ltime stamp etc., done
  if (old.getltime()<u.getltime()) {
    (*iold).second.dupID(u);
    // Punch it into the shared memory from buffer
    sense_write_raw(u.getstart(), u.getlen(), &buf[bufpos]);
  }
  // Either way we have used some UC's from the buffer
  bufpos+=u.getlen();
  return;
}
// If we are here, we need to do general block insert
do {
  if ((old.getstart()==u.getstart()) && (old.getend()==u.getend())) {
    //   This is case 1
    //   Replace ltime stamp etc., done
    if (old.getltime()<u.getltime()) {
      (*iold).second.dupID(u);
      // Punch it into the shared memory from buffer
      sense_write_raw(u.getstart(), u.getlen(), &buf[bufpos]);
    }
    // Either way we have used some UC's from the buffer
    bufpos+=u.getlen();
    return;
  }
  if (u.getstart()==old.getstart()) {
    if (u.getend()<old.getend()) {
      // This is case 2 - Partial overlap from start
      if (old.getltime()<u.getltime()) {
// Adjust start of old
        (*iold).second.setstart(u.getend()+1);
// insert new - done
        m[u.getend()]=u;
// Enter the updates into memry from buf
sense_write_raw(u.getstart(), u.getlen(), &buf[bufpos]);
      }
      // Adjust bufpos
      bufpos+=u.getlen();
      return;
    }
    else {
//    This is case 3 - Extends beyond from start
//    We know that it extends beyond old - we totally update old
      if (old.getltime()<u.getltime()) {
//      Update ltime etc. of old
        (*iold).second.dupID(u);
// Punch updates into memory
```

```
sense_write_raw(old.getstart(), old.getlen(), &buf[bufpos]);
//      new.start=old.end+1 ' do another iteration
      }
      u.setstart(old.getend()+1);
      bufpos+=old.getlen();
    }
  }
  else
//  If we are here, we know that current.start<new.start
    if (old.getend()==u.getend()) {
      // Case 4 - partial overlap to end
      // Remove old from map
      if (old.getltime()<u.getltime()) {
        m.erase(iold);
        old.setend(u.getstart()-1);
        m[old.getend()]=old;
        m[u.getend()]=u;
// Enter updates into memory
sense_write_raw(u.getstart(), u.getlen(), &buf[bufpos]);
      }
      bufpos+=u.getlen();
      return;
    }
    else
      if (old.getend()>u.getend()) {
// Case 5 - new is totally enclosed in current
        if (old.getltime()<u.getltime()) {
          m.erase(iold);
          Tsense_dirunit frag1, frag2;
          frag1.setstart(old.getstart());
          frag1.setend(u.getstart()-1);
  frag1.dupID(old);
          frag2.setstart(u.getend()+1);
          frag2.setend(old.getend());
  frag2.dupID(old);
          m[frag1.getend()]=frag1;
          m[u.getend()]=u;
  sense_write_raw(u.getstart(), u.getlen(), &buf[bufpos]);
          m[frag2.getend()]=frag2;
}
bufpos+=u.getlen();
        return;
      }
      else {
//      case 6 - new extends beyond current
int howmuch=(old.getend()-u.getstart())+1;
        if (old.getltime()<u.getltime()) {
          m.erase(iold);
          Tsense_dirunit fragold, fragnew;
          // One fragment of old from start to start of new block
```

```
          fragold.setstart(old.getstart());
          fragold.setend(u.getstart()-1);
  fragold.dupID(old);
          m[fragold.getend()]=fragold;
          // One frament of new from start of new to end of old
          fragnew.setstart(u.getstart());
          fragnew.setend(old.getend());
  fragnew.dupID(u);
          m[fragnew.getend()]=fragnew;
  sense_write_raw(fragnew.getstart(), fragnew.getlen(), &buf[bufpos]);
}
// Adjust size of new
u.setstart(old.getend()+1);
bufpos+=howmuch;
      }
  iold=m.lower_bound(u.getstart());
  old=(*iold).second;
  }
 while (true);
}

// Dups the copyset from s to d
inline void dupcopyset(const Tsense_updqunit s, Tsense_updqunit &d) {
  bool tmpcpset[sense_nodes];
  s.getcopyset(tmpcpset);
  d.setcopyset(tmpcpset);
}

// Inserts a queueblock in a map, considering L-time in ex-
isting blocks
// This is for shortening a list of updates sent at sync
// points (lock or barr)
// Presently assumes that map is prepared with one big block, so as
// to reuse algorithm from wflush
// ToDo: We may want to break up the blocks, even if ltime is <=
// in order to further the breaking up of memory into LLO's
void sense_toqmap(Tsense_updmap &m, Tsense_updqunit u)
{
  if (check_dirblock) {
    if (u.info.getlen()<=0) {
      cerr << "SENSE error (sense_general_tomap): ``;
      cerr << trying to enter a block with a length ";
      cerr << "<=0 (" << u.info.getlen() << "), data follows\n";
      u.info.print();
      cout << "\n";
    }
  }
// find block in dir containing .start
Tsense_updmap::iterator iold=m.lower_bound(u.getinfo().getstart());
// Make a copy for convinience
```

```
Tsense_dirunit old=(*iold).second.getinfo();
// if total match (common case after few iter. of user prog. -
hopefully)
 if ((old.getstart()==u.info.getstart()) &&
     (old.getend()==u.getinfo().getend())) {
   //   This is case 1
   //   Replace ltime stamp etc., done
   if (old.getltime()<u.info.getltime()) {
     (*iold).second.info.dupID(u.info);
     dupcopyset(u, (*iold).second);
   }
   return;
 }
// If we are here, we need to do general block insert
do {
 if ((old.getstart()==u.info.getstart()) &&
     (old.getend()==u.getinfo().getend())) {
   //   This is case 1
   //   Replace ltime stamp etc., done
   if (old.getltime()<u.info.getltime()) {
     (*iold).second.info.dupID(u.info);
     dupcopyset(u, (*iold).second);
   }
   return;
 }

  if (u.info.getstart()==old.getstart()) {
    if (u.info.getend()<old.getend()) {
//       This is case 2 - Partial overlap from start
      if (old.getltime()<u.info.getltime()) {
//      Adjust start of old
        (*iold).second.info.setstart(u.info.getend()+1);
//       insert new - done
        m[u.info.getend()]=u;
        }
      return;
      }
    else {
//    This is case 3 - Extends beyond from start
//    We know that it extends beyond old - we totally update old
      if (old.getltime()<u.info.getltime()) {
//      Update ltime etc. of old,
        (*iold).second.info.dupID(u.info);
//      Update copyset
        dupcopyset(u, (*iold).second);
//      new.start=old.end+1 ' do another iteration
      }
      u.info.setstart(old.getend()+1);
    }
  }
```

```
  else
//  If we are here, we know that current.start<new.start
    if (old.getend()==u.info.getend()) {
//    Case 4 - partial overlap to end
      // Remove old from map
      if (old.getltime()<u.info.getltime()) {
        Tsense_updqunit tmp;
        tmp=(*iold).second;
        m.erase(iold);
        tmp.info.setend(u.info.getstart()-1);
        m[tmp.info.getend()]=tmp;
        m[u.info.getend()]=u;
        }
      return;
      }
    else
      if (old.getend()>u.info.getend()) {
//         Case 5 - new is totally enclosed in current
        if (old.getltime()<u.info.getltime()) {
          Tsense_updqunit tmp;
          tmp=(*iold).second;
          m.erase(iold);
          Tsense_updqunit frag1, frag2;
          frag1.info.setstart(old.getstart());
          frag1.info.setend(u.info.getstart()-1);
  frag1.info.dupID(old);
          dupcopyset(tmp, frag1);
          frag2.info.setstart(u.info.getend()+1);
          frag2.info.setend(old.getend());
  frag2.info.dupID(old);
          dupcopyset(tmp, frag2);
          m[frag1.info.getend()]=frag1;
          m[u.info.getend()]=u;
          m[frag2.info.getend()]=frag2;
          }
        return;
        }
      else {
//      case 6 - new extends beyond current
        if (old.getltime()<u.info.getltime()) {
          Tsense_updqunit fragold, fragnew;
          dupcopyset((*iold).second, fragold);
          m.erase(iold);
          // One fragment of old from start to start of new block
          fragold.info.setstart(old.getstart());
          fragold.info.setend(u.info.getstart()-1);
  fragold.info.dupID(old);
  dupcopyset(old, fragold);
          m[fragold.info.getend()]=fragold;
          // One fragment of new from start of new to end of old
```

```
          fragnew.info.setstart(u.info.getstart());
          fragnew.info.setend(old.getend());
  fragnew.info.dupID(u.info);
          dupcopyset(u, fragnew);
          m[fragnew.info.getend()]=fragnew;
}
// Adjust size of new
u.info.setstart(old.getend()+1);
      }
  iold=m.lower_bound(u.info.getstart());
  old=(*iold).second.info;
  }
while (true);
}
```

C.6 Lamport.h, Lamport.o

```
/*
Declares the local l-time counter

*/

//----------------------------------------------------------
---
#ifndef LamportH
#define LamportH
//----------------------------------------------------------
---
extern int sense_ltime;
#endif

#include "Lamport.h"
int sense_ltime=0; // Allocates sense_ltime and initializes to 0
```

C.7 nodeaddr.h, nodeaddr.cpp

```
/*
Here we define the addresses of the involved nodes etc.

A note on port numbers:
There is no way of knowing which ports are in use on your com-
puters, except
for asking your local friendly sysop. GENERALLY it's safe to as-
sume that
ports above #5000 are available, but it may not be the case. PLEASE consult
with your sysop before blindly running SENSE.

SENSE uses rather a lot of port numbers, as it tries to keep connections
alive during the computation in order to avoid a performance penalty for
```

```
creating a connection. Thus it uses 1 connection pr. node to lock-
manager,
1 connection pr. node from the lockmanager and 1 connection from each node
to every other node.
*/

#ifndef nodeaddrH
#define nodeaddrH
#include "Defaults.h"

// The IP-addresses of the computational nodes
extern char *sense_nodes_IP [sense_nodes];

// The first port on computational nodes for requesting mem-
ory updates
// These will be server sockets, one for each of the other computational
// nodes
const int sense_reqmem_port_start = 8665;

// The port on computational nodes for requesting a lock
// The lock manager connects one pr. computational node
// This will be a server socket on the recieving end
const int sense_node_lock_port_start = 8765;

// The port the lockman forwards lockrequests to on the
// computational nodes
const int sense_lockman_forward_port = 8865;

// The IP-address of the lock-manager
extern char *sense_lockman_IP;

// The first port number on lock manager for requesting a lock
// These are server sockets on the lockmanager. They also handle
// coordination of setting up server- and client ports
const int sense_lockman_port_start = 8965;

// The first port number on lock manager for connections responsible
// for coordinating initialization
const int sense_lockman_init_port_start = 9065;

const int sense_bar_port_start = 8565;
#endif

/*
Allocates memory for char *'s pointing to IP's
*/

#include "nodeaddr.h"

char *sense_nodes_IP [sense_nodes] = {
```

```
  "192.168.0.2" , // Pluto - this is node 0!!!
  "192.168.0.1"}; // Rattatosk

char *sense_lockman_IP = "192.168.0.2"; // Rattatosk
```

C.8 portabillity.h

```
//-----------------------------------------------------------
-------------------
#ifndef portabilityH
#define portabilityH
//-----------------------------------------------------------
-------------------

typedef unsigned int Tsense_UC;

typedef /*unsigned*/ int Tsense_addr;

typedef unsigned char byte;

typedef unsigned int dword;

#endif
```

C.9 sense_typeconv.h, sense_typeconv.cpp

```
/*

Functions for converting ints to bytes etc.

Note that byte ordering (big/littleendian) is no problem, as we AS-
Sume the
nodes have the same byte ordering
*/

#ifndef sense_typeconvH
#define sense_typeconvH

// converts a byte to an int
int bytestoint(unsigned char *);

// converts an int to four bytes
void inttobytes(int, unsigned char *);

#endif

#include "sense_typeconv.h"
```

```
// converts a byte to an int
int bytestoint(unsigned char *buf) {
union {
int i;
struct {
unsigned char c0;
unsigned char c1;
unsigned char c2;
unsigned char c3;
} s;
} i2b;
i2b.s.c0=buf[0];
i2b.s.c1=buf[1];
i2b.s.c2=buf[2];
i2b.s.c3=buf[3];
return i2b.i;
}

// converts an int to four bytes
void inttobytes(int n, unsigned char *buf) {
union {
int i;
struct {
unsigned char c0;
unsigned char c1;
unsigned char c2;
unsigned char c3;
} s;
} i2b;
i2b.i=n;
buf[0]=i2b.s.c0;
buf[1]=i2b.s.c1;
buf[2]=i2b.s.c2;
buf[3]=i2b.s.c3;
}
```

C.10 sense_connections.h, sense_connections.cpp

```
/*
Here we define a wrapper for client and server sockets. Presently the sockets
use TCP to provide reliable connection-oriented transport. Any other protocol
with the same attributes may be used.

*/
#ifndef sense_connectionsH
#define sense_connectionsH

// If we don't trust the sockets, enable define below
//#define socks_debug
```

```
// Needed for pthreads
#define _REENTRANT
// Includes for socket communication
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <iostream.h>
// Used in cpp file
#include "sense_typeconv.h"
#include "directory.h"
#ifdef SENSE_stats
#include "stats.h"
#endif

unsigned char* sockread(int, int &);
void sockwrite(int, unsigned char*, int);
int sockreadint(int);
void sockwriteint(int, int);
void sockreadnbytes(int, unsigned char [], int );
void sockwritenbytes(int, unsigned char [], int);

// Class for client side of socket connection.
class Tsense_clisock {
 private:
  struct sockaddr_in serv_addr;
  int sockfd;
 public:
  unsigned char * bufptr;
  int msgsize;
  // Empty constructor for use in arrays of connections etc.
  Tsense_clisock() {;}
  // Constructor - creates and connects socket
  Tsense_clisock(const char serverIP[], int serverport, bool ex-
pedite=false) {
    init(serverIP, serverport, expedite);
  }
  // Init function usualle called by constructor, but avail-
able for
  // stuff like arrays of connections
  void init(const char serverIP[], int serverport, bool ex-
pedite=false) {
    // Null bufptr;
    bufptr=0;
    // Setup socket
```

```
    memset((char*) &serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(serverIP);
    serv_addr.sin_port = htons(serverport);
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0))<0)
      cerr << "Tsense_clisock: can't open stream socket\n";
    // Set options for the socket
    int on=1;
    // Expedite is useful for sending small messages
    if (expedite) {
      if (setsockopt(sockfd, IPPROTO_TCP,
TCP_NODELAY, (char *) &on, sizeof(on))< 0)
cerr << "Tsense_clisock: couldn't set TCP_NODELAY option\n";
    }

    // Just testing - increasing the recieve buffer size
    int bs=500000;
    if (setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, (char *) &bs, sizeof(bs))<0)
      cerr << "Tsense_clisock: coudn't set SOL_RCVBUF\n";
    if (setsockopt(sockfd, SOL_SOCKET, SO_SNDBUF, (char *) &bs, sizeof(bs))<0)
      cerr << "Tsense_clisock: coudn't set SOL_RCVBUF\n";

    // With this option we can reuse port/socket combinations (=restart SENSE
    // without irritating error messages
    on=1;
    if (setsockopt(sockfd, SOL_SOCKET,
SO_REUSEADDR, (char*) &on, sizeof(on))<0)
      cerr << "Tsense_clisock: couldn't set SO_REUSEADDR option\n";
    // Finaly connect the socket to a server socket
    if (connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr))<0)
      cerr << "Tsense_clisock: can't connect to server\n";
    }
  // Destructor - closes the socket
  ~Tsense_clisock() {
    close(sockfd);
    if (bufptr) delete [] bufptr;
  }
  void set_expedite() {
    int on=1;
    if (setsockopt(sockfd, IPPROTO_TCP,
TCP_NODELAY, (char *) &on, sizeof(on))< 0)
      cerr << "Tsense_servsock: couldn't set TCP_NODELAY option\n";
    }
  // Writes nbytes from buf to socket
  void sendmsg(unsigned char * buf, int nbytes) {
    #ifdef SENSE_stats
    stats_bytes_out+=nbytes+sizeof(int);
    #endif
    sockwrite(sockfd, buf, nbytes);
  }
```

```
  // Recieves into buf. Allocates new memory for buf and re-
turns number of
  // bytes read in nbytes

  void getmsg() {
    if (bufptr) delete [] bufptr;
    bufptr=sockread(sockfd, msgsize);
    #ifdef SENSE_stats
    stats_bytes_in+=msgsize+sizeof(int);
    #endif
  }
  void send(int i) {
    sockwriteint(sockfd, i);
    #ifdef SENSE_stats
    stats_bytes_out+=sizeof(int);
    #endif
  }
  void send(unsigned char * buf, int nbytes) {
    sockwritenbytes(sockfd, buf, nbytes);
    #ifdef SENSE_stats
    stats_bytes_out+=nbytes;
    #endif
  }
  int recieve() {
    #ifdef SENSE_stats
    stats_bytes_in+=sizeof(int);
    #endif
    return sockreadint(sockfd);
  }
  void recieve(unsigned char * buf, int nbytes) {
    sockreadnbytes(sockfd, buf, nbytes);
    #ifdef SENSE_stats
    stats_bytes_in+=nbytes;
    #endif
  }
};

class Tsense_servsock {
 private:
  int sockfd, newsockfd;
  socklen_t clilen;
  struct sockaddr_in cli_addr, serv_addr;
 public:
  unsigned char * bufptr;
  int msgsize;
  // Empty constructor for use in arrays of sockets
  Tsense_servsock() {;}
  // Main constructor for allocating a single connection
  Tsense_servsock(int serverport, bool expedite=false) {
    init(serverport, expedite);
```

```
  }
  // Init usually called from constructor, but available for ar-
rays of sockets
  void init(int serverport, bool expedite=false) {
    // Null the bufptr
    bufptr=0;
    // Setup socket
    if ((sockfd= socket(AF_INET, SOCK_STREAM, 0)) <0 )
      cerr << "Tsense_servsock: Can't open stream socket\n";
    int on=1;
    if (expedite) {
      if (setsockopt(sockfd, IPPROTO_TCP,
TCP_NODELAY, (char *) &on, sizeof(on))< 0)
cerr << "Tsense_servsock: couldn't set TCP_NODELAY option\n";
    }

    // Just testing - increasing the recieve buffer size
    int bs=500000;
    if (setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, (char *) &bs, sizeof(bs))<0)
      cerr << "Tsense_clisock: coudn't set SOL_RCVBUF\n";
    if (setsockopt(sockfd, SOL_SOCKET, SO_SNDBUF, (char *) &bs, sizeof(bs))<0)
      cerr << "Tsense_clisock: coudn't set SOL_RCVBUF\n";

    on=1;
    if (setsockopt(sockfd, SOL_SOCKET,
SO_REUSEADDR, (char*) &on, sizeof(on))<0)
      cerr << "Tsense_servsock: couldn't set SO_REUSEADDR option\n";
    memset((char*) &serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(serverport);
    if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr))<0)
      cerr << "Tsense_servsock: can't bind local address\n";
    listen(sockfd, 5);
  }
  ~Tsense_servsock() {
    if (bufptr) delete [] bufptr;
    close (newsockfd);
  }
  void accept() {
    clilen=sizeof(cli_addr);
    newsockfd = ::accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
    if (newsockfd <0)
      cerr << "Tsense_servsock: accept error\n";
  }
  void set_expedite() {
    int on=1;
    if (setsockopt(sockfd, IPPROTO_TCP,
TCP_NODELAY, (char *) &on, sizeof(on))< 0)
      cerr << "Tsense_servsock: couldn't set TCP_NODELAY option\n";
```

```
  }
  void sendmsg(unsigned char * buf, int nbytes) {
    sockwrite(newsockfd, buf, nbytes);
    #ifdef SENSE_stats
    stats_bytes_out+=nbytes+sizeof(int);
    #endif
  }
  void getmsg() {
    if (bufptr) delete [] bufptr;
    bufptr=sockread(newsockfd, msgsize);
    #ifdef SENSE_stats
    stats_bytes_in+=msgsize+sizeof(int);
    #endif
  }
  void send(int i) {
    sockwriteint(newsockfd, i);
    #ifdef SENSE_stats
    stats_bytes_out+=sizeof(int);
    #endif
  }
  void send(byte * buf, int nbytes) {
    sockwritenbytes(newsockfd, buf, nbytes);
    #ifdef SENSE_stats
    stats_bytes_out+=nbytes;
    #endif
  }
  int recieve() {
    #ifdef SENSE_stats
    stats_bytes_in+=sizeof(int);
    #endif
    return sockreadint(newsockfd);
  }
  void recieve(unsigned char * buf, int nbytes) {
    sockreadnbytes(newsockfd, buf, nbytes);
    #ifdef SENSE_stats
    stats_bytes_in+=nbytes;
    #endif
  }
};

#endif


/*
Implements the functions for reading and writing n bytes. This is a rather
naive implementation, that is just begging to be optimized.
*/

#include "sense_connections.h"
#include "sense_typeconv.h"
```

```
#include <errno.h>

void sockwritebyte(int sd, unsigned char c){
int rc;
  do {
    rc=write(sd, &c,1);
  }
  while (rc==0);
  #ifdef socks_debug
  cout << "Wrote " << rc << " bytes to sock, contained "
<< (short int) c << "\n";
  #endif
  if (rc==1)
    return;
  cerr << "sockwritebyte: Error " << rc << " writing byte to socket!\n";
}

void sockwritenbytes(int sd, unsigned char buf[], int nbytes) {
  int rc=0;
  int sofar=0;
  #ifdef socks_debug
  cout << "sockwritenbytes: About to write " << nbytes << " bytes\n";
  #endif
  do {
    rc=write(sd, buf+sofar, nbytes);
#ifdef socks_debug
    cout << "Wrote " << rc << " bytes from pos " << sofar << "\n";
#endif
    if (rc>=0) {
      nbytes-=rc;
      sofar+=rc;
    }
    else {
      cerr << "sockwritenbytes: Error " << rc
<< " writing buffer to socket\n";
      cerr << "Errorno is " << errno << "\n";
      cerr << "This means ";
      switch (errno) {
      case EINTR:
cerr << "EINTR"; break;
      case EAGAIN:
cerr << "EAGAIN"; break;
      case EIO:
cerr << "EIO"; break;
      case EISDIR:
cerr << "EISDIR"; break;
      case EBADF:
cerr << "EBADF"; break;
      case EINVAL:
cerr << "EINVAL"; break;
```

```
      case EFAULT:
cerr << "EFAULT"; break;
      default:
cerr << "other"; break;
      }
      cerr << "\n";
      cerr << "Called with sd=" << sd << ", nbytes=" << nbytes << "\n";
      exit (-1);
      return;
    }
  }
  while (nbytes>0);
}


void sockwriteint(int sd, int i) {
unsigned char buf[4];
 inttobytes(i, buf);
 sockwritenbytes(sd, buf, 4);
}

/*
Sockwrite writes nbytes of a buffer to a socket sd using a very simple
protocol. First is an int saying how many bytes are arriv-
ing, then the bytes.
*/

void sockwrite(int sd, unsigned char buf[], int nbytes) {

// Send number of bytes to reciever
  sockwriteint(sd, nbytes);
  if (nbytes<0) {
    cerr << "sockwrite: called with negative number (";
    cerr << nbytes << ") of bytes!\n";
  }
  if (nbytes>0)
    sockwritenbytes(sd, buf, nbytes);

}

void sockreadbyte(int sd, unsigned char &c){
int rc;
  do {
    rc=read(sd, &c,1);
  }
  while (rc==0);
  #ifdef socks_debug
  cout << "Got " << rc << " bytes from socket, contained "
<< (short int) c << "\n";
  #endif
```

```
  if (rc==1)
    return;
  cerr << "Error reading byte from socket, return value was " << rc << "\n";
  cerr << "Errorno is " << errno << "\n";
  cerr << "This means ";
  switch (errno) {
  case EINTR:
    cerr << "EINTR"; break;
  case EAGAIN:
    cerr << "EAGAIN"; break;
  case EIO:
    cerr << "EIO"; break;
  case EISDIR:
    cerr << "EISDIR"; break;
  case EBADF:
    cerr << "EBADF"; break;
  case EINVAL:
    cerr << "EINVAL"; break;
  case EFAULT:
    cerr << "EFAULT"; break;
  default:
    cerr << "other"; break;
  }
  cerr << "\n";
  return;

}

void sockreadnbytes(int sd, unsigned char buf[], int nbytes) {
  int rc=0;
  int sofar=0;
  do {

    rc=read(sd, buf+sofar, nbytes);
    if (rc==0)
      //      cout << "Recieved retval 0 from read\n";
      ;
    else {
      if (rc>0) {
  #ifdef socks_debug
cout << "Got " << rc << " bytes from socket\n";
  #endif
nbytes-=rc;
sofar+=rc;
      }
      else {
cerr << "Error reading buffer from socket, return value was "
<< rc << "\n";
cerr << "Errorno is " << errno << "\n";
cerr << "This means ";
```

```
switch (errno) {
case EINTR:
  cerr << "EINTR"; break;
case EAGAIN:
  cerr << "EAGAIN"; break;
case EIO:
  cerr << "EIO"; break;
case EISDIR:
  cerr << "EISDIR"; break;
case EBADF:
  cerr << "EBADF"; break;
case EINVAL:
  cerr << "EINVAL"; break;
case EFAULT:
  cerr << "EFAULT"; break;
default:
  cerr << "other"; break;
}
cerr << "\n";
return;
      }
    }
  }
  while (nbytes>0);
}

int sockreadint(int sd) {
unsigned char buf[4];
  sockreadnbytes(sd, buf, 4);
  return bytestoint(buf);
};


/*
Sockread reads an incomming message from a socket. Note that sock-
read allocates
memory for the buffer! Caller *must* delete the buffer af-
ter use!!!!
*/
unsigned char* sockread(int sd, int &nbytes) {

 // Read number of bytes to reciever
 nbytes=sockreadint(sd);
  #ifdef socks_debug
 cout << "Expecting " << nbytes << " bytes from socket\n";
  #endif

 if (nbytes>0) {
   // Allocate the memory
   unsigned char * buf=new unsigned char[nbytes];
```

```
    sockreadnbytes(sd, buf, nbytes);
    return buf;
  }
  else
    return 0;
}
```

C.11 thread_support.h, thread_support.cpp

```
/*

This is a support unit with functions to distribute the job of cre-
ating a
number of server socket connections on a bunch of threads.

*/
#ifndef thread_supportH
#define thread_supportH

// Uncomment to make threads print debug information
//#define SENSE_debug_threads

#define _REENTRANT
#include "sense_connections.h"
#include <pthread.h>

// Type function of type void * -> void *
typedef void * (voidpfnvoidp) (void *);

void spawn_server_connections(Tsense_servsock [], int ,
int , pthread_t [], int, voidpfnvoidp *);

void join_server_connections(pthread_t [], int, int);

#endif

#include "thread_support.h"

// This function is used for accepting the connections for the server
// sockets
void * acceptthreadfun(void * ptr) {
  Tsense_servsock *sockptr = (Tsense_servsock *) ptr;
  sockptr->accept();
  return 0;
}

void acceptthread(Tsense_servsock &sock,
pthread_t &tid, pthread_attr_t attr, voidpfnvoidp * fnptr) {
  // If no thread function supplied, we use the default one
  if (fnptr==0) {
```

```
#ifdef SENSE_debug_threads
    cout << "Using default accept thread for connection\n";
#endif
    fnptr=acceptthreadfun;
  }
  int res=pthread_create(&tid, &attr, fnptr, (void *) &sock);
  if (res<0)
    cerr << "Lock: Error creating thread for accepting connection\n";
}


// Inits and accepts nsocks server sockets at ports from first-
port to
// firstport+nsocks-1
void spawn_server_connections(Tsense_servsock socks[],
int nsocks, int firstport, pthread_t tids[],
int dontdo, voidpfnvoidp *fnptr) {

  pthread_attr_t attr;
  pthread_attr_init(&attr);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);


  // Init sockets and spawn the threads
  for (int n=0; n<nsocks; n++) {
    if (n!=dontdo) {
      socks[n].init(firstport+n);
#ifdef SENSE_debug_threads
      cout << "Spawning thread at port " << firstport+n << " with tid ";
#endif
      acceptthread(socks[n], tids[n], attr, fnptr);
#ifdef SENSE_debug_threads
      cout << tids[n] << " - done.\n";
#endif
    }
  }
}

void join_server_connections(pthread_t tids[], int nthreads, int dontdo){

  // Join with threads
  for (int n=0; n<nthreads; n++) {
    if (n!=dontdo) {
#ifdef SENSE_debug_threads
      cout << "Joining with thread with tid " << tids[n] << "...\n";
#endif
      pthread_join(tids[n], 0);
#ifdef SENSE_debug_threads
      cout << "done.\n";
#endif
```

```
    }
  }

}
```

C.12 sense_control.h, sense_control.cpp

```
/*

Functions for handeling control flow during startup phase

*/

#ifndef sense_controlH
#define sense_controlH

// Initializes client connection on computational node
void client_init_control();

// Initializes lockman's control sockets
void master_init_control();

// Called by computational node on arrival at internal barrier
void arrive();

// Called by lockman to wait at barrier
void barrier();

#endif

#include "sense_control.h"
#include "sense_connections.h"
#include "thread_support.h"
#include <pthread.h>
#include "Defaults.h"
#include "nodeaddr.h"

pthread_t ctrltids[sense_nodes];
Tsense_clisock ctrlcli;
Tsense_servsock ctrlmas[sense_nodes];

// Initializes client connection on computational node
void client_init_control() {
ctrlcli.init(sense_lockman_IP, sense_lockman_init_port_start+sense_thisnode);
}

// Initializes lockman's control sockets
void master_init_control() {
  spawn_server_connections(ctrlmas, sense_nodes,
```

```
sense_lockman_init_port_start, ctrltids, -1, 0);
  join_server_connections(ctrltids, sense_nodes, -1);
}

// Called by computational node on arrival at internal barrier
void arrive() {
int tmp=0;
 ctrlcli.send(tmp);
 tmp=ctrlcli.recieve();
}

// Called by lockman to wait at barrier
void barrier()
{
int tmp;
 for (int n=0; n<sense_nodes; n++) {
   tmp=ctrlmas[n].recieve();
 }
 for (int n=0; n<sense_nodes; n++) {
   ctrlmas[n].send(tmp);
 }
}
```

C.13 sense_barrier.h, sense_barrier.cpp

```
/*

Declares the functions needed for barrier support

*/
#define _REENTRANT
#ifndef sense_barrierH
#define sense_barrierH

void bar_init_servsocks();
void bar_init_clisocks();
void bar_kill_sockets();

void sense_barrier();

void sense_barrier_upd();

#endif

#include "sense_barrier.h"

// Included to give access to types for sense_dirunit etc.
#include "directory.h"

#include "thread_support.h"
```

```
#include "directory.h"
#include "Defaults.h"
#include "Lamport.h"
#include "nodeaddr.h"
#include <pthread.h>
#include <sched.h>
#include "sensememory.h"
#include "sense_timer.h"

// Sockets to recieve updates
Tsense_servsock bar_in[sense_nodes];

// Sockets to xmit updates
Tsense_clisock bar_out[sense_nodes];

// Highest Ltime seen during bar
int sense_bar_ltime_max=0;

// TID's for threads spawned
pthread_t Btids[sense_nodes];

// Attributes for creating threads
pthread_attr_t bar_attr;

// Mutexes for synchronizing the barrier threads
pthread_mutex_t bar_mutex_startthread[sense_nodes];
pthread_mutex_t bar_mutex_startintegrate[sense_nodes];

// Mutex for coordinating the communication
pthread_mutex_t bar_mutex_sync_read;
pthread_mutex_t bar_mutex_sync_write;

// For timing the damn thing
struct timeval threadfinish, threadjoin;

// class for parameters to update-based barriers
class Tsense_barthreadpar {
public:
  Tsense_servsock * sockp;
  int sockno; // Which socket is this - used in experimen-
tal new update barrier
  byte * buf;
  Tsense_barthreadpar() {sockp=0; buf=0;}
  void init(Tsense_servsock *p) {sockp=p; buf=0;}
  ~Tsense_barthreadpar() {
    if (buf) delete [] buf;
  }
};

// Array of parameters for the new update barrier threads
```

```
Tsense_barthreadpar barthreadpars[sense_nodes];

// Thread function to handle the server socket part of barrier
void * bar_thread_fun(void * p) {
  #ifdef SENSE_stats
  stats_msg_in++;
  #endif
  Tsense_servsock * sockp = (Tsense_servsock *) p;

  #ifdef SENSE_debug
  cout << "BarThread: Ready to accept requests\n";
  #endif
  int tmpltime=sockp->recieve();
  #ifdef SENSE_debug
  cout << "Got remote Ltime - it's " << tmpltime << "\n";
  #endif
  // Do we have to go?
  if (tmpltime==-1) {
    cout << "Barrier thread killing itself\n";
    pthread_exit(p);
  }
  sockp->getmsg();
  #ifdef SENSE_debug
  cout << "BarThread: got " << sockp->msgsize/sizeof(Tsense_dirunit)
<< " updates\n";
  #endif
  pthread_mutex_lock(&dirlock);
  // Find highest Ltime - this is done atomically, as we have the lock
  if (tmpltime>sense_bar_ltime_max)
    sense_bar_ltime_max=tmpltime;
  pthread_mutex_unlock(&dirlock);
  return 0;
}

// Thread function to handle the server socket part of bar-
rier if using update
// based protocol
void * bar_thread_fun_upd(void * p) {
  Tsense_barthreadpar *par= (Tsense_barthreadpar *) p;
  Tsense_servsock * sockp = par->sockp;
  #ifdef SENSE_debug
  cout << "BarThread: Ready to accept requests\n";
  #endif
  int tmpltime=sockp->recieve();
  #ifdef SENSE_debug
  cout << "Got remote Ltime - it's " << tmpltime << "\n";
  #endif
  // Do we have to go?
  if (tmpltime==-1) {
    cout << "Barrier thread killing itself\n";
```

```
    pthread_exit(p);
  }
  sockp->getmsg();
  #ifdef SENSE_debug
  cout << "BarThreadUpd: got " <<
   sockp->msgsize/sizeof(Tsense_dirunit)
   << " updates\n";
  #endif
  int nbytes=sockp->recieve();
  #ifdef SENSE_debug
  cout << "BarThreadPar: About to recieve " << nbytes/sizeof(Tsense_UC);
  cout << " units of shared memory\n";
  #endif
  if (nbytes>0) {
    par->buf = new byte[nbytes];
    sockp->recieve(par->buf, nbytes);
    #ifdef SENSE_debug
    cout << "BarThreadPar: Got them!\n";
    #endif
  }
  #ifdef SENSE_debug
  cout << "BarThreadPar: Increasing Ltime\n";
  #endif
  pthread_mutex_lock(&dirlock);
  // Find highest Ltime - this is done atomically, as we have the lock
  if (tmpltime>sense_bar_ltime_max)
    sense_bar_ltime_max=tmpltime;
  pthread_mutex_unlock(&dirlock);
  #ifdef SENSE_debug
  cout << "BarThreadPar: Done\n";
  #endif
  return 0;
}

// Thread function to handle the server socket part of bar-
rier if using update
// based protocol. This version is experimental for replac-
ing the original
// version, which was spawned every time
void * bar_thread_fun_upd_new(void * p) {
  Tsense_barthreadpar *par= (Tsense_barthreadpar *) p;
  Tsense_servsock * sockp = par->sockp;
  int idx=par->sockno; // Index into arrays of mutexes - also # of re-
mote node

  sockp->accept();
  do {
    // Wait until the barrier funtion lets us execute
    pthread_mutex_lock(&bar_mutex_startthread[idx]);
    #ifdef SENSE_debug
```

```
    cout << "BarThread: Ready to accept requests\n";
    #endif
    // If we start out as writers, we'd better yeild to the main thread
    if (sense_thisnode>idx) {
      cout << "Thread: I'm bigger, so I'd better yield\n";
      sched_yield();
    }
    // Guard our permission to write
    cout << "Thread: Grabbing readsync\n";
    pthread_mutex_lock(&bar_mutex_sync_read);
    int tmpltime=sockp->recieve();
    #ifdef SENSE_debug
    cout << "Got remote Ltime - it's " << tmpltime << "\n";
    #endif
    // Do we have to go?
    if (tmpltime==-1) {
      cout << "Barrier thread killing itself\n";
      pthread_exit(p);
    }
    struct timeval rcvstart, rcvend;
    gettimeofday(&rcvstart, 0);
    sockp->getmsg();
    gettimeofday(&rcvend, 0);
    cout << "Thread: Time to recieve "<< sockp->msgsize/sizeof(Tsense_dirunit);
    cout << " dir updates: ";
    printtime(rcvstart, rcvend);
    cout << "Thread: Releasing writesync\n";
    pthread_mutex_unlock(&bar_mutex_sync_write);
    cout << "Thread: Grabbing readsync\n";
    sched_yield();
    pthread_mutex_lock(&bar_mutex_sync_read);
    #ifdef SENSE_debug
    cout << "BarThreadUpd: got "
<< sockp->msgsize/sizeof(Tsense_dirunit) << " updates\n";
    #endif
    int nbytes=sockp->recieve();
    #ifdef SENSE_debug
    cout << "BarThreadPar: About to recieve " << nbytes/sizeof(Tsense_UC);
    cout << " units of shared memory\n";
    #endif
    if (nbytes>0) {
      par->buf = new byte[nbytes];
      gettimeofday(&rcvstart, 0);
      sockp->recieve(par->buf, nbytes);
      gettimeofday(&rcvend, 0);
      cout << "Thread: Time to recieve " << nbytes/sizeof(Tsense_UC);
      cout << " UC's of memory: ";
      printtime(rcvstart, rcvend);
      #ifdef SENSE_debug
      cout << "BarThreadPar: Got them!\n";
```

```
      #endif
    }
    cout << "Thread: Releasing writesync\n";
    pthread_mutex_unlock(&bar_mutex_sync_write);
    //pthread_mutex_lock(&bar_mutex_sync_read);
    #ifdef SENSE_debug
    cout << "BarThreadPar: Increasing Ltime\n";
    #endif
    //cout << "BarThread: About to get dirlock\n";
    //pthread_mutex_lock(&dirlock);
    // Find highest Ltime - this is done atomically, as we have the lock
    if (tmpltime>sense_bar_ltime_max)
      sense_bar_ltime_max=tmpltime;
    //pthread_mutex_unlock(&dirlock);
    #ifdef SENSE_debug
    cout << "BarThreadPar: Done\n";
    #endif
    // Allow the main function to proceed with integration of data
    pthread_mutex_unlock(&bar_mutex_startintegrate[idx]);
    gettimeofday(&threadfinish, 0);
  } while (true);
  return 0; // Not really necessary - here to keep -Wall from complaining
}

void sense_barrier_inv() {
  // Everyone is contacted twice - once for sync and once for updates
  #ifdef SENSE_stats
  stats_msg_in+=(sense_nodes-1)*2;
  stats_msg_out+=(sense_nodes-1)*2;
  #endif
  sense_dir.wflush();
  pthread_attr_t attr;
  pthread_attr_init(&attr);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

  int dummy=0;
  // First we make sure everyone is arrived, and tell them all we are here
  for (int n=0; n<sense_nodes; n++) {
    if (n!=sense_thisnode) {
      bar_out[n].send(dummy);
    }
  }
  for (int n=0; n<sense_nodes; n++) {
    if (n!=sense_thisnode) {
      int dummy=bar_in[n].recieve();
    }
  }
  #ifdef SENSE_debug
  cout << "Everyone has arrived now\n";
  #endif
```

```
  // Our own is the highest seen yet
  sense_bar_ltime_max=sense_ltime;
  // Fire up threads for recieving updates
  #ifdef SENSE_debug
  cout << "Starting threads for reading updates\n";
  #endif
  for (int n=0; n<sense_nodes; n++) {
    if (n!=sense_thisnode) {
      int res=pthread_create(&Btids[n], &attr,
bar_thread_fun, (void *) &bar_in[n]);
      if (res<0) {
cerr << "barrier: Error creating thread for accepting connection\n";
exit(-1);
      }
    }
  }
  // We want to avoid flooding poor node 0, so we start at next node
  // and progress
  int n=sense_thisnode;
  for (int donode=0 ; donode<(sense_nodes-1); donode++) {
    n=(n+1)%sense_nodes;
    pthread_mutex_lock(&dirlock);
    bar_out[n].send(sense_ltime);
    #ifdef SENSE_debug
    cout << "Bar: Queue size before compress is ";
    cout << sense_upd.lists[n].size() << "\n";
    cout << "Bar: allocating fastdirq\n";
    #endif
    Tsense_fastdiruq q;
    #ifdef SENSE_debug
    cout << "Bar: done allocating\n";
    #endif
    sense_upd.compressto(n, q);
    #ifdef SENSE_debug
    cout << "Bar: Queue size after compress is ";
    cout << q.size() << "\n";
    #endif
    int qlen=q.size();
#ifdef SENSE_debug
    cout << "Ready to send " << qlen << " updates to node " << n;
    cout << " (that is " << sizeof(Tsense_dirunit)*qlen << " bytes)\n";
#endif
#ifdef SENSE_debug
    cout << "Sending updates now\n";
#endif
    bar_out[n].sendmsg((byte *) q.data, sizeof(Tsense_dirunit)*qlen);
#ifdef SENSE_debug
    cout << "Done\n";
#endif
    pthread_mutex_unlock(&dirlock);
```

```
  }
  // Join with the threads
  join_server_connections(Btids, sense_nodes, sense_thisnode);
  // Integrate the recieved updates
  for (int q=0 ; q<sense_nodes ; q++) {
    if (q!=sense_thisnode) {
      int nupdates=bar_in[q].msgsize/sizeof(Tsense_dirunit);
      #ifdef SENSE_debug
      cout << "Integrating " << nupdates << " updates\n";
      #endif
      Tsense_dirunit * buf = (Tsense_dirunit *) bar_in[q].bufptr;
      for (int i=0; i<nupdates; i++) {
#ifdef SENSE_debug
cout << "Integrating update ";
buf[i].print();
cout << "\n";
#endif
sense_general_tomap(sense_dir.dir, buf[i]);
      }
      #ifdef SENSE_debug
      cout << "Done integrating updates\n";
      #endif
    }
  }
  sense_ltime=sense_bar_ltime_max+1;
  sense_dir.wprime();
  sense_dir.rprime();
}

void bar_upd_xmitto(const int n) {
    bar_out[n].send(sense_ltime);
    struct timeval cmpstart, cmpend;
    gettimeofday(&cmpstart, 0);
    #ifdef SENSE_debug
    cout << "Bar: Queue size before compress is ";
    cout << sense_upd.lists[n].size() << "\n";
    #endif
    Tsense_fastdiruq q;
    //sense_upd.compressto(n, q);
    sense_upd.moveto(n, q);
    #ifdef SENSE_debug
    cout << "Bar: Queue size after compress is ";
    cout << q.size() << "\n";
    #endif
    gettimeofday(&cmpend, 0);
    cout << "Time used on compress is: ";
    printtime(cmpstart, cmpend);
    struct timeval colstart, colend;
    gettimeofday(&colstart, 0);
    int qlen=q.size();
```

```
    int nUCs=0;
    for (int x=0; x<qlen; x++)
      nUCs+=q.data[x].getlen();
    #ifdef SENSE_debug
    cout << "Collecting " << nUCs << " memory updates in buffer\n";
    #endif
    Tsense_UC * membuf = new Tsense_UC[nUCs];
    int bufpos=0;
    for (int x=0; x<qlen; x++) {
      sense_read_raw(q.data[x].getstart(),
q.data[x].getlen(), &membuf[bufpos]);
      bufpos+=q.data[x].getlen();
      // Now that we use update proto, let them know they are gonna get it!
      // We just might be using update barriers in combina-
tion with a
      // (partial) update lock protocol
      q.data[x].setatnode(n);
    }
    gettimeofday(&colend, 0);
    cout << "Time used on collecting memory updates: ";
    printtime(colstart, colend);
#ifdef SENSE_debug
    cout << "Done collecting memory updates\n";
    cout << "Ready to send " << qlen << " updates to node " << n;
    cout << " (that is " << sizeof(Tsense_dirunit)*qlen << " bytes)\n";
#endif
    struct timeval memxmitstart, memxmitend;
    gettimeofday(&memxmitstart, 0);
    bar_out[n].sendmsg((byte *) q.data, sizeof(Tsense_dirunit)*qlen);
    gettimeofday(&memxmitend, 0);
    cout << "Time used on xmitting " << qlen << " dir updates is: ";
    printtime(memxmitstart, memxmitend);
#ifdef SENSE_debug
    cout << "Done\n";
    cout << "Sending memory updates\n";
#endif
    int nbytes=nUCs*sizeof(Tsense_UC);
    gettimeofday(&memxmitstart, 0);
    bar_out[n].send(nbytes);
    if (nbytes>0)
      bar_out[n].send((byte*) membuf, nbytes);
    gettimeofday(&memxmitend, 0);
    cout << "Time used on xmitting " << nUCs << " memory up-
dates is: ";
    printtime(memxmitstart, memxmitend);
    //#ifdef SENSE_debug
    cout << "Done transmitting memory updates\n";
    //#endif
    delete [] membuf;
}
```

```
void bar_upd_rcvfrom(const int idx, Tsense_barthreadpar* par) {
  cout << "Entering rcvfrom\n";
    Tsense_servsock *sockp=par->sockp;
    int tmpltime=sockp->recieve();
    //#ifdef SENSE_debug
    cout << "Got remote Ltime - it's " << tmpltime << "\n";
    //#endif
    struct timeval rcvstart, rcvend;
    gettimeofday(&rcvstart, 0);
    sockp->getmsg();
    gettimeofday(&rcvend, 0);
    cout << "Thread: Time to recieve "<< sockp->msgsize/sizeof(Tsense_dirunit);
    cout << " dir updates: ";
    printtime(rcvstart, rcvend);
    #ifdef SENSE_debug
    cout << "BarThreadUpd: got "
<< sockp->msgsize/sizeof(Tsense_dirunit) << " updates\n";
    #endif
    int nbytes=sockp->recieve();
    #ifdef SENSE_debug
    cout << "BarThreadPar: About to recieve " << nbytes/sizeof(Tsense_UC);
    cout << " units of shared memory\n";
    #endif
    if (nbytes>0) {
      par->buf = new byte[nbytes];
      gettimeofday(&rcvstart, 0);
      sockp->recieve(par->buf, nbytes);
      gettimeofday(&rcvend, 0);
      cout << "Thread: Time to recieve " << nbytes/sizeof(Tsense_UC);
      cout << " UC's of memory: ";
      printtime(rcvstart, rcvend);
      #ifdef SENSE_debug
      cout << "BarThreadPar: Got them!\n";
      #endif
    }
    #ifdef SENSE_debug
    cout << "BarThreadPar: Increasing Ltime\n";
    #endif
    //cout << "BarThread: About to get dirlock\n";
    //pthread_mutex_lock(&dirlock);
    // Find highest Ltime - this is done atomically, as we have the lock
    if (tmpltime>sense_bar_ltime_max)
      sense_bar_ltime_max=tmpltime;
    //pthread_mutex_unlock(&dirlock);
    #ifdef SENSE_debug
    cout << "BarThreadPar: Done\n";
    #endif
}
```

```
class par {
public:
  int val;
};

// Handles barrier xmit/rvc threaded
void * sense_bar_new_updt(void *p) {
  cout << "Thread spwaned\n";
  par *pnode=(par *) p;
  int node=pnode->val;
  cout << "Barrier thread doing node " << node << "\n";
  // We block here waiting for goahead from remote node
  int dummy=bar_in[node].recieve();
  if (sense_thisnode>node) {
    cout << "Bar: I'm bigger, and I get to xmit first\n";
    bar_upd_xmitto(node);
    bar_upd_rcvfrom(node, &barthreadpars[node]);
  }
  else {
    cout << "Bar: I'm smaller, and I get to recv first\n";
    bar_upd_rcvfrom(node, &barthreadpars[node]);
    bar_upd_xmitto(node);
  }
}

void sense_barrier_upd() {
  struct timeval barstart, barend;
  gettimeofday(&barstart, 0);
  // Everyone is contacted twice - once for sync and once for updates
  #ifdef SENSE_stats
  stats_msg_in+=(sense_nodes-1)*2;
  stats_msg_out+=(sense_nodes-1)*2;
  #endif
  sense_dir.wflush();
  // Our own is the highest seen yet
  sense_bar_ltime_max=sense_ltime;
  int dummy=0;
  // First we make sure everyone is arrived, and tell them all we are here
  for (int n=0; n<sense_nodes; n++)
    if (n!=sense_thisnode)
      bar_out[n].send(dummy);
  // Fire up threads for recieving updates
  par pars[sense_nodes];
  for (int n=0; n<sense_nodes; n++)
    if (n!=sense_thisnode) {
      cout << "Spawning thread for node " << n << "\n";
      pars[n].val=n;
      int res=pthread_create(&Btids[n],
&bar_attr, sense_bar_new_updt, (void *) &pars[n]);
      if (res<0) {
```

```
cerr << "barrier: Error creating thread for accepting connection\n";
exit(-1);
      }
      else
cout << "Done..\n";
    }

  // Join with threads
  join_server_connections(Btids, sense_nodes, sense_thisnode);
  cout << "Bar: Done communication - about to integrate updates\n";
  // Integrate the recieved updates
  for (int node=0 ; node<sense_nodes ; node++) {
    if (node!=sense_thisnode) {
      int membufpos=0;
      int nupdates=bar_in[node].msgsize/sizeof(Tsense_dirunit);
#ifdef SENSE_debug
      cout << "Integrating " << nupdates << " updates\n";
#endif
      struct timeval intstart, intend;
      gettimeofday(&intstart, 0);
      Tsense_dirunit * buf = (Tsense_dirunit *) bar_in[node].bufptr;
      for (int upd=0; upd<nupdates; upd++) {
#ifdef SENSE_debug
cout << "Integrating update ";
buf[upd].print();
cout << "\n";
#endif
sense_general_tomap_upd(sense_dir.dir, buf[upd],
(Tsense_UC *) barthreadpars[node].buf, membufpos);
      }
      gettimeofday(&intend, 0);
      cout << "Time spent integrating updates in local mem: ";
      printtime(intstart, intend);
      delete [] barthreadpars[node].buf; barthreadpars[node].buf=0;
      #ifdef SENSE_debug
      cout << "Done integrating updates\n";
      #endif
    }
  }
  sense_ltime=sense_bar_ltime_max+1;
  sense_dir.wprime();
  sense_dir.rprime();
  //cout << "*barr: exiting - hit any key\n";
  //char dummyc;
  //cin >> dummyc;
  gettimeofday(&barend, 0);
  cout << "Barrier time used total: ";
  printtime(barstart, barend);
}
```

```
void sense_barrier() {
#ifdef SENSE_stats
stats_barriers++;
#endif
#ifdef SENSE_barriers_update
sense_barrier_upd();
#endif
#ifndef SENSE_barriers_update
sense_barrier_inv();
#endif
}

// Gets server sockets initialized
void bar_init_servsocks() {
  // Initialize the thread attributes
  pthread_attr_init(&bar_attr);
  pthread_attr_setscope(&bar_attr, PTHREAD_SCOPE_SYSTEM);
  spawn_server_connections(bar_in, sense_nodes, sense_bar_port_start,
Btids, sense_thisnode, 0);
}

void bar_init_clisocks() {
  // Get client sockets up and running, lock the mutexes
  for (int n=0; n<sense_nodes; n++) {
    if (n!=sense_thisnode) {
      barthreadpars[n].sockno=n;
      barthreadpars[n].sockp=&bar_in[n];
      #ifdef SENSE_debug
      cout << "Connecting client thread to IP " << sense_nodes_IP[n];
      cout << " port " << sense_bar_port_start+n << "\n";
      #endif
      bar_out[n].init(sense_nodes_IP[n], sense_bar_port_start+sense_thisnode);
    }
  }
  // Now join with accept threads for ingoing communication
  join_server_connections(Btids, sense_nodes, sense_thisnode);
}

void bar_kill_sockets(){
  // Do nothing - not necessary!
}
```

C.14 stats.h, stats.cpp

```
/*
Here we define the variables used for gathering statistics and the functions
for printing them. Presently it is assumed that increment-
ing these variables
is atomic, so they are not protected by a mutex, even though that re-
ally ought
```

```
to be done.

Convention: A group of writes to a socket is considered ONE mes-
sage. Same goes
for reads
*/

#ifndef statsH
#define statsH
#define _REENTRANT
#include "Defaults.h"
#include <sys/time.h>
#include "sense_timer.h"
// Inits the stat variables
void stats_init();

// Prints current stat variables
void stats_print();

void stats_start_timing();

void stats_end_timing();


// Number of messages sent
extern int stats_msg_out;

// Number of messages recieved
extern int stats_msg_in;

// Number of bytes sent
extern int stats_bytes_out;

// Number of bytes recieved
extern int stats_bytes_in;

// Number of read faults resulting in a fetch from another node
extern int stats_read_faults;

// Number of reads
extern int stats_reads;

// Number of reads satisfied from readblock
extern int stats_reads_cached;

// Number of writes
extern int stats_writes;

// Number of flushes of write unit
extern int stats_write_flushes;
```

```
// Number of writes satisfied from writeblock
extern int stats_writes_cached;

// Number of locks taken
extern int stats_locks_taken;

// Number of which were satisfied locally
extern int stats_locks_local;

// Number of barriers met
extern int stats_barriers;

// Number of sequential reads detected
extern int stats_reads_sequential;

// Max size of update queues
extern int stats_updq_maxsize;

// Max usage of memory
extern int stats_mem_maxsize;

// Number of lock handoff requests recieved
extern int stats_lock_req_in;

// Number of these recieved while lock was locked
extern int stats_req_while_taken;

// Total time spent acquiring locks
extern double stats_lock_acqtime;

// Total time spent releasing locks
extern double stats_lock_reltime;

// Total time taken doing lock handoff
extern double stats_lock_handofftime;

// Total time spent holding locks
extern double stats_lock_heldtime;

#endif


#include "stats.h"
#include <iostream.h>
#include <stdio.h>

// Number of messages sent
int stats_msg_out;
```

```
// Number of messages recieved
int stats_msg_in;

// Number of bytes sent
int stats_bytes_out;

// Number of bytes recieved
int stats_bytes_in;

// Number of read faults resulting in a fetch from another node
int stats_read_faults;

// Number of reads
int stats_reads;

// Number of reads satisfied from readblock
int stats_reads_cached;

// Number of writes
int stats_writes;

// Number of writes satisfied from writeblock
int stats_writes_cached;

// Number of flushes of the write unit
int stats_write_flushes;

// Time when started
struct timeval stats_start;

// Time when finished
struct timeval stats_end;

// Number of locks taken
int stats_locks_taken;

// Number of which were satisfied locally
int stats_locks_local;

// Number of barriers met
int stats_barriers;

// Number of sequential reads detected
int stats_reads_sequential;

// Max size of update queues
int stats_updq_maxsize;

// Max usage of memory
int stats_mem_maxsize;
```

```
// Number of lock handoff requests recieved
int stats_lock_req_in;

// Number of these recieved while lock was locked
int stats_req_while_taken;

// Total time spent acquiring locks
double stats_lock_acqtime;

// Time spent doing handoff
double stats_lock_handofftime;

// Total time spent releasing locks
double stats_lock_reltime;

// Total time spent holding locks
double stats_lock_heldtime;


// Inits the stat variables
void stats_init() {
stats_msg_out=0;
stats_msg_in=0;
stats_bytes_out=0;
stats_bytes_in=0;
stats_read_faults=0;
stats_reads=0;
stats_reads_cached=0;
stats_reads_sequential=0;
stats_writes=0;
stats_writes_cached=0;
stats_write_flushes=0;
stats_locks_taken=0;
stats_locks_local=0;
stats_lock_acqtime=0.0;
stats_lock_reltime=0.0;
stats_lock_handofftime=0.0;
stats_lock_heldtime=0.0;
stats_barriers=0;
stats_updq_maxsize=0;
stats_mem_maxsize=0;
stats_lock_req_in=0;
stats_req_while_taken=0;
}

void stats_start_timing() {
  gettimeofday(&stats_start, 0);
}
```

```
void stats_end_timing() {
  gettimeofday(&stats_end, 0);
}

// Prints current stat variables
void stats_print() {
  cout << "* Time:\n";
  cout << "-------\n";
cout << "Stat: ";
fprintf(stdout, "Elapsed time: %.2f seconds\n",
          (((stats_end.tv_sec * 1000000.0) + stats_end.tv_usec) -

          ((stats_start.tv_sec * 1000000.0) + stats_start.tv_usec))
/ 1000000.0);

 cout << "\n";
 cout << "* Network\n";
 cout << "---------\n";
cout << "Stat: ";
cout << "Messages sent " << stats_msg_out;
cout << "\n";
cout << "Stat: ";
cout << "Messages recieved " << stats_msg_in;
cout << "\n";
cout << "Stat: ";
cout << "Bytes sent " << stats_bytes_out;
cout << "\n";
cout << "Stat: ";
cout << "Bytes recieved " << stats_bytes_in;
cout << "\n";

 cout << "\n";
 cout << "* SENSE memory subsystem\n";
 cout << "------------------------\n";
cout << "Stat: ";
cout << "Memory blocks requested " << stats_read_faults;
cout << "\n";
cout << "Stat: ";
cout << "Number of reads to shared memory ";
cout << stats_reads;
cout << "\n";
cout << "Stat: ";
cout << "Number of reads satisfied by read cache " << stats_reads_cached;
cout << "\n";
cout << "Stat: ";
cout << "Number of sequential block reads detected by read cache ";
cout << stats_reads_sequential;
cout << "\n";
cout << "Stat: ";
cout << "Number of writes to shared memory " << stats_writes;
```

```
cout << "\n";
cout << "Stat: ";
cout << "Number of writes satisfied by write cache " << stats_writes_cached;
cout << "\n";
cout << "Stat: ";
cout << "Number of flushes of the write unit " << stats_write_flushes;
cout << "\n";
cout << "Stat: ";
cout << "Max. size of updatequeues " << stats_updq_maxsize;
cout << "\n";
cout << "Stat: ";
cout << "UC's of memory allocated " << stats_mem_maxsize;
cout << "\n";

 cout << "\n";
 cout << "* Locks and barriers\n";
 cout << "--------------------\n";
cout << "Stat: ";
cout << "Number of locks taken " << stats_locks_taken;
cout << "\n";
cout << "Stat: ";
cout << "Number of these satisfied locally " << stats_locks_local;
cout << "\n";
cout << "Stat: ";
cout << "Number of lock requests recieved " << stats_lock_req_in;
cout << "\n";
cout << "Stat: ";
cout << "Number of these recieved while lock was held "
<< stats_req_while_taken;
cout << "\n";
cout << "Stat: ";
cout << "Average time to acquire a lock "
<< stats_lock_acqtime/stats_locks_taken;
cout << "\n";
cout << "Stat: ";
cout << "Average time to release a lock "
<< stats_lock_reltime/stats_locks_taken;
cout << "\n";
cout << "Stat: ";
cout << "Average time to handoff a lock to remote "
<< stats_lock_handofftime/stats_lock_req_in;
cout << "\n";
cout << "Stat: ";
cout << "Average time lock is held "
<< stats_lock_heldtime/stats_locks_taken;
cout << "\n";

cout << "Stat: ";
cout << "Number of barriers met " << stats_barriers;
cout << "\n";
```

```
}
```

C.15 sense_unit.h, sense_unit.cpp

```
// Contains the units that have to be visible to the user program
// Also the main init function

//-----------------------------------------------------------
-------------------
#ifndef sense_unitH
#define sense_unitH
//-----------------------------------------------------------
-------------------

// sense_malloc, sense_read, sense_write, shared datatypes
#include "sensememory.h"
// Locks
#include "Lock.h"
// Barriers
#include "sense_barrier.h"

// Main init function etc.
void sense_init();

// Main end function
void sense_finish();

// compatibillity stuff for Treadmarks programs
inline void Tmk_lock_acquire(const int i) {
  sense_acq(i);
}

inline void Tmk_lock_release(const int i) {
  sense_rel(i);
}

inline void Tmk_barrier(const int i) {
  sense_barrier();
}

inline void Tmk_startup(const int a,  char **b) {
sense_init();
}

inline void Tmk_exit(const int i) {
  sense_finish();
}

const int Tmk_proc_id=sense_thisnode;
const int Tmk_nprocs=sense_nodes;
```

```
// Same for CVM
inline void cvm_lock(const int i) {
  sense_acq(i);
}

inline void cvm_unlock(const int i) {
  sense_rel(i);
}

inline void cvm_barrier(const int i) {
  sense_barrier();
}

inline int cvm_pid() {
  return sense_thisnode;
}

inline void cvm_lock_reduce(const int i, const char* p, const int ii) {
  cerr << "SENSE equivivalent to cvm_lock_reduce NOT implemented\n";
}

inline void cvm_zero_stats() {
  ; // do nothing
}

inline void cvm_yield() {
  cerr << "SENSE equivivalent to cvm_yield NOT implemented\n";
}

inline int cvm_tid() {
  return 0; // Only one thread in SENSE
}

const int cvm_threads_proc=1; // Only one thread pr. node in SENSE
const int cvm_num_procs=sense_nodes;

inline void cvm_finish() {
  sense_finish();
}

#endif

#include "sense_unit.h"
#include "sense_control.h"
#ifdef SENSE_stats
#include "stats.h"
#endif

void sense_init()
```

```
{
  cout << "This is node " << sense_thisnode << " of " << sense_nodes << "\n";
  cout << "The protocol for locks is ";
  #ifdef SENSE_pure_update
  cout << "pure update\n";
  #endif
  #ifdef SENSE_pure_invalidate
  cout << "pure invalidate\n";
  #endif
  #ifdef SENSE_hybrid
  cout << "hybrid update/invalidate\n";
  #endif
  cout << "The protocol for barriers is ";
  #ifdef SENSE_barriers_update
  cout << "pure update\n";
  #endif
  #ifndef SENSE_barriers_update
  cout << "pure invalidate\n";
  #endif

  // Init client side of internal barrier
  client_init_control();
  bar_init_servsocks();
  #ifdef SENSE_debug
  cout << "Barrier server sockets are up\n";
  #endif
  mem_init_servsocks();
  #ifdef SENSE_debug
  cout << "Memory server sockets are up\n";
  #endif
  lock_init_servsocks();
  #ifdef SENSE_debug
  cout << "Lock server sockets are up\n";
  #endif
  arrive();
  bar_init_clisocks();
  #ifdef SENSE_debug
  cout << "Barrier client sockets are up\n";
  #endif
  mem_init_clisocks();
  #ifdef SENSE_debug
  cout << "Memory client sockets are up\n";
  #endif
  lock_init_clisocks();
  #ifdef SENSE_debug
  cout << "Lock client sockets are up\n";
  cout << "All sockets are up - waiting at barrier for the other nodes.\n";
  #endif
  arrive();
  // Start timing
```

```
  #ifdef SENSE_stats
  stats_start_timing();
  #endif
 // Do nothing
}

void sense_finish()
{
  cout << "Computation finished, waiting at barrier for lock-
manager and other";
  cout << " nodes\n";
  arrive();
  #ifdef SENSE_stats
  stats_end_timing();
  #endif
  cout << "Telling sockets to commit suicide\n";
  bar_kill_sockets();
  mem_kill_sockets();
  lock_kill_sockets();
  arrive();
  #ifdef SENSE_stats
  stats_print();
  #endif
  cout << "Done, bye now!\n";
  arrive();
}
```

D Bibliografi

Her bør det nævnes at [7] vides at være publiceret, men forfatterne har på trods af gentagne henvendelse undladt at fremsende mig de nødvendige detaljer.

Litteratur

[1] A. V. AHO, R. SETHI, AND J. D. ULLMAN, *Compilers: Principles, Techniques and Tools*, Addison-Wesley Publishing Company, Reading, Massachusetts (1986).

[2] C. AMZA, A. L. COX, S. DWARKADAS, P. KELEHER, H. LU, R. RAJAMONY, W. YU, AND W. ZWAENEPOEL, Treadmarks: Shared Memory Computing on Networks of Workstations, *IEEE Computer* **29**, 2 (Feb. 1996), 18–29.

[3] A. W. APPEL, *Modern Compiler Implementation in ML*, Cambridge University Press, Cambridge, United Kingdom (1998).

[4] J. BENTLEY, *Programming Pearls*, Addison-Wesley Publishing Company, Reading, Massachusetts (1989).

[5] B. N. BERSHAD, M. J. ZEKAUSKAS, AND W. A. SAWDON, The Midway Distributed Shared Memory System, *Proceedings of the IEEE CompCon Conference* (1993), .

[6] J. L. HENNESSY AND D. A. PATTERSON, *Computer Architecture, a Quantitative Approach*, 2 ed., Morgan Kaufmann Publishers Inc., San Francisco, California (1996).

[7] Z. HUANG, C. SUN, AND A. SATTAR, Updates Propagation based on Regional Locality in Distributed Shared Memory ().

[8] K. L. JOHNSON, M. F. KAASHOEK, AND D. A. WALLACH, CRL: High-Performance All-software Distributed Shared Memory, *Operating Systems Review* **29**, 5 (1995), 213–228.

[9] P. KELEHER, The Coherent Virtual Machine, Technical Report TR93-215, University of Maryland (1995).

[10] P. KELEHER, *Lazy Release Consistency for Distributed Shared Memory*, Ph.D. Thesis, Rice University, Houston, Texas (1995).

[11] P. KELEHER, The Relative Importance of Concurrent Writers and Weak Consistency Models, Technical Report CS-TR-3543, University of Maryland (May 1996).

[12] P. KELEHER, A. L. COX, S. DWARKADAS, AND W. ZWAENEPOEL, An Evaluation of Software-Based Release Protocols, *The Journal of Parallel and Distributed Computing* **29** (Sept. 1995), .

[13] L. LAMPORT, Time, Clocks and the Ordering of Events in a Distributed System, *Communications of the ACM* **21** (1978), 558–656.

[14] L. LAMPORT, Concurrent Reading and Writing of Clocks, *ACM Transactions on Computer Systems* **8** (1990), 305–310.

[15] K. LI, IVY: A Shared Virtual Memory System for Parallel Computing, *Proceedings of the 1988 International Conference on Parallel Processing* **II** (1988), 94–101.

[16] H. LU, Message Passing Versus Shared Memory on Networks of Workstations, M.Sc. Thesis, Rice University, Houston, Texas (May 1995).

[17] D. R. MUSSER AND A. SAINI, *STL Tutorial and Reference Guide*, Addison-Wesley Publishing Company, Reading, Massachusetts (1996).

[18] M. O. SVENDSEN, ORDER - an Efficient All-Software DSM system with a Dynamic Adaptable Unit of Coherence, Performance Evaluation, M.Sc. Thesis, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark (1996).

[19] A. S. TAENENBAUM, *Distributed Operating Systems*, Prentice-Hall, Inc., Upper Saddle River, New Jersey (1995).

[20] A. S. TAENENBAUM, *Computer Networks*, 3 ed., Prentice-Hall, Inc., Upper Saddle River, New Jersey (1996).