View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by NORA - Norwegian Open Research Archives

UNIVERSITETET I OSLO
Institutt for informatikk

Distribuert fellesminne i software over SCI

Henning Spjelkavik

Hovedfagsoppgave

1. mai 1999

Distribuert fellesminne i software over SCI

Henning Spjelkavik

1. mai 1999

«For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers in such a manner as to permit cooperative solution.»

Dr. Gene M. Amdahl, IBM Corporation, 1967, [2]

Sammendrag

Denne oppgaven diskuterer utnyttelse av SCI i et system for distribuert fellesminne implementert i software. Foruten å benytte SCI som et medium for rask meldingsutveksling, er også muligheten for å skrive direkte til minnet i en annen maskin via SCI-kortet testet. Konklusjonen er at man har mulighet til å oppnå bedre ytelse ved å utnytte SCI og den ekstra funksjonaliteten. Dette krever dog at programmereren har kjennskap til programmets datastrukturer og bruk av data for å få en effektiv utnyttelse.

Forord

Denne hovedoppgaven er skrevet til graden Candidatus Scientiarum ved Institutt for Informatikk, Universitetet i Oslo. Arbeidet med oppgaven skal utgjøre to av de totalt tre semestre hovedfagsstudiet varer.

Min veileder har vært Øystein Gran Larsen. Jeg vil rette en takk for en spennende oppgave, for konstruktive kommentarer og kritikk rundt oppgaven og et godt samarbeid.

Videre vil jeg gjerne takke Knut Omang som har vært til stor hjelp underveis og som ikke minst har inspirert meg faglig. Med sine grundige kunnskaper om SCI har muligheten for å diskutere med ham vært meget viktig.

I tillegg fortjener mine korrekturlesere i innspurten, Endre, mor og far en takk for rettelse av lumske skrivefeil som man ikke oppdager etter å ha lest sin egen tekst for mange ganger. Dessuten vil jeg takke mine studiekamerater ved Ifi, ogspesielt en takk til de som har hatt lesesalsplass samtidig med meg på Forskningsparken, for en hyggelig studietid.

Til slutt en takk til min venninne for å ha ha lest oppgaven med en helt annen synsvinkel og med det tvunget frem klarere formulereringer, og for oppmuntring og støtte under arbeidet.

Blindern, 1 mai 1999

Henning Spjelkavik

Innhold

1 Innledning **1**
1.1 Historie . . . 1
1.2 Organisering av oppgaven . . . 4

2 Metoder **7**
2.1 Bootstrapping . . . 7
2.2 Fremgangsmåte . . . 8
2.2.1 Endrede protokoller – *remote store* . . . 9
2.2.2 Implementasjon . . . 9
2.3 Oppsummering . . . 10

3 Bakgrunn **11**
3.1 Minnearkitekturer . . . 11
3.1.1 Uniform minnetilgang (UMA) . . . 11
3.1.2 Ikke-uniform minnetilgang (NUMA) . . . 12
3.2 Prosessoren - regneenheten . . . 15
3.3 Multiprosessor eller multicomputer . . . 16
3.3.1 De raskeste maskiner i 1998 . . . 18
3.3.2 Klassifisering av MIMD maskiner . . . 18
3.3.3 Multicomputer kan lage en abstraksjon av DSM . . . 19

4 Konsistensmodeller **21**
4.1 Innledning . . . 21
4.2 Sekvensiell konsistens . . . 22
4.2.1 Videreutvikling av modellen . . . 23
4.3 Prosessorkonsistens (PC) . . . 23
4.4 Svak konsistens . . . 24
4.5 Release konsistens . . . 24

5 Protokoller og CVM 27
5.1 Protokoller for DSM . . . 27
5.1.1 Generelle betraktninger og terminologi . . . 27
5.1.2 Ulike konsepter . . . 28
5.1.3 Lokalisering av side . . . 28
5.1.4 En protokoll for sekvensiell konsistens . . . 29
5.1.5 Ulike implementasjoner av RC . . . 29
5.1.6 Lazy release consistency (LRC) . . . 30
5.1.7 Automatic update release consistency (AURC) . . . 32
5.1.8 Home-based lazy release consistency (HLRC) . . . 33
5.1.9 Sammenligning . . . 33
5.2 Implementasjon av HLRC i CVM . . . 33
5.2.1 Sidefeil . . . 34
5.2.2 Leseforespørsel . . . 34
5.2.3 Synkroniseringspunkter . . . 34

6 Nettverk i eksisterende DSM 37
6.1 De ulike teknologiene . . . 38
6.1.1 HIPPI . . . 38
6.1.2 Memory Channel(MC) . . . 38
6.1.3 Scalable Coherent Interface . . . 40
6.2 Resultater . . . 40
6.3 Diskusjon – konsekvenser for DSM . . . 43
6.3.1 Kjernemodus vs brukermodus og interrupter . . . 43
6.3.2 Ingen kopiering («zero copy») . . . 44
6.3.3 Resultatene fra SoftFLASH . . . 45
6.3.4 Sidestørrelser . . . 46
6.4 Konklusjon . . . 47
6.4.1 Nyere DSM-systemer . . . 47
6.4.2 Konsekvenser . . . 47

7 Ytelse 49
7.1 En definisjon av ytelse . . . 49
7.1.1 Diskusjon . . . 50
7.1.2 Skalering . . . 50
7.1.3 Amdahls lov . . . 51
7.2 Tradisjonelle ytelsestester . . . 51

7.2.1 Kjerner . . . 52
7.2.2 Syntetiske tester . . . 52
7.2.3 Suiter . . . 52
7.2.4 Hva tester programmene? . . . 53
7.3 Testapplikasjoner . . . 53
7.3.1 Barrier . . . 54
7.3.2 Globalsum . . . 54
7.3.3 FFT . . . 54
7.3.4 Water . . . 54
7.3.5 Hint . . . 55
7.4 Rapportering av resultater . . . 55

8 Utvidelse – Nye protokoller 57
8.1 Relaterte prosjekter . . . 57
8.2 Plattform . . . 59
8.3 Kommunikasjon i CVM . . . 59
8.3.1 UDP . . . 60
8.3.2 MPI . . . 60
8.3.3 Organisering av kapittelet . . . 61
8.4 Første versjon - Naiv MPI/SCI v1.00 . . . 61
8.4.1 Løsning: et forenklet MPI-lag . . . 61
8.4.2 Diskusjon . . . 62
8.5 Andre versjon av MPI/SCI . . . 62
8.5.1 En annen mulig løsning: felles mottagerbuffer . . . 63
8.5.2 Optimalisering av meldingsutveksling . . . 64
8.6 Protokoller som utnytter SCI . . . 65
8.7 AURC over SCI . . . 66
8.7.1 Minneallokering . . . 66
8.7.2 Diskusjon . . . 66
8.7.3 Sideobjekt . . . 67
8.7.4 Protokoll . . . 67
8.7.5 Oppsummering . . . 68
8.8 AURC2 – Remote store ved skrivefeil . . . 69
8.8.1 Sidefeilhåndtering . . . 69
8.9 AURC3 – Remote store for utvalgte sider og dynamisk valg av hjemmenoder 70
8.9.1 Sidefeilhåndtering . . . 70

9 Eksperimenter og diskusjon **71**
9.1 Primitiver . . . 71
9.1.1 Diskusjon . . . 72
9.2 Forbedring fra UDP til MPI/SCI 1 . . . 73
9.2.1 Skalering fra én til to noder . . . 75
9.2.2 Oppsummering . . . 75
9.3 Skalering til flere noder – MPI/SCI 2 . . . 75
9.3.1 Barrier . . . 76
9.3.2 Globalsum . . . 77
9.3.3 FFT . . . 78
9.3.4 Water . . . 81
9.3.5 Hint . . . 82
9.4 Forskjellen mellom MPI/SCI 1 og MPI/SCI 2 . . . 85

10 Konklusjon og videre arbeid **87**
10.1 Oppsummering av resultater . . . 87
10.1.1 Lav forsinkelse i nettverket . . . 87
10.1.2 Utnyttelse av funksjonalitet i SCI . . . 88
10.1.3 Fordeling av sidenes hjemmenode . . . 88
10.2 Konklusjon . . . 88
10.3 Videre arbeid . . . 89
10.3.1 Minnebruk . . . 89
10.3.2 Ny arkitektur . . . 90

A Minnebusser **91**

B Resultater i tabellform **93**
B.1 Barrier . . . 93
B.1.1 Quantify-resultater . . . 93
B.2 Globalsum . . . 94
B.3 Resultater for FFT . . . 95
B.3.1 Statistikk for 100 iterasjoner av FFT . . . 97
B.4 Resultater for Water . . . 97
B.4.1 MPI/SCI-statistikk med Water . . . 97

Figurer

2.1 Arkitektur 9

3.1 Blokkdiagram for UMA (fra [33, side 20] 12
3.2 Blokkdiagram for SMP basert på Pentium Pro (omarbeidet etter [65, side 223] 13
3.3 Blokkdiagram for NUMA 14

4.1 Illustrasjon av sekvensiell konsistens 22
4.2 Alle tidligere *store* må være utført før en *load* 24
4.3 Kategorisering av ulike skriveaksesser 25

5.1 Illustrasjon av LRC 31
5.2 Illustrasjon av AURC 32

6.1 Arkitektur 39
6.2 Forsinkelser 41
6.3 Forsinkelser i «Memory Channel 2» 43

8.1 Tilstandsdiagram for AURC 1 68
8.2 Tilstandsdiagram for AURC 2 69

9.1 Primitiver 72
9.2 Kjøretid for programmene 74
9.3 Forbedring? 74
9.4 Kjøretid for barrier 76
9.5 Kjøretid for globalsum 78
9.6 Kjøretid for fft ved første forsøk 78
9.7 Kjøretid for fft etter modifisering 79
9.8 Kjøretid for water 82
9.9 Hint kurve for 2 noder 83
9.10 Hint kurve for 3 noder 84

9.11 Hint kurve for 4 noder 85

Tabeller

1.1 De tre raskeste maskinene pr. november 1998 ifølge Top500 Supercomputing sites[17] 2

6.1 Oversikt over forsinkelser (se kildehenvisninger på side 41) . . . 41
6.2 Fordeling av tiden som én remote store bruker 43
6.3 Hvor lang tid tar det å levere én melding? 44
6.4 Gjennomstrømning, reciever copy. Hentet fra [61] 45
6.5 Skalering av forsinkelse 46

9.1 Tid i μs for minnerelaterte primitiver 72
9.2 Resultater for MPI/SCI 1 74
9.3 «Forbedring» fra én til to noder 75
9.4 Resultater for Barrier med MPI/SCI 2 77
9.5 Statistikk for FFT 80
9.6 Statistikk for Water med 4 noder 83
9.7 Forbedring fra første til andre generasjon 86

Kapittel 1

Innledning

1.1 Historie

Siden datamaskinen ble introdusert på slutten av 1940-tallet, har det blitt utviklet stadig raskere maskiner og maskinsystemer. Man ønsker likevel å ha enda større regnekraft enn det én maskin kan tilby. Derfor kobles regneenheter sammen for å lage et kraftigere system.

De raskeste systemene, gjerne kalt for supercomputere eller superdatamaskiner, er og har vært meget dyre. Disse maskinene har vært utviklet spesielt for sitt formål – å utføre store mengder beregninger på kortest mulig tid – og utviklerne har derfor kunnet bruke store ressurser på å utvikle spesielle deler for sine prosjekter. Både nettverksteknologien og regneenhetene er ofte spesielle for prosjektet. Det har vært store variasjoner i hvordan enhetene blir benyttet og koblet sammen. Dette har i mange tilfeller også medført at det er store forskjeller i hvordan de skal programmeres. Ikke bare fra leverandør til leverandør, men også fra modell til modell. Dette gjør disse systemene meget dyre, ikke bare i innkjøp og drift, men også ved at de menneskene som utfører denne jobben må lære seg nye måter å programmere maskinen på, og deretter skrive om algoritmer for å utnytte maskinen[33, side 13].

Den teknologiske utviklingen innen integrert kretsteknologi har medført at prosessorer implementert i CMOS idag omtrent har tatt over markedet for regneenheter i datamaskiner. Med VLSI-teknologi kan man nå ha mange millioner transistorer på én brikke, og produsert i stort volum blir slike brikker meget rimelige.

Et annet viktig bidrag er at algoritmer og teknikker som ble utviklet til mainframes og supermaskiner allerede på 60-tallet for å utnytte parallellitet i kode idag igjen blir benyttet, slik som f.eks Tomasulos algoritme[56, side 251]. Den gang ble algoritmene implementert på kretskort, idag blir de laget i silisium i chipene.

De maskinene som idag regnes for å ha størst regnekraft, er derfor ba-

sert på prosessorer i CMOS. Ved Universitetet i Mannheim vedlikeholder de en liste over de maskinene som er skal være raskest i verden til å løse et ligningssystem fra ytelsestesten Linpack[17]. Som vist i tabell 1.1 ser vi at prosessorene er det som har blitt kalt skalare mikroprosessorer, og foruten at de brukes i noder for kraftige systemer, benyttes disse prosessorene også i vanlige arbeidsstasjoner og PCer.

Det tallet som brukes til å rangere systemene er fra den rapporterte maksimale ytelsen under Linpackkjøringen (R_{max}), her gjengitt med enhet Tflop/s.

	prosessor	maks Tflop/s
Intel ASCI	Intel Pentium Pro	1.338
Cray T3E	DEC Alpha 21164	0.891
IBM SP-2	PowerPC 604e	0.468

Tabell 1.1: De tre raskeste maskinene pr. november 1998 ifølge Top500 Supercomputing sites[17]

En annen type arkitektur som har vært meget vanlig, er basert på å bruke vektorprosessering i tillegg til skalarprosessorer [33, 10]. Flere japanske firmaer, som Fujitsu og Hitachi, har implementert vektorprosessorer i CMOS [10].

Blant de mest kjente produsentene, og i sin tid av mange regnet som den ledende av denne typen maskiner finner vi Cray, som gikk konkurs tidlig på nittitallet. Rettighetene ble delvis kjøpt opp av Silicon Graphics, og delvis av Sun. Det selges fortsatt Cray-maskiner, både basert på Crays tradisjonelle vektorkonsept og på vanlige skalarprosessorer.

Hvordan?

Det finnes mange måter å lage parallelle datamaskiner på. Én inndeling er å se på hvordan de ulike enhetene kommuniserer, og hvordan de har tilgang til minnet. Det er to hovedtyper. Enten er maskinen basert på en eller annen form for delt tilgang til minnet, noe jeg vil kalle for fellesminne i denne oppgaven («shared memory»), eller den benytter meldingsutveksling («message passing»).

Denne inndelingen er imidlertid ganske grov. Hvorledes, og på hvilket nivå i maskinen dette implementeres, varierer. Dette vil vi komme grundigere tilbake til, fordi det er viktig for forståelsen av distribuert felles minne i software.

Nettverk av arbeidsstasjoner

Tidligere var de raskeste maskinene basert på spesielle prosessorer og arkitekturer. Idag benyttes det i stor grad prosessorer som vi har i vanlige arbeidsstasjoner. Burde det ikke være mulig å koble sammen arbeidsstasjonene våre og få de til å oppføre seg som én kraftig maskin?

Et «Nettverk av arbeidsstasjoner» (NOW)[5] er nettopp denne idéen. Å benytte rimelige arbeidsstasjoner, koble disse sammen med nettverksteknologi og få en klynge med høy ytelse til lav pris er ønsket. Dette skal være utstyr som produseres i store volum og følgelig er nodene forholdsvis rimelige, spesielt sett i forhold til nodene i et tradisjonelt supersystem.

NOW har benyttet tradisjonelle protokoller (TCP/IP) over vanlige linker som Ethernet (10 MBit) og i det siste 100 Mbit Ethernet og ATM. Disse har relativt høye forsinkelser – flere ordner – sammenlignet med den interconnect-teknologien som benyttes på bakplanet i klynger og supermaskiner.

Kommunikasjonen over nettverket i et NOW er meldingsutveksling[5], fordi vanlige nettverk ikke tilbyr noen annen mulighet, som for eksempel å skrive rett inn i bestemte adresser i minnet til maskinen i den andre enden.

Med interconnect har man tradisjonelt ment en teknologi for sammenkobling som har høy båndbredde og lav forsinkelse. Ofte har de vært proprietære, og meget dyre. Det tradisjonelle nettverket Ethernet er kjennetegnet ved høy forsinkelse og lav båndbredde.

Distribuert fellesminne i software (DSM)

Fellesminne er en interessant modell for programmering av parallelle programmer, fordi den byr programmereren fordeler i forhold til tradisjonell parallellprogrammering ved hjelp av meldingsutveksling. Abstraksjonen av fellesminne gjør at prosessene kan bruke datastrukturer som aksesseres på vanlig måte i programmene, men som likevel er globale for alle prosessene i applikasjonen.

Ved vanlig meldingsutveksling må programmereren ta stilling til hva som skal sendes når og til hvem. Med DSM er kommunikasjonen transparent, og forhåpentlig vil dette medføre at utvikleren kan konsentrere seg mer om de parallelle algoritmer som brukes, og ikke med hvordan kommunikasjonen skal implementeres.

Kanskje er det også lettere å porte programmer fra kraftigere maskiner med fellesminne til denne modellen, enn til en meldingsutvekslingsmodell, siden abstraksjonen er den samme[3].

Teknikker for implementasjon

Distribuert fellesminne kan implementeres ved å bruke det virtuelle minnesystemet til å detektere bruk av minne som er delt. Dette kaller vi for et sidebasert DSM-system, og det er denne tilnærmingsmåten vi ser på i denne oppgaven[1].

Granulariteten, hvor fin oppdeling vi får av de delte sidene, avhenger av minnesystemet, men størrelsen kan ikke være mindre enn én fysisk side i maskinen. Det er typisk 4 kB eller 8 kB på de maskiner som benyttes.

Ved forsøk på å lese eller skrive til delte lokasjoner, må systemet sørge for å innhente korrekte data slik at leseoperasjonen returnerer det den skal, og dessuten sørge for at det ved skriving til en side ett sted ikke medfører at gamle data blir lest et annet sted.

Motivasjon

Feltet DSM ble foreslått og først implementert av Kai Li med prosjektet Ivy på slutten av åttitallet[48].

På 1990-tallet har flere prosjekter basert på disse idéene vært utført. Vi har tatt utgangspunkt i Erlichsons artikkel om SoftFLASH[20]. Artikkelen bygger på hans doktorgrad om SoftFLASH, og den beskriver det han opplevde som problemer med DSM på dette tidspunktet.

Erlichson implementerte et system for distribuert felles hukommelse, basert på FLASH-protokollen, utviklet ved Stanford for et distribuert hardware system[44]. Han benyttet HIPPI som interconnect[21].

Konklusjonen i artikkelen er at overhead, begrenset båndbredde og den høye forsinkelsen på nettverket, hindrer god ytelse for kommunikasjonsintensive applikasjoner. Likevel avslutter han:

> «Overall, this approach still appears promising, but our results indicate that large low latency networks may be needed to make cluster-based virtual shared-memory machines broadly useful as large-scale shared-memory multiprocessors.»

Scalable Coherent Interface (SCI) har tradisjonelt blitt omtalt som interconnect. Protokollen er designet for cache-coherens, og for å sitte på minnebussen. Hos oss sitter SCI-kortet på I/O-bussen og brukes i første omgang som nettverk.

Vi benytter SCI som er et nettverk med stor båndbredde og meget lav latency sammenlignet med de fleste nettverksteknologier som tidligere er benyttet. Derfor burde SCI være godt egnet for DSM. Vi ser nærmere på dette temaet i kapittel 6.

[1] Det finnes også mange andre tilnærmingsmåter. Peter Keleher vedlikeholder en oversikt over mange ulike prosjekter[41].

1.2 Organisering av oppgaven

I kapittel 2 vil vi presentere grundigere målet med oppgaven, og hvilke metoder som er benyttet for undersøkelsen.

Deretter vil de innledende kapitlene beskrive ulike måter å bygge maskiner med høy ytelse og høy grad av parallellitet. Vi vil klassifisere ulike arkitekturer på ulike nivåer og konsistensmodeller, og forsøke å sette dette i sammenheng med hvordan vi implementerer DSM.

Videre vil vi gå igjennom protokoller og implementasjon av disse i flere DSM-systemer. For å begrunne hvorfor vi tror at vi kan få bedre ytelse med SCI istedet for de nettverksteknologier som tidligere er brukt, vil vi se på hva som har vært flaskehalsen i andre systemer.

For å sammenligne ulike systemer må vi definere hva vi sammenligner. Vi vil derfor diskutere ulike forståelser av ytelse, hvordan dette kan måles, og ikke minst drøfte en del av svakhetene.

Dernest vil vi presentere våre implementasjoner. Vi vil se på både hvilke valg som ble gjort, og ikke minst hvorfor.

Til slutt går vi igjennom resultater fra ulike tester kjørt med de ulike protokollene, vi identifiserer problemer og ser på mulige forbedringer.

Kapittel 2

Metoder

Målet for oppgaven var å undersøke distribuert fellesminne i software (Software DSM) med bruk av SCI som nettverk.

Vi kunne enten starte med å bygge opp et nytt system fra bunnen, eller vi kunne utvide og tilpasse et eksisterende system, for å undersøke problemstillingen.

Én ulempe ved å starte forfra, er at det ville kunne ta mye tid å utvikle et fungerende system, før vi kunne komme til konkrete eksperimenter med SCI. En fare med dette var å miste fokus fra målet, som var ikke å studere hvordan utvikle et software DSM system, men å studere hvordan man kan benytte SCI i et slikt system.

For å kunne sammenligne ytelse og egenskaper med andre nettverk og protokoller, ville det også være gunstig å benytte et system som man allerede har mange resultater og testprogrammet til.

Vi har derfor valg å ta utgangspunkt i Pete Kelehers Coherent Virtual Machine (CVM)[38].

2.1 Bootstrapping

Coherent Virtual Machine

Peter Keleher deltok i utviklingen av TreadMarks[3]. Senere har han laget Coherent Virtual Machine – CVM, som er laget spesielt for å eksperimentere med nye protokoller[38]. CVM er fritt tilgjengelig under GNU lisens[23]. Mens TreadMarks idag er kommersielt tilgjengelig, er CVM ment for å eksperimentere, og med en lisens som tillater endringer og spredning av kildekoden.

Drivere for SCI – Slib

Det finnes flere ulike drivere for SCI-kortene, noe avhengig av hvilken maskinarkitektur vi ville velge å bruke. Dolphin Interconnect Solutions, som har levert kortene, leverer med én driver. I tillegg har det ved Institutt for Informatikk blitt utviklet en flerlags driver (kalt ICM) som har et grensesnitt som er implementert på flere plattformer[59], inkludert Solaris og Windows NT, men denne driveren fungerer bare med PCI buss i maskinen.

CVM er kun tilgjengelig på Solaris (SPARC) av våre platformer, og vi fikk den ikke til å kjøre under Solaris x86 under innledende forsøk. I det tilgjengelige klusteret ved instituttet[51] finnes det 4 UltraSPARC 1 koblet sammen med blant annet en SCI ring med SCI-kort basert på SBus.

Knut Omang ved instituttet har skrevet et bibliotek som tilbyr meldingsutveksling, opprettelse av delte variabler, låser med mere, over SCI til arbeidet med sin doktorgradsavhandling[55, 52]. Denne kan benytte både ICM og Dolphins driver for kommunikasjon med SCI-kortet.

Ved å velge Slib som grensesnitt mot SCI burde vi kunne bli med til en eventuell ny generasjon av UltraSparc basert på PCI-bus, noe som var en potensiell mulighet da prosjektet startet.

2.2 Fremgangsmåte

CVM har to arkitekturer for kommunikasjon mellom nodene. I første omgang benyttet CVM UDP/IP til kommunikasjon, men ved porting til en IBM SP-2 klynge (som er basert på IBM RS/6000 arbeidsstasjoner) ble det skrevet støtte for å benytte Message Passing Interface (MPI) i tillegg[50].

MPI tilbyr pålitelig levering av melding mellom noder, mens når man benytter UDP må applikasjonen selv sørge for å håndtere tap av datagrammer [73].

Koden for kommunikasjon over UDP/IP er derfor betydelig mer kompleks enn over MPI. Dette kan illustreres ved å nevne at det er ca 1000 kodelinjer for vanlig kommunikasjon og drøyt 550 for MPI.

Siden Slib tilbyr pålitelig meldingsutveksling over SCI så vi det som naturlig å starte med å skrive et relativt enkelt lag som mappet MPI-kall til Slib-kall. Figur 2.1 viser en lagdelt modell for kommunikasjonen i min modifiserte utgave av CVM.

CVM benytter et meget lavt antall MPI-kall, og kun funksjoner knyttet til initialisering, status, og pålitelig sending og mottak av meldinger. Denne løsningen var en effektiv måte å få et fungerende system som benyttet SCI.

Applikasjon
CVM
MPI/SCI
Slib
SCI driver
SCI kort

Figur 2.1: Arkitektur

2.2.1 Endrede protokoller – *remote store*

SCI tilbyr funksjonalitet som vanlige nettverksteknologier ikke tilbyr. Vi ønsket å undersøke hvordan vi kunne benytte denne funksjonaliteten i forbindelse med DSM.

For spesielt interessant regnet vi muligheten for *remote store*, det vil si at man kan skrive fra en maskin (A) direkte til minnet på en annen maskin (B) uten å belaste annet enn minnebussen og SCI-kortet på maskin B. For å utføre skriveoperasjonen gjør man som vanlig når man skriver til minne via instruksjonen *store* eller tilsvarende på andre arkitekturer.

Dette settes opp på forhånd ved at deler av minnet på maskin B deles ut via SCI-kortet. På maskin A kan vi sette opp en mapping fra en virtuell adresse i maskin A til det delte minnet på maskin B.

Vi har brukt mye tid på å vurdere og teste ulike måter å implementere og utnytte remote store. For å undersøke hvordan en protokoll med remote store kunne implementeres, har vi laget mange små testprogrammer som bruker driverene på ulike måter, kombinert med måling av tids- og ressursbruk.

2.2.2 Implementasjon

Resultatene av ulike undersøkelser, kombinert med krav til protokollen vi har ønsket å lage, medførte at vi valgte å bruke systemkallet mmap(3) direkte mot Dolphins driver for å håndtere oppsettet av hvor og hvordan de ulike sider i minnet skulle peke, enten til en annen node, eller lokalt. Dette er det nivået som ligger nærmest driveren, men som likevel lar oss gjøre det vi ønsker fra brukermodus.

Det finnes klare ulemper ved dette valget. Denne løsningen er kun portabelt til så lenge denne driveren finnes til ønskede kombinasjonen av operativsystem og hardware, og kallene har den samme semantikken.

Videre var det også flere valg. Først testet vi å benytte en filpeker (file

descriptor) pr. side som kunne brukes med remote store. Dette ville kunne lette håndteringen når sider skulle settes opp til å peke til andre noder. En slik løsning ville nok være lettere å porte til nye drivere, fordi hver side behandles isolert, og ikke er avhengig av hvordan de andre sidene settes opp.

Det største problemet var begrensninger i antall åpne filer, både i operativsystemer og for driveren. Dette låste seg ved alt for få sider (under hundre), og løsningen ble forkastet.

Vi endte opp med å velge å mappe hele CVM-minnet fysisk via SCI-driveren og tilbake til det virtuelle minnet. I forhold til de andre nodene mappet vi inn alt remote minne fysisk. En klar ulempe er at dette bruker mye av adresserommet i SCI, noe som vil begrense antallet delte sider totalt i systemet. Dette vil igjen medføre en begrensning i hvor stort system vi kan bygge.

Når vi skal sette opp bruk av minne på andre maskiner, trengs det kun et munmap/mmap-par. Dette tar mellom 50 og 100 mikrosekunder, se i avsnitt 9.1, noe som er raskere enn å hente over en kopi av siden.

Vi har benyttet denne funksjonaliteten i flere protokoller som ligner Automatic Update Release Consistency [1] (AURC) [11, 79]. Dette er en hjemmenodebasert protokoll med release consistency som tillater flere skrivende noder, og som bruker en form for remote store for å oppdatere hjemmenoden.

Et annet alternativ ville være å la Slib håndtere delte sider, og abstrahere oss fra driveren. Problemet var at Slib ikke har nødvendig funksjonalitet for å flytte hjemmenoden til delte variabler (i.e. sider). I alle tilfeller måtte det altså skrives et nytt grensesnitt mot driveren for å kunne manipulere sidetabellene slik vi ønsket.

2.3 Oppsummering

Vi håpet å kunne få bedre ytelse og god skalering med testprogrammene. Det vil si lavere kjøretid enn ved å bruke det vanlige nettverket. Med god skalering mener vi ihvertfall at kjøretiden synker når vi øker antall noder. Idéelt sett bør kjøretiden halveres når antall noder dobles. En alternativ skalering er at utført arbeid dobles ved doblet antall noder med konstant kjøretid.

[1] home based LRC, multiple writers.

Kapittel 3

Bakgrunn

Vi vil først studere ulike former for tilgang til fellesminne i multiprosessorer, og deretter se på to ulike typer prosessorer. Til slutt skal vi diskutere kort forskjellen på en multiprosessor og en multicomputer.

3.1 Minnearkitekturer

Maskiner eller systemer som benytter flere enn én prosessor, kan deles inn grovt etter hvordan kommunikasjonen mellom de ulike prosessorene utføres. Vi snakker vanligvis i dag om to metoder. Enten er maskinen basert på meldingsutveksling, eller så er den basert på en eller annen form for fellesminne. Denne inndelingen kan brukes på flere lag i maskinen, og det er fullt mulig at maskinvaren benytter fellesminne, men at programmereren ser et meldingsutvekslingsgrensesnitt, eller motsatt.

En maskin med fellesminne vil for eksempel ønske å støtte gjenbruk av eksisterende kode basert på Message Passing Interface (MPI), og kan da implementere meldingsutveksling. Det kan gjøres meget effektivt ved å utveksle pekere til bufre i fellesminne for meldinger som skal overføres.

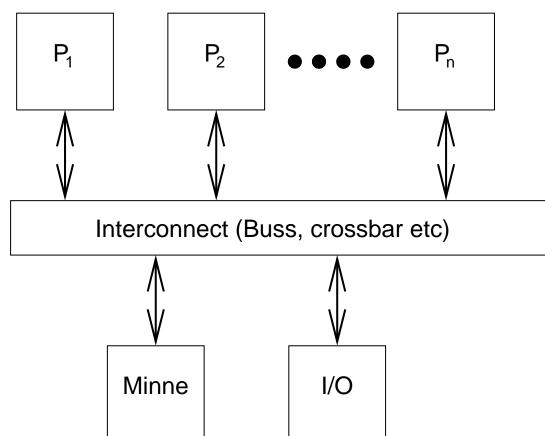
3.1.1 Uniform minnetilgang (UMA)

En vanlig PC består vanligvis av én prosessor (CPU), minnebrikker og et antall periferienheter (I/O) koblet sammen ved hjelp av databusser. For å øke prosesseringskraften kan man sette flere prosessorer sammen i en maskin, og la disse prosessorene dele de andre ressursene.

I UMA-modellen for multiprosessor, deles tilgangen til minnet mellom alle prosessorene. Alle prosessorer har lik tilgangstid til minnet, og vi sier derfor at delingen er uniform. Dog kan hver prosessor ha sin egen cache. Kommunikasjonen skjer gjerne via en buss. Dette har betydning for hvor godt denne

tilnærmingsmåten skalerer.

Slike systemer kalles tett koblede fordi ressursene deles i så stor grad. Når alle prosessorer har lik tilgang til alle periferienheter kalles systemet for en symmetrisk multiprosessor (SMP). En skjematisk fremstilling av «Uniform Memory Architecture» (UMA) er vist i figur 3.1.



Figur 3.1: Blokkdiagram for UMA (fra [33, side 20]

Mange produsenter har multiprosessorutgaver av sine uniprosessorer, og en slik utvidelse kan derfor være relativt rimelig.

Et eksempel: Intel Pentium Pro

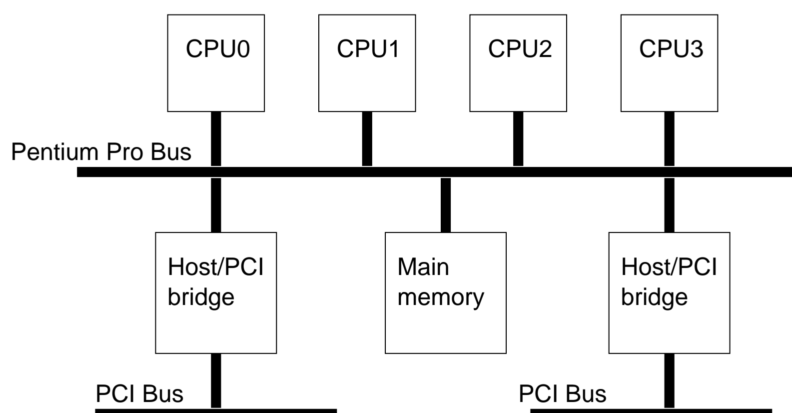
Et konkret eksempel på en produsent som tilbyr både vanlig uniprosessor og symmetrisk multiprosessering er Intels arkitektur for Pentium Pro. Arkitekturen er laget for at inntil fire prosessorer kan kobles sammen på denne måten[65, side 223]. Det finnes i dag leverandører som bygger maskiner med enda flere prosessorer, 8 og 10 veis, men det er foreløpig usikkert om ytelsen skalerer[24].

Blokkdiagrammet i figur 3.2 illustrerer arkitekturen.

3.1.2 Ikke-uniform minnetilgang (NUMA)

Et system med fellesminne hvor tilgangstiden til et ord i minnet varierer med hvor dette ordet er lokalisert, benytter ikke-uniform minnetilgang. Denne modellen kalles for «Non-uniform Memory Architecture» (NUMA).

Det er vanlig i slikt design at minnet er distribuert sammen med hver prosessor. Hver prosessor har eget minne, i tillegg til tilgang til de andre prosessorenes minne. Minnet, både lokalt og lokalisert hos andre prosessorer, er



Figur 3.2: Blokkdiagram for SMP basert på Pentium Pro (omarbeidet etter [65, side 223]

tilgjengelig i et globalt adresserom.

Tilgangstiden er lavere til minne som er lokalt for prosessoren enn til minne hos en annen prosessor, på grunn av forsinkelsen i sammenkoblingsnettverket.

Dessuten kan man ha rene minnemoduler – det vil si minne uten lokal prosessor – koblet til sammenkoblingsnettverket (interconnect). Dette minnet kalles da for globalt fellesminne. Figur 3.3 demonstrerer konseptet.

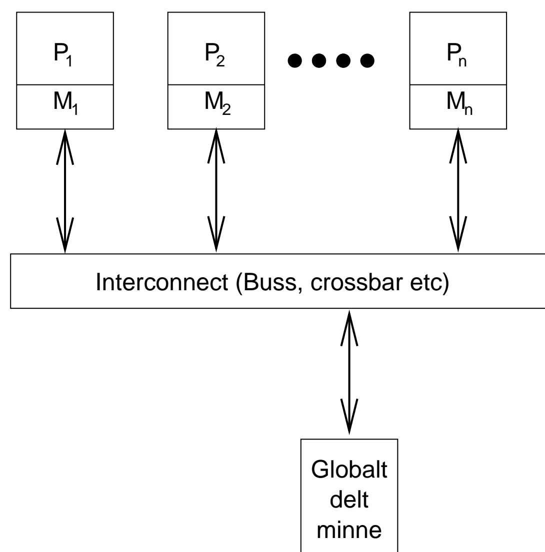
Det er idag vanlig at prosessorer har cache for å forsøke å skjule den store forskjellen i hastighet mellom prosessor og minne. I den arkitekturen som er forklart over, antar vi at cachene er koherente innenfor visse betingelser som bestemmes ved arkitekturen.

Dersom ikke-lokalt minne caches, så må det finnes en protokoll for å sørge for at gamle, ugyldige data ikke blir benyttet. Slike protokoller kalles for koherensprotokoller.

NCC-NUMA

Et system basert på en NUMA arkitektur, men som ikke har cache-koherens, kaller vi for «Non-cache coherent NUMA» (NCC-NUMA). Dette betyr at minne som finnes i cache lokalt i flere noder, kan bli inkonsistente dersom én node oppdaterer sin lokale kopi.

Dette i motsetning til en cache-koherent maskin (CC-NUMA), hvor lokale kopier enten vil bli ugyldiggjort eller oppdatert ved endringer. I systemer hvor alle oppdateringer mot minne går via én felles minnbuss, kan koherens implementeres ved at de lokale cachene lytter («snooper») på minnebussen. Det vil si at de undersøker alle skriveoperasjoner, og dersom adressen det skri-



Figur 3.3: Blokkdiagram for NUMA

ves til finnes lokalt, så oppdateres den lokale cachen, enten med nye data, eller ved å gjøre kopien ugyldig.

Diskusjon

Ved en bussarkitektur er båndbredden låst, men det er rimelig å koble til en ekstra node på bussen. Imidlertid øker faren for at bussen blir mettet, og ikke tåler et større antall noder.

Dersom man benytter en svitsj og kobler til en ny node med en ekstra port, vil båndbredden øke, men denne ekstra enheten tilkoblet er dyr.

I en ring vil en ekstra tilkoblet node øke den totale båndbredden, men siden avstandene blir større vil forsinkelsen (latency) øke.

Grovt sett er UMA best egnet for en buss, siden tilgangstiden skal være lik uansett hvor data befinner seg, men man kan også tenke seg mer kompliserte løsninger.

Svitsjer, ringer og ulike sammensetninger blir mer aktuelle når vi ønsker å skalere til et stort (hundretalls) noder. Da er nok også NUMA arkitekturer enklere å implementere enn UMA.

Ett eksempel på en slik maskin er (SGI) Cray T3E som skalerer til inntil 2048 CPUer (DEC Alpha EV5) med et logisk globalt adresserom, men fysisk distribuert minne med inntil 2GB pr. CPU. Den benytter ikke-uniform minnetilgang (NUMA). Sammenkoblingen er i en tredimensjonal kube med GigaRing interconnect[26].

3.2 Prosessoren - regneenheten

Vi kan dele inn prosessorene i to hovedtyper etter hvilken enhet de bruker i sine matematiske operasjoner: skalare og vektorbaserte prosessorer.

Skalare mikroprosessorer

Den dominerende typen prosessorer i dag kalles for skalare mikroprosessorer. De produseres i meget stort volum, noe som igjen har medført at de er meget rimelige. Til tross for stadige påstander om at metoden vil nå sin begrensning i ytelse, ser vi fortsatt en stadig forbedring i ytelsen fra år til år.

Den grunnleggende enheten som man regner med i en skalarprosessor vil være enten ett heltall eller ett flyttall. Størrelsen på tallet har øket, og idag er det vanligst med 32-bits heltall, men med 64-bits på vei inn (Alpha, Merced, UltraSparc).

Blant den skalare typen prosessorer finner vi Intels serie fra 8086 og opp til dagens Pentium II med andre implementasjoner av denne arkitekturen (AMD K5, Cyrix 6x86 osv), MIPS, Sparc og mange andre.

Introduksjonen av VLSI-teknologi, og kontinuerlige forbedringer i denne teknologien, har gjort det mulig å benytte et meget stort antall transistorer på én silisiumbrikke, og med økende klokkefrekvens.

Vektorbaserte prosessorer

Mens supermaskinene på 1960-tallet utnyttet muligheter for pipelining og dynamisk skedulering av instruksjoner, ble vektorprosessering den dominerende teknologien blant de raskeste maskiner på 1970-tallet. Vektorprosessorer opererer på en serie av heltall eller flyttall, altså på vektorer, i motsetning til skalareprosessorer som brukte ett og ett tall.

I mange vitenskapelige beregninger regner man på vektorer i matriser. Istedet for en løkke for å beregne hvert element i vektoren, så beregnes her alle elementene samtidig. Antall flyttallsoperasjoner pr klokkeperiode kan dermed bli meget høyt, og dette har vært nøkkelen til denne teknologiens tradisjonelt høye ytelse. Man får altså en mye høyere grad av parallellitet hvis man klarer å utnytte vektorer.

Maskiner fra Cray-selskapene er kanskje de mest kjente basert på dette konseptet. Idag leverer japanske selskaper som Hitachi, Fujitsu og NEC maskiner med vektorprosessorer, implementert i CMOS.

Normalt har man ikke benyttet vektorprosessoren alene. Istedet har man levert en eller flere vektorprosessorer sammen med en skalarprosessor som hovedprosessor. Denne kan så programmeres til å sette opp og bruke vektorprosessoren etter programmererens ønske. Det vil her være et kompromiss

mellom den kostnad det er med å sette opp vektorenheten, i forhold til å utføre de samme beregningene i en løkke.

De to maskinene som blir regnet som de første vektorprosessorene var CDC Star-100 og TI ASC[56, side B-38], begge annonsert i 1972. Maskinene hadde en relativt treg skalarprosessor, og jobbet rett mot minnet, uten registre. Oppstartskostnaden ved å benytte vektorenheten var meget høy, skjæringspunktet for når det lønte seg å benytte vektorprosessoren kunne være på over 50 elementer.

Seymour Cray ved Cray Research annonserte CRAY-1 i 1976, og denne hadde en meget rask skalar prosessor med en klokkeperiode på 12.5 ns (80 MHz), og meget lave oppstartskostnader for bruk av vektorer. Denne brukte registre i beregningene, istedet for å jobbe rett mot minnet. Etterfølgende systemer fra Cray, som CRAY X-MP, delvis CRAY-2 og CRAY Y-MP har blitt regnet som ledende da de ble annonsert.

Konvergens?

Vi har allerede sett at mikroprosessorene har tatt opp idéer fra supermaskinene som ble bygget på 1960-tallet med hensyn til pipelining og dynamisk skedulering.

Nye instruksjoner for multimedia og beregninger i tre dimensjoner i de skalare mikroprosessorene, som AMDs 3D-Now og Intel KNI og til dels MMX (Matrix Math Extensions, eller Multi Media Extensions som markedsføringen kalte det) har et innslag av vektorer.

Intels nyeste prosessor våren 1999, Pentium III, har som omtrent eneste nyhet nye instruksjoner, forkortet KNI, som kan utføre beregninger på fire elementers vektorer, mens AMD K6-2 som ble lansert våren 1998 kunne utføre operasjoner på to elementers vektorer. Ulike RISC prosessorer har hatt dette enda lenger, for eksempel finnes det såkalte multimediaoperasjoner i UltraSparc-arkitekturen, i tillegg til 64 bits *load*og *store*.

Kanskje ser vi en konvergens ved at de skalare mikroprosessorene også begynner å plukke opp idéer fra 1970-tallets vektorprosessorer?

3.3 Multiprosessor eller multicomputer

Michael Flynn introduserte i 1972 en klassifikasjon av datamaskinarkitekturer basert på instruksjons- og datastrømmer. Konvensjonsjonelle énprosessors maskiner kan klassifiseres som SISD (single instruction stream over a single data stream).

Dagens parallelle maskiner er MIMD (multiple instruction streams over multiple data streams), og disse henter instruksjoner og data fra lokalt minne.

Dette kan være vanlige mikroprosessorer satt sammen til en parallell maskin.

SIMD (single instruction stream over multiple data streams) er gjerne brukt ved å la én kontrollenhet styre et større antall passive prosesseringselementer (PE) med eget minne ved å kringkaste instruksjonene til elementene. De er ikke nødvendigvis generelle prosessorer, heller spesialtilpasset for oppgaven.

Hennessey og Patterson hevder at det ikke er bygget kommersielle maskiner basert på MISD[56, side 637], mens Hwang mener at systolske array kan klassifiseres som MISD fordi en felles datastrøm styres gjennom at array av prosesseringselementer som kan utføre ulike instruksjoner[33, side 11]. Man kan kanskje drøfte om systoliske arrays har vært et kommersielt alternativ.

SIMD kan gjøres ved å la en sentral enhet dekode instruksjonene, og på lik linje med en vektormaskin utføre operasjonene enten på den lokale skalarprosessoren, eller utføre vektoroperasjoner. Forskjellen er at vektoroperasjoner distribueres til et antall prosesseringselementer (PE) som er passive ALUer som utfører instruksjoner som den sentrale kontrollenheten bestemmer.

For å løse de største problemer vil man koble sammen flere tusen prosessorer eller prosesseringselementer i noe som har blitt kalt Massive Parallell Processing (MPP).

Ett eksempel på en slik maskin er Goodyear MPP med blant annet 16384 prosesseringselementer basert på SIMD modellen. Den brukte en Digital PDP-11 minimaskin til vanlige beregninger (Program and Data Managment Unit og en VAX11/780 som styremaskin (Host), i tillegg til selve arrayet av prosesseringselementer og en egen kontrollenhet for disse[8]. Andre eksempler er MasPar MP-1 og Thinking Machines CM-2 og CM-5, som vi kommer kort tilbake til.

Hva er en multiprosessor?

En maskin som betegnes multiprosessor er en maskin med flere prosessorer som er koblet tett sammen. Med tett sammenkobling menes at ressurser, som for eksempel I/O, deles i høy grad. Den fysiske sammenkoblingen skjer gjerne med en buss, svitsj, ring eller spesielle nettverk[33, side 19].

Det enkleste er en maskin med to prosessorer koblet symmetrisk (SMP) med buss som kommunikasjon mot felles I/O og minnesystem (UMA). Sentralisert fellesminne arkitektur kan vi kalle en slik løsning hvor alt minnet deles og er samlet på ett sted. Denne arkitekturen er vanskelig å skalere til et stort antall noder.

Med mer avanserte konfigurasjoner med dyrere, og mer avanserte interconnect teknologi, kan man lage multiprosessorer med et meget stort antall prosesseringselementer eller CPUer hvor minnet er fysisk distribuert. Dette kalles også for skalerbare distribuerte multiprosessorer.

Hva er en multicomputer?

En multicomputer består av autonome maskiner som på en eller annen måte er koblet sammen slik at de fungerer som eller kan abstraheres som én maskin.

Disse autonome maskinene kan godt være multiprosessorer. Scali leverer sine klynger med doble Pentium II-prosessorer, koblet sammen med SCI som interconnect.

Kommunikasjonen i en multicomputer skjer vanligvis via meldingsutveksling, siden det er den kommunikasjonsmodellen som støttes av vanlig nettverksteknologi.

3.3.1 De raskeste maskiner i 1998

De største maskiner, i form av antall prosessorer, har blitt kalt for Massive Parallell Processors (MPP)-maskiner, og betegnelsen har gjerne vært forbeholdt maskiner med mer enn 100 eller kanskje 1000 prosessorer.

Betegnelsen var meget populært på åttitallet, og på begynnelsen av nittitallet, gjerne i form av SIMD arkitektur, for eksempel med maskiner fra Thinking Machines CM-2, tidligere nevnte Goodyear MPP og MasPar MP-1. Thinking Machines gikk bort ifra en ren SIMD arkitektur med sin neste maskin, CM-5, blant annet fordi maskinen kun kunne kjøre én veldig stor jobb om gangen. Dessuten var skalarytelsen dårlig, noe som medførte at den kun kunne brukes til store, parallelle, og av andre grunner grovkornede parallelle jobber[9]. Dette gjorde maskinen for lite kost-effektiv.

Idag er de raskeste MPP-maskiner stort sett basert på mikroprosessorer som er hyllevare. CM-5 var basert på kortet til en Sun SparcStation 1, Cray T3E på DEC Alpha og Intel Paragon på Pentium Pro. Dessuten finnes det fortsatt en del vektorprosessorer på topp-500 listen , først og fremst representert ved japanske maskiner fra NEC (SX-4), Hitachi (SR2201/1024, pseudovektor) og Fujitsu (VPP700/116), men også Cray T90 (forøvrig basert på ECL istedet for CMOS). En oversikt over ulike typer superdatamaskiner finnes i [69].

3.3.2 Klassifisering av MIMD maskiner

Vi kan dele inn Flynns MIMD maskiner etter 2 hovedprinsipper; i multiprosessorer eller multicomputer, og etter om minnet er distribuert eller sentralt. Når en skalerer en maskin til mange noder, blir et sentralt minne en flaskehals[9]. Nøkkelen til skalerbarhet er derfor distribuert minne.

Man kan benytte kommunikasjonsarkitekturen til å vise forskjellen mellom en multiprosessor og en multicomputer, og bruke dette i en definisjon. Bell definerer at en multicomputer benytter eksplisitte meldinger for å aksessere

minne på andre noder, mens han definerer en multiprosessor som en maskin som har ett felles adresserom, altså fellesminne[9].

3.3.3 Multicomputer kan lage en abstraksjon av DSM

Vi har tidligere sett hvordan meldingsutveksling kan implementeres over fellesminne, for å kjøre programmer som er skrevet for eksplisitt meldingsutveksling. Kan vi gjøre det motsatte?

La oss anta at vi har et parallelt program som har deklarert noen variabler som globale. Dette programmet kjører vi så på to ulike maskiner med et nettverk i mellom, en slags multicomputer.

Minnesystemet i en maskin håndterer oversettelse fra virtuelle adresser (som programmet bruker), til fysisk minne i maskinen. Tilknyttet denne oversettelsen er det også ulike beskyttelsesmuligheter; slik at man kan hindre henholdsvis all tilgang til siden, kun gi lesetilgang eller full tilgang. Dersom programmet forsøker å utføre en ulovlig tilgang, kalles dette en sidefeil, og operativsystemet vil enten kalle en håndterer for dette, eller terminere programmet.

Ved å hekte oss på sidefeilshåndtereren og ved å manipulere beskyttelsestabellen kan vi vite når programmet prøver å aksesserere de globale variablene. Hvis vårt system vet hvilken node som sist skrev til denne variabelen, så kan vi innhente de riktige verdiene, og rette opp beskyttelsen, før vi lar programmet fortsette der det stoppet.

Med denne metoden kan vi lage en fellesminneabstraksjon over maskiner som tilbyr meldingsutveksling. Det har den fordel at vi kan skrive programmer som bruker fellesminne, istedet for å bruke eksplisitt meldingsutveksling. Eksisterende programmer skrevet for multiprosessorer bør med minimal innsats kunne kjøre på et slikt system. Vi kaller dette for distribuert fellesminne implementert i software («software distributed shared memory»).

For at dette skal fungere særlig bra, er vi avhengig av raske nettverk med relativt bra båndbredde, men ikke minst lav forsinkelse, en konsistensmodell og protokoller for utveksling av data[19].

Kapittel 4

Konsistensmodeller

4.1 Innledning

Vanlige uniprosessor datamaskiner tilbyr en enkel og intuitiv minnemodell til programmereren. En *load* skal alltid returnere den siste verdien skrevet til denne adressen i minnet, og verdien skrevet med en *store* blir returnert av alle etterfølgende *load* fra denne adressen frem til en ny *store* på samme adresse. Modellen er at minnet oppdateres sekvensielt etter rekkefølgen av instruksjoner i kildekoden.

Maskinen behøver imidlertid ikke å utføre operasjonene nøyaktig i den spesifiserte rekkefølgen. Både kompilatorer, moderne prosessorerer og minnesubsystemet kan bytte om på rekkefølgen for å oppnå høyere ytelse, bare resultatet er det samme som om instruksjonene hadde blitt utført sekvensielt, dvs at kontroll- og dataavhengigheter blir respektert. Dette gjøres ved flere lag med cacher, buffere, og pipelineteknikker og andre teknikker og algoritmer[56, 35].

For multiprosessorsystemer er dette vanskeligere. En del av begrepene blir mer uklare. Hva menes for eksempel med «siste verdien skrevet» når flere prosessorer har adgang til det samme minnet? I en NUMA arkitektur vil dessuten flere *store*-instruksjoner fra ulike prosessorer/noder til samme lokasjon bruke ulik tid på å nå frem til minnet[33].

Ett av aksiomene for distribuerte systemer er at vi ikke kan forutsette en global klokke[74]. Vi kan vanskelig vite når to minneoperasjoner mot samme adresse fra to noder ble sendt i forhold til hverandre. Rekkefølgen operasjonen mottas av minnemodulen kan være motsatt av programrekkefølgen. Ved kommunikasjonsproblemer, eller dersom den ene prosessoren jobber mot lokalt minne, og den andre kommer via et nettverk, vil vi kunne ha flere ordner ulik gangtid.

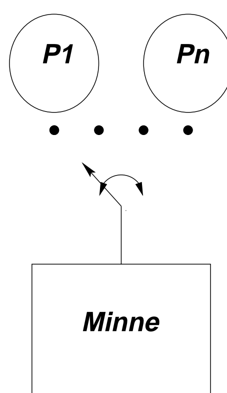
Dette skulle begrunne at vi trenger klare modeller som entydig definerer konsistens, slik at vi kan få deterministiske resultater.

4.2 Sekvensiell konsistens

Det er etablert flere modeller for konsistens i flerprosessorsystemer. Modellene spesifiserer i hvilken rekkefølge handlinger (events) utført av én prosessor kommer til syne hos de andre prosessorene.

Den første modellen vi vil nevne, er den klassiske «sequential consistency», som er definert av Leslie Lamport i [45]. Denne kommer nærmest vår intuitive modell for en uniprosessor.

> En multiprosessor er sekvensielt konsistent hvis og bare hvis resultatet av en hvilken som helst utførelse er den samme som om operasjonene til alle prosessorene ble utført i en sekvensiell orden, og at operasjonene på hver individuelle prosessor kommer til syne i denne sekvensen i den orden den er spesifisert av sitt program.



Figur 4.1: Illustrasjon av sekvensiell konsistens

Vi kan illustrere dette ved å tenke oss et minne med kun én port, og n prosessorer som ønsker tilgang. Tilgangen styres av en svitsj som alternerer mellom prosessorene tilfeldig, se figure 4.1.

Denne modellen er konseptuelt enkel – den er lett å forstå, kanskje fordi den minner mest om vår vanlig modell. Dessverre ser det ut til å være vanskelig å implementere denne modellen effektivt, fordi den legger store restriksjoner på mulighetene til å utnytte optimaliseringsmuligheter som allerede finnes i maskinvaren[25].

Mark Hill argumenterer for at fordelen med å programmere med sekvensiell konsistens oppveier den, etter hans mening, minkende forskjellen mellom sekvensiell konsistens og svakere konsistensmodeller. Hans konklusjon er at multiprosessorer bør støtte sekvensiell konsistens[31].

4.2.1 Videreutvikling av modellen

En noe mer formell beskrivelse ble etablert av Dubois m.fl[18]. Begrepene som de definerer er meget nyttige når vi skal diskutere andre, mer løse modeller også.

Definisjon 1 *Performing a memory request*

A load by P_i is considered performed with respect to P_k at a point in time when the issuing of a store to the same address by P_k cannot affect the value returned by the load. A store by P_i is considered performed with respect to P_k at a point in time when an issued load to the same address by P_k returns the value defined by this store. (or a subsequent store to the same location). An access is performed when it is performed with respect to all processors.

Definisjon 2 *Performing a load globally*

A load is globally performed if it is performed and if the store that is the source of the returned value has been performed.

Med disse to definisjonene kan vi se på følgende betingelse for sekvensiell konsistens, noe endret etter [18] i [25], og illustrert ved figur 4.2:

Betingelse 1 *Sufficient conditions for sequential consistency*

- *before a load is allowed to perform with respect to any other processor, all previous load accesses must be globally perfomed and all previous store accesses must be performed, and*
- *before a store is allowed to perform with respect to any other processor, all previous load accesses must be globally performed and all previous store accesses must be performed.*

4.3 Prosessorkonsistens (PC)

Prosessorkonsistens[25] er en mellomting mellom sekvensiell og svak konsistens, som forklares i seksjon 4.4. Denne blir benyttet i betingelsene for release-konsistens.

PC krever at store fra én prosessor kommer til syne i riktig rekkefølge, men den sier ikke noe om hvordan store fra to ulike prosessorer synes hos hverandre eller én tredje prosessor. En annen endring er at en load etter en store kan passere og hente verdien, uten å måtte vente på at store er utført globalt (globally performed).

Figur 4.2: Alle tidligere *store* må være utført før en *load*

4.4 Svak konsistens

Om vi innfører synkroniseringspunkter i programmene, og definerer minneoperasjonene i forhold til disse, får vi en svakere modell.

En optimalisering som vil bryte med sekvensiell konsistens, er pipelining av *store*-operasjoner. Etter betingelsen 1, må enhver *store* være fullført før en ny *store* kan utføres. Med synkronisering må programmereren, eventuelt en kompilator, identifisere og legge inn tilstrekkelig med synkronisering.

Svak konsistens (weak consistency) ble definert av Dubois et al i [18]:

Betingelse 2 *Conditions for Weak Consistency*

- *before an ordinary* load *or* store *access is allowed to perform with respect to any other processor, all previous synchronization accesses must be performed, and*
- *before a synchronization access is allowed to perform with respect to any other processor, all previous ordinary* load *and* store *accesses must be performed, and*
- *synchronization accesses are sequentially consistent with respect to one another.*

4.5 Release konsistens

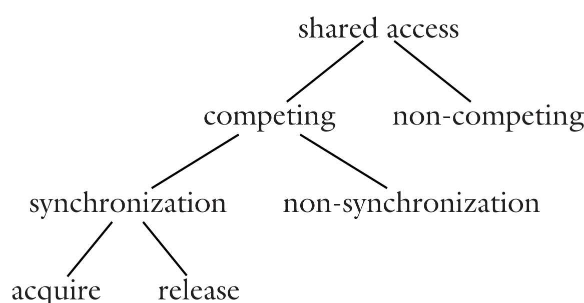
I [25] presenterer forfatterne en modell de kaller for «release consistency».

To aksesser er i konflikt dersom de er mot samme adresse, og minst én av dem er en *store*. Gitt at disse to aksessene utføres fra to prosessorer, a_1 og a_2, vil her ha et race. Dette paret av operasjoner kalles et konkurrerende par.

I et produsent-forbruker (producer-consumer) eksempel, vil vi ofte bruke en flaggvariabel for å signalisere til den andre prosessen når det finnes data. Andre ganger vil man beskytte oppdateringen av en datastruktur ved hjelp av lock og unlock-operasjoner. Slike aksesser kalles for synkroniseringsaksesser.

Disse har to viktige karakteristikker:

- Det er konkurrerende aksesser, ved at den ene prosessoren leser en variabel, mens den andre vil skrive til den,
- dessuten brukes disse aksessene til å styre programflyten slik at andre aksesser ikke blir i konflikt.



Figur 4.3: Kategorisering av ulike skriveaksesser

De ulike typene skriveaksesser kan kategoriseres slik som vist i figur 4.3.

En acquire-aksess skaffer tilgang til et kritisk område (i.e. lock) mens release går ut (i.e. unlock). Dersom vi vet hva de ulike aksesser gjør, kan vi utnytte dette i en løsere minnemodell. Artikkelen innfører begrepet «Properly labeled (PL) programs», og viser at programmer hvor konkurrerende aksesser er markert, kan utføres med release consistency og likevel være ekvivalente med sequential consistency.

At et program er PL vil si at et tilstrekkelig antall med aksesser er merket som acquire eller release, slik at for alle lovlige interleavinger av aksesser, er hvert par av aksesser som er i konflikt separert av release/acquire. Ifølge beviset skal resultater da være ekvivalent med sekvensiell konsistens[25].

Betingelse 3 *Conditions for Release Consistency*

- *before an ordinary* load *or* store *is allowed to perform with respect to any other processor, all previous acquire accesses must be performed, and*
- *before a release access is allowed to perform with respect to any other processor, all previous ordinary* load *and* store *accesses must be performed, and*
- *special accesses are processor consistent with respect to one another.*

Vi kan utnytte den kunnskapen vi får om minnebruken ved disse merkelappene (labler) til å utsette spredningen av de oppdaterte data.

Kapittel 5

Protokoller og CVM

5.1 Protokoller for DSM

Det finnes mange måter å implementere de ulike konsistensmodellene i et DSM-system. I dette kapittelet vil vi gå igjennom ulike implementasjoner av konsistensmodellene. For de protokollene som allerede er implementert i CVM, vil vi også vise til hvordan implementasjonen er gjort der, i den grad det er relevant for forståelsen.

5.1.1 Generelle betraktninger og terminologi

Siden kommunikasjonen er en begrensende faktor, vil vi ønske både å redusere antallet meldinger og lengden av dem. Avhengig av den underliggende kommunikasjonsteknologien vil man kunne gjøre ulike kompromisser. F.eks vil det med ethernet i en tradisjonell implementasjon være så stor oppstartskostnad i forhold til kostnaden pr. sendte byte, at det vil være viktigere med få meldinger enn korte meldinger[42].

Et sidebasert DSM-system deler større enheter enn et cachelinjebasert system, den har en grovere granularitet. Dette kan gi øket grad av falsk deling («false sharing»).

Falsk deling vil si at to prosesser skriver til samme side, men ulike adresser i siden. Med optimalt fin granularitet hadde ikke de to prosessene skrevet til samme side. Dersom vi har at kun én prosess kan skrive til en side av gangen, vil vi risikere at skrivetillatelsen til siden hopper mellom de to prosessene. Dette kalles for ping-pong effekten og det vil oftest være meget merkbart for ytelsen til programmet.

Utnyttelse av konsistensmodellen *release consistency* medfører at man kan få redusert kommunikasjonen ved ulike optimaliseringer[37].

5.1.2 Ulike konsepter

Når en side hos en node a ikke er oppdatert fordi en annen node (b) har skrevet siden senere, nevner litteraturen to måter å sørge for at ikke utdaterte data blir lest. Den éne metoden, kalt *invalidate*, invaliderer sider som er skrevet til. Disse sidene må da hentes på nytt når noden ønsker å lese eller skrive. Den andre varianten, *update*, medfører at noden a sørger for å få overført endringer fra de noder som har skrevet til siden, slik at siden blir oppdatert riktig ved synkroniseringspunkter.

Dersom en side skal oppdateres kan man enten oversende hele siden, eller bruke en annen metode, som kalles *twin & diff* for å oppdatere endringer fra andre noder. Idet en side blir skrevet til for første gang, vil DSM-systemet lage en kopi – en tvilling (twin). Når så endringene skal propageres, sammenlignes den skrevne siden med originalen. Resultatet blir en diff, en melding som er kortere enn siden, som oppsummerer hvilke endringer som må gjøres for å gjøre en tidligere kopi oppdatert. Dette gir kortere meldinger enn om hele siden skulle vært overført, men det tar selvsagt tid å beregne forskjellene og bygge opp diffen.

Flere av protokollene som implementerer RC kan varieres mellom at én node har eksklusiv skrivetilgang (*single writer – SW*), eller at flere noder kan skrive samtidig (*multiple writers – MW*). Spesielt ved MW kan det være nødvendig å kunne overføre bare endringer fra flere noder, og så settes disse sammen til en riktig, oppdatert side ved hjelp av twin & diff. Med SW kan man overføre hele siden, men det vil medføre lengre meldinger enn om man bare sender endringene. Hva som vil lønne seg, kan variere fra applikasjon til applikasjon[42]

5.1.3 Lokalisering av side

Når en prosessor[1] ønsker å lese fra en side som den ikke har rettigheter til, må vi ha en metode for å lokalisere hvor denne siden befinner seg. I CVM har Keleher valgt å ha ha en *manager* for hver side. Denne vet hvem som til enhver tid har siden. Er det den selv, trengs det én melding for å finne ut hvor siden befinner seg, hvis ikke videresendes forespørselen til den registrerte eieren. I tillegg til svaret tilbake til spørrende node krever dette maksimalt to meldinger ($O(2)$)[42].

I det første sidebaserte DSM-systemet, IVY, brukte man en kjede av sannsynlige eiere(*probable owners*). Protokollen fungerer slik at alle prosessorer som har hatt en side husker hvem de har gitt den videre til. For å lokalisere en side spør man først den opprinnelige eieren, og deretter følger man kjeden. Li og

[1] I de følgende avsnitt vil vi bruke prosessor synonymt med prosess. Det kan imidlertid være en forskjell dersom man i en node kjører flere prosesser pr. prosessor, eller om man kjører med et SMP-system. Dette er faktorer som vi ikke ser på i denne oppgaven.

Hudak viste at man i verste fall vil ha $O(n + k \log n)$ meldinger for å lokalisere en side i et n-prosessor system med k sidefeil[48]. I gjennomsnitt gir dette maksimalt $\log n$ meldinger pr feil. Allerede ved åtte noder vil man med en statisk manager pr. side ha færre antall meldinger i verste fall (worst-case), enn med probable owner, siden $2 < \log 8$. Kelehers simulering viste at ved $n = 8$, så gav statisk eierskap et snitt på 1.83 meldinger for å lokalisere siden pr miss, mens probable owner gav 1.86[42].

5.1.4 En protokoll for sekvensiell konsistens

I CVM er protokollen implementert slik at det enten eksisterer én node med skrivetillatelse til siden, eller n lesere, hvor n er mellom 0 og antall noder. Eierskapet til en side flyttes til den som etterspør siden, uavhengig om det var en skrive- eller leseforespørsel.

For å unngå ping-pong effekten, garanteres en prosessor å ha en side i et visst tidsrom (100 ms) før den kan flyttes. Uten denne garantien økte eksekveringstiden i visse tilfeller med opptil 5 ganger[42].

Når en node ber om skrivetilgang til en side, må alle noder som har en kopi av siden invalidere sine kopier.

5.1.5 Ulike implementasjoner av RC

I DASH har RC blitt brukt slik at man pipelinet skriveoperasjoner til minnet. Først ved en release ble prosessoren stoppet for å vente på at alle operasjonene var utført[47]. Dette medfører at man reduserer ventingen ved flere etterfølgende skriveoperasjoner. Man slipper med totalt én venteperiode, istedet for én pr. skriveoperasjon.

For å redusere antallet meldinger, brukte man i Munin en «delayed update que» (DUQ) hvor alle skriveoperasjoner ble bufret. Alle operasjonene ble satt sammen til én melding og sendt samtidig ved release[13]. Denne kaller vi for «eager release consistency» (ERC), fordi den oppdaterer i synkroniseringspunktet. Dette er en ivrig oppdatering, fordi det er mulig å vente til et senere tidspunkt, noe vi skal se straks.

I CVMs implementasjon av ERC, blir endringer propagert til andre prosesser med kopi av siden når en skrivende prosess utfører release. For å redusere lengden på meldingene, beregnes det en diff som propageres. Prosessen blokkeres frem til den har mottatt bekreftelse på at diffen er mottatt og utført på samtlige noder med kopi[36].

Når acquire skal utføres, lokaliseres den prosessoren som sist gjorde en release, men ingen konsistensinformasjon overføres.

Ved miss lokaliseres eieren av siden via sidens manager, og siden overføres så til den som hadde lesefeil.

5.1.6 Lazy release consistency (LRC)

Keleher viste ved et eksempel at å sende oppdateringer ved release kan medføre unødvendige meldinger, dersom noen av prosessorene ikke benytter disse dataene senere. Det er tilstrekkelig å oppdatere kopien av en side som skal endres først ved acquire, istedet for ved release. Da unngår man å sende oppdateringer som aldri vil bli benyttet. Dette kaller Keleher for *lazy release consistency* (LRC)[36].

Når en prosessor utfører en acquire, må den finne ut hvilke endringer den må se, for å oppdatere sin lokale kopi slik at vi følger definisjonen av RC. I CVM bruker Keleher *happened-before-1* (hb1) partiell orden for å definere rekkefølgen på hendelser. HB1 ble definert i [1], men vi velger å presentere Kelehers forenklede variant fra [36]:

Definisjon 3 *Happened-before-1 partial order*

Shared memory accesses are partially ordered by happened-before-1*, denoted* $\stackrel{hb1}{\rightarrow}$*, defined as follows:*

- *If* a_1 *and* a_2 *are accesses on the same processor, and* a_1 *occours before* a_2 *in program order, then* $a_1 \stackrel{hb1}{\rightarrow} a_2$.
- *If* a_1 *is a release on processor* p_1*, and* a_2 *is an acquire on the same memory location on processor* p_2*, and* a_2 *returns the value written by* a_1*, then* $a_1 \stackrel{hb1}{\rightarrow} a_2$.
- *If* $a_1 \stackrel{hb1}{\rightarrow} a_2$ *and* $a_2 \stackrel{hb1}{\rightarrow} a_3$*, then* $a_1 \stackrel{hb1}{\rightarrow} a_3$.

RC krever at før en prosessor fortsetter forbi en acquire, må samtlige tidligere aksesser etter $\stackrel{hb1}{\rightarrow}$være utført på prosessoren. I CVM garanteres dette ved at det propageres skrivevarsler (*write-notices*). Vi deler opp eksekveringen i intervaller og nummerer slik at et nytt intervall påbegynnes ved hver spesialaksess (i.e. release/acquire, eller barriére).

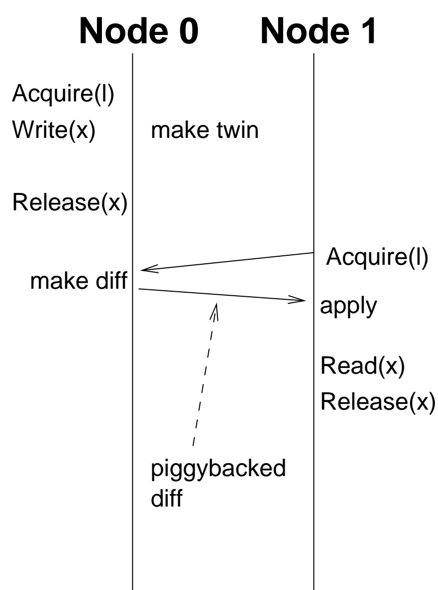
Hver prosessor har et vektortidsstempel (*vector timestamp*) for hvert intervall. La oss kalle denne $V^p(i)$, hvor i er intervallet hvor dette tidsstempel gjelder for prosessor p. Vektoren inneholder ett element for hver prosessor q, og verdien av elementet for prosessor q er det seneste intervall som er utført (performed) på p. Elementet til p er pr definisjon lik i.

Ved en aksessmiss må det, om det ikke finnes en lokal kopi, hentes en kopi av siden. Uansett må differ innhentes for å oppdatere siden slik endringer som har skjedd siden kopien sist ble oppdatert er gjenspeilet.

Ved acquire sender prosessoren p sitt tidsstempel til den som sist utførte en release, q. Prosessor q bruker denne informasjonen til å videresende alle skrivevarsler som den har mottatt for alle intervaller som ikke er utført på p.

Dersom vi kjører en invalideringsprotokoll, invalideres sider som p nå har mottatt skrivevarsler fra. Er det derimot en oppdateringsprotokoll hentes diff'er fra de prosessorer som har skrevet, og disse må så utføres i riktig rekkefølge på den lokale siden. Fordelen med en invalideringsprotokoll er at vi kun henter inn differ når de absolutt trengs, det vil si ved en sidefeil når programmet trenger tilgang til siden. På den annen side vil dette kreve flere meldinger.

Figur 5.1 demonstrerer protokollen.



Figur 5.1: Illustrasjon av LRC

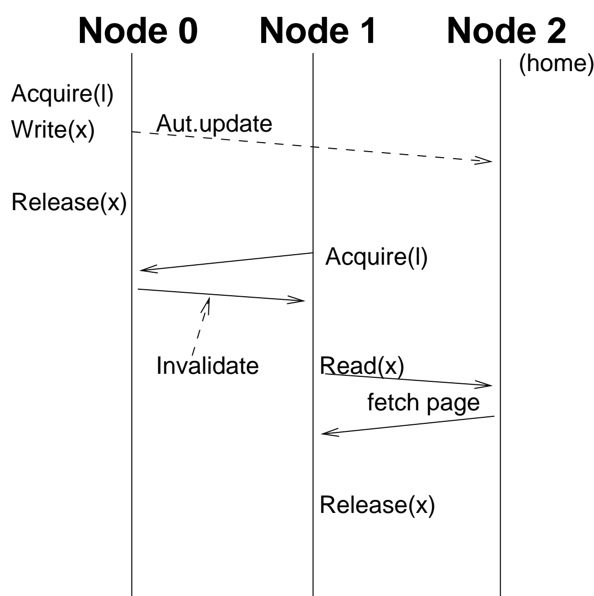
Single writer Denne protokollen tillater kun én skrivende prosessor. Dersom den ønsker å skrive, må den innhente eksklusiv skrivetilgang til siden. Dog kan det finnes mange lesere til én side. Dersom to prosessorer konkurrerer om å skrive til den samme siden, risikerer vi ping-pong effekten. Protokollen kalles for «Lazy consistency, Single Writer», forkortet LSW.

Multiple writers (LMW) I motsetning til LSW, tillates at flere prosessorer skriver til den samme siden. Dette håndterer falsk deling («false sharing») slik at vi unngår ping-pong effekten, ved at siden hopper mellom prosessorene. Flere prosessorer kan skrive til samme side, og deretter settes endringene sammen ved hjelp av diffene.

En annen fordel ved «Lazy consistency, Multiple Writers» (LMW) er at om en prosessor ønsker å skrive til en side som den allerede har cachet lokalt, er dette en avgjørelse som ikke krever kommunikasjon – den kan tas lokalt.

5.1.7 Automatic update release consistency (AURC)

Til SHRIMP Multicomputer[11] har man utviklet nettverksgrensesnitt som kan mappes inn i virtuelt minne, i likhet med Digitals Memory Channel[46, 22], og Dolphins SCI-kort[16].



Figur 5.2: Illustrasjon av AURC

For å utnytte denne funksjonaliteten har de laget en protokoll som automatisk oppdaterer hjemmenoden. Den illustreres med et eksempel i figur 5.2.

Ved å sette opp en mapping fra en side i den skrivende prosessen til sidens hjemmenode, vil alle endringer automatisk bli propagert. Når først denne mappingen er satt opp, er denne oppdateringen transparent og skjer uten at CPUen belastes med overføringen, slik at videre beregning kan skje mens oppdateringene kommuniseres – vi har en overlapping mellom beregning og kommunikasjon.

Når en node trenger en oppdatert kopi av side, altså når den får en sidefeil (page fault), vil den hente over en kopi av siden fra eiernoden.

Denne protokollen utnytter også RC, ved at den først ved synkroniseringspunkter venter til alle automatiske oppdateringer som er på vei er kommet frem.

I simuleringer fikk de i gjennomsnitt bedre resultater med denne protokollen enn med LRC[34]. En ulempe med denne protokollen er at den krever spesiell hardware; nettverkskort som kan mappes inn i virtuelt minne.

5.1.8 Home-based lazy release consistency (HLRC)

Denne protokollen minner litt om AURC, men den krever ikke spesiell hardware. Alle sider har med denne protokollen en hjemmenode, og det er denne som etter ethvert synkroniseringspunkt har en oppdatert kopi.

Hovedidéen er at man ved slutten av et intervall (i.e. ved synkronisering) undersøker om siden er oppdatert lokalt og isåfall beregner og sender en diff til sidens hjemmenode.

Diffens levetid er meget kort, fordi den kan beregnes, og deretter sendes til hjemmenoden hvor siden blir oppdatert. Når mottaket er bekreftet kan diffen fjernes fra minnet.

Når en node får lesefeil på siden, henter den en oppdatert kopi med én forespørsel til hjemmenoden, altså med én round-trip.

Istedet for å hente inn diffene fra alle samtidige skrivere, så sendes diffene til en sides hjemmenode, og utføres der. Ved synkronisering av sidene, overføres den ferdige siden fra hjemmenoden til den som ønsker å lese[79]. Denne protokollen gav i gjennomsnitt det beste resultatet av alle protokoller for 4kB sidestørrelse i én undersøkelse[80].

5.1.9 Sammenligning

Keleher har i [42] sammenlignet protokollene som er implementert i CVM, og prøvd å finne ut hva som er viktigst, svakere konsistensmodeller eller én eller flere skrivere. (SW/MW)

Han testet med MPI over en høyytelses bi-directional link, med båndbredde på opptil 40MB/s. Startkostnaden for en melding var høy, på grunn av stor software overhead. Den relative kostnaden pr. byte overført var forholdsvis mye lavere. Dette kan forklare at SW-LRC for 6 av 8 applikasjoner i snitt var bedre enn MW-LRC. SW-LRC var i snitt 34% bedre enn SW-SC.

5.2 Implementasjon av HLRC i CVM

Våre modifiserte protokoller som utnytter ekstra funksjonalitet i SCI er basert på HLRC. Derfor skal vi se mer nøye på hvordan denne protokollen er implementert.

Alle noder i systemer er nummerert stigende fra 0. Med denne protokollen har alle noder en hjemmenode, og initielt har alle sider node 0 som hjemmenode. Dette kan endres på flere måter.

Det finnes mulighet for å la hjemmenoden være den noden som sist har endret på en side ved en barriére. Dette kalles for migrerende hjem. Videre er det mulig fra applikasjonen å tildele en side en statisk hjemmenode.

Ved oppstart er alle sider som befinner seg på hjemmenoden for denne siden, leselige lokalt. Alle andre sider er beskyttet mot både lesing og skriving.

5.2.1 Sidefeil

Når operativsystemet fanger en sidefeil («segmentation fault»), startes CVMs sidefeilshåndterer. Denne regner ut hvilken side feilen kom på, og sender kaller sideobjektets feilhåndterer.

Dersom siden er på hjemmenoden, skal det være en skrivefeil. Siden gjøres da skrivbar, og markeres i datastrukturen som skrevet (dirty). Protokollen holder rede på alle som har en skrivbar kopi, og legger derfor til denne noden i kopisettet for siden. Deretter returneres kontrollen til programmet.

Nå vet vi at vi ikke er på en hjemmenode. Dersom vi ikke har en lesbar kopi, så må vi innhente denne. Det sendes en leseforespørsel til hjemmenoden. I svaret følger siden med. Den kopieres inn i riktig adresseområde, og siden settes deretter lesbar, eller skrivbar hvis dette var en skrivefeil.

Dersom vi hadde en skrivefeil, opprettes det dessuten en identisk kopi (twin), som brukes når hjemmenoden skal oppdateres. Den legger til seg selv i sin lokale kopisett, og setter siden skrivbar.

Kontrollen returneres til applikasjonen.

5.2.2 Leseforespørsel

En leseforespørsel skal kun komme til hjemmenoden. Den svarer med å sende den oppdaterte siden tilbake. Dessuten degraderes siden fra skrivbar til lesbar.

5.2.3 Synkroniseringspunkter

Noder som har skrevet til en side som de ikke eier, må ved synkroniseringspunkter oppdatere hjemmenodens kopi. Dette gjøres ved å sammenligne den skrevne siden i minnet med kopien som ble opprettet før siden ble skrevet til. Resultatet av sammenligningen er en diff, en melding som inneholder alle endringer som er gjort. Denne meldingen sendes til hjemmenoden.

Hjemmenoden tar imot differ, og utfører disse på sin kopi av sidene. Dessuten tar den vare på hvilken node som sendte inn diffen, og øker versjonsnummeret på siden. Hvem som sist skrev til siden kan benyttes til å migrere hjemmenoden etter hvem som skriver til en side. Versjonsnummeret økes ved alle synkroniseringspunkter hvor siden er endret, for å kunne vite om to noder har lik kopi av en side, uten å måtte sammenligne ord for ord.

Det finnes to ulike typer synkroniseringspunkter: låser og barriérer. Når en side tar en lås (acquire), må den for det første innhente låsen fra den som

allerede holder den. Dessuten må den oppdatere sine lokale sider slik at de er av siste versjon.

Kapittel 6

Nettverk i eksisterende DSM

I dette kapittelet vil vi først se på de tre sammenkoblingsteknologiene (interconnect) HIPPI, Scalable Coherent Interface og Memory Channel, og oppsummere ulike tall for ytelse som har fremkommet i ulike prosjekter.

Deretter vil vi diskutere forskjellene mellom disse tre, og litt om hvilke konsekvenser disse ulikhetene kan ha for implementasjon av et system for distribuert felles hukommelse.

SoftFLASH

Som nevnte i innledning implementerte Erlichson et system for distribuert fellesminne basert på FLASH protokollen fra DASH-maskinen. Han benyttet HIPPI som interconnect. Et resultat han kom frem til var at en kjerneimplementasjon av protokollen var 3–4 ganger raskere enn en implementasjon på brukernivå.

Likevel ble konklusjonen i artikkelen at overhead, begrenset båndbredde på nettverket og den høye forsinkelsen hindret god ytelse for kommunikasjonsintensive applikasjoner.

Nyere interconnect

«Scalable Coherent Interface» (SCI) er en IEEE standard som i tillegg til overføring via DMA tilbyr memory-mapping med direkte lese og skrivetilgang via de vanlige minneoperasjonene.

Det har vært stor aktivitet knyttet til «Scalable Coherent Interface» ved Institutt for Informatikk[64], og det finnes et antall rapporter med ulike ytelsesmålinger. Det er skrevet flere biblioteker og SCI har vært benyttet på én eller annen måte i flere hovedoppgaver og doktorgradsavhandlinger.

Et annet system som ligner SCI er «Memory Channel» (MC). Dette leveres av Digital Equipment Corporation (DEC), og ble lansert i 1996. DSM-

prosjektet Cashmere benyttet Memory Channel til kommunikasjon, og har interessante resultater [70].

6.1 De ulike teknologiene

6.1.1 HIPPI

HIPPI har mye til felles med tradisjonell svitsjet nettverksteknologi. Den tilbyr pålitelig levering av pakker i riktig rekkefølge, og med flytkontroll. Adapterene benytter DMA til å kopiere pakker til og fra hukommelsen i maskinen. Når en pakke skal sendes, må driveren første sette opp DMA-overføring fra internhukommelsen til adapteret, og først når alle dataene er mottatt, vil adapteret sende pakken.

Konseptet er store-and-forward, og vanlig for nettverksteknologi. Dette er en stor begrensning på forsinkelsen [19].

HIPPI er forbindelsesorientert i flere lag. Både den fysiske sammenkoblingen og den logiske forbindelsen i overføringen er punkt til punkt.

Båndbredden er 100Mbyte/s og omtrent hele båndbredden kan benyttes, imotsetning til f.eks ATM hvor en stor andel forsvinner i overhead. Bussbaserte systemer vil ha arbitreringsproblemer, fordi flere sendere må konkurrere om sendetid [21].

Opprinnelig ble HIPPI benyttet i supermaskiner, til å koble sammen enhetene, men den synkende prisen (fra $15000 til ned mot $2000 for adapterene), har gjort HIPPI attraktiv i miljøer som ønsker nettverk med lavere forsinkelse og høy båndbredde [76].

HIPPI benytter parallelle kobberkabler, eller optisk fiber (seriell) som fysisk medium ved overføringen.

Det finnes ANSI-standarder for de ulike lagene i HIPPI. I rfc2067 [58], som har status som draft-standard, dokumenteres IP over HIPPI.

Idag finnes det mange leverandører av HIPPI-adaptere basert på ANSI-standardene, blant annet Essential Communications, Cray Research, Network Systems, SGI og IBM.

I Erlichsons doktorgrad [19] ble det benyttet kort fra Essential Communications. Han har estimert at en side på 16kbyte, bruker 163 μs på å overføre pakken på kabelen. I tillegg regner han 794 μs på oppsett av DMA i begge ender.

6.1.2 Memory Channel(MC)

Memory Channel er basert på delt minne. Den har en teoretisk båndbredde på 100MB/s, og forsinkelse på under 5 μs for en minimal overføring, 8 bytes.

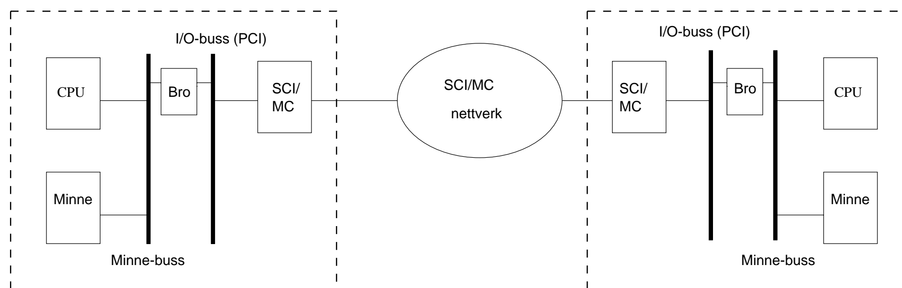
Adresseringen av adapterene skjer via et 512 MB stort adresserom, skjønt den første implementasjonen tillot kun 128 MB [46]. På det fysiske laget er Memory Channel basert på en delt buss. Dette har betydning for skalerbarheten, i.e. muligheten for å koble på et stort antall noder, siden disse må konkurrere om sendetid. I Memory Channel 2 benyttes svitsjede forbindelser [22].

Systemet tilbyr skriveaksess fra en region (virtuell adresse) i én maskin, til en region i fysisk hukommelse på én eller flere andre maskiner. Vi kan si at operasjoner mot det lokale minnet reflekteres til minne i andre maskiner[60].

Disse logiske forbindelsene er punkt til punkt, men man kan også sette opp flere mottagere av skriveoperasjoner til ett område. Siden Memory Channel er bussbasert, bør slik broadcasting ikke koste stort ekstra.

Den første generasjonen Memory Channel tilbyr kun skriveaksess (remote store). Remote read var ikke mulig – en prosess kan ikke lese direkte fra en annen maskin. Dette ble påpekt som en begrensing for Cashmere [70]. Den andre generasjonen har også implementert lesemuligheter (remote read)[22].

Oppsett av en forbindelse skjer ved at det i klienten settes inn peker til Memory Channel-adapterets adresseområde i tabellen for oversettelse fra virtuell til fysisk hukommelse. I tillegg må klienten sette opp Memory Channel-adapteret, slik at det reagerer på mottak til denne adressen og sender dataene riktig.



Figur 6.1: Arkitektur

På tjeneren må siden hvor dataene skal ligge bli låst, slik at den ikke kan swappes ut til disk. Dette er nødvendig for å håndtere skriveoperasjoner fra andre maskiner. Dessuten må Memory Channel-adapteret settes opp slik at mottatte data plasseres riktig i hukommelsen, altså en mapping fra adresserommet i Memory Channel til det fysiske minnet lokalt.

Biblioteket som benyttes vil i tillegg utføre ugyldiggjøring (invalidering) av cachelinjer ved mottak av data, slik at prosessoren (eller prosessorene) oppdager at hukommelsen har blitt oppdatert.

Overføringen skjer ved at klienten skriver til en adresse i hukommelsen som peker på mc-adapterets i/o-område, ved hjelp av *store*-instruksjonen.

Adapteret mottar dataene, og skriver dem til det minnet som er satt opp som mottagere av denne adressen.

Digital presenterer Memory Channel dets arkitektur og ytelse, samt et bibliotek for programmering (TruCluster) i [46]. Erfaringer og mål for en forbedret utgave av Memory Channel er publisert i [22]. Figur 6.1 demonstrerer arkitekturen.

Det finnes ingen publisert standard for Memory Channel, produktet er proprietært og leveres kun av Digital. Det finnes dog flere lignende konsepter, for eksempel de spesialbygde adapterene for prosjektet Shrimp[11].

6.1.3 Scalable Coherent Interface

Scalable Coherent Interface har mange fellestrekk med Memory Channel. Begge to er basert på delt hukommelse, og kan skrive direkte til minnet. I motsetning til Memory Channel er ikke Scalable Coherent Interface bussbasert, men basert på punkt-til-punkt sammenkobling. Blokkskjema for arkitekturen til både Memory Channelog Scalable Coherent Interface vises i figur 6.1.

SCI-enheter kan kobles sammen på ulike måter. Den rimeligste varianten er back-to-back for to enheter, og ring for flere enheter. Større løsninger kobles sammen via en eller flere svitsjer.

SCI støtter remote read, i tillegg til remote store, har en båndbredde på over 1GB/s i de raskeste implementasjonene, og en énveis forsinkelse på under 3 μs for en minimal overføring, typisk 4 bytes. Resultatene er hentet fra artikler som alle benytter implementasjonene fra Dolpin Interconnect Solutions.

Én SCI-pakke kan inneholde inntil 64-bytes. Dersom det prosessoren skriver til er påfølgende adresser, blir disse om mulig, og etter bestemte regler, flettet til én pakke. Ved å kjenne disse reglene, behøver ikke en slik opphopning av skriveoperasjoner medføre forsinkelse for applikasjonen.

SCI ble standardisert i 1992, og finnes som ANSI, IEEE og ISO-standard.

6.2 Resultater

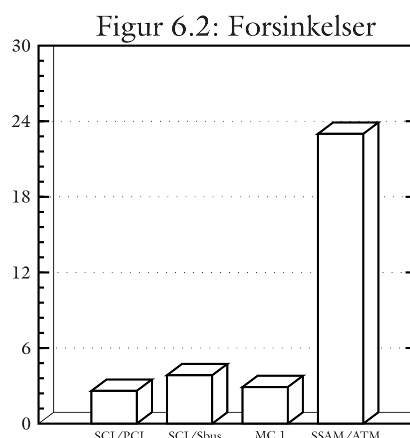
Når man prøver å finne en nedre grense for forsinkelse ved hjelp av repeterte målinger, vil man om mulig benytte polling, fordi forsinkelsen knyttet til interrupt langt vil overstige kommunikasjonsforsinkelsen for raske interconnect. Vi velger å kalle den målte nedre grense for minimal forsinkelse.

Tabell 6.1 viser en oversikt over minimale forsinkelser for ulike sammenkoblinger. I den grafiske fremstillingen i figur 6.2 er kun de raskeste metodene tatt med (≤ 23 μs). Kolonnen lengst til høyre viser tid pr. byte. Merk at denne tiden ikke sier stort om hvor lang tid en lengre melding enn den minimale vil

ta. Dette ser vi nærmere på i avsnitt 6.5.

tid (i μs)	system	nyttelast	tid pr.byte
2.6	SCI/PCI	4 bytes	0.65 μs
3.84	SCI/Sbus	4 bytes	0.96 μs
2.9	MC 1	8 bytes	0.73 μs
23.	SSAM/ATM	32 bytes	1.39 μs
163.	HIPPI (wire-time)	16kB	0.01 μs
957.	HIPPI, total kommunikasjonsforsinkelse	16kB	0.06 μs
750.	HIPPI, estimert forsinkelse ved kort melding	Ukjent	
280.	TCP/IP/ATM	4 byte	

Tabell 6.1: Oversikt over forsinkelser (se kildehenvisninger på side 41)



Memory Channel 1 og SCI forsinkelsene er for en remote store[1], hvor mottageren poller for å registrere at data er mottatt. Tallet for Memory Channel er hentet fra [46], for SCI/PCI fra [62], og for ATM/SSAM og SCI/Sbus fra [12]. Årsaken til at PCI-implementasjonen er raskere enn Sbus-implementasjonen, skyldes at mens PCI bruker en ASIC-brikke, så er Sbus-kortet implementert med FPGA.

SSAM/ATM er en forkortelse for «SuperSparc Active Messages» over ATM. Active Messages går utenom den tradisjonelle protokollstakken for å få utnyttet den lavere forsinkelsen som er mulig over ATM. Protokollen benytter 32 bytes av én ATM-celle for å overføre data, og de resterende bytes (16) benyttes til blant annet flytkontroll og andre system og metadata. Meldinger kortere

[1] Alpha har 64-bits store, mens Intel Pentium har 32-bits store, følgelig er nyttelasten for MC 8 byte og PCI/SCI 4 byte.

enn 32 bytes har ikke merkbar lavere forsinkelse[49].

TCP/IP over ATM er hentet fra [12], og viser en sammenligning med en løsning som er ganske hyllevare.

I ytelsesmålingene for SCI har det blitt benyttet ulike kombinasjoner av Dolphins adaptere (basert på enten LC-1 eller LC-2 chiper), og busser (PCI og SBus). De beste målingene har blitt gjort for LC-2 over PCI-buss. Den nyeste versjonen har også mulighet for «store reordering», noe som kan ha konsekvenser for hvilke protokoller som kan benyttes, eller er mest effektive til synkronisering og meldingsutveksling [53].

Erlichson estimerer at HIPPI bruker 163μs på å overføre en 16kbyte side på kabelen. 794μs beregner han går med til kommunikasjonsoverhead. Disse tallene er ikke så veldig relevante, siden vi i disse målingene for HIPPI overfører en mye større mengde data enn i de målingene jeg tidligere har nevnt. Imidlertid synker ikke kommunikasjonsoverhead noe særlig, selv om vi sender en kortere melding med HIPPI. Dette skyldes at data overføres ved at software-driveren setter opp en DMA-overføring fra hukommelsen i maskinen og til adapteret. Erlichson mener at ved optimalisering for korte meldinger kan han få forsinkelser ned mot ca 750μs over HIPPI [19].

Oppsummering

Det skulle være klart utifra denne oversikten at det er mulig å oppnå betydlig lavere minimal forsinkelse (en forskjell på 10^2 for korte meldinger) ved å benytte SCI eller MC som interconnect istedet for HIPPI. Om dette vil gjenspeile seg i bedre ytelse for programmene er likevel ikke sikkert – dersom programmene ikke er begrenset av nettverkets ytelse vil vi ikke kunne se noen forbedring.

Forbedringer

For å finne ut hvordan adapterene kan forbedres, bør vi undersøke nærmere hva som er begrensningene i kortene. Noen kandidater for nærmere undersøkelse er I/O-bussen og minnebussen i maskinen, hastighet i adapteret, og hastigehten på linken.

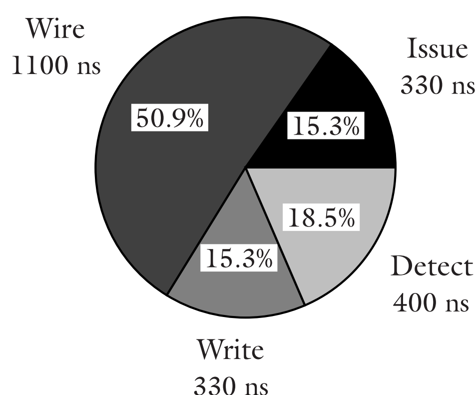
I [22] er de 2.17μs som Memory Channel 2 bruker på én remote store brutt ned. Tabell 6.2 er en tilpasning av tabellen fra artikkelen, mens en grafisk fremstilling ser vi i figur 6.3.

Overføringen mellom de to adapterene tar mer enn halvparten av tiden, og er det klart største enkelt bidraget. Likvel har vi en ikke ubetydelig maskinarkitekturavhengig forsinkelse på 1060ns, som kan bedres ved å ha raskere minnebuss, PCI-buss og raskere CPU.

330ns	15%	CPU instr-issue →PCI-buss$_1$
1100ns	52%	Adapt/PCI$_1$ →MC →Adapt/PCI$_2$
330ns	15%	Adapt/PCI$_2$ →hukommelse
400ns	18%	polling returnerer positivt

Tabell 6.2: Fordeling av tiden som én remote store bruker

Figur 6.3: Forsinkelser i «Memory Channel 2»



6.3 Diskusjon – konsekvenser for DSM

6.3.1 Kjernemodus vs brukermodus og interrupter

Erlichson lagde både en implementasjon av FLASH som kjørte i kjernen på IRIX 6.2, og en versjon som kjørte på brukernivå. Å hente en 16kbyte side, tok i hans sammenligning 3216μs i brukernivå mot 1164μs i kjernen.

Tiden det tar fra operativsystemet oppdager en sidefeil, og til håndtereren starter er 41μs på IRIX [20].

Ved overføring av en kort melding over SCILAN mellom Windows NT-maskiner, finner Ryan og Bryhni en forskjell på ca 100μs avhengig av hvilken strategi som brukes. Ved polling er forsinkelsen 16.1μs, mens ved interruptdrevet levering 179.8μs [61].

Tilsvarende resultater har vi også for SS20 maskiner med Solaris. Bryhni og Omang demonstrerer dette i [12], hvor de sammenligner meldingsutveksling over Ethernet, ATM, SCI via operativsystemet og SCI via et eget bibliotek.

I artikkelen som beskriver Memory Channel og dens driver TruCluster [46], rapporterer de å ha målt at et bytte fra bruker- til kjernemodus koster ca 30 μs på Digital Unix og AlphaServer 4100.

Tabell 6.3 viser hvor lang tid det tar å overføre en melding for ulike nett-

Tabell 6.3: Hvor lang tid tar det å levere én melding?

Ethernet	ATM	SCI via OS	SCI via lib
461μs	280μs	186μs	18μs

verk og driverløsninger.

Dette bekrefter at levering via operativsystemet, som benytter både interrupt og innebærer skift fra brukermodus til kjernemodus, er tregt. Begge operasjonene er kostbare, og bruken av dette bør minimaliseres, om vi ønsker å utnytte den gode ytelsen i nettverket.

Mens interruptforsinkelsen utgjør noen få prosent av den totale forsinkelsen for SoftFLASH, vil den være flere ganger større enn kommunikasjonsforsinkelsen for en meldingsutvekslingsprotokoll basert på delt minne på brukernivå.

På den annen side, dersom hver maskin kun har én prosessor, vil denne ikke kunne utføre andre operasjoner dersom vi benytter polling for å registrere mottak av pakker. Vi vil altså sløse CPU ved «busy wait». En løsning på dette er å utføre polling inntil en bestemt tid har gått, deretter blokkere prosessen og vente på interrupt[4]. Imidlertid vil vi kanskje ikke kjøre andre applikasjoner samtidig med større beregninger, og da har vi ingen gevinst. Én løsning for å skjule latency er å ha flere tråder som beregner pr. prosessor. Da kan vi håpe på at en annen tråd har mer jobb å utføre, slik at den kan arbeide når den éne tråden er blokkert i påvente av data.

Keleher har undersøkt effekten av å ha flere tråder pr. node for å skjule forsinkelse ved synkonisering og venting på data fra andre noder. Hans resultater var en noe forbedring (inntil 20%) i noen tilfeller, men at det kan kreve en del innsats i form av tilpasning av applikasjonen å få god ytelse[75].

6.3.2 Ingen kopiering («zero copy»)

I forbindelse med meldingsutveksling ønsker vi ideelt sett at data kan skrives via *store* direkte fra applikasjonens buffer over SCI og rett inn i mottagerapplikasjonens buffer. Imidlertid vil vi nok måtte kopiere data, ihvertfall hos mottageren fra et kommunikasjonsbuffer og inn i applikasjonsbufferet. Dette er nødvendig for å kunne ta imot større mengder data. En slik løsning sparer dessuten globalt adresserom, fordi vi bare allokerer et begrenset område (bufferet) som delt.

Ved en løsning med mottagerbuffer vil altså data bli sendt fra SCI adapteret via minnebussen til et mottagerbuffer. En driver, som forøvrig godt kan kjøre i brukermodus, vil så kopiere disse fra driverbufferet til applikasjonsbufferet. Da går data fra hukommelsen via CPU og tilbake til hukommelsen. Altså

passerer et ord (typisk 64 bit på en UltraSparc) tre ganger på minnebussen til mottageren.

Prosessor	(MB/s)
Pentium 200 MHz, HX	36
UltraSPARC-I 167 MHz (E3000)	178
UltraSPARC-II 296 MHz (Quark)	214

Tabell 6.4: Gjennomstrømning, reciever copy. Hentet fra [61]

Dette kan være en alvorlig begrensning. Tabell 6.4 viser maksimal gjennomstrømning på minnebussen for ulike arkitekturer. Flere resultater viser at PC arkitekturen har en svært dårlig minnebuss, sammenlignet med UltraSparc. Forskjellen er i størrelsesorden en faktor 5 [54, 61]. Ifølge [61] blir ikke minnebussen på UltraSparc mettet i disse testene, mens både minnebussen på SparcStation 20 og PCer[2] ble det.

Kopieringens innvirkning på ytelsen

Ryan og Bryhni gjorde målinger av tre ulike operasjoner[61]. «SCI memset» skriver en konstant over SCI med remote store. «SCI memcpy» leser data fra lokal hukommelse og skriver via remote store. Forskjellen i ytelse mellom memset og memcpy skyldes bruk av minnebussen hos sender. Den tredje operasjonen, SCILAN, innebærer kopiering som nevnt tidligere, fra mottagerbuffer via CPU til applikasjonsbuffer.

For meldingsstørrelser under 10000 bytes, er det ingen forskjell i ytelse mellom memset og memcpy. For meldingsstørrelser mellom 15000 og 45000 byte er grafene jevne. SCI memset har en gjennomstrømning på ca 70MB/s, SCI memcpy 55 MB/s, og SCILAN på 20MB/s.

Linkhastigheten på de nyeste chipene fra Dolphin (LC-2) er på 500MB/s, men andre komponenter på PCI/SCI mettes ved 88MB/s. Dette var langt over kapasiteten til minnebussen på en PC høsten 1997. Vi har gjort lignende målinger på utstyr som ble solgt i 1998, og de viser en stor forbedring for gjennomstrømning på minnebussen på PCer. Disse resultatene finnes i appendiks A.

6.3.3 Resultatene fra SoftFLASH

I sin doktorgradsavhandling konkluderer Erlichson med at software-basert DSM har dystre utsikter. Resultatene viser at protokollen for SW (single-writer) stort sett var mer effektiv enn MW (multiple-writers), ihvertfall for hans system.

[2] PCene er pentium-maskiner, 100 og 133 MHz, med HX-chipset

SoftFLASH er et sidebasert DSM, med 16kB store sider. Fremtidige arkitekturer vil antagelig benytte større og større sider, hevder Erlichson, for å unngå øket antall TLB-miss, for å holde størrelsen på datastrukturene nede og effektiviteten til sidehåndteringen i operativsystemet oppe. Når sidene blir større og større, krever det enten et nettverk med høy båndbredde, eller at en benytter MW-protokollen, som han mener generelt gir dårligere ytelse enn SW.

Når vi øker antall prosessorer i en klynge, ønsker vi en tilsvarende forbedring i hastighet ved kjøring. For eksempel ønsker vi at programmet skal kjøre 32 ganger raskere dersom vi benytter 32 prosessorer istedet for én. Dette er det ideelle resultatet.

Forbedringene med SoftFLASH varierte mellom 4 og 21.6 ganger, når det ble benyttet 32 prosessorer, og dette er en god del dårligere enn det hardwarebaserte systemer klarer[20].

Siden SoftFLASH med SW var begrenset av forsinkelsen i nettverket, ønsket Erlichson å få en tilnærming til hvordan systemet ville virke ved forbedret nettverk. Han har tatt resultatene fra sine kjøringer, redusert forsinkelsen med en faktor 2 og en faktor 4, og ekstrapolert resultatene. Dette gir ikke så veldig store utslag, bortsett fra for applikasjonen Ocean som har tilnærmet ideell forbedring [19].

6.3.4 Sidestørrelser

La oss regne ut hvor lang tid det vil ta å overføre en 16384 bytes stor side, ved å skalere de forsinkelsene som fremkommer ved en ping-pong test, se tabell 6.5. Data er hentet fra rapporten til Omang, [54].

Metode		tid	$\frac{\mu s}{\text{byte}}$	tid for 16kB
SCI PIO	4 byte	2.90	0.725	11878 μs
SCI PIO	8 byte	3.02	0.377	6184 μs
SCI PIO	64 byte	4.73	0.074	1210 μs
SCI PIO	1024 byte	20.94	0.020	335 μs
SCI PIO	65536 byte	811.56	0.012	203 μs
HIPPI DMA	16374 byte	957.0	0.058	957 μs

Tabell 6.5: Skalering av forsinkelse

Heldigvis bruker vi ikke $n * 2.90\mu s$ på å overføre n bytes. Dette fordi nettverket er så raskt at selve overføringen av ekstra bytes ikke tar særlig lenger tid[54].

6.4 Konklusjon

6.4.1 Nyere DSM-systemer

De som lagde Cashmere-2L kjente til Erlichsons resultater, og de har benyttet Memory Channel som vi har sett har betydelig lavere forsinkelse enn HIPPI. I testene har de benyttet mange av de samme applikasjonene som Erlichson. Med 32 prosessorer, fordelt på 8 noder med 4 prosessorer i hver, har de en forbedring på mellom 8.5 og 31.5 ganger.

Den beste forbedringen i Cashmere er vesentlig bedre enn den beste for SoftFLASH, og nærmer seg det ideelle (1:1 forbedring).

6.4.2 Konsekvenser

Det kan synes nødvendig å lage kode som strever mot å

- Unngå interrupter
- Unngå å (skift til kjernemodus), altså implementere mest mulig i brukermodus
- Unngå kopiering mellom bufre

Kapittel 7

Ytelse

I vår undersøkelse av distribuert fellesminne vi det være interessant å sammenligne hvordan og hvor godt løsninger basert på SCI er i forhold til mer tradisjonelle løsninger. En vanlig måte å gjøre dette på, er å kjøre ett eller flere programmer med både SCI og det vi ønsker å sammenligne med, og deretter se på ytelsen til programmene i forhold til hverandre.

Da bør vi første diskutere hva vi mener med ytelse, og hvordan vi ønsker å måle denne. Vi vil derfor først se generelt på en definisjon av ytelse, ulike former for skalering, og ulike suiter av programmer som har vært vanlig å benytte i *benchmarks*.

7.1 En definisjon av ytelse

Hennesey and Patterson definerer i [56, side 18] ytelse som den inverse av kjøretid. Dersom man sier at maskin X er n ganger raskere enn Y (også kalt «speedup» mellom to maskiner), menes

$$\frac{\text{Kjøretid}_Y}{\text{Kjøretid}_X} = n$$

Videre skriver de:

> Det er forfatternes mening at den eneste konsistente og pålitelige målingen av ytelse er kjøretiden til ekte programmer, og at alle foreslåtte alternativer til tid som metrikk, eller (alternativer) til ekte programmer, har (eventually) medført misledende påstander eller til og med tabber i computer design.

7.1.1 Diskusjon

Det er likevel flere problemer med dette med å måle tid. Hvilken tid skal vi måle? Totalt forløpt tid, også kalt veggklokketid er kanskje det mest nærliggende. Det vil altså si å måle hvor lang tid som har forløpt fra programmet startes til det avslutter.

Dersom programmet kjøres på en flerbrukermaskin, vil større eller mindre deler av den forløpte veggklokketid gå med til å kjøre andre programmer. Unix kan rapportere hvor mye tid som er benyttet i brukermodus og i systemmodus under programmet. Problemet er at blokkerende venting på I/O ikke kommer med i noen av disse to tidene, og dermed vil vi risikere at forsinkelse i nettverket ikke blir en del av målingen.

Dersom vi benytter veggklokketid, og kjører testene på maskiner som ikke har andre brukere, og som ikke benyttes som servere samtidig, burde vi kunne eliminere mesteparten av den tiden som går med til andre programmer.

7.1.2 Skalering

For å sammenligne to kjøretider bør det arbeidet som utføres, og som vi dermed sammenligner kjøretiden på, være likt.

Vi får raskere og raskere maskiner, og et program som tok 2 minutter å utføre for få år siden, kan kanskje gå på sekunder idag. Da kan vi møte på en del sperrer for høyere hastighet som vi ikke ønsker (f.eks på grunn av kostnaden, eller at det ikke er hensiktsmessig), eller har særlig mulighet til å gjøre noe med.

Dersom vi altså bruker tid som mål og holder arbeidet konstant, vil vi sannsynligvis få problemer med å sammenligne maskiner over flere generasjoner.

Om vi ønsker å analysere hvor godt et system skalerer når man legger til flere noder eller prosesseringselementer, må vi også kunne sammenligne ytelsen. Dersom problemets størrelse er låst, kan vi få dårlig skalering rett og slett fordi problemets størrelse er så liten at det ikke er mulig å hente ut nok parallellitet til å la et større antall noder jobbe med problemet.

Dette medfører at vi rapporterer dårlig ytelse for vår parallelle maskin, til tross for at den kanskje er egnet til å løse større problemer.

Man kan skalere arbeidet ved å holde kjøretiden konstant, og la arbeidet vokse. Dersom algoritmen er av orden $O(n^2)$, burde vi ved å ha fire ganger så mange prosessorer og dobbelt arbeidsmengde, få samme kjøretid, dersom vi har ideell forbedring[56, side 722].

Et problem her kan være at usikkerheten i resultatet ved fysiske beregninger ofte øker ved økning av arbeidsmengden, slik at vi får dårligere kvalitet på resultatet[30].

7.1.3 Amdahls lov

Den som leser litteratur om skalering ved bruk av parallelle prosessorer, kan vanskelig unngå å lese om Amdahls lov.

I sin artikkel formulerer han dog ingen lov, men han drøfter karakteristikker som begrenser ytelsen på parallelle prosessorer[2]. Han hevder at den andelen av total kjøretid som brukes til bokholderi av data er relativt høy, og dessuten at denne delen så ut til å være sekvensiell.

Dersom hele programmet er parallelt, vil vi kunne få en idéell (lineær) forbedring med 16 ganger bedre ytelse ved 16 ganger så mange prosessorer, og tilstrekkelig (uendelig) med båndbredde og minne, og lav (ingen) forsinkelse[1]. Hvis ikke hele programmet kjører parallelt vil forbedringen av å kjøre deler av programmet parallelt være begrenset av den andelen av tiden som dette tar av den totale kjøretid.

En annen konklusjon han trakk var at bestrebelser etter høy parallell ytelse ikke vil være tilstrekkelig, dersom man ikke klarer å få til en lignende forbedring for sekvensielle prosessering.

7.2 Tradisjonelle ytelsestester

Om et system skal benyttes for å utføre én bestemt oppgave, er antagelig den mest hensiktsmessige ytelsestesten å kjøre oppgaven på de ulike systemene som er under vurdering.

Dette kan imidlertid være meget vanskelig. Å oversette, og tilpasse en stor applikasjon til flere ulike typer systemer, vil antagelig være tidkrevende. Ofte vil også kjøringene ta så lang tid, at en slik evalueringsform vil være lite hensiktsmessig.

Kanskje er det ikke bare én oppgave maskinen skal utføre, men mange ulike, og å teste alle kombinasjoner av oppgaver på alle mulige systemer er antagelig enda vanskeligere.

Flere forfattere hevder at mange av studiene har i for stor grad benyttet leketøysprogrammer som ligner lite på de applikasjonene det er ønskelig at systemene skal kjøre. Mange av disse programmene er for små, bruker for lite minne til å være realistiske og har for få likheter med virkelige applikasjoner. Blant eksempler på slike programmer nevnes Tårnet i Hanoi, Dronningproblemet, og heap og quicksort blant mange andre[67, 56, 29].

[1] Det er rapportert om superlineær forbedring i noen tilfeller, for eksempel 3D-PDE under IVY[48]. Dette skyldes at datasettet passet bedre inn i cachene og unngikk paging til disk ved 16 noder enn med én.

7.2.1 Kjerner

En mulighet er å plukke ut små, viktige deler fra store programmer, og bruke disse til evaluering. Livermore Loops og Linpack er kanskje best kjent. Dette kan brukes til å isolere og evaluere karakteristikker[56].

Linpack tester matriser, og mesteparten av tiden går med i indre løkker for matrisetransformasjoner, mens Livermore Loops (Livermore FORTRAN kernels) inneholder kjerner fra vitenskapelige FORTRAN programmer.

7.2.2 Syntetiske tester

Tidligere har man benyttet såkalte syntetiske benchmarks som whetstone og dhrystone.

Whetstones skulle ha en instruksjonsstrøm som var representativ for flyttallsberegninger, og var basert på statistikk fra programmer som utførte vitenskaplige beregninger skrevet i Algol-60[56, side 56]. Dhrystones kodemiks skulle være representativ for systemprogrammer som brukte heltall.

Ingen av disse programmene bruker så mye minne at dataene som brukes havner utenfor cache i nyere systemer, dessuten er det meget lett for kompilatorer å forkaste store deler av koden, og utføre transformasjoner som neppe kan gjøres i vanlige programmer[56, side 47].

7.2.3 Suiter

SPEC Organisasjonen SPEC har som mål å utvikle, vedlikeholde og støtte et standarisert sett av relevante ytelsestester for den siste generasjonen av datamaskiner[15]. De mest brukte testene fra SPEC i fagbladenes analyser av ytelse, var SPECmark89, og senere SPECint og SPECfloat 1992 og 1995-utgavene. Ved å regne sammen kjøretid på ulike programmer kommer man frem til ett resultat, et tall som benyttes til å sammenligne ytelse.

Et problem med disse testene var at produsentene begynte å optimalisere kompilatorer for nøyaktig de versjonene av programmene som fantes i suiten. Slik fikk man kunstige forbedringer, i den betydning at en tilsvarende forbedring ikke ville oppnåes med andre programmer, og i noen tilfelle har slike forbedringer produsert gal kode[56].

For å oppdatere programmene til stadig kraftigere maskiner, og for å rette opp og fjerne programmer som det er for lett å jukse med, har suiten blitt endret omtrent hvert tredje år. (1989, 1992, 1995)

SPLASH En viktig grunn for å lage suiter av programmer for å måle ytelse, er at man i større grad kan sammenligne testresultater. Ett av målene med Stanford Parallel Applications for Shared-Memory (SPLASH), var å lage en

suite av realistiske applikasjoner som kan benyttes til å evaluere og teste design av systemer basert på fellesminneprogrammering[67].

Denne suiten har blitt videreutviklet under navnet SPLASH-2, både med forbedrede algoritmer, bedre tilpassning til moderne minnesystemer og med programmer som skalerer til et større antall noder[78].

NAS Numerical Aerospace Simulation (NAS) Parallel Benchmarks (klasse 1), utviklet ved NASA Ames Research Center, var spesiell fordi den spesifiserte algoritmen som skulle benyttes, men overlot selve implementasjonen til de som ønsket å utføre tester. De argumenterte med at nye, avanserte parallelle systemer ofte trengte nye tilnærminger og metoder som langt i fra lignet tradisjonell kode for en vektor- eller vanlige, sekvensiell maskin. Testene burde være generelle og ikke favorisere noen spesiell arkitektur. Derfor utelukket de meldingsutveksling som et krav for å kunne kjøre testene. Korrekthet, både for resultatene og for ytelsesmålingene måtte være enkelt verifiserbart, og minnekrav og kjøretid måtte på enkel måte kunne endres for å teste nyere, kraftigere systemer[6]. De stilte forøvrig mange krav til implementasjonen, blant annet at den måtte skrives i C eller Fortran-90 og benytte 64 bits flyttall.

Det ville kreve mye ressurser å implementere testene etter slike spesifikasjoner, og derfor ble også en referanseimplementasjon skrevet i Fortran distribuert.

Det viste seg at NAS-1 likevel ikke skalerte så godt til nye og raskere maskiner. Dessuten var det et problem at leverandørenes implementasjoner var hemmelige og meget nøye tilpasset for å få gode tall, noe som gjorde det vanskelig for andre å reprodusere resultatene som ble publisert. Derfor ble en annen utgave publisert i 1995 som skulle supplere NAS-1[7] og imøtegå noen av problemene.

7.2.4 Hva tester programmene?

Hvis vi kjører et program og måler eksekveringstiden på ulike konfigurasjoner, hva er det egentlig vi tester?

Et lite program som går i én løkke med få beregninger tester kanskje flyttallsoperasjonene i CPUen. Et program som bruker en liten del av minnet tester kanskje cache. Andre programmer som er avhengig av viktige tjenester fra operativsystemet, tester kanskje implementasjonen av nettverksprotokollen eller interrupthåndtering i operativsystemet.

7.3 Testapplikasjoner

Pete Keleher har benyttet CVM til å demonstere ulike teknikker, protokoller og problemer ved DSM i en rekke artikler[39, 40, 42]. For å demonstrere

og sammenligne ytelse har han benyttet en rekke applikasjoner og kjerne fra programmer som utfører vitenskapelige beregninger.

Vi har valgt å benytte et utvalg av hans programmer. Dette er gunstig for å kunne sammenligne resultatene av våre studier med hans. Dessuten lettet det vårt arbeid, siden vi slapp å tilpasse programmer. Videre har vi funnet frem til en annen interessant måte å måle ytelse, prosjektet Hint, som vi vil se litt nærmere på.

Til slutt har vi også ønsket å undersøke den potensielle ytelse til *remote store*, og derfor har vi tatt med et lekeprogram som vi har kalt «Globalsum».

Resten av kapittelet bruker vi til å gå igjennom de ulike programmene, fortelle om karakteristikkene deres, og hva de er egnet til å demonstrere.

7.3.1 Barrier

Barrier er ett av testprogrammene som følger med CVM, og det er egnet til å teste forsinkelsen i meldingsutveksling. En barriere krever $2(n-1)$ (i dette tilfellet: korte) meldinger. Programmet tar tiden på 1024 barriérer, og regner ut antall millisekunder i snitt per barriére.

Programmet sier svært lite om ytelsen til applikasjoner som beregner noe meningsfylt, men gir oss litt informasjon om primitivene i systemet.

7.3.2 Globalsum

I en del type applikasjoner deles løsningsrommet mellom nodene, og hver node beregner sitt svar og til slutt samles svarene. Det kan implementeres ved at alle nodene skriver sitt svar inn i et globalt array for løsninger. Dette arrayet kan typisk ligge på én maskin.

Programmet «Globalsum» er meget lite, men tester dette viktig aksessmønsteret i distribuerte programmer. Det lar hver node skrive et tall basert på sin nodeid inn i tildelte områder av et globalt array. Vi får dermed testet egenskapene når mange noder skriver samtidig til en eller et fåtall sider.

7.3.3 FFT

Dette programmet utfører 3-D FFT beregninger, og er hentet fra NAS Parallel Benchmark suite i Fortran, oversatt til C og tilpasset til CVM.

7.3.4 Water

«Water» er en simulering av molekylær dynamikk. Den simulerer krefter i et system med vannmolekyler i væskeform. (N-body molecular dynamics)

Programmet er opprinnelig fra Perfect Club benchmark suite. Deretter er det oversatt til C med PARMACS makroer i forbindelse med Splash-1[67] og senere til TreadMarks og CVM[37]. Det finnes to vanlige utgaver av «water», og dette er som ofte kalles «n-squared».

7.3.5 Hint

Hierarkisk integrasjon (Hint) beregner de rasjonelle grensene for arealet i xy planet hvor $x\epsilon[0,1]$ og $y\epsilon[0,\frac{1-x}{1+x}]$.

Den beskrevne algoritmen sier at man skal dele inn x og y i et antall delintervaller som er en potens av 2. Deretter finner man ut hvor bidraget til usikkerheten er størst, og forbedrer dette området. Dette gjøres helt til man ikke klarer å allokere mer minne, presisjonen til tallene ikke lenger er god nok, eller kjøringen tar for lang tid.

Det er ingen begrensning matematisk på hvor høy kvalitet man kan nå, men presisjonen og tilgjengelig minne i maskinen setter en praktisk begrensning oppad. Ifølge forfatterne betyr dette at Hint skalerer uendelig[30].

Den adaptive algoritmen hevdes også å fange karakteristikker fra applikasjoner som Barnes-Hut, Greengard for n-body dynamikk, og Quasi-Monte Carlo. Metodene som er benyttet i disse programmene tar vare på de største bidragene til feil eller usikkerhet, og forfiner deretter svaret ved å beregne disse områdene på nytt.

Kvaliteten på svaret er definert som den inverse av differansen mellom øvre og nedre grense. Den vil øke som en stegvis funksjon av tid for hver forbedring som beregnes.

Resultatet i Hint gis som en kurve over Quality Improvements Pr Second (QUIPS), som viser kvalitetsforbedring som en funksjon av kjøretiden. For de som ønsker å få ett enkelt nummer ut, har de definert NetQuips som arealet under QUIPS-kurven med en logaritmisk tidsskala.

Programmet har lite kommunikasjon underveis, kun en kollaps av sum i slutten av hver iterasjon. Dessuten kommuniseres informasjon fra sjefsnoden om blant annet hvilket intervall som skal beregnes i neste iterasjon.

Forfatterne hevder at enkeltpunkter på Hint-kurven samsvarer meget godt med ulike av de tradisjonelle benchmarks[29], at Hint nærmest er et supersett av tradisjonelle benchmarks.

7.4 Rapportering av resultater

Vi har valgt å kjøre testprogrammene først én gang for å fylle cachene, og deretter flere ganger hvor vi har målt kjøretiden. Det som rapporteres er gjennomsnittlig kjøretid, standardavvik og antall kjøringer.

Et alternativ var å kjøre mange repetisjoner, og velge den laveste (beste) verdien. Imidlertid kunne det tenkes at en spesiell situasjon medførte at denne verdien sjelden ville oppstå. Derfor vil vi ved å repetere testene og rapportere standardavviket sikre at resultatene er reproduserbare. I de tilfeller hvor standardavviket har vært usedvanlig høyt, har vi sett etter andre prosesser som har blitt kjørt i klusteret samtidig, fjernet disse og repetert testen, slik at standardavviket har kommet ned i et akseptabelt nivå.

Keleher oppfordrer nå om at man rapporterer resultater som kjøretid fra og med andre iterasjon til terminering i iterative programmer[38]. Dette begrunner han med at oppstartskostnader kan være store og varierende, og ha lite med selve protokollens eller systemets totale ytelse å gjøre. Ved å måle tiden etter første iterasjon, burde systemet ha nådd en steady-state. Dessuten kan man argumentere for at denne første iterasjonens andel av total kjøretid vil bli meget lav i en større kjøring, og at vi derfor kan se bortifra dette. Vi kaller denne protokollen for «indreløkke-måling».

Denne måleprotokollen kan kanskje gi et riktigere inntrykk av hvordan ytelsen skalerer ved et stort antall repetisjoner, fordi andelen av total kjøretid som går til flytting av hjemmenoder – som skjer etter første iterasjon, og initiell kommunikasjon da vil bli veldig liten[39]. Det er ihvertfall grunn til å tro at rapportert speedup vil bli bedre (i.e. høyere tall) med denne måleprotokollen.

Tidsmålingene som vi ser på er i hovedsak startet idet programmet har initialisert seg, stopper når det terminerer. Initialisering vil si at prosessene er startet på alle nodene, og at det delte minnet er satt opp i programmene. Beregningene har ikke startet. Vi har dessuten kjørt enkelte av testene i tillegg med indreløkke-målinger, og kan dermed sammenligne resultatene fra de to målemetodene.

Problemer med rapportering av resultater De fleste artikler vi har sett på dette området nevner ofte bare speedup i forhold til én node, eller i forhold til en annen protokoll. Noen ganger nevnes også kjøretiden, eventuelt normalisert kjøretid. Svært sjelden rapporteres noen form for varians mellom to eller flere kjøringer.

Problemet med dette er at vi ikke vet usikkerheten i de rapporterte speedupratioene. Til tross for at det i noen tilfeller konkluderes med at én løsning er raskere enn en annen, finner vi sjelden noen avklaring av det statistiske signifikansnivået av resultatene.

I en artikkel som publiseres våren 1999 argumenterer forfatterne Shi og Tang for at man bør benytte konfidensintervall og aksepterte statistiske metoder når man sammenligner ulike systemer og metoder. De går igjennom flere sentrale artikler og demonstrerer problemer med de analysene som er gjort ved å sammenligne ratioer uten hensyn til usikkerhet[66].

Kapittel 8

Utvidelse – Nye protokoller

I dette kapittelet vil vi først se kort på relaterte DSM-prosjekter. Deretter etablerer vi bakgrunnen for eksperimentene, før vi ser på og diskuterer våre løsninger og implementasjoner.

8.1 Relaterte prosjekter

Vi har nå sett på ulike konsistensmodeller, protokoller og nettverksteknologier. Dermed har vi også diskutert ulike begreper som er nødvendige når vi skal se på de vesentligste egenskapene ved relaterte DSM prosjekter.

La oss derfor se på et utvalg av tidligere implementasjoner av DSM.

Ivy kalles den første implementasjonen av virtuelt fellesminne («shared virtual memory»)[48]. Systemet implementerte sekvensiell konsistens. Drøftelsene om systemet gikk på effektiviteten av ulike algoritmer for å holde sidene synkronisert, altså koherensstrategier.

Munin tilbød flere konsistensmodeller, deriblant release consistency. Dette gjorde at de kunne bringe inn noe annet nytt i forhold til tidligere systemer, f.eks Stanford DASH, nemlig at utgående skriveoperasjoner ble lagt buffer og sendt samtidig i én melding[13].

SoftFLASH er en implementasjon av DSM i software av Stanfords FLASH protokoll. Også denne baserer seg på SMP-noder, og maskinene er koblet sammen med HIPPI interconnect. Systemet ble utviklet som en del av Andrew Erlichsons doktorgrad ved Standford University, CA, og det implementerer release consistency, men med kun én skrivende node.

Han rapporterer forbedringer mellom 6.9 og 21.6 for 32 prosessorer (4 noder * 8 CPUer). Det dårligste resultatet fikk han med FFT med en forbed-

ring på 6.9. Han mener at det høye antaller prosessorer pr. node medfører problemer blant annet fordi det er så dyrt å synkronisere TLBene. Dessuten at nettverket var for tregt og hadde for lav båndbredde[20].

Shrimp består av 16 PC-noder med interconnect fra Intel Paragon. De har utviklet sitt eget nettverkskort som er koblet til minnebussen på Pentium 60 maskinene. Dette systemet tilbyr i tillegg til tradisjonell overføring via bufre også automatisk oppdatering av minnet på en annen maskin ved at en store til et lokalt minne som er delt ut, også sendes til den andre maskinen[11].

I dette systemet har de implementert blant annet AURC-protokollen som utnytter denne automatiske oppdateringen. For applikasjoner med stor grad av falsk skrivedeling, dvs. at to noder skriver til samme node samtidig, men til ulike deler av en side, var AURC-protokollen raskere enn HLRC (mellom 9% og 79% raskere). Til forskjell fra vår AURC har de lokal cache av sidene, slik at det ikke er dyrere å lese en ikke-lokal side.

Cashmere fra University of Rochester og DEC Cambridge Research Lab, MA, bruker maskiner bestående av flere prosessorer (SMP), sammenkoblet med første generasjons Memory Channel. De utnytter remote store funksjonaliteten til å implementere lazy release consistency med flere skrivende noder, hvor alle sider har en hjemmenode.

Ved oppdatering av hjemmenoden sendes differ med remote store, i tillegg til en variant som ligner AURC. Siden MC ikke hadde mulighet for remote read, så brukte de eksplisitt meldingsutveksling for å be motparten sende over en side som skulle leses.

Med 32 prosessorer (8 noder * 4 CPUer) rapporterer de en forbedring på mellom 8 og 31. For Water (som vi også benytter) rapporterer de 28 med 32 prosessorer. Systemet kjører stort sett i brukermodus, men de har gjort enkelte endringer i kjernen på Digital Unix.

Cashmere hadde bedre resultater med flere CPUer i nodene enn i Soft-FLASH, og finner svært liten kostnad ved å synkronisere TLBene. Dette skyldes i hovedsak at de har færre slike synkroniseringer ved å tillate flere skrivende noder og en katalog som gir bedre oversikt over hvilke noder som trenger synkronisering. I tillegg er deres mekanisme for TLB synkronisering mer effektiv[70].

TreadMarks er en forløper for CVM. Systemet ble utviklet som en del av Kelehers doktorgrad[37], og er et system for fellesminne implementert i brukermodus for Unix. Det tilbyr lazy release konsistens[3]. TreadMarks er nå kommersialisert og portet til mange Unixen og dessuten Windows NT. I tillegg til bindinger mot C og C++, støttes også Java og Fortran[77].

JIAJIA er utviklet ved Chinese Academy of Sciences. Systemet er hjemmenodebasert, men skiller seg fra flere andre systemer (som f.eks CVM og TreadMarks) ved at det totale minnet teoretisk tilgjengelig er summen av det utdelte minnet fra alle maskiner. Dette implementeres ved å sette av noe plass til caching av sider fra andre maskiner, og gjenbruke dette området senere. Dette kan sammenlignes med forholdet mellom sider i RAM og sider skrevet til disk i en vanlig maskinorganisering. Vi kan ha langt flere sider på disk, enn vi har mulighet til å holde i RAM[32].

SVMlib er en implementasjon av DSM som kjører i brukermodus på NT og Solaris x86, og som implementerer sekvensiell konsistens. Den kan bruke både TCP/IP og dessuten SCI som transportmekanisme. I tillegg støtter den programmeringsgrensesnittet til CVM, SPLASH, JIAJIA og et lokalt utviklet grensesnitt kalt SMI[72]. Utviklingen skjer ved Rheinisch Westfälische Technische Hochschule i Aachen, Tyskland. Dette prosjektet kommuniserer over SCI med Dolphins SCI-kort. De har således en lignende plattform til den vi benytter her.

8.2 Plattform

Vi vil først se litt nærmere på hvilken plattform eksperimentene har blitt utført på.

Laboratoriet, SCI-labben ved Institutt for Informatikk, har fire Sun UltraSparc maskiner med 128 MB internminne. Prosessoren er UltraSparc I klokket på 167 MHz og arkitekturen er dermed «sun4u/sparc». Hver maskin har et Dolphin SCI-kort basert på LC-2 chip og med Sbus-grensesnitt mot maskinen. SCI-kortene er koblet sammen i en ringkonfigurasjon, med alle fire noder på ringen. Operativsystemet er Solaris 2.6.

Nettverket ble byttet i perioden. I de innledende eksperimentene benyttet vi 10Mbit/s Ethernet, men i alle eksperimentene som sammenlignes med MPI/SCI 2 er det kjørt med 100Mbit/s svitsjet Ethernet.

8.3 Kommunikasjon i CVM

CVM har støtte for to ulike måter å kommunisere mellom maskiner, via en meldingsutvekslings protokoll laget over UDP/IP [57] eller via MPI [50]. Kommunikasjonen er skilt ut i en egen klasse, Communication Manager, som sender meldinger ved hjelp av instanser av klassen Message. Siden vi overfører instanser av objektene binært, er vi med CVM avhengig av at alle maskinene i systemet har lik arkitektur med hensyn på hvordan data i et objekt legges ut i minnet.

Vi studerte kildekoden for kommunikasjon via UDP/IP og koden for MPI. Det ble klart at koden for MPI var kortere og mer konsis, og inneholdt færre spesielle ting (signal-håndtering, sekvensnummerering m.m) enn UDP-koden.

CVM benytter kun 10 ulike MPI-funksjoner. Vi undersøkte derfor både syntaksen og semantikken til disse ti funksjonene, ved å studere manualen til en implementasjon av MPI[28] og standarden [50].

Vi så en rekke mulige løsninger for å benytte SCI som transportmedium, og vi skiller disse utifra hvordan vi forbinder oss med CVM.

8.3.1 UDP

Det finnes implementert en UDP-stack over SCI for Dolphins kort. Vi har ikke hatt denne tilgjengelig, men den kunne vært en mulighet. Vi kunne dessuten implementere et UDP-lag over SCI.

Dette er en unødvendig komplisert løsning. UDP er en forbindelsesløs, og ikke pålitelig protokoll, mens vi har mulighet for forbindelsesorientert, pålitelig overføring over SCI ved hjelp av raske biblioteker.

8.3.2 MPI

Den mest nærliggende løsningen er derfor å ta utgangspunkt i MPI, som fra et applikasjonssynspunkt bruker pålitelig overføring.

Vi så på følgende muligheter:

MPICH I en hovedoppgave ved instituttet har Uwe Kubosch arbeidet med å tilpasse et MPI-bibliotek til å benytte SCI som nettverk[43]. Han tok utgangspunkt i MPICH[28, 27] som har meget klare grensesnitt mellom selve kommunikasjonen og MPI-kallene, noe som gjør at det å implementere støtte for en annen transportmekanisme er relativt oversiktlig. MPICH er en full implementasjon av MPI.

ScaMPI Scali AS har utviklet et MPI-bibliotek som benytter SCI til kommunikasjon i sine systemer (klynger av arbeidsstasjoner). ScaMPI implementerer hele MPI, og bruker ulike strategier for å overføre en melding avhengig av størrelsen[63]. Dette gir oss muligheten til å se hvordan en kommersiell, og fullstendig MPI-implementasjon fungerer sammen med CVM.

Minimalt MPI-lag Vi observerer at CVM kun bruker en liten del av MPIs mulige funksjonalitet. Derfor burde et minimalt MPI-lag, optimalisert for relativt små meldinger over SCI kunne gi bedre ytelse enn de to nevnte løsninger.

8.3.3 Organisering av kapittelet

I resten av kapittelet vil vi gå igjennom de ulike protokoller og transportløsninger som vi har benyttet.

Omtrent alle klynger vil kunne kjøre CVM med UDP som nettverk over for eksempel ethernet. Vi vil derfor bruke UDP som et grunnlag, og sammenligne våre algoritmer og etterhvert utvidede protokoller mot UDP. I tillegg har vi kjørt ScaMPI på mange av testene, og får således også en sammenligning på hvor effektivt et minimalt lag er i forhold til en full MPI-implementasjon.

Vår versjon av ScaMPI har dessverre hatt en god del problemer, og terminerte med feil under alle protokoller unntatt LMW. Idag foregår utviklingen kun mot PCI-baserte kort, så det er grunn til håpe at en nyere ScaMPI under en annen arkitektur ville fungere mer stabilt.

8.4 Første versjon - Naiv MPI/SCI v1.00

Vi vil her forklare hvordan vi utviklet et forenklet MPI-lag og tilpasset CVM for å benytte kommunikasjon over SCI. Til slutt vil vi drøfte hvorfor denne løsningen ikke var god nok.

8.4.1 Løsning: et forenklet MPI-lag

MPI/SCI fungerer i denne versjonen kun for to noder. Den benytter Omangs protokoll for å overføre meldinger i brukermodus[51], kalt for Low-Latency Messages, LLM.

Biblioteket kopierer meldingsobjektet fra CVM til et buffer med adresse delelig på 64 (64 byte aligned) i biblioteket, og kaller deretter Slibs sendfunksjon.

Selve LLM-protokollen fungerer som en strøm av 64 byte store blokker, og har intet begrep om meldingsgrenser. LLM sender en melding med et bestemt antall bytes, en størrelse som må være en multippel av 64, men mottageren må selv vite hvor mange bytes som skal mottas.

Med tradisjonell sockets over TCP ser applikasjonen en kontinuerlig strøm av data fra sender til mottager. For UDP er enheten datagrammer, men mottageren får oppgitt størrelsen på datagrammet, og behøver ikke å vite denne på forhånd.

Siden vi overfører meldinger med varierende lengde, er det opp til vårt lag å håndtere slike meldinger korrekt. Vi trenger å opplyse mottageren om størrelsen på meldingen.

Vi løser dette med følgende protokoll. Først overføres én blokk på 64 bytes med meta-data om meldingen, informasjon om spesielle tagger knyttet til

meldingen fra CVM og størrelsen. Umiddelbart deretter sendes selve meldingen. Det er ikke mulig at en annen melding kan bli sendt i mellom disse to meldinger.

Mottageren forsøker å lese en melding på 64 bytes. Når en slik melding blir mottatt, ekstraheres meta-data, og det korrekte antall bytes blir så lest fra den påfølgende sending fra samme senderadresse. Siden det kun er én sender pr. node, vil dette fungere.

8.4.2 Diskusjon

Denne implementasjonen er ganske enkel og naiv. Den fungerer kun mellom to noder og bruker et simplest mulig lag mellom CVM og SCI ved å nesten koble MPI-kallene direkte til tilsvarende Slib-kall.

En ulempe er at implementasjonen flere ganger kopierer mellom bufre. Ved sending kopieres først data fra CVM til et 64-byte alignet buffer og deretter til Slib. Dessuten kopieres det ved mottak fra et Slib-buffer til et 64-byte aligned internt buffer i MPI/SCI og til slutt tilbake til CVM. I tillegg benytter den to sendekall for hver hele melding som overføres. Alt dette bidrar til å øke overhead ved kommunikasjonen.

Videre fungerer den kun med to noder, og det er for lite. For å se hvordan mulighetene er for skalering bør vi ha en løsning som kan vokse til et større antall noder. Siden vår lab har fire maskiner tilgjengelig, burde vi ihvertfall ha en effektiv implementasjon som utnytter alle maskinene.

8.5 Andre versjon av MPI/SCI

Erfaringene med å utvikle den første versjonen av MPI/SCI, ble benyttet for å arbeide mot målet om å ha en løsning som både virker med og skalerer ytelsesmessig til flere enn to noder.

Som tidligere nevnt har Uwe Kubosch i en hovedoppgave ved instituttet arbeidet med å tilpasse et MPI-bibliotek til å benytte SCI som nettverk[43].

I oppgaven kommer han frem til en algoritme og implementerer rutiner for meldingsutveksling over SCI med Slib som fungerer for flere noder enn to. Vi tok derfor utgangspunkt i hans algoritme og tilpasset den til vår implementasjon i CVM.

Det ville selvsagt vært mulig å benytte hele MPICH, men det ville gi økt overhead, og siden vi ikke benytter så store deler av funksjonaliteten i MPI, mener vi at det var bedre å lage et bibliotek som var minimalt for å få bedre ytelse.

Algoritme

For hvert mottager-sender par opprettes det en forbindelse. Nodene er altså koblet alle-til-alle (fullt koblet). Det betyr at antall forbindelser vokser kvadratisk med antall noder, $O(n^2)$.

En slik kostnad ved skalering vil begrense hvor godt løsningen skalerer til et stort antall noder, av flere grunner. Hvert buffer tar opp en del minne som går på bekostning av minne til applikasjonen. I det vi bruker så mye minne at sider som brukes av programmet blir lagt ut til disk (såkalt *paging*), vil ytelsen gå dramatisk ned, siden aksesstiden til sider på disk er mye, mye større enn mot primærminnet.

Slib benytter ikke interrupter ved mottak av meldinger, og det er én av grunnene til den lave forsinkelsen som er mulig ved bruk av meldingsutveksling i Slib. For å undersøke om det har kommet en melding til et buffer, må det derfor sjekkes et flagg som markerer innkommende meldinger, det vil si at vi benytter en pollestrategi.

Det vil være et mottagerbuffer for hver node på på alle maskiner. Hver gang CVM sjekker om det har kommet en melding, sjekkes samtlige bufre for nye meldinger. Tiden dette tar øker linært med antall noder, $O(n)$.

Det er derfor god grunn til å drøfte andre mulige løsninger som kanskje kan unngå noen av disse problemene.

8.5.1 En annen mulig løsning: felles mottagerbuffer

Det kan virke som det er ganske vanlig at implementasjoner av MPI har ett, felles buffer for mottak av meldinger – én mottagerkø pr. node[63, side 29]. I ScaMPI benytter Scali én kø for hver node sender, slik som vi har valgt her.

Dersom man ønsker å bruke ikke-blokkerende lesekall fra MPI, vil man først sjekke om det finnes en melding klar til levering, ved hjelp av funksjonen kalt «probe». Deretter henter man meldingen ved hjelp av «recieve».

Litt slurvete bruk av MPI-funksjoner gjør at den meldingen som man tror man mottar etter en probe, ikke nødvendigvis er den man får ved recieve, dersom MPI implementasjonen bruker flere mottagerkøer. Veldig mange programmer har implisitt forutsatt én mottagerkø for meldinger, deriblant CVM. Denne feilen i applikasjonenes bruk av MPI er dog enkel å korrigere[63].

Forutsetninger for å bruke felles mottagerbuffer

Ville det være mulig å benytte ett, felles buffer for mottak av meldinger også for MPI/SCI, men likevel beholde den lave forsinkelsen?

En mulig implementasjon av dette kunne benytte en lås av mottagersidens buffer. Når en node (n_1) ønsker å sende til en annen node (n_2) må n_1 sørge

for eksklusiv tilgang til mottagerbufferet hos n_2 ved hjelp av en lås (*mutex*). Slib har støtte for låser, men vi støter på et arkitekturproblem.

Dersom alle noder skriver til ett felles buffer i hver mottager, får vi problemer med låsingen på våre maskiner. De er UltraSPARC 1 basert med Sbus. Å få en lås med SCI-kortene fra Dolphin for Sbus med LC-2 tar minst $60\mu s$[53], mens det på PCI-kortene basert på LC-2 er mye raskere, drøyt $10\mu s$[54]. Dette vil altså si at et lock/unlock par vil ta over $120\mu s$, og det er mye. Forsinkelsen når man overfører én melding er nede i $28\mu s$ for en eldre versjon av kortene[51].

En viktig innvending mot denne algoritmen er problemer som vil oppstå dersom alle nodene ønsker å sende til én node samtidig, såkalt «*hot reciever*». Da vil samtlige noder konkurrere om låsene, og vi vil få en forsinkelse på minst $n * 120\mu s$ før den siste noden er ferdig med sin melding.

Her kan vi også få en utsultingseffekt (*starvation*), dersom nodene ikke plasseres i en rettferdig kø for å få tilgang, men konkurrerer om å vinne låsen i hvert tilfelle.

Denne løsningen vil skalere bedre med hensyn på plass, men gitt den store kostnaden ved låsing, vil konkurransen om låsene være en sterkt begrensende faktor ved et større antall noder. Dessuten vil tiden for å få en lås overstige den tiden vi håpet å benytte for å overført en hel melding.

8.5.2 Optimalisering av meldingsutveksling

Vår første implementasjon av MPI/SCI for flere noder bygget på resultatene fra den mest avanserte versjonen i [43]. Ved polling av de ulike meldingskøene, ble alle køene sjekket fra nummer 1 og oppover til det ble funnet en melding som ikke var lest, eller at samtlige køer var tomme.

Round-robin

Vi observerer at dersom noden som har meldingskø nummer 1 sender kontinuerlig, vil ikke meldinger fra de andre meldingskøene (nodene) bli lest i det hele tatt, og vi får utsulting.

Hvilke andre varianter kunne tenkes? Man kunne velge hvilken node som skal sjekkes tilfeldig. En annen variant som er ganske vanlig ved et slikt problem, er å sjekke køene etter tur, slik at dersom meldingskø nr 2 mottok en melding sist, så starter vi med å sjekke nr 3. Dette kalles for *round-robin* algoritmen.

Effektivisering av metadata

Når en melding skal overføres i MPI/SCIble det i Kuboschs implementasjon, og i vår første utgave, overført to meldinger med LLM. Den første inneholdt meta-data, og deretter ble meldingen sendt. Hvert kall til LLM bruker ca 20μs fordi rutinen ikke returnerer før meldingen er sendt og mottatt i minnet på den andre maskinen.

Vi legger merke til at mottageren ser en strøm av blokker. Derfor kan vi sette sammen metadata og nyttelasten, selve meldingen, til en stor melding som vi sender med LLM. Mottageren leser som tidligere først 64 bytes, hvor lengden på resten av meldingen ligger beskrevet. Dette sparer ett sendekall pr. sending og gav store forbedringer, noe vi kommer tilbake til senere.

8.6 Protokoller som utnytter SCI

SCI har funksjonalitet for å mappe inn minne fra en annen maskin inn i det lokale virtuelle minnet. Lesing og skriving til dette minnet gjøres med prosessorens vanlige *load* og *store*-instruksjoner. MPI/SCI har utnyttet dette til å få høyhastighets meldingsutveksling som benyttes til kommunikasjon mellom nodene i CVM.

Det andre målet med oppgaven var å undersøke om vi kan utnytte noe av SCIs spesielle funksjonalitet i et software DSM. Konkret ville vi lage en protokoll som utnytter remote store.

I [79] viser forfatterne at LRC[36] ikke skalerer veldig godt med et stort antall noder, fordi det genereres veldig mange meldinger. De foreslår og har implementert to protokoller som bygger på LRC: Automatic Update Release Consistency (AURC) og Home-based Lazy Release Consistency (HLRC). Disse protokollene er forklart grundigere i henholdsvis avsnittene 5.1.7 og 5.1.8.

Nettverksgrensesnittet som ble utviklet til Shrimp kan snoope på minnebussen og dermed se oppdateringer til fysisk minne. Dette har de utnyttet til å lage automatisk oppdatering av sider.

Man kan sette opp en mapping fra en lokal fysisk side til en side på en annen node. Når applikasjonen skriver til den lokale siden, sendes det automatisk oppdatering til den andre noden i bakgrunnen.

Ved å utnytte denne funksjonaliteten, kan man bruke det lokale minnet som cache, slik at dersom man ved miss først overfører hele siden, vil man kunne lese med hastighet til det lokale minnet, og oppdatere både siden på hjemmenoden (remote) og lokalt samtidig. Da slipper man å oppdatere sidene ved synkronisering slik man gjør tradisjonelt ved enten å overføre hele siden, eller å generere, overføre og anvende differ.

Denne funksjonaliteten er ikke tilgjengelig med Dolphins SCI-adaptre. Vi

må derfor enten velge om en side skal mappes til hjemmenoden (remote), og da vil både lesing og skriving gå over SCI, eller benytte en vanlig protokoll og ikke utnytte spesiell SCI funksjonalitet.

8.7 AURC over SCI

Vi vil nå se grundigere hvordan AURC over SCI er implementert i CVM. Først ser vi på den grunnleggende implementasjonen, og til slutt på to varianter som tilbyr remote store på utvalgte sider etter ulike vi kan lage mer avanserte protokoller som bygger på viderekriterier.

8.7.1 Minneallokering

Ved initialisering av CVM settes det av et kontinuerlig minneområde med funksjonen valloc. Størrelsen kaller vi arenastørrelse og er produktet av maksimalt antall sider og sidestørrelsen.

Deretter opprettes det en Slibområdepeker pr node, og vi deler ut et område med arenastørrelse for tilknytning fra andre noder. Dette minnet blir frosset, slik at det ikke kan pages ut til disk.

Deretter ber vi SCI-driveren om å sette opp en mapping fra alle andre noders arena inn til I/O-driveren, men vi utsetter å sette opp mappingen fra det virtuelle minnet og til de andre nodenes minne.

Det vil altså si at alle mulige mappinger er satt opp, men at applikasjonene ikke har tilgang til de andre nodenes minne. For å få denne tilgangen behøver vi kun ett mmap-kall. Dette forbereder oss på å kunne få tilgang til de andre nodenes minne meget hurtig.

Inn i vårt adresserom for arenaen mappes så minnet som er delt ut for SCI fra denne noden. I utgangspunktet er altså vårt lokale minne det som er satt opp, men tilgangen skjer via SCI-kortet.

8.7.2 Diskusjon

Ved denne metoden er det flere klare begrensninger. Vi har et stort forbruk av adresserom. Det er en begrensning på hvor mye minne SCI-driveren klarer å håndtere, og i tillegg vil det være en begrensning i størrelsen på I/O-minne, altså fysiske minneadresser som er satt av til et I/O-kort.

SCI har et adresserom på 48 bit pr. node, noe som neppe vil være noen begrensning for oss.

Det fysiske adresserom tilgjengelig på en Ultra2 41 bits[71]. Problemet er at etter SBus-spesifkasjonen er det kun 28 bits tilgjengelig pr. kort[71], det vil si 256 megabyte adresserbart minne.

Siden vi har støtte for at hjemmenoden for hver enkelt side kan endre seg under kjøring, må vi la alle nodene kunne dele ut minne. Imidlertid tar det relativt lang tid å sette opp utdeling av minne, derfor bør vi gjøre (og har gjort) dette før selve beregningene starter.

Dersom alle sider hadde hatt statisk hjemmenode ville vi ikke hatt den samme begrensningen på adresserom. Imidlertid måtte vi krevd apriori statisk plassering av sidene på hjemmenode. Plasseringen av sider i en hjemmenode-protokoll er helt kritisk for ytelsen[39].

Selv om vi har en kopi av en side lokalt, så skriver vi til minne som er delt ut. Dette har da fysisk adresse til I/O-minnet, og er under kontroll av en I/O-driver.

Minnesystemet vil ikke kunne cache data fra slike sider, fordi endringer skal være synlig umiddelbart. Derfor vil minneaksess lokalt være tregere med denne protokollen enn med de protokoller som ikke kan benytte remote store, men dermed kan utnytte cachene.

8.7.3 Sideobjekt

I CVM har hver side et objekt som inneholder tilstandsinformasjon om siden og dessuten prosedyrer for å håndtere meldinger, sidefeil og annet.

Vi har lagt til et boolsk attributt i siden som sier om den for øyeblikket er mappet til lokalt, fysisk minne, eller om sidens adresseområde peker til en annen maskin.

8.7.4 Protokoll

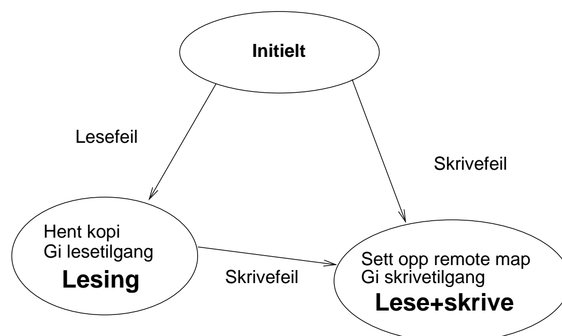
For å implementere selve protokollen, tok vi utgangspunkt i CVMs hjemmenodebaserte protokoll med flere skrivende noder, HLRC (Homebased lazy release consistency). Men istedet for å hente over en kopi av siden ved skrivefeil, så bruker vi en mapping direkte til hjemmenodens minne. Et tilstandsdiagram for protokollen vises i figur 8.1.

Ved synkroniseringspunktene må vi velge om vi ønsker å beholde mappingen, eller om vi skal skru dem av. Dersom vi ikke skrur av mappingene, vil vi kunne få meget dårlig ytelse dersom en node begynner å lese fra en side den tidligere har skrevet til, siden lesing fra en annen node over SCI er flere ordner tregere enn å lese fra lokalt minne. I denne første versjonen av AURC beholdes mappingen ved synkroniseringspunkter.

Protokollen HLRC har mulighet for å endre hjemmenode for en side etter hvilken node som sist skrev til siden, eller etter applikasjonens ønske. Dersom en sides hjemmenode endres, må vi skru av alle remotemappinger, og dessuten sette den gamle lokale kopien til å være ugyldig.

Dersom det har kommet skrivenotiser på en side som vi har mappet til

hjemmenoden, behøver vi ikke å gjøre noe som helst. Er siden ikke mappet derimot, men lesbar eller skrivbar, så må en lokal kopi av den invalideres, og siden må eventuelt hentes på nytt dersom den skal benyttes igjen.



Figur 8.1: Tilstandsdiagram for AURC 1

Det er to viktige endringer i sidehåndteringskoden:

Mapping Vi har skrevet en ny funksjon for å sette opp mapping av en side i CVM til en ønsket node. Den har signaturen `map_page (int node)`, og beregner offset inn i arenaen for SCI-minnet for siden. Deretter fjerner den mappingen fra den gamle virtuelle adressen, og setter opp en mapping til SCI-minnet for den siden på den ønskede noden. I tillegg finnes det statistikkfunksjoner som tar vare på hvor mye tid som går med til å utføre disse mappingene og antall ganger.

Sideobjektets mapped_remote attributt blir oppdatert til å gjenspeile om siden for øyeblikket er lokal eller peker til en annen node.

Sidefeil En sidefeil behandles forskjellig avhengig av typen. Dersom vi får en lesefeil, så sendes en lese-forespørsel til hjemmenoden, og i retur får vi siden som legges inn i det lokale minnet. Ved skrivefeil mappes derimot siden til hjemmenoden, og gjøres skrivbar.

8.7.5 Oppsummering

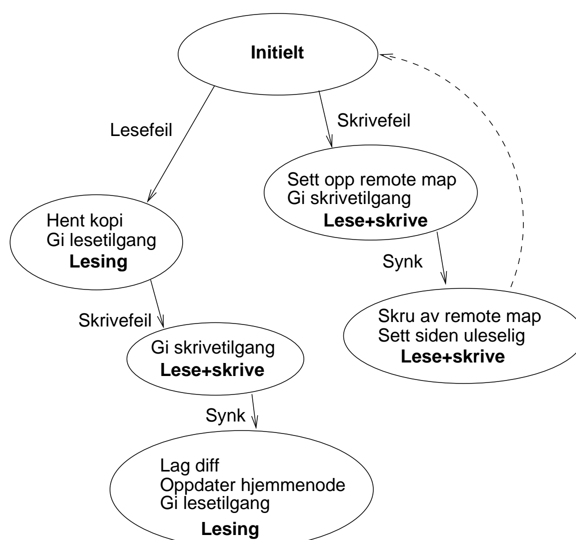
En side som skrives til på en node, vil altså settes opp mot hjemmenoden, og forbli oppkoblet til programmet terminerer, eller at sidens hjem flyttes. En slik flytting kan skje eksplisitt i programmet, eller etter en viss kjøretid ved hjelp av parametre som gis til CVM ved oppstart.

8.8 AURC2 – Remote store ved skrivefeil

Vi ønsket å la sider som primært ble skrevet fra en lokal node være koblet mot hjemmenode, og ellers la de andre sidene benytte den vanlige protokollen, HLRC.

Dette fordi vi vet at å lese over SCI fra hjemmenoden tar meget lang tid. Derfor var det først ved skrivefeil vi satte opp koblingen mot hjemmenoden.

Uheldigvis støtter de færreste MMUer kun skrivetilgang til en side, og heller ikke våre UltraSparcer. Vi legger merke til at programmer som kun skal fortelle en annen node en verdi, ikke trenger å lese lokasjonen først. Derfor implementerte vi AURC2 slik at en kobling mot hjemmenoden kun ble opprettet dersom vi hadde en skrivefeil til en side som vi ikke allerede hadde lesetilgang til.



Figur 8.2: Tilstandsdiagram for AURC 2

Dersom vi ved synkroniseringspunkter mottar en skrivenotis fra en annen node vil vi skru av all tilgang til denne siden lokalt, og dersom siden var mappet til hjemmenoden, så blir mappingen fjernet. Det siste er nødvendig for å håndtere en senere lesefeil, for da ønsker vi at siden skal hentes over lokalt, og leses fra lokalt minne.

8.8.1 Sidefeilhåndtering

Ved en skrivefeil, så vil siden kobles til hjemmenoden hvis og bare hvis siden nå er mappet lokalt, men ikke har lesetilgang. Ellers vil vi ved skrivefeil benytte HLRC, altså hente siden fra hjemmenoden, og lage en kopi (twin) som be-

nyttes til å generere en diff ved synkronisering. Dersom vi får en lesefeil, gjør vi som tidligere og henter en oppdatert kopi fra hjemmenoden.

Tilstandsdiagramet i figur 8.2 beskriver AURC2 og den ulike håndteringen avhengig av sidens nåværende rettigheter.

8.9 AURC3 – Remote store for utvalgte sider og dynamisk valg av hjemmenoder

Den siste protokollen vi implementerer har to viktige endringer. En side kobles nå til en hjemmenode hvis og bare hvis programmereren eksplisitt har sagt at denne siden skal benytte remote store.

Programmeren angir hvilke sider som skal ha remote store ved et nytt kall vi har innført i grensesnittet mellom applikasjonen og CVM.

Alle de større applikasjonene som benyttes som testprogrammer er iterative. Dette gjelder generelt for mange programmer som utfører vitenskapelige beregninger[39]. Basert på lignende idéer som Keleher presenterer i [39], har vi innført dynamisk plassering av sider til hjemmenoder etter første iterasjon.

I forbindelse med siste barriére i den første iterasjonen flyttes hjemmenoden for sidene til den noden som sist skrev til siden. Beslutningen tas av den noden som er sjef for barriéren, og distribueres til de andre nodene med meldingen om at de kan forlate barriéren.

8.9.1 Sidefeilhåndtering

Sidefeilshåndtering i AURC3 er som følger: En side kobles til hjemmenoden hvis og bare hvis siden er markert for å støtte remote store. En tilkobling oppheves kun ved migrering av hjemmenoden for siden.

Kapittel 9

Eksperimenter og diskusjon

Vi vil i dette kapittelet gå igjennom de ulike eksperimentene som er utført. Først vil vi se på et lite utvalg «microbenchmarks» som sier oss noe om de kostnadene til de primitive operasjoner i systemet. Deretter går vi igjennom de ulike versjonene av MPI/SCI, og dessuten AURC-protokollene, og drøfter årsaken til de resultatene vi ser.

9.1 Primitiver

For å kunne diskutere resultatene, vil det være nyttig å vite litt om hvor lang tid grunnleggende operasjoner tar. Dette vil være til hjelp når vi vil prøve å forstå hvorfor programmer og protokoller oppfører seg slik de gjør. Vi måler dette med noen «microbenchmarks», det vil si små ytelsestester som forsøker å isolere enkelte deler av systemet.

Operasjonene «remote miss», «remote read» og «remote write» er testet ved å foreta en serie med 200 målinger på rad hvor snittiden er resultatet. Snittet som oppgis i tabellen er snitt og standardavvik for 4 slike serier.

Remote (write) miss Hvor lang tid applikasjonen må vente (steile) før den kan fortsette ved remote miss. Remote miss vil si at siden ikke finnes lokalt. Dette måler altså hvor lang tid det tar å få tilgang til en side for de ulike protokoller. I dette tilfellet er det brukt en write remote miss, uten at det finnes kopi lokalt. For protokollene som benytter differ, opprettes det dessuten en kopi (twin).

Remote read Hvor lang tid det tar å lese et ord fra en side som ikke ligger lokalt i utgangspunktet, men som allerede er satt opp med leserettigheter. Dette skjer enten via en lokal kopi, eller via SCI remote read.

Remote write Hvor lang tid det tar det å skrive til en side som ikke er lokal, men som er satt opp med skriverettigheter. Enten skrives det til en lokal

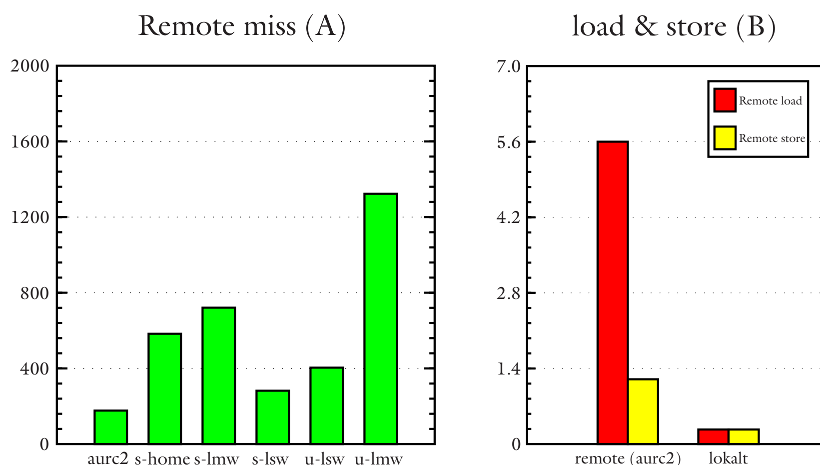
kopi, eller via SCI.

Resultatet av målingene vises i tabell 9.1 og også fremstilt grafisk i figur 9.1.

	read (μs)	write (μs)
sci	5.66	1.17
lokalt	0.28	0.27

	Miss (μs)
sci-aurc2	177
sci-home	580
sci-lsw	280
sci-lmw	720
udp-lsw	400
udp-lmw	1320

Tabell 9.1: Tid i μs for minnerelaterte primitiver



Figur 9.1: Primitiver

9.1.1 Diskusjon

Samtlige protokoller, unntatt AURC, vil ved remote miss hente over en kopi av siden fra hjemmenoden eller den noden som har en lovlig kopi for øyeblikket.

AURC setter i stedet opp en mapping mot sidens hjemmenode. Å utføre denne mappingen tar knapt 180μs, mens å overføre en hel side tar mellom 580μs og 720μs for protokollene hvor flere noder kan skrive samtidig, jf den grafiske fremstillingen i 9.1 A. Merk at LSW, som ikke behøver å ta hensyn til at det kan finnes et stort antall skrivende noder, er mye kjappere, bare drøyt 50% tregere enn AURC.

Når vi ønsker å lese et ord fra en side under AURC, så leser vi direkte fra minnet til hjemmenoden. Dette tar omtrent 20 ganger så lang tid som å lese fra lokalt minne som ikke er i cache. CPUen stopper mens den venter på at ordet skal overføres, og vi kan derfor ikke skjule latency ved å la CPUen utføre andre operasjoner mens vi venter.

Hadde vi kunnet legge leseforespørslene i pipeline, slik at vi kunne ha flere utestående leseoperasjoner, kunne vi kanskje fått omtrent normal forsinkelse på de neste ordene, gitt at de ble lest fortløpende.

Ved remote write derimot behøver ikke prosessoren å vente på at ordet er skrevet på hjemmenoden, men kan fortsette så snart nettverkskortet på I/O-bussen har bekreftet mottaket av ordet fra minnesystemet. Remote write er drøyt 4 ganger så tregt som skriving til lokalt minne, dersom man skriver mange ord tett i tett.

Vi har også forsøkt å måle latency ved skriving av færre enn 16 ord på rad, men vi får da problemer med at funksjonen som kalles for å måle tiden dominerer forholdsmessig for mye av den totale tiden. En bedre måling kan tenkes å bli gjort ved å lese av et klokkeregister og deretter skrive 16 ord og lese av klokkeregisteret igjen, og skrive denne funksjonen i assembler. Dette er dog ikke gjort.

Dersom vi kun skriver til en side, ser vi at vi kan utføre mange remote write mot en side før det ville ha lønt seg å hente over en kopi og skrive lokalt. Dette betyr at støtte for remote store *kan* gi bedre ytelse. Fordelen med remote store er bedre enn tallene her viser, fordi det i tillegg kommer kostnader i forbindelse med synkronisering av sidene, i form av å lage og utføre differ, eller overføring av hele den modifiserte siden.

9.2 Forbedring fra UDP til MPI/SCI 1

Resultatene er målt med en versjon av CVM 0.2 modifisert for å kjøre med vårt bibliotek for kommunikasjon over SCI. Sammenligningen gjøres med en umodifisert versjon 0.2 av CVM over 10 Mbit/s UDP/IP. Protokollen er LMW som er den protokollen som automatisk benyttes i CVM om intet annet angis.

I tabell 9.2 betegner n antall ganger programmet er kjørt, $\bar{t}$ gjennomsnittlig kjøretid, og sd standardavviket.

Resultatene fra tabellen er fremstilt grafisk i figur 9.2.

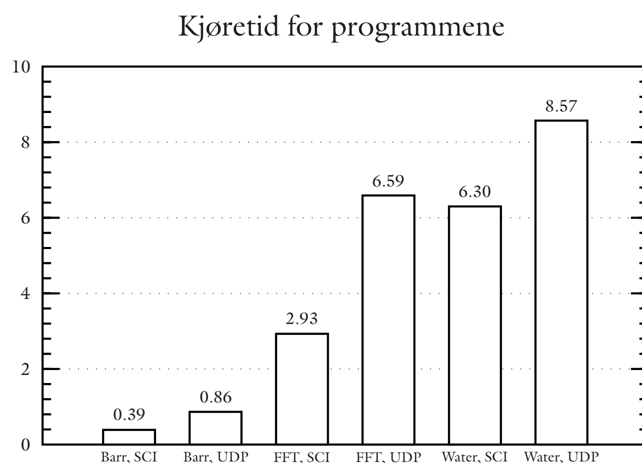
Figur 9.3 viser en sammenstilling av forbedringen, uttrykt ved speedup, ved bytte fra UDP til SCI som nettverk. Speedup er definert ved

$$\text{Speedup} = \frac{\text{Kjøretid UDP}}{\text{Kjøretid SCI}}$$

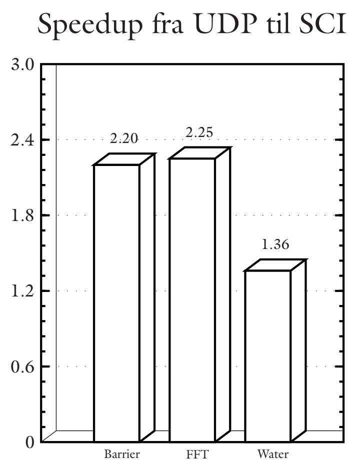
.

Barrier, SCI	$n = 10$	$\bar{t} = 0.388$	$sd = 0.003$
Barrier, UDP/IP	$n = 10$	$\bar{t} = 0.865$	$sd = 0.034$
FFT, SCI	$n = 5$	$\bar{t} = 2.93$	$sd = 0.09$
FFT, UDP/IP	$n = 5$	$\bar{t} = 6.59$	$sd = 0.13$
Water, SCI	$n = 5$	$\bar{t} = 6.30$	$sd = 2.229$
Water, UDP/IP	$n = 5$	$\bar{t} = 8.57$	$sd = 0.414$

Tabell 9.2: Resultater for MPI/SCI 1



Figur 9.2: Kjøretid for programmene



Figur 9.3: Forbedring?

9.2.1 Skalering fra én til to noder

Siden vi benytter to maskiner til å beregne resultatet parallelt, burde vi kunne forvente noe forbedring når vi kjører med to maskiner. Tabell 9.3 viser i første kolonne kjøretid i sekunder for én node, for to noder med SCI, og i tredje kolonne forholdet mellom én node og to noder, altså speedup.

	Én node	To noder	Speedup
FFT	2.54	2.93	0.87
Water	7.00	6.30	1.11

Tabell 9.3: «Forbedring» fra én til to noder

Med Water over SCI fikk vi 10% lavere kjøretid, altså en liten forbedring ved å øke til to noder. Resultatet med FFT var derimot en liten *forverring* med SCI, og en stor med UDP.

9.2.2 Oppsummering

Vi ser en relativt stor forbedring fra å benytte kringkastet 10Mbit/s ethernet med UDP til SCI, demonstrert med speedup i figur 9.3.

Likevel så gir ikke den forbedringen særlig gode argumenter for å bruke DSM; ytelsen var omtrent den samme om man bruker én eller to noder. Med FFT gikk beregningene til og med fortere for én node.

Vi ser her noe av faren ved å bruke speedup som mål for forbedring, ved at man må være oppmerksom på hva man sammenligner med. Til tross for gode resultater med endring av nettverk, var ikke resultatet særlig oppløftende når vi ser på applikasjonens ytelse fra én til to noder.

Dette illustrerer antagelig at forholdet mellom tidsbruk til beregning og til kommunikasjon er for dårlig.

Dette kan forbedres ved å ha raskere kommunikasjon, ved lavere forsinkelse i nettverket (fysisk) eller i protokollstacken (software). En annen mulighet er bedre protokoller som klarer å skjule eller minimere kommunikasjonen. En annen mulighet er å skrive om programmet slik at utnyttelsen av parallelliteten fungerer bedre med CVM.

Vi undersøkte flere av disse alternativene med MPI/SCI 2.

9.3 Skalering til flere noder – MPI/SCI 2

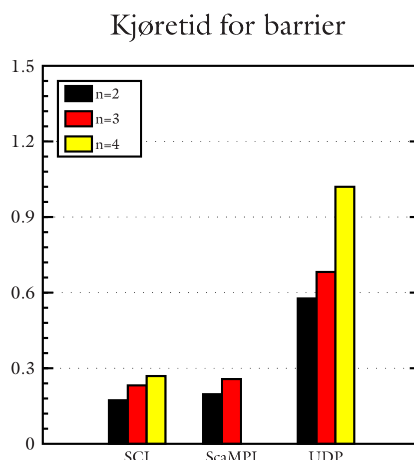
MPI/SCI 2 er implementert i CVM 0.33. Vi har kjørt programmene under denne versjonen av CVM, men på de samme maskinene som de første målingene.

Det finnes et stort antall kombinasjoner av protokoller og nettverkstyper. Vi har kjørt testene på de fleste kombinasjonene som terminerte og var praktisk mulige. Imidlertid vil fokus i dette kapittelet være på hvordan vi kan utnytte SCI til å få bedre ytelse enn med tradisjonelt nettverk. Samtlige resultater finnes i vedlegg B. De resultater som er viktige for diskusjonen blir gjengitt her og fremstilt grafisk.

Programmene er forklart i et tidligere kapittel, 7.3, og kun hovedpunkter som forklarer resultatene vil bli gjentatt og diskutert her.

9.3.1 Barrier

Tabell 9.4 viser hvor lang tid Barrier bruker for ulike nettverk for 2, 3 og 4 noder, fremstilt grafisk i 9.4.



Figur 9.4: Kjøretid for barrier

Vi ser at Barrier kjører mellom 3 og 4 ganger så fort med SCI som nettverk enn med UDP. ScaMPI er litt, men ikke mye, tregere enn MPI/SCI. Mens MPI/SCI er optimalisert for rask meldingsutveksling med relativt små meldinger, har ScaMPI den ulempe i denne sammenheng at den støtter full MPI og skal fungere effektivt for både store (flere hundre kilobyte) og små meldinger. Dette medfører antagelig større overhead ved sending av meldinger, enn med MPI/SCI.

Forbedringen fra UDP til SCI kan skyldes at selve SCI-nettverket har lavere forsinkelse, eller i kombinasjon med at implementasjonen av meldingsmottak er mer effektiv med MPI/SCI enn med operativsystemets UDP-stack. Mens operativsystemet bruker interrupts og leverer meldinger til CVM via et signal (SIGIO), poller MPI/SCI og Slibs LLM etter nye data. Levering av

	Antall noder: 2		
Barrier, SCI	$n = 10$	$\bar{t} = 0.173$	$sd = 0.005$
Barrier, ScaMPI	$n = 10$	$\bar{t} = 0.197$	$sd = 0.005$
Barrier, UDP	$n = 10$	$\bar{t} = 0.577$	$sd = 0.005$
	Antall noder: 3		
Barrier, SCI	$n = 10$	$\bar{t} = 0.212$	$sd = 0.004$
Barrier, ScaMPI	$n = 10$	$\bar{t} = 0.257$	$sd = 0.005$
Barrier, UDP	$n = 10$	$\bar{t} = 0.666$	$sd = 0.02$
	Antall noder: 4		
Barrier, SCI	$n = 4$	$\bar{t} = 0.269$	$sd = 0.003$
Barrier, UDP	$n = 4$	$\bar{t} = 1.02$	$sd = 0.02$

Tabell 9.4: Resultater for Barrier med MPI/SCI 2

meldinger skjer dessuten kun i brukermodus, og man unngår dermed kontekstskifte til kjernemodus, noe som er relativt kostbart.

Programmet Quantify fra Rational[14] utfører en profilering ved å instrumentere ferdig kompilerte objektfiler med spesielle instruksjoner, slik at det under kjøring logges hvor i programmet tiden brukes. Med SCI går mellom 80% og 90% av tiden til å vente mellom hver polling ved mottak av data i systemfunksjonen nanosleep. Med UDP brukes ca 60% i _so_recvmsg og ca 10% i _so_sendmsg. At såvidt stor andel tid brukes på å sende meldinger, kan tyde på at det er mye overhead ved bruk av operativsystemets UDP-stack. Dette er også påvist i mange undersøkelser, se f.eks [61].

9.3.2 Globalsum

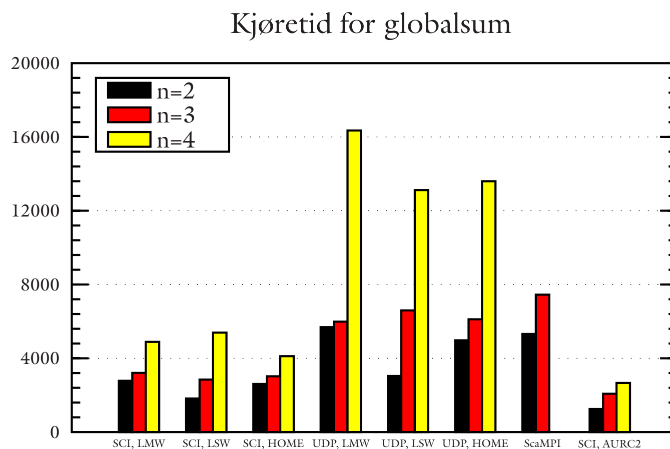
Oppstillingen i tabell B.2 i vedleggene viser hvor lang tid som ble brukt for ulike kombinasjoner av protokoller og nettverk. Resultatene er fremstilt grafisk i figur 9.5.

Siden dette lille programmet skulle demonstrere at remote store fungerer, var det ventet at AURC-protokollene ville ha klart best ytelse. AURC er rundt 30% raskere enn protokoller som tillater én enkelt skrivende node, slik som hjemmenodebasert (HLSW), lazy release konsistens (LSW) og sekvensiell konsistent protokoll (SEQ). De som tillater flere enn én, hjemmenodebasert (HOME) og lazy release konsistens (LMW) med flere skrivende noder, brukte dobbelt så lang tid som AURC.

Ytelsen med UDP var enda dårligere, og blir dessuten vesentlig dårligere både ved økning til 3 og ved økning til 4 noder.

Ping-pong effekten ved sekvensiell konsistens reduseres ved at noden er garantert å ha siden i et visst tidsrom. Med UDP tar antagelig kommunikasjon så mye av denne tiden at vi får en ping-pong effekt allerede ved to noder. Dette

demonstreres tydelig ved at UDP er nesten 15 ganger så tregt som SCI ved 3 noder og sekvensiell konsistent protokoll.

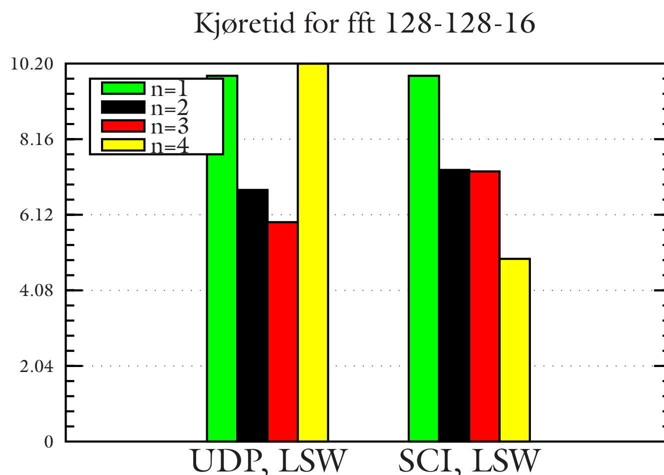


Figur 9.5: Kjøretid for globalsum

ScaMPI har dårligere ytelse for denne applikasjonen enn UDP for to og tre noder.

9.3.3 FFT

Fullstendige resultater finnes i appendix B.3. Den grafiske fremstilling i figur 9.6 viser de beste resultatene fra den innledende undersøkelsen.



Figur 9.6: Kjøretid for fft ved første forsøk

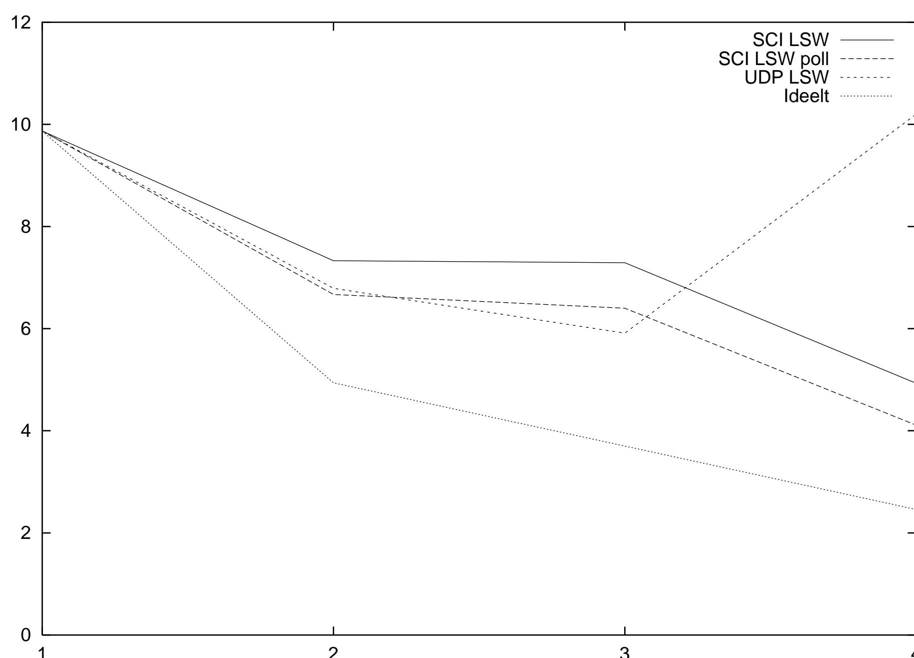
Vi har initielt kjørt FFT med størrelsen 128x128x16 i 5 iterasjoner. Resultatene viser at protokollen LSW var raskest for alle kombinasjoner av protokoller og nettverk.

Ved to noder ser vi at UDP var raskere enn SCI! Dette var uventet, og vi har derfor undersøkt nærmere for å se hva kan dette skyldes. Dessuten observerer vi at ytelsen for UDP faller katastrofalt når vi øker til fra tre til fire noder.

En undersøkelse av kommunikasjonen under SCI med to noder viste at ved flere av forespørslene har avsendernoden ventet i over 500ms for å få svar. Med en forventet kjøretid på under 8 sekunder er dette veldig mye.

Mottak av meldinger med UDP skjer asynkront, og CVM får melding via et signal (SIGIO). Med MPI som kommunikasjonsgrensesnitt i CVM sjekkes det etter innkommende meldinger ved sidefeil, ved synkroniseringspunkter og ved bruk av kall til CVM.

La oss se på et eksempel som viser hvordan det i verste fall kan bli. Dersom en vilkårlig node A, og bare A, får en sidefeil rett etter at alle nodene har forlatt en barriére, vil den måtte vente på svar til mottageren har kommet til neste synkroniseringspunkt. Mellom disse to synkroniseringspunktene vil utførelsen altså være sekvensiell.



Figur 9.7: Kjøretid for fft etter modifisering

Kjøringer av FFT med 2 noder viser at dette eksempelet ikke bare er teo-

Tabell 9.5: Statistikk for FFT

	UDP, LSW			SCI n=4		
	n=4	n=2	n=3	LSW	AURC3	HOME
bytes	9.2MB	6.09MB	12.1MB	9.2MB	15MB	15MB
msgs_sent	1215	793	1574	1207		1454
number of diffs	-	-	-	-	1884	1902
diffbytes	-	-	-	-	14.7MB	14.7MB
remote_misses	1182	786	1554	1178	1122	1139
segvs	1950	1298	2408	1950	3262	3292
sigios	1170	785	1561	0	0	0
barrierDelay (msec pr.proc)	955	36	172	463	31	55
diff_apply (msec pr.proc)					1339	1622
fault (msec pr proc)	3430	923	1122	615	538	755
fault (usec pr.proc.pr.fault)	1758	713	460	315	152	229

retisk. Et lignende tilfelle skjer rett etter en barriére hvor node 2 trenger data fra node 1, men node 1 er allerede er inne i en løkke med beregninger, og den har alle sidene den trenger, slik at den ikke får noen sidefeil. Node 2 får ikke svar får node 1 har nådd neste barriére. Utførelsen av dette området i programmet blir nærmest sekvensiell.

For å forbedre dette har vi derfor satt inn eksplisitte kall for å sjekke meldinger på kritiske steder i programmet, og dette gir stor forbedring, som vist i figur 9.7.

Vi ser at FFT over SCI med LSW protokoll etter denne endringen er raskere enn tilsvarende protokoll over UDP. I form av speedup er forbedringen henholdvis 1.5 ganger med 2 noder og 2.4 ganger med 4 noder.

Dersom vi bruker indreløkke målinger som grunnlag for speedupberegningene får vi litt bedre tall, henholdsvis 1.7 for 2 noder og 2.6 for 4 noder. Det er likevel ingen forskjell i den innbyrdes rekkefølgen mellom protokollenes kjøretid.

Ytelsesreduksjon for UDP

Hvorfor går ytelsen så drastisk ned for 4 noder med UDP? Vi ser at tiden som går med til å behandle sidefeil øker med 3 ganger fra tre til fire noder. I sidefeilsbehandlingen oppdateres sidene og det endres rettigheter til sidene via systemkallet mprotect.

Problemet kan skyldes skaleringsproblemer enten i operativsystemet eller på nettverket. Enten har nettverket store problemer med 4 noder som sender samtidig, eller så har vi funnet problemer med protokollstacken i operativsystemet, og at forsinkelsen ved bruk av et signal for å levere meldingene blir

for stor. Likevel ser vi at det sendes flere meldinger og totalt flere bytes ved 3 noder enn ved 4, noe som skulle tyde på at det er heller antall sendere, enn mengden data som er problemet.

Hjemmenodebaserte protokoller

De hjemmenodebaserte protokollene gjør det ikke like bra som LSW i dette programmet med 5 iterasjoner; de har omtrent samme ytelse som LMW.

Vi ser av statistikken i tabell 9.5 at det går med veldig mye tid til å anvende differ. I tillegg overføres det over 50% mere data ved disse protokollene enn med LSW.

FFT benytter store delte datastrukturer som benyttes på flere av nodene. Vi har kun benyttet remote store på en liten datastruktur, og gir utslag i noen færre remote miss (17 av ca 1100 totalt), og dermed færre meldinger og mindre data som overføres.

Imidlertid kan vi med de korte kjøretidene ikke påpeke noen signifikant forskjell. La oss derfor se hva som skjer når vi kjører 100 iterasjoner istedet for 5. Statistikken fra kjøringen vises i tabell B.3.1, men vi oppsummerer her resultatene for 4 noder med SCI:

	kjøretid (s)	speedup i forhold til én node
HOME	124	1.54
AURC3	97	1.98
LSW	115	1.67
LSW $n = 1$	192	1.00

Som forventet har AURC3 færrest sidefeil og færrest sendte meldinger. Ved denne større belastningen ser vi også at AURC3 har den laveste kjøretiden, og dessuten den mest stabile kjøretiden, jf standardavviket på side 97.

Dette resultatet viser at AURC3 kan skalere godt ved større beregninger når programmet er tilpasset for å utnytte remote store.

9.3.4 Water

Resultatene for den raskeste protokollen er fremstilt grafisk i figur 9.8. Fullstendig oversikt over resultatene finnes i tabellen i appendix B.4.

Protokollen LMW gav for dette programmet den beste ytelsen både med SCI og med UDP, med 5 iterasjoner (som er standard i CVM) og med 100.

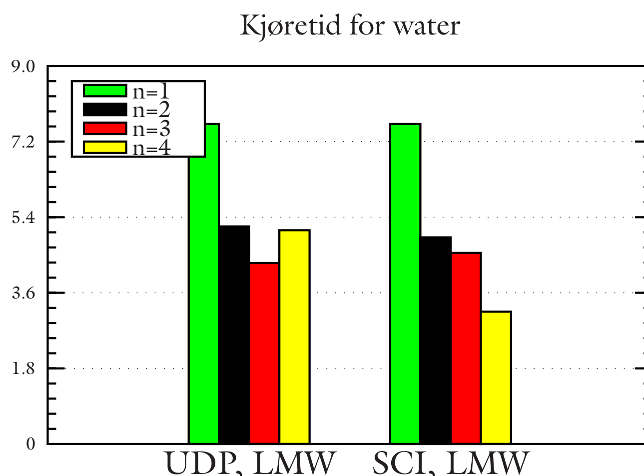
Vi har i kommunikasjonslaget (MPI/SCI) målt hvor lang tid det tar å sende en melding, definert ved hvor lang tid biblioteket steiler før det kan returnere til applikasjonen. Med 4 noders Water, bruker den i snitt ca 46 μs pr.melding. Resultatet av disse målingene finnes i appendiks B.4.1.

Av statistikken i tabell 9.6 ser vi at AURC3 som ventet sender færre meldinger enn HOME, med henholdsvis 9577 og 9747 meldinger. Men LMW bruker bare 7484 meldinger, altså 20% færre. Målt i antallet differ som er generert er forskjellen enda større; 743 for LMW og 3776 og 3960 for AURC3 og HOME.

Her ser vi én av ulempene med denne typen hjemmenodebaserte protokoller: noder som skriver til en side som de ikke eier, vil uansett automatisk oppdatere hjemmenoden ved synkroniseringspunkter. Water benytter låser til synkronisering og det utveksles konsistensinformasjon ved hver eneste lås. LMW sender ikke differ før noen trenger dem.

En mulig forbedring kan være å passe på at alle variabler som endres i forbindelse med låser finnes i minne for remote store, og undersøke om vi da kan utelate utveksling av konsistensinformasjon ved alle innhenting av låser.

Ved å la all datastruktur være på sider som oppdateres med remote store, vil vi ikke generere noen differ. Siden data blir aksessert fra flere noder, er dette lite effektivt, kjøringen tok 72 sekunder, mot 3 sekunder med LMW. Antall meldinger var i dette tilfellet 6139. Kanskje kan dette forbedres ved en optimal plassering av sidene på nodene, utifra grundig kjennskap til hvordan Water benytter data på de ulike nodene.



Figur 9.8: Kjøretid for water

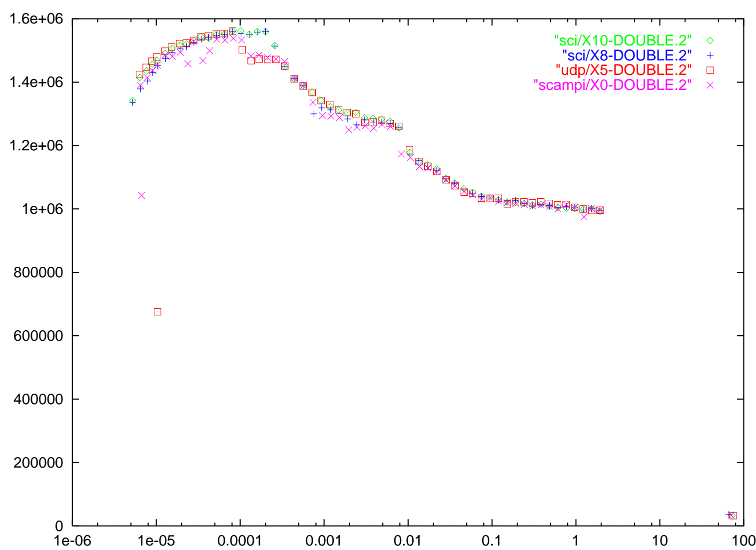
9.3.5 Hint

Hint-kurver med de beste protokollene for 2 og 3 noder er vist i figurene 9.9 og 9.10. Y-aksen viser QUIPS, mens X-aksen har enhet sekunder.

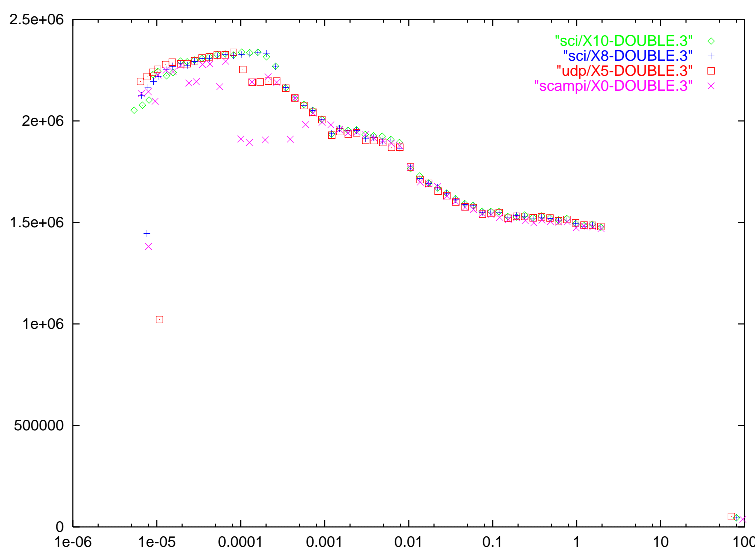
Vi ser at kurvene har to markerte knekkpunkter. De markerer grensen for

Tabell 9.6: Statistikk for Water med 4 noder

		SCI		UDP
	LMW	AURC3	HOME	HOME
elapsed inner	3.02	4.03	4.12	7.18
elapsed inner (sd)	0.01	0.02	0.03	0.03
msgs_sent	7484	9577	9747	9756
number of diffs	743	2716	2788	2788
diffbytes	1939720	329364	330996	330996
remote_miss	1027	704	790	792
segvs	1769	3776	3960	3961
barrierDelay	77	199	222	1265.0
fault	484	460	469	1276.6
fault pr	273	122	118	322



Figur 9.9: Hint kurve for 2 noder



Figur 9.10: Hint kurve for 3 noder

hvor datasettene passer inn i cachenivåene på maskinen.

Resultatene lengst til venstre, altså med lavest x-verdier, sier noe om forsinkelse (latency) i nettverket. Desto lenger tid det tar for å få det første resultatet, desto større er minste forsinkelse.

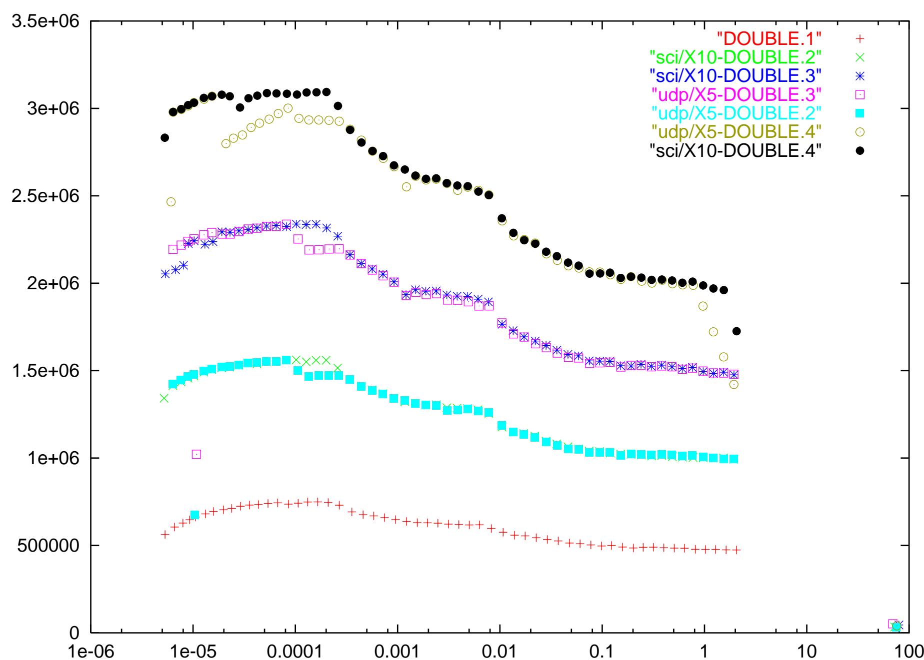
Vi ser at AURC-protokollene er de som holder seg lengst før de får det første knekkpunktet. Dette knekkpunktet er grensen for primærcache, og skulle tyde på at AURC bruker mindre minne enn de andre protokollene. Dette er riktig, fordi den ikke bruker minne til å lage og ta vare på diff'er eller mellomlagring av sider som skal overføres i bufre.

Merk at ScaMPI viser ganske varierende resultater, og tyder på at vår versjon av biblioteket har ujevn ytelse. Denne variasjonen indikerer høy varians i forsinkelse når meldinger sendes med ScaMPI.

Det er meget liten forskjell mellom resultatene med SCI som nettverk og med UDP som nettverk for Hint. Dette tyder på at programmet ikke er så veldig godt egnet til å se på relativt små forskjeller i båndbredde og forsinkelse på nettverket, dersom nodene ellers er like. Dog vil varians i forsinkelsen kunne komme ganske tydelig frem, slik vi har kjørt Hint-testene.

Skalering

Figur 9.11 viser resultatkurver fra én til fire noder. Vi observerer at avstanden mellom kurvene nesten er konstant, og at verdiene for kurven med to noder er nesten dobbelt av verdien for én node. Dette viser at Hint skalerer meget



Figur 9.11: Hint kurve for 4 noder

bra fra én til fire noder. Vi ser altså et potensiale for meget god skalering for vår løsning med DSM.

9.4 Forskjellen mellom MPI/SCI 1 og MPI/SCI 2

Vi bør se på om det gav noen forbedring å optimalisere MPI/SCI-biblioteket, for å vurdere om denne innsatsen gav resultater.

Forbedringen fra MPI/SCI 1 til MPI/SCI 2 var ganske store både funksjonalitetsmessig, ved at vi kan kjøre på et større antall noder, men også ved at applikasjonene fikk bedre ytelse. Tabell 9.7 viser forbedringen i kjøretid fra første til andre utgave av MPI/SCI med to noder og protokollen LMW.

Det gav en betydlig forbedring å optimalisere meldingsbiblioteket. Sett i lys av at vi ønsker at programmet skal kjøre fortere når vi benytter to maskiner i stedet for én, virker det klart at dette arbeidet var *helt nødvendig* for å få noe som helst forbedring i ytelse.

Lav forsinkelse i meldingsutvekslingen ser altså ut til å være en vesentlig faktor for å oppnå god ytelse med et distribuert fellesminne system i software. Dette er i samsvar med andre undersøkelser[20].

Barrier, MPI/SCI 2	$\bar{t} = 0.173$
Barrier, MPI/SCI 1	$\bar{t} = 0.388$
FFT 64x64x16, MPI/SCI 1	$\bar{t} = 2.93$
FFT, MPI/SCI 2 poll	$\bar{t} = 2.16$
Water, MPI/SCI 1	$\bar{t} = 6.30$
Water, MPI/SCI 2	$\bar{t} = 4.92$

Tabell 9.7: Forbedring fra første til andre generasjon

Kapittel 10

Konklusjon og videre arbeid

Målet for oppgaven var å undersøke distribuert fellesminne i software over SCI, både med bruk av SCI som et raskt nettverk for meldingsutveksling, men også ved å utnytte *remote store* i én eller flere protokoller.

Før arbeidet med oppgaven hadde vi indikasjoner på at den lave forsinkelsen og høye båndbredden i SCI ville være fordelaktig. Vi hadde ikke sett andre prosjekter som har undersøkt den samme kombinasjonen.

10.1 Oppsummering av resultater

La oss nå oppsummere de viktigste observasjoner og resultater vi har sett på i denne oppgaven.

10.1.1 Lav forsinkelse i nettverket

Med den første utgaven av MPI/SCI hadde vi ingen forbedring, og til dels forverring ved to noder i forhold til én. Den videreutviklede varianten, MPI/SCI 2, har lavere forsinkelse, og dette medførte forbedring i kjøretiden for applikasjonene fra én til fire noder.

Dette bekrefter tidligere resultater som hevder at lav forsinkelse er en kritisk faktor med distribuert fellesminne.

På den annen side fungerer UDP over svitsjet 100Mbit/s Ethernet bra med to noder. Grunnen til at SCI i våre forsøk ikke er raskere i dette tilfelle, kan være at vi ikke har asynkron meldingsutveksling, men må vente på synkroniseringspunkter eller sidefeil. I slike tilfeller får vi nesten sekvensiell kjøring.

10.1.2 Utnyttelse av funksjonalitet i SCI

Ett av hovedmålene var å se om vi kunne utnytte remote store funksjonaliteten i SCI til å få bedre ytelse.

Vi har sett at remote store kan gi bedre ytelse. Potensialet demonstrerte vi ved lekeprogrammet globalsum, men også ved at FFT gikk nesten 20% fortere ved ved større problemstørrelser og lengre kjøretid[1].

Dersom man lar alle sider ukritisk bruke remote store, fikk vi en forverring, det vil si en meget stor økning i kjøretiden, jf resultatene for protokollene AURC og AURC2. Vi har altså ingen mekanisme for å bruke remote store slik at vi får eksisterende skrevne programmer til å kjøre raskere uten noen form for tilpasning.

Å utnytte remote store effektivt vil derfor kreve litt av programmereren. Programmet må modifiseres slik at det markerer hvilke deler av minnet som skal være tilgjengelig med remote store. For at dette skal fungere og gi bedre ytelse, må de data som ligger på disse sidene brukes på en slik måte i programmet at dette er mer effektivt enn å hente over en lokal kopi. Dette krever kjennskap til hvordan datastrukturene i programmet utnyttes.

10.1.3 Fordeling av sidenes hjemmenode

Noe annet som er kritisk for alle hjemmenodebaserte protokoller, er tildeling av sidenes hjem. Vi har for iterative applikasjoner benyttet omplassering (migrering) etter første iterasjon til den noden som sist skrev til siden i AURC3. Ved å vite hvilke noder som skriver til sidene, kan programmereren antagelig angi en mer optimal plassering.

En annen mulig løsning er å plassere sidene statisk etter round-robin prinsippet, tilfeldig, eller ved første berøring; det vil si at vi lar den noden som først bruker en side få eierskapet. Dette er metoder som er benyttet i Cashmere[70].

10.2 Konklusjon

Vi konkluderer med at å bruke SCI kan gi til dels stor forbedring i ytelse for et system med distribuert fellesminne i software i forhold til å benytte vanlig nettverksteknologi.

Det er også mulig i en del tilfeller å utnytte spesiell SCI-funksjonalitet til å få enda større forbedring, men dette vil kreve en del kjennskap til utnyttelsen av datastrukturene hos rogrammereren.

[1] 100 iterasjoner med problemstørrelse 128x128x16.

10.3 Videre arbeid

Ett av problemene ved få noder var at vi ikke hadde asynkron meldingshåndtering over MPI. Vi løste dette ved å legge inn eksplisitt polling i applikasjonene, når de ble kjørt med få noder. En annen mulig løsning ville være å bruke en egen tråd som håndterer meldinger.

Sekvensiell konsistens er den programmeringsmodellen de fleste er vant med. Det ville vært interessant å undersøke om vi kunne bruke SCI til å tilnærme ytelsen med en sekvensiell konsistensmodell til de svakere modellene. En mulighet som kunne undersøkes er å bruke remote store i visse tilfeller med en slik protokoll.

Vår versjon av ScaMPI hadde varierende ytelse i disse undersøkelsene. Det burde undersøkes om disse problemene finnes i nyere utgaver av ScaMPI, og om nødvendig finne årsaken til denne rare oppførselen. CVM med ScaMPI som transportlag terminerte stort sett på grunn av ville pekere for alle andre protokoller enn LMW. Feilen kan skyldes både CVM og ScaMPI, men siden CVM fungerer med andre MPI implementasjoner, er det nærliggende å tro at noe er unormalt med vår versjon av ScaMPI[2].

10.3.1 Minnebruk

En stor begrensning med implementasjonen av remote store, er skalering til antall noder. Det er en øvre grense for hvor mye minne som kan deles ut, ved tilgjengelig adresserom for et I/O-kort i maskinene (256 MB). Vi har delt ut alt minne som brukes av CVM i alle maskiner. Bruken av totalt adresserom til delte sider blir

$$\text{Max antall sider} * \text{Sidestørrelse} * \text{Antall noder}$$

og vi når fort opp mot grensen. I tillegg til dette brukes det minne til buffere for meldingsutveksling, og noen buffere til statusinformasjon i Slib. Vi har kjørt med 1600 sider og en sidestørrelse på 8192 bytes. Ved fire noder bruker vi ca 52 MB av adresserommet til delte sider.

Vår implementasjon tillater at vilkårlige sider kan deles ut over SCI til enhver tid, og dessuten flyttes. Ved å kun dele ut sider som skal benyttes med remote store, vil vi kunne redusere minnebruken betraktelig.

Som et eksempel kan vi legge merke til FFT som med problemstørrelse 128x128x16 bruker alle 1600 sider til datastrukturer i fellesminne. Av disse er det kun to sider som brukes med remote store.

Utdeling av sider med remote store kan skje ved initialisering. Det betyr at hvilke sider som skal deles ut må være klart, før selve den parallelle prosesse-

[2] ScaMPI Release 1.4.3 March 11 1998

ringen starter. Alternativt kan dette skje etter den første iterasjonen, samtidig med omplassering av hjemmenode.

10.3.2 Ny arkitektur

Scalis klynger er basert på Solaris x86 og Pentium II-prosessorer med PCI-buss, sammenkoblet med SCI. Det ville være interessant å porte MPI/SCI og AURC-protokollene til å benytte denne arkitekturen, fordi det er til denne platformen de nyeste kortene og programvaren finnes.

Et EU/Esprit prosjekt, «Standard Software Infrastructures for SCI-based Parallel Systems» (SISCI), har definert standardiserte grensesnitt mellom applikasjoner og driver/kort. Målet for dette arbeidet har vært å få laget et grensesnitt som gjør at applikasjoner skal kunne være portable mellom ulike operativsystemer og kortleverandører[68]. Ved å tilpasse CVM til dette grensesnittet, vil vi kunne få en portabel implementasjon av CVM mot SCI. Alternativt kunne man prøve å ta utgangspunkt i TreadMarks og tilpasse til dette nye grensesnittet.

Tillegg A

Minnebusser

	memcpy	memset
Intel Pentium II,430TX, 233MHz, 64 MB ECC, 512kB Cache	76.89	86.00
Intel Pentium II, 350 MHZ, 128MB ECC, 100 MHz buss 440BX	116.25	162.41
AMD K6-2 350 MHz, Acer Aladdin5, 128 MB, 100MHz,	61.23	84.18
Ultra1 (vestavind) 167MHz	154.22	355.26

Tillegg B

Resultater i tabellform

B.1 Barrier

	Antall noder: 2		
Barrier, SCI	$n = 10$	$\bar{t} = 0.173$	$sd = 0.005$
Barrier, ScaMPI	$n = 10$	$\bar{t} = 0.197$	$sd = 0.005$
Barrier, udp	$n = 10$	$\bar{t} = 0.577$	$sd = 0.005$
	Antall noder: 3		
Barrier, SCI	$n = 10$	$\bar{t} = 0.212$	$sd = 0.004$
Barrier, ScaMPI	$n = 10$	$\bar{t} = 0.257$	$sd = 0.005$
Barrier, udp	$n = 10$	$\bar{t} = 0.666$	$sd = 0.02$
	Antall noder: 4		
Barrier, SCI	$n = 4$	$\bar{t} = 0.269$	$sd = 0.003$
Barrier, udp	$n = 4$	$\bar{t} = 1.02$	$sd = 0.02$

B.1.1 Quantify-resultater

	SCI 2 noder	
83%	_libc_nanosleep	polling ved mottak av data
13%	i ioctl (oppsett av driveren)	
<1%	for hver av de andre prosedyrene	
	UDP 2 noder	
60%	_so_recvmsg	biblioteksrutine for mottak av data via UDP
11%	_so_sendmsg	biblioteksrutine for sending av data

B.2 Globalsum

Antall noder: 2

SCI, LMW,	$n = 10$	$\bar{t} = 2784\mu s$	$sd = 9$
SCI, LSW,	$n = 10$	$\bar{t} = 1821\mu s$	$sd = 23$
SCI, SEQ,	$n = 10$	$\bar{t} = 1850\mu s$	$sd = 50$
SCI, HLSW,	$n = 10$	$\bar{t} = 1861\mu s$	$sd = 118$
SCI, HLSW,	$n = 10$	$\bar{t} = 1859\mu s$	$sd = 89$
SCI, HOME,	$n = 10$	$\bar{t} = 2608\mu s$	$sd = 9$
SCI, AURC,	$n = 10$	$\bar{t} = 1326\mu s$	$sd = 18$
SCI, AURC II,	$n = 10$	$\bar{t} = 1251\mu s$	$sd = 4$
UDP, LMW,	$n = 10$	$\bar{t} = 5689\mu s$	$sd = 1479$
UDP, LSW,	$n = 10$	$\bar{t} = 3041\mu s$	$sd = 43$
UDP, SEQ,	$n = 10$	$\bar{t} = 22434\mu s$	$sd = 2529$
UDP, HLSW,	$n = 10$	$\bar{t} = 5816\mu s$	$sd = 994$
UDP, HOME,	$n = 10$	$\bar{t} = 4976\mu s$	$sd = 59$
SCAMPI, LMW,	$n = 10$	$\bar{t} = 5319\mu s$	$sd = 13$

Antall noder: 3

SCI, LMW,	$n = 10$	$\bar{t} = 3208\mu s$	$sd = 80$
SCI, LSW,	$n = 10$	$\bar{t} = 2845\mu s$	$sd = 27$
SCI, SEQ,	$n = 10$	$\bar{t} = 2944\mu s$	$sd = 40$
SCI, HLSW,	$n = 10$	$\bar{t} = 2845\mu s$	$sd = 18$
SCI, HOME,	$n = 10$	$\bar{t} = 3025\mu s$	$sd = 32$
SCI, AURC,	$n = 10$	$\bar{t} = 2169\mu s$	$sd = 94$
SCI, AURC II,	$n = 10$	$\bar{t} = 2077\mu s$	$sd = 92$
UDP, LMW,	$n = 10$	$\bar{t} = 5986\mu s$	$sd = 46$
UDP, LSW,	$n = 10$	$\bar{t} = 6596\mu s$	$sd = 130$
UDP, SEQ,	$n = 10$	$\bar{t} = 43828\mu s$	$sd = 2911$
UDP, HLSW,	$n = 10$	$\bar{t} = 6301\mu s$	$sd = 1705$
UDP, HOME,	$n = 10$	$\bar{t} = 6115\mu s$	$sd = 1424$
SCAMPI, LMW,	$n = 10$	$\bar{t} = 7449\mu s$	$sd = 929$

Antall noder: 4

SCI, LSW	$n = 4$	$\bar{t} = 5391\mu s$	$sd = 41$
SCI, LMW	$n = 4$	$\bar{t} = 4890\mu s$	$sd = 30$
SCI, HOME	$n = 4$	$\bar{t} = 4115\mu s$	$sd = 15$
SCI, AURC II	$n = 4$	$\bar{t} = 2663\mu s$	$sd = 46$
UDP, LSW	$n = 4$	$\bar{t} = 16349\mu s$	$sd = 60$
UDP, LMW	$n = 4$	$\bar{t} = 13119\mu s$	$sd = 43$
UDP, HOME	$n = 4$	$\bar{t} = 13599\mu s$	$sd = 47$

B.3 Resultater for FFT

128x128x16

	Antall noder: 1		
Én node:	$n = 4$	$\bar{t} = 9.87$	$sd = 0.01$
	Antall noder: 2		
UDP, LSW	$n = 4$	$\bar{t} = 6.79$	$sd = 0.02$
SCI, LSW	$n = 4$	$\bar{t} = 7.33$	$sd = 0.01$
SCI, LSW poll	$n = 4$	$\bar{t} = 6.67$	$sd = 0.02$
	Antall noder: 3		
SCI, LMW	$n = 4$	$\bar{t} = 9.20$	$sd = 0.05$
SCI, LSW	$n = 4$	$\bar{t} = 7.29$	$sd = 0.02$
SCI, HOME	$n = 4$	$\bar{t} = 8.03$	$sd = 0.01$
SCI, AURC II	$n = 10$	$\bar{t} = 69$	$sd = 1.9$
UDP, LMW	$n = 4$	$\bar{t} = 8.30$	$sd = 0.10$
UDP, LSW	$n = 4$	$\bar{t} = 5.91$	$sd = 0.02$
UDP, SEQ	$n = 4$	$\bar{t} = 8.96$	$sd = 0.22$
SCI, HOME	$n = 4$	$\bar{t} = 8.95$	$sd = 0.12$
SCI, LSW poll	$n = 4$	$\bar{t} = 6.40$	$sd = 0.12$
	Antall noder: 4		
SCI, LSW	$n = 4$	$\bar{t} = 4.93$	$sd = 0.11$
UDP, LSW	$n = 4$	$\bar{t} = 10.20$	$sd = 0.09$
SCI, LSW poll	$n = 4$	$\bar{t} = 4.11$	$sd = 0.04$
SCI, HOME mig+poll	$n = 4$	$\bar{t} = 5.97$	$sd = 0.09$
SCI, AURC3 mig+poll	$n = 4$	$\bar{t} = 5.98$	$sd = 0.04$

64x64x16

Antall noder: 2

SCI, LMW	$n = 4$	$\bar{t} = 2.61 sec$	$sd = 0.01$
SCI, LMW poll	$n = 4$	$\bar{t} = 2.16$	$sd = 0.01$
SCI, LSW	$n = 4$	$\bar{t} = 1.81 sec$	$sd = 0.01$
SCI, SEQ	$n = 4$	$\bar{t} = 1.92 sec$	$sd = 0.02$
SCI, HOME	$n = 4$	$\bar{t} = 2.79 sec$	$sd = 0.01$
AURC	$n = 10$	$\bar{t} = 21.7 sec$	$sd = 0.02$
AURC II	$n = 4$	$\bar{t} = 21.7 sec$	$sd = 0.02$
UDP, LSW	$n = 4$	$\bar{t} = 1.71 sec$	$sd = 0.01$
UDP, LMW	$n = 10$	$\bar{t} = 2.85 sec$	$sd = 0.19$
UDP, SEQ	$n = 4$	$\bar{t} = 2.27 sec$	$sd = 0.08$

Antall noder: 3

SCI, LMW	$n = 10$	$\bar{t} = 2.92$	$sd = 0.02$
SCI, LSW	$n = 4$	$\bar{t} = 2.31$	$sd = 0.01$
SCI, HOME	$n = 10$	$\bar{t} = 3.11$	$sd = 0.19$
AURC II	$n = 4$	$\bar{t} = 19.5$	$sd = 0.23$
UDP, SEQ	$n = 4$	$\bar{t} = 3.00$	$sd = 0.02$
UDP, LMW	$n = 10$	$\bar{t} = 3.00$	$sd = 0.12$
UDP, LSW	$n = 8$	$\bar{t} = 1.59$	$sd = 0.06$
UDP, HOME	$n = 10$	$\bar{t} = 3.15$	$sd = 0.06$

Indre løkke – 128x128x16

Antall noder: 1

SCI	$n = 4$	$\bar{t} = 5.73$	$sd = 0.01$

Antall noder: 2

SCI, HOME mig	$n = 4$	$\bar{t} = 6.20$	$sd = 0.10$
SCI, HOME mig+poll	$n = 4$	$\bar{t} = 4.99$	$sd = 0.01$
SCI, LSW	$n = 4$	$\bar{t} = 3.87$	$sd = 0.01$
SCI, LSW poll	$n = 4$	$\bar{t} = 3.34$	$sd = 0.01$
SCI, AURC3 poll	$n = 4$	$\bar{t} = 4.96$	$sd = 0.00$
UDP, HOME mig	$n = 4$	$\bar{t} = 4.09$	$sd = 0.01$
UDP, LSW	$n = 4$	$\bar{t} = 3.79$	$sd = 0.02$

Antall noder: 4

SCI, HOME mig	$n = 4$	$\bar{t} = 3.74$	$sd = 0.14$
SCI, HOME mig+poll	$n = 4$	$\bar{t} = 2.74$	$sd = 0.05$
SCI, LSW	$n = 4$	$\bar{t} = 2.68$	$sd = 0.14$
SCI, LSW poll	$n = 4$	$\bar{t} = 2.24$	$sd = 0.08$
SCI, AURC3 poll	$n = 4$	$\bar{t} = 2.73$	$sd = 0.06$

UDP, HOME mig	$n = 4$	$\bar{t} = 6.53$	$sd = 0.01$
UDP, LSW	$n = 4$	$\bar{t} = 5.86$	$sd = 0.02$

B.3.1 Statistikk for 100 iterasjoner av FFT

	SCI			UDP	
	HOME	AURC3	LSW	LSW $n = 4$	LSW $n = 1$
msg_sent	37730	36937	39627		
remote_miss	36836	36340	38820		
segv	73867	72877	76638		
elapsed	124.4	97.0	115	203	192
sd	0.20	0.03	7		
inner	120.9	93.5	112	199	189
sd	0.22	0.03	7		

B.4 Resultater for Water

	Antall noder: 4		
SCI, LMW	$n = 4$	$\bar{t} = 3.15$	$sd = 0.03$
UDP, LMW	$n = 4$	$\bar{t} = 5.09$	$sd = 0.02$
SCI, AURC	$n = 1$	$t = 72.97$	
	Antall noder: 3		
ScaMPI, LMW	$n = 2$	$\bar{t} = 5.03$	$sd = 0.03$
UDP, LMW	$n = 4$	$\bar{t} = 4.31$	$sd = 0.03$
SCI, LMW	$n = 4$	$\bar{t} = 4.55$	$sd = 0.02$
	Antall noder: 2		
SCI, LSW	$n = 4$	$\bar{t} = 5.04$	$sd = 0.01$
SCI, HOME	$n = 4$	$\bar{t} = 9.14$	$sd = 0.02$
SCI, AURC	$n = 1$	$\bar{t} = 97.2$	
SCI, AURC2	$n = 4$	$\bar{t} = 9.78$	$sd = 0.02$
SCI, LMW	$n = 4$	$\bar{t} = 4.92$	$sd = 0.01$
UDP, LMW	$n = 4$	$\bar{t} = 5.18$	$sd = 0.03$
UDP, LSW	$n = 4$	$\bar{t} = 5.69$	$sd = 0.01$
UDP, SEQ	$n = 4$	$\bar{t} = 6.12$	$sd = 0.03$
UDP, HOME	$n = 4$	$\bar{t} = 7.0$	$sd = 0.2$
ScaMPI, LMW	$n = 4$	$\bar{t} = 4.55$	$sd = 0.01$
	Antall noder: 1		
	$n = 4$	$\bar{t} = 7.62$	$sd = 0.01$

B.4.1 MPI/SCI-statistikk med Water

Antall forespørsler vil si det antall ganger `MPI_Probe` er kalt fra CVM til MPI/SCI. Den tiden som totalt har gått med til pollinger er tid hvor noden

steiler i påvente av at én eller flere av de andre nodene skal svare på en forespørsel, eller ankomme en barriére. En høy total tid her, tyder på at det er for mye venting på grunn av en asynkronitet. Tiltak kan være eksplisitt polling (jf FFT-eksempelet på side 79, eller en jevnere arbeidsfordeling mellom nodene.

	snitt	totalt	kommentar
Water 4 noder, AURC3			
pollinger (probe)	9.88 μs	1.59 s	152701 neg. poll av totalt 160627
kall til s_llm_send	46.33 μs	0.27	
kall til s_llm_read	18.78 μs	0.11	
kopering mellom bufre	1.76μs	0.02	
Water 4 noder, LMW			
pollinger (probe)	9.75 μs	0.95 s	91824 neg. av totalt 97260
kall til s_llm_send	45.38 μs	0.17 s	
kall til s_llm_read	18.80 μs	0.07 s	
kopering mellom bufre	1.69 μs	0.013 s	

Bibliografi

[1] Sarita V Adve og Mark D Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and distributed systems*, 4(6):613–624, juni 1993.

[2] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS*, side 483–485, april 1967.

[3] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu og W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, februar 1996.

[4] Thomas E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, januar 1990.

[5] Thomas E. Anderson, David E. Culler, David A. Patterson et al. A Case for Networks of Workstations: NOW. *IEEE Micro*, februar 1995. http://now.cs.berkeley.edu/Case/case.ps.

[6] D. Bailey, E. Barzscs, J. Barton, D. Browning, R. Crter, L. Dagum, R. Fatoohi, S. Fineberg, P. Fredrickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan og S. Weeratunga. The nas parallel benchmarchs. Rapport RNR-94-007, NASA Ames Research Center, mars 1994.

[7] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo og Maurice Yarrow. The nas parallel benchmarks 2.0. Rapport NAS-95-020, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, desember 1995.

[8] Kenneth E. Batcher. The massively parallel processor system overview. I *Parallel Processing – Addresses, essays, lectures*, side 142–149. MIT Press series in scientific computation, 1985.

[9] Gordon Bell. Ultracomputers – a teraflop before it's time. *CACM*, 35(8):26–47, august 1992.

[10] Gordon Bell. 1995 Observations on Supercomputing Alternatives: Did the MPP Bandwagon Lead to a Cul-de-Sac. *CACM*, 39(3):11–15, mars 1996.

[11] Matthias A. Blumrich, Richard D. Alpert, Yuqun Chen, Douglas W. Clark, Stefanos N. Damianakis, Cezary Dubnicki, Edward W. Felten, Liviu Iftode, Kai Li, Margaret Martonosi og Robert A. Shillner. Design choices in the shrimp system: An empirical study. I *Proceedings of 25th Annual ACM/IEEE International Symposium on Computer Architecture*, juni 1998.

[12] Haakon Bryhni og Knut Omang. A Comparison of Network Adapter based Technologies for Workstation Clustering. I *Proceedings of 11th International Conference on Systems Engineering*, juli 1996.

[13] J B Carter, J K Bennett og W Zwaenepoel. Implementation and performance of munin. I *Proceedings of the 13th ACM Symposium on Operating System Principles*, side 152–164, oktober 1991.

[14] Rational Software Corporation. Rational Quantify. http://www.rational.com/products/quantify/.

[15] Standard Performance Evaluation Corporation. Frequently asked questions. http://www.spec.org/spec/faq/.

[16] Dolphin Interconnect Solutions, Inc. http://www.dolphinics.com/.

[17] Jack J Dongarra, Hans W Meuer og Erich Strohmaier. Top500 supercomputer sites. http://www.top500.org/lists/1998/11/top500_9811.ps.gz, november 1998.

[18] Michel Dubois, Christoph Scheurich og Faeyé Briggs. Memory access buffering in multiprocessors. I *Proceedings of the 13th Annual International Symposium on Computer Architechture*, side 434–442, juni 1986.

[19] Andrew Erlichson. *Clustered Distributed Virtual Shared Memory for Large-Scale Multiprocessing*. Doktorgradsoppgave, Stanford University, http://www-flash.stanford.edu/~aje/thesis.ps, juni 1997.

[20] Andrew Erlichson, Neal Nuckolls, Greg Chesson og John Hennessy. Softflash: Analyzing the performance of clustered distributed virtual shared memory. I *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[21] Essential Communications, HIPPI - Technology Overview. http://www.esscom.com/techoverview.html.

[22] Marco Fillo og Richard B Gillett. Architecture and implementation of memory channel 2. *Digital Tech Journal*, 9(1):27–41, 1997.

[23] Free Software Foundation. Gnu general public license. http://www.gnu.org/copyleft/gpl.html, 1989–1991.

[24] Gregory C. Garry. Intel in the data center: A three-trick pony? *Unix Review's Performance Computing*, 17(2):13–19, februar 1999.

[25] Kourosh Gharachorloo. Memory consistency and event ordering in scalable shared-memory multiprocessors. I *Proceedings of 17th Annual International Symposium on Computer Architecture*, bind 18, side 15–26. ACM, juni 1990.

[26] Silicon Graphics. Cray t3e home page. http://www.sgi.com/t3e/.

[27] W. Gropp, E. Lusk, N. Doss og A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, september 1996. ftp://ftp.mcs.anl.gov/pub/mpi/mpich/papers/mpicharticle.ps.

[28] William D. Gropp og Ewing Lusk. *User's Guide for* mpich*, a Portable Implementation of MPI.* Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439, 1996. "ANL-96/6, ftp://ftp.mcs.anl.gov/pub/mpi/userguide.ps".

[29] John Gustafson og Rajat Todi. Conventional benchmarks as a sample of the performance spectrum. I *Proceedings of the 31st HICSS Conference, Kohala Coast, Hawaii*, januar 1998. http://www.scl.ameslab.gov/Publications/HICSS98/HICSS98.ps.

[30] John L. Gustafson og Quinn O. Snell. Hint: A new way to measure computer performance, januar 1995. ftp://ftp.scl.ameslab.gov/pub/HINT/doc/HINTpaper.ps.

[31] Mark D. Hill. Multiprocessors should support simple memory consistency models. *IEEEC*, august 1998.

[32] Weiwu Hu, Weisong Shi og Zhimin Tang. Reducing system overheads in home-based software dsms. I *13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing*, San Juan, Puerto Rico, april 1999.

[33] Kai Hwang. *Advanced computer architecture : parallelism, scalability, pro grammability*. McGraw-Hill, c1993.

[34] Liviu Iftode, Cezary Dubnicki, Edward W Felten og Kai Li. Improving release-consistent shared virtual memory using automatic update. I *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture*, februar 1996.

[35] Mike Johnson. *Superscalar microprocessor design*. Prentice Hall, c1991.

[36] P. Keleher, A. L. Cox og W. Zwaenepoel. Lazy release consistency for software distributed shared memory. I *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, side 13–21, mai 1992. http://www.cs.umd.edu/~keleher/papers/sigarch92.ps.gz.

[37] Pete Keleher. *Lazy Release Consistency for Distributed Shared Memory*. Doktorgradsoppgave, Rice University, januar 1995. http://www.cs.umd.edu/~keleher/papers/keleher_thesis.ps.gz.

[38] Pete Keleher. *CVM: The Coherent Virtual Machine v0.3*. University of Maryland, august 1998.

[39] Pete Keleher. Update protocols and iterative scientific applications. I *The 12th International Parallel Processing Symposium*, mars 1998. http://www.cs.umd.edu/~keleher/papers/homeBar.pdf.

[40] Pete Keleher, Alan L Cox, Sandhya Dwarkadas og Willy Zwaenepoel. An evaluation of software-based release consistent protocols. I *Proc. of the 20th Annual Int'l Symp. on Computer Architecture (ISCA'93)*, side 144–155, 1993.

[41] Peter Keleher. Distributed shared memory home pages. http://www.cs.umd.edu/users/keleher/dsm.html.

[42] Peter J Keleher. The relative importance of concurrent writers and weak consistency models. I *Proceedings of the IEEE COMPCON'96 Conference*, februar 1996.

[43] Uwe S.V. Kubosch. Meldingsutveksling med scalable coherent interface. Hovedfagsoppgave, University of Oslo, Department of Informatics, november 1997.

[44] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum og John Hennessy. The stanford flash multiprocessor. I *In Proceedings of the 21st International Symposium on Computer Architecture*, side 302–313, april 1994.

[45] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions On Computers*, 28(9):690–691, september 1979.

[46] James V Lawton, John J Brosnan, Morgan P Doyle, Seosamh D Ó Riordáin og Tomothy G Reddin. Building a High-performance Message-passing System for MEMORY CHANNEL Clusters. *Digital Tech Journal*, 8(2):96–116, 1996.

[47] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta og John Hennessy. The dash prototype: implementation and performance. I *Proceedings of the 17th Annual International Symposium on Computer Architecture*, bind 20 av *ACM SIGARCH Computer Architecture News*, side 92–103, mai 1992.

[48] K Li og P Hudak. Memory coherence in shared virtual memory systems. *ACMTCS*, 7(4):321–359, november 1989.

[49] R Martin et al. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. I *Proceedings of the 25th International Symposium on Computer Architecture*, side 85–97, mai 1997. http://http.cs.berkeley.edu/~culler/papers/isca97.ps.

[50] Message Passing Interface Forum. Mpi: A message-passing interface standard, juni 1995. http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html.

[51] Knut Omang. Performance results form SALMON, a cluster of Workstations connected by SCI. Rapport 208, Department of Informatics, University of Oslo, november 1995.

[52] Knut Omang. Slib: A library for sci. Utkast til en artikkel., 1996.

[53] Knut Omang. Synchronization Support in I/O adapter Based SCI Clusters. I *Proceedings of Workshop on Communication and Architectural Support for Network-based Parallel Computing, CANPC'97*, nr 1199 i Lecture Notes in Computer Science, side 158–172, februar 1997.

[54] Knut Omang. Performance of a Cluster of PCI Based UltraSparc Workstations Interconnected with SCI. I *Proceedings of Network-Based Parallel Computing, Communication, Architecture, and Applications, CANPC'98*, nr 1362 i Lecture Notes in Computer Science, side 232–246, januar 1998.

[55] Knut Omang. *SCI Clustering through the I/O bus: A performance and functionality analysis.* Doktorgradsoppgave, Department of Informatics, University of Oslo, april 1998.

[56] David A. Patterson, David Goldberg og John L. Hennessy. *Computer architecture : a quantitative approach.* Morgan Kaufmann, 1996.

[57] J Postel. User datagram protocol (rfc768). RFC768, august 1980.

[58] J Renwick. IP and ARP on HIPPI (rfc1374). rfc1374, januar 1997.

[59] Stein Jørgen Ryan. The Design and Implementation of a Portable Driver for Shared Memory Cluster Adapters. Rapport 255, Department of Informatics, University of Oslo, desember 1997.

[60] Stein Jørgen Ryan. *Efficient Middleware for IO Bus Bridging Cluster Adapters.* Doktorgradsoppgave, Department of Informatics, University of Oslo, juni 1998.

[61] Stein Jørgen Ryan og Haakon Bryhni. SCI for Local Area Networks. Rapport 256, Department of Informatics, University of Oslo, januar 1998.

[62] Stein Jørgen Ryan, Stein Gjessing og Marius Liaaen. Cluster Communication using a PCI to SCI interface. I *Proceedings of IASTED Eighth International Conference on Parallel and Distributed Computing and Systems, Chicago*, oktober 1996. http://www.ifi.uio.no/~sci/Publications/pdcs96.ps.

[63] Scali AS. *ScaMPI User's Guide*, v1.3.0 utgave, 1997.

[64] Sci-aktivitet ved ifi. http://www.ifi.uio.no/~sci/.

[65] Tom Shanley. *Pentium Pro and Pentium II System Architecture.* PC System Architecture Series. Addison Wesley, second edition utgave, 1997.

[66] Weisong Shi og Zhimin Tang. Using confidence interval to summarize the evaluating results of dsm systems. *Journal of Computer Science and Technology*, 1999. Accepted. Published by Academia Sinica Chinese Computer Federation. ftp://ftp.ict.ac.cn/incoming/chpc/dsm/paper/jcst99a.ps.

[67] Jaswinder Pal Singh, Wolf-Dietrich Weber og Anoop Gupta. Splash: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5–44, 1992.

[68] Standard software infrastructures for sci-based parallel systems. http://www.parallab.uib.no/projects/sisci/. EU Esprit project nr 23174.

[69] Aad J van der Steen og Jack J Dongarra. Overview of recent supercomputers. http://www.fys.ruu.nl/~steen/overview/contents.html.

[70] Robert Stets, Sandhya Dwarkadas, Nikolaos Hardavellas, Galen Hunt, Leonidas Kontothanassis, Srinivasan Parthasarathy og Michael Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered

Remote-Write Network. I *Proceedings of the Sixteenth ACM Symposium on Operating systems Principles*, Saint-Malo, France, oktober 1997.

[71] Sun Microsystems Inc. *Writing Device Drivers, Solaris 7.* Tilgjengelig fra http://docs.sun.com/.

[72] Svmlib. http://www.lfbs.rwth-aachen.de/~karsten/projects/SVMlib/index.html.

[73] Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall, 4. utgave, 1996.

[74] Andrew S. Tanenbaum. *Modern operating systems.* Prentice Hall, c1992.

[75] Kritchalach Thitikamol og Pete Keleher. Multi-threading and remote latency in software dsms. I *The 17th International Conference on Distributed Computing Systems*, mai 1997.

[76] Don Tolmie og Don Flanagan. HIPPI: It's not just for supercomputers anymore. *Data Communications Magazine, a publication of the McGraw-Hill*, 1995.

[77] The treadmarks distributed shared memory (dsm) system. http://www.cs.rice.edu/~willy/TreadMarks/overview.html.

[78] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh og Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. I *In Proceedings of the 22nd International Symposium on Computer Architecture*, side 24–36, juni 1995. ftp://www-flash.stanford.edu/pub/splash2/splash2_isca95.ps.Z.

[79] Yuanyuan Zhou, Liviu Iftode og Kai Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. I *Proceedings of the 2nd Symposium on Operating System Design and Implementation*, oktober 1996.

[80] Yuanyuan Zhou, Liviu Iftode, Jaswinder Pal Singh, Kai Li, Brian R. Toonen, Ioannis Schoinas, Mark D. Hill og David A. Wood. Relaxed consistency and coherence granularity in dsm systems: A performance evaluation. I *SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, juni 1997.