Universitetet i Oslo
Institutt for informatikk

Global Arrays over SCI

Enveis kommunikasjon i clustere

Kai-Robert Bjørnstad

Hovedfagsoppgave

Februar 2003

Forord

Denne rapporten er et resultat av hovedfagsarbeid utført av underskrevne som student ved Institutt for Informatikk (IfI) ved Universitetet i Oslo. Det praktiske arbeidet ble gjennomført i perioden 2002–2003 ved Scali AS. Arbeidet leder frem til en Cand.Scient-grad i informatikk.

Som et utgangspunkt for oppgaven hadde jeg lite annet enn noe programmeringserfaring. Teknologier som clustere, SCI og Global Arrays var fullstendig ukjente og dermed meget spennende. Gjennom arbeidet med denne hovedfagsoppgaven samt noe deltidsarbeide for Scali AS er situasjonen nå en helt annen. Prosessen har vært meget lærerik og jeg sitter nå med et godt grunnlag for både implementasjon, benchmarking og bruk av parallelle systemer.

Oppgaven har også vekket interesse hos utviklerene av Global Arrays, Jarek Nieplocha og medarbeidere ved Pacific Northwest National Laboratory (PNNL) i USA. Jeg ser således for meg en mulig senere publisering av både kode og resultater for videre utvikling og optimalisering.

Scali AS har som firma gitt meg hjelp og støtte over all forventning. Denne oppgaven dekker over et stort område av forskjellige teknologier og metoder. Hjelp og tips fra Scali AS sin utviklingsstab har således vært uvurderlig.

Jeg vil begynne med å takke veilederene mine Øystein Gran Larsen og Håkon O. Bugge fra Scali AS sammen med Dag Langmyhr fra IfI, for å ha gitt meg både akademiske og faglige råd. Videre vil jeg takke Lars Paul Huse for hjelp med både akademiske spørmål, dokumentasjon og forståelse av både MPI, SCI og ScaFun. Steffen Persvold har gitt meg støtte og forklaringer rundt ScaSCI-driveren og ScaIP, Terje Eggestad programmeringsmessige “tips og triks” og Ole W. Saastad har hjulpet meg å se kompleksiteten i kunsten å benchmarke.

Jeg vil benytte anledningen til å anbefale eventuelle hovedfag hos Scali AS for fremtidige studenter. Det hyggelige arbeidsmiljøet, den høye kompetansen og deres vilje til å sette av tid til studenter har vært over all forventning!

Blindern, 01.02.2003

Kai-Robert Bjørnstad

Innhold

Kapittel 1

Innledning

Verdens krav til datakraft fortsetter å øke, ikke bare når det kommer til avansert grafikk, store databaser og bredbåndsnettverk, men også behovet for rå regnekraft. Spesielt forsknings- og ingeniørmiljø kommer stadig opp med nye oppgaver hvor store databeregninger er absolutt nødvendige for resultatet. Man kan her trekke fram bruksområder som værsimuleringer på bakgrunn av satellittdata, optimaliseringsberegninger for både konstruksjon og kollisjonssikkerhet i bil- og flyindustrien, eller simuleringer og beregninger både innenfor seismikk, kjemi og atomfysikk. En fellesnevner for slike beregninger er store og kompliserte matematiske beregninger, også kalt tallknusing. Slike problem blir ofte referert til som "The Grand Challenges" i litteraturen.

Et uttrykk som benyttes for å beskrive dette fagfeltet er "High Performance Computing" (HPC). Selve maskinene refereres ofte til som "Superdatamaskiner" (Supercomputers). Det finnes ingen statisk definisjon på hva HPC er, men det impliserer store, raske og dyre programvare og hardware. Nøkkelordet er *ytelse*. Man ser også at bruk av samtidighet eller parallellisme har en økende tendens, da spesielt innenfor HPC.

Ytelsen på prosessorer har siden 1960-tallet hatt eksponentiell vekst. For å beskrive denne veksten refereres det ofte til Moores lov[97, 98] fra 1965. Moore fremsatte en teori om at kapasiteten på prosessorer og internminne ville dobles hver 18. måned[1]. Som en følge av dette kan man si at at ethvert endelig problem kan løses innenfor en gitt prosesseringstid. Problemet er at man i enkelte tilfeller kan bli nødt til å vente lenge på en kraftig nok maskin. Dette er ofte lite praktisk da man gjerne ønsker å løse problemet nå og ikke om flere år. Et annet aspekt i så måte er at problemene også har en tendens til å vokse. Enten ved problemet øker i sin økt datamengde, løsningen krever større nøyaktighet eller brukere fremsetter krav om mindre regnetid.

HPC bransjen ble tidligere dominert av såkalte MPP-maskiner (se A.1) fra en rekke produsenter (for eksempel IBM, Cray og SGI). Det ble dermed flere biblioteker og API (Application Programming Interface) til bruk for kommunikasjon mellom prosessorene i disse maskinene. Dette førte til problemer med hensyn på portabilitet. Forskjellige produsenter utviklet egne grensesnitt for kommunikasjon over sine proprietære nettverk og vanskeliggjorde dermed flytting/*porting* av applikasjoner til andre nyere og/eller billigere nettverksteknologier og arkitekturer.

Som et resultat av ble et felles grensesnitt kalt MPI (Message Passing Interface [100]) for prosess-

[1] Moores lov refererer i utgangspunktet kun direkte til halvlederteknologi. Moore fremsatte at man kunne halvere størrelsen på en transistor hver 18. måned. I forretningssammenheng ble det således utledet en tommelfingerregel om at ytelsen og kapasiteten for komponenter, bestående av transistorer, ville fordobles som en konsekvens.

kommunikasjon utviklet. Grensesnittet ble utviklet med hensyn på å være uavhengig av underliggende maskinarkitektur og nettverksteknologi. MPI-standarden er i dag godt utbredt og anerkjent og blir i dag benyttet av de fleste applikasjoner som grensesnitt for kommunikasjon mellom parallelle prosesser.

Konseptet med lavkostnads-*clustere* (se avsnitt 2.3) ble introdusert sent i 80-årene og formalisert av Pfister i 1995[118]. I de siste årene har fremveksten av slike systemer økt. Clustere er i motsetning til proprietære maskiner fra for eksempel IBM og CRAY, i utgangspunktet basert på såkalt COTS-hardware (Commercial Off-The-Shelf) som i praksis følger Moores lov. Som oftest benyttes det relativt rimelige, men kraftige, arbeidsstasjoner og servere (noder) med fra en til fire prosessorer. Disse kobles sammen med et høyhastighets LAN (Local Area Network) for kommunikasjon mellom nodene.

Utbredelsen av clustere satte også i gang utvikling av mer spesialiserte nettverksteknologier for høyytelses kommunikasjon. Slike teknologier har primærfokus på egenskaper som høy båndbredde og lav forsinkelse. Eksempelvis kan jeg nevne Myrinet(avsnitt 3.3.2) og SCI(avsnitt 3.3.3).

Enkelte applikasjonstyper og algoritmer vanskelig å implementere over MPI-grensesnittet[2]. MPI-standarden fordrer en tosidig kommunikasjonsmodell hvor både sender og mottaker eksplisitt må delta i kommunikasjonen. Dette fører ofte til at en mottakerprosess må avbryte sin prosessering/regning for å sende og motta data fra andre prosesser.

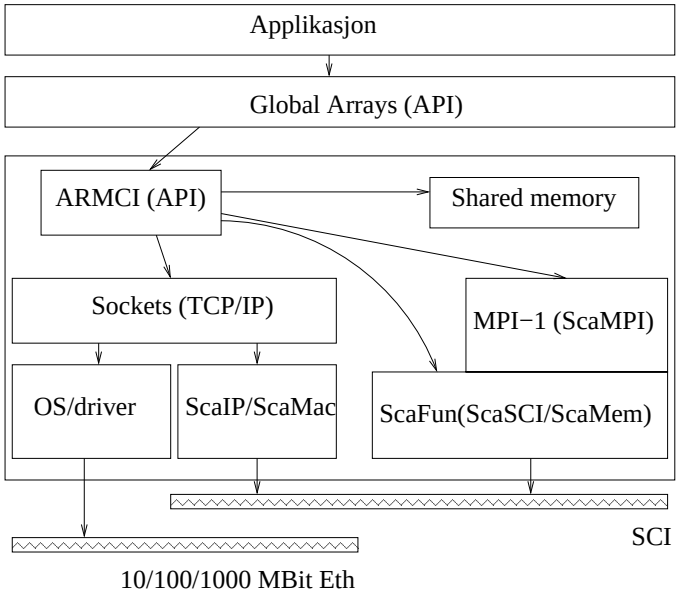
Ensidig kommunikasjon baserer seg derimot kun på deltakelse fra senderprosessen ved utveksling av data (typisk kommunikasjonsmetode for SMP-maskiner). Hovedforskjellen på ensidig og tosidig kommunikasjon ligger i hvilken programmeringsmodell som tilbys, med andre ord hvordan applikasjonsprogrammereren må forholde seg til kommunikasjon mellom prosesser. Algoritmer og grensesnitt (for eksempel SHMEM[22] som ble utviklet for MPP-maskinen Cray T3D/T3E), er utviklet for maskiner hvor prosessorene ikke har et delt adresserom.

Clustere har i utgangspunktet ikke et delt adresserom, noe som skaper problemer ved enveis kommunikasjon. Som et resultat har man på mange måter forsøkt å skape en illusjon av shared memory ved hjelp av både hardware og/eller programvare, et såkalt DSM (Distributed Shared Memory). I denne sammenheng har et bibliotek kalt Global Arrays (GA, nærmere beskrevet i 3.4.5) blitt utviklet. GA gir applikasjonprogrammerere blant annet muligheten til å opprette distribuerte arrayer på clustere, som videre oppfattes som en enkelt enhet. Operasjoner på disse arrayene gjøres ved enkle ensidige operasjoner, hvor all kommunikasjon gjennomføres av underliggende bibliotek. Sentrale operasjoner i så måte er *put*, *get* og *accumulate*.

GA er implementert over kommunikasjonsbiblioteket ARMCI (se avsnitt 4.1). ARMCI er et portabelt kommunikasjonsbibliotek med fokus på implementasjon av effektive ensidige kommunikasjonsoperasjoner i clustere og shared memory-maskiner. Grensesnittet er ikke standardisert, men benyttes likefullt av flere applikasjoner, for eksempel NWCHEM[110] og GAMESS-UK[50] (gjennom GA). Problemet i så måte er at det ikke finnes noe standardiserte API på linje med MPI for ensidig kommunikasjon. ARMCI benytter derfor sitt eget, med fokus på ikke-sekvensielle dataoverføringer.

En viktig del av datakommunikasjon er å gjøre denne så effektiv som mulig. Spesielt innenfor HPC fokuseres på å benytte minst mulig tid på kommunikasjon (*overhead*) og mest mulig på pro-

[2] I denne rapporten refererer MPI til MPI-1-standarden. MPI-2-standarden refereres til som MPI-2 (se forøvrig avsnitt 3.4.3 på side 38)



Figur 1.1: **Lagdeling for utvidelser av ARMCI.** Figuren viser laginndelingen og overordnet design for de nye utvidelsene av ARMCI og den eksisterende Sockets-versjonen.

sessering/regning. Med dette som utganspunkt ønsker jeg derfor å undersøke mulighetene for å la ARMCI kunne kommunisere over høyhastighetsnettverket SCI.

Gjennom å implementere eksempelimplementasjoner av ARMCI over SCI ønsker jeg å belyse problemstillinger som:

- Kan ARMCI benyttes over SCI gjennom eksisterende programvarebibliotek, og fortsatt oppnå en høy ytelse?
- Kan ARMCI gjøres mer portabel gjennom å utelukkende benytte MPI-grensesnittet? Kan man tilby god ytelse for ensidig kommunikasjon ved bruk av tosidig kommunikasjonsbibliotek?
- Er ARMCI egnet for en implementasjon over SCI? Kan SCI-teknologien utnyttes?
- Hva avdekker en ARMCI-implementasjon over MPI-grensesnittet? Hva avdekkes ved bruk av lavnivå SCI?
- Kan applikasjoner som benytter ARMCI over SCI (i en eller annen form) nyte fordel av dette med hensyn på kjøretid?

Oppgaven er delt inn i to hovedfaser:

1. fase: Implementasjon og prototyping

Som et utgangspunkt skisserte jeg tre hovedmetoder for implementasjon (se figur 1.1); ARMCI over SCI via TCP/IP, MPI og SCI-driver. Jeg forventet at implementasjonene ville ha en forskjellig grad av kompleksitet og ytelse:

TCP/IP Målsettingen var å få ARMCI til å benytte SCI gjennom den eksisterende implementasjonen av ARMCI over Sockets og TCP/IP. Jeg ønsket her å prøve en eksisterende implementasjon Sockets eller TCP/IP over SCI. Jeg forventet her en relativt lav implementasjonstid, ettersom både ARMCI og TCP/IP-implementasjonen praktisk talt skulle være COTS. Jeg regnet med at ytelsen ville bære preg av flere biblioteksslag og protokoller, men ha en høy grad av portabilitet.

MPI Målsettingen her var å implementere en utvidelse av ARMCI, som lot biblioteket kjøre som et rent MPI-program. Ved å la ARMCI kun forholde seg til det standardiserte MPI-grensesnittet for datakommunikasjon, ville jeg teoretisk sett kunne kjøre ARMCI over ethvert mellomnodenett hvor det fantes en MPI-implementasjon. Ytelsen ble her forventet å avhenge noe av den underliggende MPI-implementasjonen. Likevel forventet jeg høy ytelse ettersom MPI-bibliotek er optimalisert for kommunikasjon på sitt underliggende nettverk (i dette tilfellet SCI). MPI-grensesnittet regnes som et relativt lavnivå kommunikasjons grensesnitt i parallelle system og betydelig implementasjonstid ble forventet.

SCI-driver Den potensielt raskeste implementasjonen av ARMCI over SCI, forventet jeg direkte over SCI-driveren. Her vil jeg ha mulighet for å eliminere mellomliggende protokoller og bibliotek. En implementasjon av ARMCI direkte over SCI-driveren ble forventet å være den mest avanserte og tidskrevende fremgangsmåten. Ettersom det finnes flere API for slike drivere og ingen gjennomført standard, forventet jeg lav portabilitet ettersom jeg ville forhold meg utelukkende til en konkret SCI-driver.

2. fase: Verifisering og ytelsestesting

Jeg så for meg verifikasjon og ytelsestester i to hovedklasser:

Benchmarker Både GA- og ARMCI-distribusjonene inneholder flere mindre verifikasjons- og ytelsestester. Som en målsetting ønsket jeg å kunne kjøre ytelsestestene og passere verifikasjonstestene, på alle mine nye implementasjoner av ARMCI over SCI. Jeg håpet å kunne benytte disse testene som et sammenlikningsgrunnlag for de forskjellige implementasjonene.

Applikasjoner I tillegg til de mindre testene ønsket jeg også å kjøre verifikasjons- og ytelsestester på reelle produksjonsapplikasjoner. Mindre tester og verifikasjoner er ofte designet for å måle og/eller verifisere en spesifikk del av biblioteket man tester. Ved å benytte reelle applikasjoner regnet jeg med å få et mer realistisk bilde av betydningen av GA-/ARMCI-implementasjonenes ytelse og deres eventuelle bieffekter.

1.1 Forutsetninger og grunnlag

Sammenliknet med nettverk som Gigabit Ethernet (GbE) og Myrinet, er SCI lite utbredt. Det er i dag kun Dolphin ICS[28] som leverer COTS PCI-SCI-adaptere for cluster-teknologi. På programvaresiden fins det kun to driverimplementasjoner for disse kortene: Dolphins SISCI-driver (Software Infrastructure for SCI)[40, 130] og Scali AS[127] sin ScaSCI-driver (se avsnitt 6.1).

Hovedforskjellen på disse driverene er deres utgangspunkt for design og implementasjon. SISCI-driveren er utviklet som en mer generell driver, mens ScaSCI-driveren er utviklet og spesielt optimalisert mot cluster-systemer og HPC gjennom ScaFun (. ScaSCI-driveren gir dermed en bedre ytelse, men samtidig mindre feiltoleranse og feilhåndtering enn SISCI-driveren. SISCI-driveren distribueres Open Source, mens ScaSCI-driveren er et kommersielt produkt.

MPI-implementasjoner over SCI er også en sjeldenhet. Scali AS har utviklet en høyytelses implementasjon kalt ScaMPI([67], se avsnitt 5.1) basert på sin ScaSCI-driver. Et alternativ til denne er SCI-MPICH[145, 129] utviklet ved RWTH Aachen[123] basert på SMI-grensesnittet. SMI (Shared Memory Interface)[144] er et abstraherende grensesnitt med støtte for flere drivere, deriblant SISCI-driveren (ikke ScaSCI). ScaSCI-driveren har støtte for SISCI-grensesnittet, men benytter i hovedsak API-et basert beskrevet av Ryan[126, 124]. SCI-MPICH er i likhet med SISCI-driveren Open Source. ScaMPI er derimot en kommersiell implementasjon av MPI-standarden.

Jeg har i dette prosjektet valgt å basere meg på Scali AS sin programvarepakke (med ScaFun og ScaMPI) SSP (Scali Software Package)[13]. Det er seks hovedgrunner for dette valget:

1. *Tilgjengelighet av hardware og programvare.* Som en del av sin støtte til dette prosjektet, tilbød Scali AS meg tilgang til både SCI-baserte clustere og ubegrenset bruk av deres SSP.

2. *Tilgjengelighet på og støtte fra Scali AS sin utviklingsgruppe.* Mulighet for konferering og diskusjon med utviklerne av både ScaMPI og ScaSCI-driveren åpnet for mer effektiv bruk av implementasjonene.

3. *Programvarestabilitet og funksjonalitet.* Scali AS sin SSP er kommersiell og blir benyttet av flere kunder. Kravene til stabilitet, implementasjon og funksjonalitet (blant annet full MPI-kompatibilitet og *thread-safeness*) er derfor høyere og mer fastsatt enn hva man kan forvente av Open Source kode.

4. *Debugging og overvåkingsmuligheter.* Implementasjonene i SSP pakken er tilrettelagt for debugging og kjøring av programmer i forskjellige miljø. Pakken inneholder også programmer for overvåking og monitorering av applikasjoner og clustere som helhet.

5. *Ytelse.* Scali AS sin programvare har vist seg å være ytelsesmessig best sammenliknet med tilsvarende programpakker.

6. *Kompatibilitet.* Scali AS sin forskjellige programvare (både SCI og annen) er garantert å fungere sammen. Bruk av programvare fra flere kilder garanterer ikke kompatibilitet og deling av ressurser.

Prototypeimplementasjonene beskrevet i denne oppgaven er dermed utviklet og testet på Scali programvare. Jeg har dessverre ikke hatt mulighet til å utføre tester og sammenlikninger med annen programvare (andre SCI-drivere og MPI-implementasjoner).

1.2 Kapittelinndeling og terminologi

En nærmere beskrivelse av ARMCI og ensidig kommunikasjon over SCI beskrives i følgende kapitler:

Kapittel 2 setter oppgaven i en større sammenheng ved å forklare begreper som programmeringsmodell og skalerbarhet. Kapittelet beskriver også cluster-arkitekturen i sammenheng med andre maskinarkitekturer.

Kapittel 3 introduserer den formelle modellen LogP og utvidelser for analyse av effektiviteten av datakommunikasjon. Et utvalg av de mest relevante mellomnodenettverk for clustere beskrives sammen med et utvalg velkjente kommunikasjonsbibliotek, deriblant GA.

Kapittel 4 gir en detaljert beskrivelse av ARMCI-biblioteket og grensesnittet. Prinsippet bak "perfekt" ensidig kommunikasjon over SCI diskuteres i avsnitt 4.2. Kapittelet beskriver også bruk av den eksisterende ARMCI-implementasjonen med en TCP/IP-implementasjon over SCI.

Kapittel 5 beskriver en eksempelimplementasjon av ARMCI over MPI-grensesnittet. Kapittelet beskriver implementasjon, mulige optimaliseringer og bruk av ScaMPI som underliggende MPI-implementasjon for ARMCI.

Kapittel 6 beskriver en eksempelimplementasjon av ARMCI over SCI basert på ScaSCI-driveren (og ScaMem) fra Scali AS. I denne sammenheng beskrives Dolphin sine PCI-SCI-adaptere utviklet for tilkobling på I/O-bussen. Kapittelet skisserer både implementasjon og mulige optimaliseringer.

Kapittel 7 beskriver testmiljø og resultater for gjennomførte verifikasjon og ytelsestester av de forskjellige eksempelimplementasjonene beskrevet i kapittel 5 og 6.

Kapittel 8 inneholder relatert arbeide, konklusjon, erfaringer og videre arbeide.

Terminologi

Fagområdet informatikk bærer sterkt preg av engelske ord og uttrykk. I denne oppgaven er derfor de engelskspråklige faguttrykkene i stor utstrekning benyttet av mangel på egnede norske erstatningsord og for å forhindre misforståelser.

Engelske ord er fornorsket og bøyes deretter, for eksempel *en request, den requesten, flere requester, alle requestene*. Forkortelser bøyes derimot med "-" og norsk endelse, for eksempel *en CPU, den CPU-en, flere CPU-er, alle CPU-ene*.

I tillegg A er enkelte av de mest brukte uttrykkene kort forklart. Begreper rundt maskinarkitekturer (for eksempel SMP, PVP, DSM og MPP) er beskrevet i tillegg A.1.

Andre ord og uttrykk som ikke regnes som allmenkunnskap innenfor fagfeltet gjennomgås fortløpende i rapporten. Nye engelske ord og uttrykk introduseres ved at ordet utheves i *kursiv*. Funksjoner kjennetegnes ved at de er uthevet i kursiv og har en etterfølgende parentes; *funksjon()*.

Figurer

Jeg har forsøkt å i benytte norsk språk i de fleste figurer. Bruken av engelske faguttrykk har likevel gjort dette vanskelig i noen tilfeller. Enkelte figurer har derfor engelske markeringer.

Flere av figurene som viser grafer er skissert med en eller flere logaritmiske akser. Hvis så er tilfellet, begynner figurteksten med: *(Log <akse>)* hvor *<akse>* er X eller Y. Noen av figurene navngir sine kurver med *zc*, *oc* eller *dc*, for eksempel *Puma_ScaMPI_dc_SCI*. Dette er forkortelser for henholdsvis *zero-copy*, *one-copy* og *double-copy* (se avsnitt 3.4.1).

Kapittel 2

Bakgrunn og motivasjon

Dette kapittelet peker jeg på sammenhengen mellom programmeringsmodeller, maskinarkitekturer og maskinklassifiseringer. Jeg forsøker her å beskrive noen av de viktigste sammenhengene mellom disse begrepene. I tillegg pressiseres begrepet skalerbarhet nærmere. I HPC bransjen blir dette begrepet mye benyttet, både for å beskrive maskinarkitekturer, nettverk, applikasjoner og programmeringsmodeller.

Det fokuseres her på programmeringsmodeller og arkitekturer relatert til cluster-teknologien.

2.1 Parallelle programmeringsmodeller

Hwang[69, s.547] beskriver programmeringsmodeller som en samling programabstraksjoner som gir programmereren et forenklet og transparent bilde av hardware- og programvaresystemet. Parallelle modeller ble i begynnelsen spesialdesignet for multiprosessorer, løst koblede system eller andre typer maskiner. Hwang[70, s.607] forklarer hvordan flere parallelle programmeringsmodeller har konvergert til tre essensielle; shared memory[1], meldingsutveksling og dataparallell.

Det er viktig å merke seg at de tre overnevnte modellene benytter såkalt *eksplisitt parallellisme*. Dette betyr at parallelliteten blir eksplisitt spesifisert av applikasjonsprogrammereren. Dette gjøres for eksempel ved spesielle språkkonstruksjoner, kompilatordirektiver eller bibliotekskall. Hvis programmereren ikke eksplisitt spesifiserer parallelismen, men lar kompilatoren og *run-time* systemet automatisk utnytte parallellitet, har man *implisitt parallellisme*.

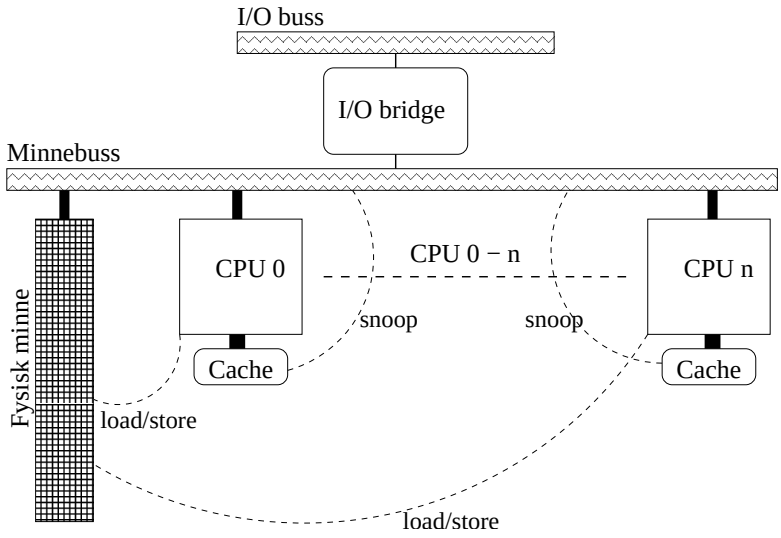
Denne rapporten fokuserer på eksplisitt parallellitet i shared memory- og meldingsutvekslingsmodellene.

2.1.1 Shared memory

Flerprosessorprogrammering i shared memory-modellen er basert på delte variabler i et felles minne. Kommunikasjon (IPC - Inter Process Communication) utføres ved *load* og *store* mot disse variablene. En delt variabel forutsetter et shared memory (se figur 2.1) og mekanismer for gjensidig ekskludering (*mutual exclusion)* mellom de prosesser som aksesserer det samme settet med variable. Hovedproblemene med bruk av denne modellen er beskyttet aksess av *critical sections*, *cache koherens*, atomisitet og koherens ved minneoperasjoner, rask synkronisering, delte datastrukturer og teknikker for hurtig flytting av data.

Shared memory-modellen likner dataparallellmodellen i at den har et enkelt (globalt) adresseområde og meldingsutvekslingsmodellen i at den er flertrådet (*multithreaded*) og asynkron. Likevel

[1] Shared memory blir også referert til som Shared variabel.



Figur 2.1: **Shared memory-modellen.** To eller flere CPU-er deler den samme minnemodulen. Cache-oppdateringer gjøres automatisk av hardware gjennom for eksempel buss-*snooping*. Modellen tillater dermed load og store til minnet av en prosessor, uten eksplisitt deltagelse fra andre prosessorer/prosesser. Synkronisering og konsistens i datastrukturene må likefullt utføres av programmereren.

trenger ikke data i det delte adresseområdet å bli eksplisitt allokert som delt, eller arbeidslasten å bli eksplisitt definert. Synkronisering er likefullt eksplisitt.

2.1.2 Meldingsutveksling

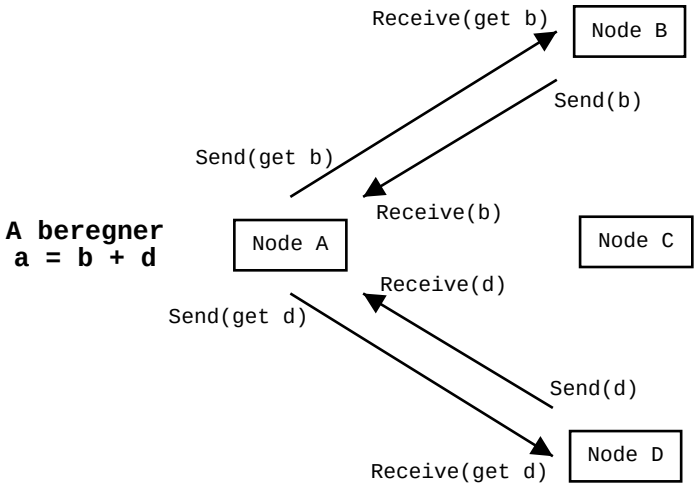
To prosesser kan kommunisere ved hjelp av meldinger implementert over shared memory eller et mellomnodenettverk (et såkalt *interconnect*). Meldingene kan inneholde for eksempel instruksjoner, data, synkroniserings- eller interrupt-signaler. Kommunikasjonsforsinkelsen forårsaket av et interconnect er betydelig større enn ved bruk av shared memory.

Prosesser som benytter meldingsutveksling har separate adresseområder. Prosesser i meldingsutvekslingsmodellen må gjøre eksplisitt allokering og definering av data og arbeidslast. Figur 2.2 viser et eksempel på bruk av meldingsutveksling mellom tre prosesser i et parallelt program. Prosess A trenger data fra to andre prosesser og sender en melding til disse. Prosessene B og D må således forvente en melding fra A i sin applikasjonskode, for så så sende tilbake dataene A spør etter. Prosess C derimot kan fortsette med sine beregninger uten å måtte vente eller på annen måte bli berørt av meldingene mellom de andre prosessene. Den interne protokollen i meldingene blir bestemt av applikasjonen.

Meldingsutvekslingsmodellen kan igjen deles inn i to undermodeller; synkron og asynkron meldingsutveksling.

Synkron meldingsutveksling

Siden prosesser basert på meldingsutveksling ikke har noe delt adresseområde, er det ikke noe behov for gjensidig utelukking (mutual exclusion). Ved synkron meldingsutveksling må både sender- og mottakerprosessen være synkronisert i tid og “rom”. “Rom” er her ekvivalent med hvor prosessen er i sin kjøring. Hvis bare den ene prosessen er klar for kommunikasjon, vil denne blokkeres (må vente)



Figur 2.2: **Meldingsutvekslingsmodellen.** Node A skal legge sammen dataelementet **b** (som ligger i minnet til node B) og **d** (som ligger hos node D) og legge resultatet i variabel **a** hos seg selv. Anta at node B og D på utvekslingstidspunktet ikke vet hvilke dataelement A trenger. A må dermed sende forespørsler (get) til de andre nodene/prosessene og motta de ikke-lokale dataene (receive).

til kommunikasjonspartneren er klar.

Asynkron meldingsutveksling

Asynkron meldingsutveksling krever ikke en synkronisering mellom de kommuniserende prosessene i tid og "rom". Buffere blir ofte benyttet i kanalene og resulterer i en ikke-blokkerende modell. For å unngå overflyt av meldinger må disse bufferene være store nok til å kunne håndtere trafikken. Denne asynkrone modellen kan føre til forsinkelser ved kommunikasjon hvor sender krever svar tilbake fra mottaker. Siden det ikke er definert når mottaker leser meldinger kan man heller ikke forutsi når svaret kommer tilbake. På den annen side kan sender sende sin melding uavhengig av om mottaker er klar eller ikke. Asynkron meldingsutveksling letter programmererens jobb, men kan innføre en noe større overhead enn ved synkron meldingsutveksling.

2.1.3 Abstraksjoner over modeller

Modellene skissert over i avsnittene 2.1.1 og 2.1.2 beskriver eksplisitte kommunikasjonsmodeller for både applikasjoner og biblioteker. Det er viktig å trekke en skillelinje mellom hvilken abstraksjonsmodell selve applikasjonskoden (applikasjonsprogrammereren) forholder seg til og hvilke modeller maskinarkitekturen og eventuelle underliggende bibliotek benytter. For eksempel kan en applikasjon forholde seg til en meldingsutvekslingsmodell, mens selve biblioteket for denne modellen er implementert over shared memory. Abstraksjonsnivået for selve applikasjonskoden kan dermed være på et høyere nivå enn programmeringsmodellene beskrevet over.

Kommunikasjonsabstraksjoner applikasjonsprogrammereren forholder seg til kan således deles inn i to hovedtyper:

1. *Tosidig kommunikasjon* baserer seg på at applikasjonskoden er bevisst all kommunikasjon. Applikasjonskoden må eksplisitt kontrollere og administrere all inngående og utgående kommuni-

kasjon den selv inngår i. Ved bruk av for eksempel MPI vil man normalt følge et slikt prinsipp. Alle *requester* til prosessen må eksplisitt besvares.

2. *Ensidig kommunikasjon* krever derimot ingen aktiv deltagelse fra applikasjonskoden ved innkommende requester. Kommunikasjonen er ensidig ved at applikasjonskoden i senderprosessen gjør en request eller sender en melding til en annen prosess. Mottakende prosess har et system for mottak, tolking og utføring av denne requesten/meldingen, men dette skjer som en sideeffekt eller av hardware. Merk skillet mellom dette abstraksjonsnivået og asynkron meldingsutveksling. Ved asynkron meldingsutveksling kan applikasjonskoden behandle spørringer når det måtte passe, ved ensidig kommunikasjon trenger ikke applikasjonskoden å ta hensyn til disse i det hele tatt. De blir bare utført. Prinsippene kan minne mye om kommunikasjon over shared memory, men er til forskjell fra den modellen ikke begrenset av et felles adresseområde.

Applikasjoner kan forholde seg til et av prinsippene, eller en kombinasjon av disse. MPI (avsnitt 3.4.3) har blitt nevnt som et eksempel på tosidig kommunikasjon, SHMEM (avsnitt 3.4.4) er et eksempel på bruk av ensidig kommunikasjon. GA(avsnitt 3.4.5) tilbyr en modell som benytter en kombinasjon av både ensidig og tosidig kommunikasjon.

2.2 Skalerbarhet

Hwang og Xu[70, s. 6] definerer skalerbarhet (eller scalability) av et datasystem som følger:

> A computer system, including all it's hardware and software resources, is called *scalable* if it can *scale up* (i.e., improve it's resources) to accommodate ever-increasing performance and functionality demand and/or *scale down* (i.e., decrease it's resources) to reduce cost.

Selv om man vanligvis fokuserer på aspekter rundt oppskalering av et system, presiserer Hwang og Xu videre at skalerbarhet ikke nødvendigvis er ekvivalent med å være stor. Mer presist skisserer de tre hovedkriterier:

- *Funksjonalitet og ytelse:* Et oppskalert system bør gi mer funksjonalitet eller bedre ytelse. Den totale regnekraften av ett system bør øke proporsjonalt med dets økning i ressurser. Ideelt vil brukerene se en ytelses økning nær en faktor n når systemets ressurser øker med n.
- *Skalering i kostnad:* Kostnaden for oppskalering av et system bør være fornuftig. En tommelfingerregel er at oppskalering med en faktor på n bør ikke ha en høyere kostnad enn en faktor på n eller $n \log(n)$.
- *Kompatibilitet:* De samme komponentene, inkludert hardware, system- og applikasjonsprogramvare bør fortsatt være brukbare med små forandringer. Det er ikke å forvente at brukere vil betale for et helt nytt operativsystem og på nytt utvikle sine applikasjonskoder.

2.2.1 Amdahls lov: fast problemstørrelse

Litteraturen[69, 70] refererer til Amdahls lov[8] som en av de mest fundamentale lover når det kommer til studier av parallelle system.

I mange praktiske applikasjoner er ofte arbeidslasten fast, eller lik for hver kjøring. Ettersom antallet prosessorer øker i en parallell datamaskin, vil denne fastsatte arbeidslasten distribueres til

flere prosessorer for parallell eksekvering. Ta for eksempel en jobb med en fastsatt arbeidslast W (*workload*). Anta at W kan deles inn i to deler $W = \alpha W + (1 - \alpha)W$, hvor α er prosentandelen av W som må utføres sekvensielt, og de gjenværende $1 - \alpha$ kan eksekveres av n prosessorer samtidig. Antatt at all overhead ignoreres. Amdahl definerer dermed ytelsesøkningen for fast last (*fixed-load speedup*) som (hentet fra [70, s. 134]):

$$Sn = \frac{W}{\alpha W + (1 - \alpha)(W/n)} = \frac{n}{1 + (n - 1)\alpha} \rightarrow \frac{1}{\alpha},\ n \rightarrow \infty$$

Denne likningen har flere implikasjoner:

1. For en gitt last W, er maksimal ytelsesøkning begrenset av en øvre grense $1/\alpha$. Med andre er er den sekvensielle komponenten av programmet en flaskehals når det kommer til skalering. Ettersom α øker vil skalerbarheten, altså ytelsesøkningen ved høyere parallellitet, synke tilsvarende proporsjonalt.

2. For å oppnå god skalering er det viktig å få den sekvensielle flaskehalsen så liten som mulig.

3. Når et problem består av de to overnevnte delene, bør man få den største delen til å gå fortere. Med andre ord optimalisere for det vanlige tilfellet.

Amdahls lov tar som sagt ikke høyde for generell overhead for kommunikasjon mellom prosesser. Wang og Xu pressiserer derfor at ytelsen og skalerbarheten av et parallelt program ikke bare er begrenset av den sekvensielle flaskehalsen, men også den gjennomsnittlig overhead.

Inkludering av et formelt uttrykk for overhead i Amdahls lov er utenfor omfanget av denne rapporten. Det henvises til [70, s.135] for dette.

Det er også utarbeidet andre modeller relatert til skalering, for eksempel Gustavsons lov[55] og Amdahls Case rule[9].

2.2.2 Overhead ved parallellisering

Hwang og Xu[70] beskriver tre typer operasjoner i parallelle program:

Beregning: *(computation)* inkluderer aritmetiske/logiske operasjoner, dataoverføringer og kontrollflyt som man også finner i tradisjonelle sekvensielle program.

Parallelisering: *(parallelism)* refererer til operasjoner som er nødvendig for administrasjon av prosesser, for eksempel oppretting og terminering, prosessbytter og gruppering.

Interaksjon: *(interaction)* innebærer operasjoner er nødvendig for kommunikasjon og synkronering mellom prosesser.

Interaksjons og parallelliseringsoperasjonene kan være enten *eksplisitte* eller *implisitte*. En eksplisitt operasjon er en som er å finne i koden/teksten til det parallelle programmet, for eksempel Unix *fork()* kallet. En implisitt operasjon er ikke å finne i program koden, men blir gjort i bakgrunnen av systemet.

Interaksjon og parallelisering er kilder til *overhead*. Overhead betyr at ekstra tid og ressurser kreves i tillegg til selve arbeidslasten fra programmet (beregningen) for å utføre oppgaven. Hwang og Xu deler overheaden inn i fire typer:

- *Parallelliserings-overhead* forårsaket av prosessadministrasjon.

- *Kommunikasjons-overhead* som et resultat av at prosesser utveksler informasjon.
- *Synkroniserings-overhead* i å utføre synkroniseringsoperasjoner
- Overhead gjennom *Last-ubalanse* oppstår når noen prosesser er *idle* (venter) mens andre er opptatt.

Hvis det ikke finnes noen overhead, vil skaleringsfaktoren ved å bruke n prosessorer alltid være n.

Det er viktig å innse at forskjellige arkitekturer, programmeringsmodeller og algoritmer/applikasjoner, vil forsake mer eller mindre av de forskjellige typene overhead som en faktor i den totale kjøretiden. Et klart mål i så måte er derfor å minimalisere den delen av overheaden som har en effekt på kjøretiden og hindrer systemet å skalere med faktoren n.

2.3 Maskiner og arkitekturer

HPC-maskiner (superdatamaskiner) var tidlig preget av proprietære maskiner og arkitekturer fra produsenter som Cray, IBM og SGI. Disse maskinene var dyre, store og ofte vanskelige å oppgradere. Slike maskiner produseres og selges fortsatt i dag (for eksempel IBM pSeries), men først og fremst til organisasjoner og bedrifter med meget høye krav til ytelse (TFLOPS-klassen[2]) og god finansiell ryggdekning. Datamaskinenes bruksmuligheter i ingeniør- og forskningsarbeider har de senere årene ført til et stadig større behov for regnekraft og et sterke krav til skalerbarhet og pris.

De siste års store forbedringer i mikroprosessorer, minne, busser og nettverk har gjort det mulig å omgjøre grupper av relativt billige arbeidsstasjoner og servere til formidable superdatamaskiner. Slike grupper refereres til som lavkostnads-*clustere* (formalisert av Pfister i 1995[118]). Clustere kan med relativt få noder (enkeltmaskiner) oppnå beregninger i TFLOPS-klassen, til en betydelig lavere kostnad enn proprietære superdatamaskiner.

2.3.1 Arkitekturklassifiseringer

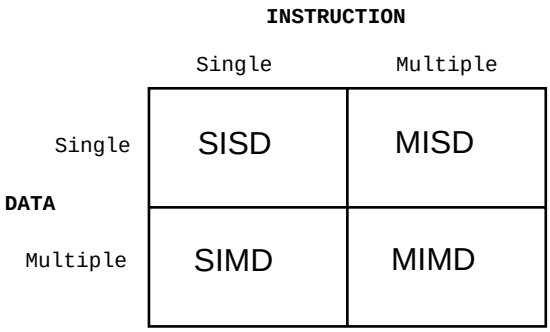
Generelt er de fleste arkitekturer rotfestet i Flynns hardware-taksonomi[46]. Teorien kan skisseres som matrisen i figur 2.3 med følgende beskrivelse:

SISD *(Single Instruction, Single Data)* klassifiserer en-prosessormaskiner konstruert med et instruksjonssettet separert fra datasettet, for eksempel en arbeidsstasjon med en prosessor. En slik seriell maskin sies å ha en Von Neumann-arkitektur[141]. Termen SISD blir ikke benyttet til vanlig. Typiske eksempler på slike er arbeidsstasjoner basert på Intel x86-arkitekturen.

SIMD *(Single Instruction, Multiple Data)* betegner maskiner som har et felles instruksjonssett som arbeider på hver sin del av datasettet. En instruksjon resulterer dermed i at flere operander blir oppdatert samtidig (Intel Pentium 3 og 4[81] sine SSE-registre (Streaming SIMD Extensions) faller inn under denne kategorien). Eksempler på SIMD-maskiner er Thinking Machines(CM2, CM5), MasPar og DAP.

MISD *(Multiple Instruction, Single Data)* MISD-termen er med på å gjøre kombinasjonen komplett.

[2] **TFLOPS**: Terra Floating-point Operations Per Second, uttales “terraflops”.



Figur 2.3: **Skissering av Flynns hardware-taksonomi.** Flynns modell ble i utgangspunktet utviklet for klassifisering av hardware og datamaskiner. Til tross for andre utarbeidelser av tilsvarene modeller, står Flynns taksonomi fra 1972 fortsatt sterkt.

MIMD *(Multiple Instruction, Multiple Data)* er maskiner som består av uavhengige prosessorer med separate datastrømmer. MIMD deles ofte i to klasser, løst koblede(NUMA) og tett koblede(UMA) system. Eksempler på tett koblede system er CRAY YMP, C90, J90, T90, SGI Origin 2000, Sun Enterprise Ultra og IBM AS400. Clustere faller inn under kategorien løst koblede systemer.

I den parallelle verden er det MIMD som i dag gjør seg gjeldene, og er grunnprinsippet for alle nyere superdatamaskiner og clustere.

2.3.2 Cluster-arkitekturen

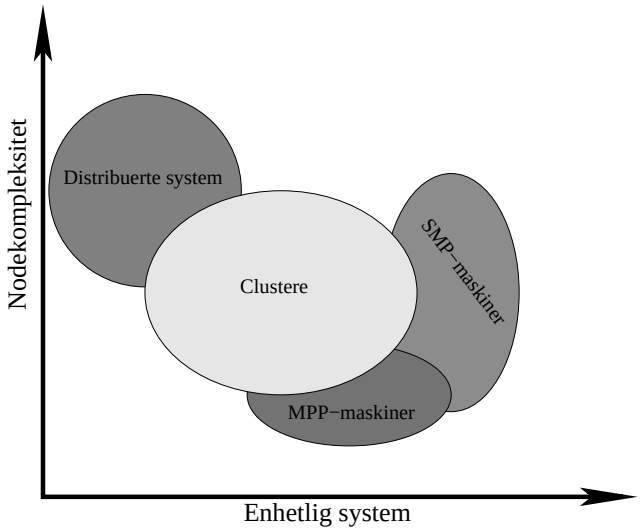
Et cluster er en gruppe uavhengige datamaskiner og utgjør dermed et løst koblet multiprosessor system (NORMA[70, s.237], underklasse av NUMA). Dedikerte superdatamaskiner derimot baseres som oftest på UMA- eller NUMA-arkitekturer[70, s.237]. De individuelle datamaskinene og kommunikasjonsnettverket er for lavkostnads-clustere COTS (hyllevare, se tillegg A).

En viktig egenskap med cluster-arkitekturen er den evne til å la alle nodene arbeide kollektivt sammen som en enkelt integrert dataressurs. I noen tilfeller kan det også være ønskelig å benytte enkeltnodene individuelt av flere brukere. Figur 2.4 viser hvordan clustere både kan implementere distribuerte og enhetlige maskiner og består ofte av MPP- og/eller SMP-noder. Ved det japanske Earth Simulation Center[39] har man vist at cluster-konseptet også kan benyttes for PVP-maskiner. Dette clusteret er basert på NEC SX6 PVP-maskiner og er i dag (januar 2003) verdens kraftigste cluster med 5120 prosessorer og en ytelse på ca. 35860 GFLOPS[3].

Cluster-arkitekturen kan summeres opp i fem hovedpunkter:

- *Cluster-noder:* Hver node er en komplett datamaskin med prosessor(er), cache, minne, disk og I/O-adaptere. Videre eksisterer det et komplett standard operativsystem på hver node. Aspekter rundt maskinarkitekturen beskrives nærmere i 2.3.3.

[3] **GFLOPS**: Giga Floating-point Operations Per Second, uttales “gigaflops”.



Figur 2.4: **Clustere i relasjon til andre arkitekturer.** Ved å se på faktorer som kompleksitet av noder og deres evne til sammen å operere som en enhet ser man hvordan cluster-arkitekturen spenner over både SMP-, MPP- og distribuerte systemer. Arkitekturen har med andre ord en høy tilpasningsevne sammenlignet med andre parallelle arkitekturer.

- *Singel-system Image:* Et cluster er logisk sett en enkelt dataressurs. Dette er i motsetning til et distribuert system, for eksempel et lokalt datanettverk, hvor alle nodene er individuelle ressurser.
- *Mellomnodekommunikasjon:* Nodene i clustere er koblet sammen og kommuniserer over et høyhastighetsnettverk. Selv om clustere til tider benytter mer eller mindre eksotiske nettverksarkitekturer, benyttes det som oftest standard protokoller for mellomnodekommunikasjon (for eksempel MPI). Clustere benytter i utgangspunktet et homogent nettverk mellom sine noder, men kan også benytte heterogene nettverk (såkalte *grid*-clustere).
- *Utvidet tilgjengelighet:* Clustere tilbyr en kostnadseffektiv måte å bygge datamaskiner med stor regnekraft.
- *Bedre ytelse:* Clustere tilbyr høy ytelse på flere områder. De kan for eksempel benyttes som superservere hvor n noder som hver kan betjene m klienter som til sammen kan betjene $n * m$ klienter. Clustere benyttes også stor grad som parallelle datamaskiner for eksekvering av større parallelle (HPC) jobber med sikte på minimal kjøretid (etter Amdahls lov).

I denne oppgaven refererer jeg i første omgang til lavkostnads-clustere (forkortet clustere) og ikke-proprietære cluster-liknende løsninger (for eksempel IBM SP2).

Skalerbare noder

Huse [68, s.36] påpeker hvordan en- og to-prosessor COTS arbeidstasjoner/servere produseres i store mengder og således kan selges til en lav pris. Dette er i motsetning til spesialdesignede stormaskiner eller PVP-maskiner som innehar spesielle egenskaper og kun produseres i moderate volum. Et annet

poeng er at slike lavkostnadsmaskiner i praksis følger Moores lov[97, 98]. Spesialdesignede maskiner derimot har en lang utviklingstid (typisk 2-3 år), noe som gjør at man for hver ny maskin må utvikle en maskin som er 4-5 ganger raskere enn dagens maskiner for å kunne holde tritt med Moores lov. Slike utviklingskostnader er vanskelig å forsvare sett fra et kommersielt synspunkt.

Huse[68, s.37] skisserer videre en skalerbar, ytelseseffektiv og kostnadseffektiv løsning ved bruk av SMP-maskiner som cluster-noder. Huse beskriver hvordan man ved bruk av SMP-maskiner kan øke antallet prosesser i clusteret totalt, samtidig som kommunikasjon over nettverket minimeres. SMP-nodene kan benytte shared memory mellom prosesser internt i SMP-noden og meldingsutveksling (for eksempel MPI eller PVM[51, 135]) mellom prosesser på forskjellige noder. Parallellisering internt i SMP-noden kan gjøres ved separate prosesser eller bruk av for eksempel OpenMP[113, 114, 115]. Likefullt fremhever Huse at valget av nodearkitektur bør reflektere oppgavene clusteret skal utføre.

De fleste cluster-integratorer bygger i dag sine clustere ved bruk av 2-prosessor SMP-maskiner basert på Intel og AMD sine 32 eller 64 bit arkitekturer.

2.3.3 COTS prosessorer og chipset

Dagens COTS prosessorer består i hovedsak av 32 bit CISC (Complex Instruction Set Computer) og RISC (Reduced Instruction Set Computer) arkitekturer. Eksempler på slike prosessorer er Intels Pentium 4 Xeon og AMD Athlon. Disse prosessorene benytter i utgangspunktet samme instruksjonssett (CISC), men har forskjellig intern implementasjon (RISC-liknende).

Ettersom det er åpenbare adresse- og instruksjonsbegrensninger i 32 bit arkitekturen har man nå begynt å se fremveksten av mer kommersielle 64 bit arkitekturer. Intel lanserte i 2000 sin Itanium-arkitektur basert på EPIC (Explicitly Parallel Instruction Computing)[128, 136]. AMD lanserte sin Athlon 64 også kjent som Clawhammer i desember 2002.

Instruksjonssett

Den enkelte prosessors instruksjonssett er avgjørende for både portabilitet mellom prosessorer og hastighet. For eksempel har Pentium 4 og AMD Athlon samme instruksjonssett (Athlon er en Intel-klone). Dette gjør at binærkode kompilert for den ene prosessoren kan kjøres uten modifikasjoner på den andre. Sammenlikner man Itanium-arkitekturen og Athlon 64 ser man derimot store forskjeller. Begge prosessorene er bakoverkompatible med 32 bit-instruksjonene og kan kjøre disse direkte, forskjellen ligger i instruksjonssettet for 64 bit kode.

For 32 bit prosessorene har man også sett endringer i instruksjonssett. Intel har med sine arkitekturer alltid hatt som mål å være bakoverkompatible med sine nye prosesorer. Dette har ført til at også dagens Pentium 4 benytter de samme kjerneinstruksjonene som sin forgjenger 8086, men har et større sett utvidelser. Man så tidlig behovet for en egen matteprosessor (x87) for 80386. Videre kom MMX (Multi Media eXtension) med Pentium MMX, SSE (Streaming SIMD Extensions) med Pentium III og SSE2 (Streaming SIMD Extensions 2) med Pentium 4 som de viktigste utvidelsene. X87-enheten åpnet for regning med høy presisjon, mens MMX, SSE og SSE2 gav rom for SIMD-eksekveringer.

Intern prosessordesign

De nye instruksjonssettene har sammen med høyere klokkefrekvenser og større cache, gitt dagens COTS prosessorer formidabel ytelse. Ser man nærmere på prosessorarkitekturene finnes det også andre deler som har gir betydelige ytelsesøkninger.

To kanskje viktige punktene er automatisk *prefetching* og *out-of-order-execution* av instruksjoner. Pentium 4 var en av de første prosessorene med en avansert hardware-implementert prefetching-heuristikk. Ved bruk av den eldre Pentium III måtte programmereren enten eksplisitt eller implisitt (gjennom kompilatoren) generere prefetch-instruksjoner for å minske overheaden ved *cache-miss*. Pentium 4 implementerer en avansert heuristikk som gjør at prosessoren gjør dette automatisk. Dette gir økt ytelse, spesielt til applikasjoner som flytter på større datamengder (for eksempel seismiske applikasjoner). I tillegg er det implementert funksjonalitet som *branch prediction* og parallell eksekvering av uavhengige instruksjoner.

Som nevnt er parallellitet en viktig metode for å oppnå økt hastighet. Bruk av tråder ved hjelp av bibliotek som Pthreads[79] og OpenMP har derfor lenge blitt benyttet for å oppnå parallellitet i programmer, da spesielt i programmer hvor I/O, låser og mutex-bruk er hyppig. For å minske overheaden ved bruk av slike teknikker har man forsøkt forskjellige teknikker på prosessornivå for mer effektivt å kunne eksekvere slik kode. Intels løsning på dette problemet, ofte referert til som *Simultanious multithreading*, har de kalt *Hyper-Threading* (HT)[90].

HT ble først introdusert i Intel Pentium 4 Xeon og lovet en generell ytelsesforbedring på 10–30% for trådbaserte applikasjoner. HT er en prosessormodus hvor man kort sagt deler en fysisk prosessor inn i to logiske. Dette lar seg gjøre ved hjelp av prosessorens evne til å utføre out-of-order-execution. Ved å la en tråd kjøre på hver logiske prosessor unngår man overhead gjennom operativsystemet ved bytte mellom tråder når en er opptatt med I/O, låser, mutexer eller liknende.

Chipset og I/O

Chipset benyttes som et samleutrykk for det man kan kalle "limet" i datamaskinen. Chipsetet er veien mellom CPU-en og I/O, det vil si PCI-*bridge* og minne kontroller. Selv om nyere prosessorer nå implementert distribuert minnekoherens (i SMP-maskiner), er chipsetet avgjørende for I/O-ytelsen.

Huse [66] forklarer hvordan forskjellige chipset-implementasjoner kan ha stor innvirkning på I/O-ytelsen og dermed effektiviteten av nettverket. Det samme fremheves av Conservative Computer, Inc.[21]. Kort summert er et bra chipset og programvare som tar hensyn til chipsetets egenskaper avgjørende for rask I/O og dermed kommunikasjonsytelse.

Dagens PCI-buss[116] står ofte for den største begrensningen når det kommer til I/O-båndbredde. En PCI-buss kan sende $32bit = 4byte$ pr. klokkesykel. En 33MHz PCI-buss har dermed en maksimal båndbredde på $33MHz * 4byte = 132MByte/s$. En 66MHz PCI-buss får tilsvarende maks på $264MByte/s$. Den nye PCI-X standarden[82] kan derimot sende $64bit = 8byte$ pr. klokkesykel og øke sykelfrekvensen til 133MHz og vil dermed heve det teoretisk maksimale til $1056MByte/s$. PCI-X standarden er i ferd med å etablere seg (blir blant annet benyttet av Gigabit Ethernet-adaptere). Likefullt er det fortsatt et tydelig skille mellom maskinenes interne minne- og I/O-båndbredde.

Kapittel 3

Parallell kommunikasjon

For å kunne implementere effektiv kommunikasjon er identifisering og eliminering av flaskehalser viktig. I dette kapittelet presenteres kommunikasjonsmodellen LogGP[5, 24]. Ved hjelp av denne modellen har jeg forsøkt å beskrive enkelte nettverk (interconnect[1]) og bibliotek sin relasjon til denne.

En kort introduksjon til forskjellige interconnect presenteres, da med vekt på Gigabit Ethernet, Myrinet og spesielt SCI. Av kommunikasjonsbibliotek har jeg forsøkt å beskrive de mest kjente bibliotek relatert til problemstillingen i denne rapporten.

3.1 Kommunikasjonsmodellen LogP (LogGP)

David Culler et. al. foreslo i 1993 LogP modellen[24] for modellering av kommunikasjon i parallelle datamaskiner. Modellen ble utviklet som en base for design og analyse av raske portable parallelle algoritmer. Culler hevder at andre modeller er urealistiske i at de ofte gjør kunstige antagelser som for eksempel ingen forsinkelse og uendelig båndbredde. For eksempel er den ofte benyttede PRAM[70, s.13] modellen urealistisk da den antar at alle prosessorer arbeider synkront og at all kommunikasjon mellom prosesser er gratis.

Culler definerer fire hovedparametere for LogP modellen:

L: en øvre grense for forsinkelse (*latency*) som oppstår ved kommunikasjon med meldinger med et *word/32 bit* (eller et mindre antall word) fra sin kildemodul til sin målmodul. Det presiseres at dette er hardware-forsinkelse for kommunikasjonen.

o: overhead defineres som den tiden prosessoren er opptatt sending og/eller mottak av hver melding. Under denne tiden kan ikke prosessoren gjøre andre operasjoner. Kopieringer som utføres av DMA-maskiner (Direct Memory Access) regnes dermed ikke som overhead (initiering av disse maskinene er derimot å regne som overhead).

g: avstanden (gap) er definert som det minste tidsintervallet mellom etterfølgende meldingstransmisjoner eller -resepsjoner for en prosessor. Parameteren inkluderer o. Den inverse av g tilsvarer den tilgjengelige per-prosess kommunikasjonsbåndbredde.

P: antallet prosessesorer/minnemoduler. Man antar enhetstid for lokale operasjoner og kaller det en sykel.

[1] Mellomnodenettverk for clustere refereres til som *interconnect* i dagligtalen.

Videre antas det at nettverket har en endelig kapasitet. Det maksimale antall meldinger som kan sendes fra enhver prosessor til enhver annen prosessor til en hver tid er $\frac{L}{g}$.

Tiden for å sende et word fra en prosess til en annen kan dermed uttrykkes som: $T_{ping} = o+L+o$ eller $T_{ping} = 2o + L$. Den første o-parameteren representerer overheaden ved sending, som er tiden det tar for en prosessor å legge et word ut på nettverket. Den andre representerer overhead ved mottak, tiden den mottakende prosessoren bruker på hente et word fra nettverket. L representerer hardware-forsinkelsen i nettverket, med andre ord tiden det tar for signalet å forplante seg gjennom adapteret og nettverket. En tilsvarende beskrivelse av en ping-pong test (sende et word med påfølgende respons fra mottaker) kan uttrykkes som $T_{ping-pong} = 4o + 2L$.

Albert Alexandrov et al.[5] fremla i 1995 en utvidelse av LogP modellen; LogGP. Den nye G-parameteren representere båndbredden for lengre meldinger. Alexandrov hevder at den opprinnelige LogP modellen ikke nøyaktig kan modellere både lange og korte meldinger. Argumentasjonen bygger på at flere parallelle maskiner har spesiell støtte for lengre meldinger og kan oppnå betydelig høyere båndbredde ved lange meldinger enn ved korte. Faktoren $\frac{g}{G}$ blir dermed en indikator på ytelsesforbedringen ved å gruppere mindre meldinger til en stor. Mange maskiner har begrensninger i meldingsprosseseringen i nettverksadaptere og ikke i nettverket selv. G bestemmer i slike tilfeller hastigheten til og fra nettverksadapteret (for eksempel gjennom DMA) isteden for båndbredden på de fysiske linkene.

3.2 LogGP og applikasjonsytelse

Valg av interconnect for et cluster kan være en komplisert vurdering. Applikasjoner som skal kjøres og ytelseskrav til disse, kan være avgjørende for valg av arkitektur. Det er ikke nødvendigvis gitt at et raskt interconnect gir kortere kjøretid. I andre tilfeller kan interconnectet være avgjørende for om applikasjonen skalerer i det hele tatt. Et viktig spørsmål i så måte er: Hva innebærer en rask kommunikasjonsforbindelse?

Richard P. Martin et. al.[92] gjorde i relevans til dette spørsmålet en meget interessant undersøkelse. Martin prøvde å bestemme applikasjoners følsomhet for forskjellige kommunikasjonskarakteristika i clustere basert på LogGP modellen.

Martin beskriver videre to viktige spørsmål:

1. Hvilke parametere påvirker applikasjonsytelsen mest?
2. Hvor mye påvirkes ytelsen?

Han presiserer videre at det å forandre ytelsesaspektene på et punkt i et parallelt program kan få uforutsette konsekvenser for andre deler av programmet. For eksempel kan forandring i overhead ha innvirkning på lastbalanse og synkronisering. Med andre ord er både modellering og forutsigbarhet av parallell kommunikasjon meget komplisert.

Martin et. al. implimenterte et kommunikasjonslag hvor man kunne forandre individuelt på parameterene i LogGP modellen. Deretter ble resultanene av kjøringer for et utvalg applikasjoner med forskjellige kommunikasjons mønstre observert.

Studien kom frem til følgende interessante hovedpunkter:

- Applikasjoner er mest sensitive for en økning i *overhead*. Selv lett kommunikasjon fører til en ytelsesforværring med en faktor på 3-5 ved kun å introdusere forsinkelsen ved mye brukte protokollstakker (for eksempel TCP/IP).

- Applikasjoner er også sensitive for økning i *gap* parameteren. Spesielt for applikasjoner med høy kommunikasjonsfrekvens.

- De fleste applikasjoner er mindre følsomme for nettverksforsinkelse (L).

- De fleste applikasjoner er relativt lite påvirket av båndbredden (G) i et kommunikasjonssystem.

Resultatene indikerte altså at applikasjoner påvirkes mest ved en økning i o-parameteren, etterfulgt av g og med L(forsinkelse) og G(båndbredden) som de minst kritiske. En interessant resultat av testene var at samtlige testapplikasjoner viste en lineær avhengighet av både overhead og gap. Martin konkluderer dermed med at en forbedring på disse punktene vil føre til en tilsvarende forbedring av ytelsen, da begrenset av Amdahls lov. Resultatene indikerer også at det i noen tilfeller kan være mer fornuftig å gjøre en investering i kommunikasjonsprogramvare og -hardware med mindre overhead og gap, enn i mer prosesseringskraft.

Applikasjonen MM5[96] er et slikt eksempel. MM5 er en modell for simulering av atmosfæriske sirkulasjoner, med andre ord værsimuleringer. Tester utført av Scali AS viser at MM5 skalerer lineært (etter Amdahls lov, da med tangens $\frac{1}{2}$) fra 1-256 noder ved bruk av SCI. Ved bruk at Fast Ethernet skalerer applikasjonen kun fra 1-16 noder. Kjemiapplikasjonen Dalton[26], har derimot ingen ytelsesendring ved bruk av SCI fremfor Fast Ethernet.

Det er viktig å merke seg at Martin sine resultater ikke direkte er overførbare til alle applikasjoner. Martin undersøkte bare et utvalg applikasjoner og algoritmer. Konkrete case-eksempler med andre algoritmer og kommunikasjonsmønster kan produsere andre resultater. For eksempel vil en godt synkronisert parallell applikasjon være mer følsom for forsinkelse, mens en applikasjon med store datasett kan være avhengig av stor båndbredde.

3.2.1 Oppsummering av LogGP

LogGP modellen skisserer et utgangspunkt for betraktinger rundt både interconnect og kommunikasjonsbibliotek. Likefullt benyttes som oftest en noe grovere modell i dagligtalen, da med fokus på forsinkelse og båndbredde:

1. Forsinkelse som summen av $g + L$. Løst definert som systemets evne til å kommunisere med flere mindre datastørrelser.

2. Båndbredde som en funksjon av både G og g, altså evnen til å flytte større mengde data på kort tid.

Studien til Richard P. Martin et. al. i avsnitt 3.2 konkluderte med at parameteren o var den viktigste for ytelsen. Likefullt bør man, i et realistisk miljø, også ta hensyn til kombinasjonen $g + L$. Dette er spesielt viktig for applikasjoner som genererer hyppig trafikk bestående av mindre datastørrelser.

Båndbredden eller $\frac{1}{G}$ og $\frac{1}{g}$ gjør seg gjeldende ved kommunikasjon av større datablokker. Her er det viktig med ytelse innenfor det området av datastørrelser man forventer å sende med en applikasjon.

Jeg kan dermed dra slutningen at det ikke bare er arkitekturen og hardwaren i interconnectet viktig, men også overheaden for eksempel gjennom en eller flere kommunikasjonsbibliotek, protokoller, bruk av interrupter, DMA (Direct Memory Access) eller PIO (Programmed I/O).

3.3 Interconnect

I dette avsnittet introduseres noen av de tilgjengelige høyytelses cluster-interconnectene på markedet i dag. Det er flere Gbps-interconnect tilgjengelig, for eksempel cLAN/GigaNet[42], Gigabit Ethernet[80], SCI[78], Myrinet[102], QsNet (Elan)[117], ATM[14], HiPPi[62], ServerNet II[56], Synfini[142] og Memory Channel[44].

Jeg vil ikke forsøke å beskrive alle disse teknologiene, men kort beskrive de tre mest aktuelle for denne rapporten; Gigabit Ethernet, Myrinet og SCI. Det vil også bli gitt en kort introduksjon til nye teknologier som kan bli viktige i clustering-sammenheng.

3.3.1 Gigabit Ethernet (IEEE 802.3z)

Gigabit Ethernet (GbE) er en utvidelse av Fast Ethernet (100 Mbps - IEEE 802.3u) og klassisk Ethernet (10 Mbps - IEEE 802.3[72]) standardene. Standarden lagt frem i 1998 og opererer med en hastighet på 1000 Mbps. GbE har begynt å bli vanlig både i clustere og vanlige LAN. I juni 2002 fikk Ethernet Task Force[6] godkjent sitt forslag til 10-GbE som IEEE Std. 802.3ae-2002[80, 73].

GbE benytter linkhastigheter på 156 MByte/s med enten et enkelt bit på 1250 MHz eller 10 parallelle bit på 125 MHz. Pakkeformatet for GbE er det samme som for de andre medlemmene av Ethernet familien, bortsett fra at den minimale pakkestørrelsen er økt fra 64 byte (Fast Ethernet) til 512 byte. Dette blir gjort for å unngå en reduksjon i kabellengden (slik som ble tilfellet for Fast Ethernet). En teknikk kalt *carrier extension* blir benyttet for å legge til data på pakker som er mindre enn 512 minimal størrelse.

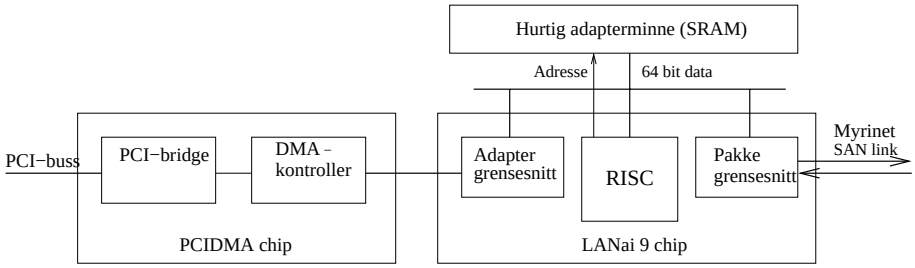
Siden GbE er svitsjbasert (selv om half-duplex modus er en del av standarden), vil dette skape økte forsinkelser (økt L). Da spesielt med tanke på billige svitsjer som benytter *store-and-forward* rutingalgoritmer. Relativt høy båndbredde(G) kan likevel oppnås ved hjelp av såkalt *packet bursting*. Denne teknikken tillater etterfølgende pakker på nettverket med et kortest mulig mellomrom (*inter-packet gap*) på inntil 8 KByte data. GbE baserer seg på at kommunikasjonsbibliotek/-programvare/-protokoller detekterer pakketap og gjør retransmisjon av disse (for eksempel med TCP/IP[27]). Dette introduserer tradisjonelt økt overhead (økt o) og lavere båndbredde enn hva man teoretisk kan oppnå.

GbE adapterene benytter interrupter for å fortelle kommunikasjonsprogramvaren at det har ankommet pakker. Også dette fører til økt overhead og forsinkelse. Flaskehalsen relatert til båndbredde er arvet fra standard og Fast Ethernet, men kan reduseres ved hjelp av såkalte *jumbo frames[7]* utviklet av Alteon Networks. Flaskehalsen er relatert til MTU (Maximum Transfer Unit) for nettverket. Ved bruk av jumbo frames, kan man øke MTU størrelsen fra standard 1518 byte til 9000 byte, noe som gir en ytelsesøkning på opptil 50% for større datastørrelser. Bruk av jumbo frames fordrer støtte for denne funksjonaliteten i mellomliggende svitsjer og rutere.

3.3.2 Myrinet (ANSI/VITA 26-1998)

Myrinet[17] er et meldingsutvekslingsbasert interconnect utviklet av Myricom, Inc[103]. Myricom sitt mål er å lage et COTS interconnect for bygging av clustere. Myrinet er basert på multicomputer og VLSI teknologi utviklet ved California Institute of Technology og ATOMIC/LAN teknologi utviklet ved University of Southern California. Teknologien har gått fra å være en proprietær SAN-protokoll til å bli en åpen standard; American National Standard: ANSI/VITA 26-1998, "Myrinet-on-VME Protocol Specification"[10, 102].

Myrinet adapteret er bygget rundt en VLSI-chip med RISC-prosessor kalt LANai (se figur 3.1). Denne chipen implementerer Myrinets grensesnitt, pakke grensesnittet, en DMA-maskin og et raskt



Figur 3.1: **Skjematisk oversikt over Myrinet-adapteret.** Myrinet-adapteret er DMA-basert og benytter en *on-board* RISC prosessor og et eget hurtig minne (SRAM) for kjøring av MCP programmet.

statisk minne. Myrinet benytter gjennomført DMA-maskinen på sine adaptere ved kommunikasjon. Minnet benyttes for pakkebuffering og inneholder et kontrollprogram kalt *Myrinet Control Program* (MCP) som eksekveres av LANai-prosessoren. Denne arkitekturen tilbyr et fleksibelt høyhastighets-grensesnitt mellom en generisk buss og en Myrinet link. MCP programmet gir mulighet for å direkte implementere kommunikasjonsprotokoller i adapteret. MCP programmet eksekveres på adapterets prosessor og unngår dermed overhead (lav o) gjennom operativsystemet.

Myrinet SAN- eller LAN-linkene er full duplex og opererer på 225 MByte/s (SAN-1800) eller 250 MByte/s (SAN-2000). Alle pakker CRC-sjekkes (Cyclic Redundancy Check) og det interne minne samt PCI-aksesser har paritetsjekk. Myrinet grensesnittet kan ikke koblets adapter-til-adapter (som for eksempel GbE), men må gå gjennom en eller flere svitsjer. Svitsjene benytter såkalt ormhull eller *cut-trough* pakkeruting liknende det som brukes i CRAY T3D og Intel Paragon.

3.3.3 SCI (IEEE Std. 1596-1992)

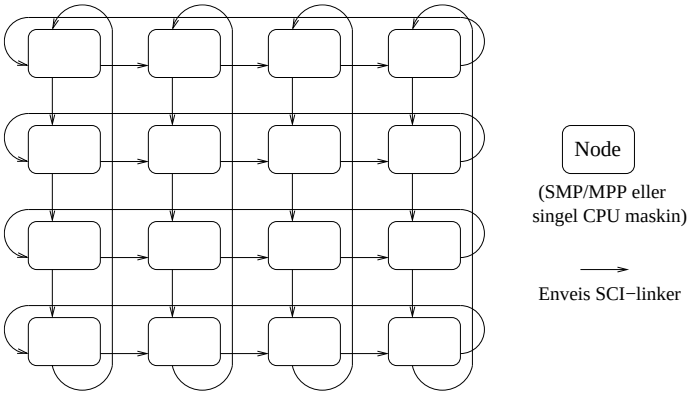
SCI (Scalable Coherent Interface) har sitt utspring fra et prosjekt utført av en gruppe busseksperter sent på 80-tallet. Disse ønsket å definere en HPC-buss eller “Superbuss” med støtte for en høy grad av multiprosessering. Gruppen innså imidlertid raskt at en tradisjonell bussarkitektur ikke ville kunne oppnå kravene man fremsatte. Hellwagner[57] beskriver hovedgrunnene:

1) Buss er en sentralisert ressurs og dermed en flaskehals som ville gjøre seg mer og mer gjeldende ettersom prosessorene ble raskere.

2) Bussignaler nærmet seg allerede sine fundamentale grenser (lysets hastighet), resulterende i elektrisk kompliserte og dyre løsninger samt korte busslengder.

Gruppen forlot dermed sitt bussorienterte syn og utviklet en distribuert løsning for å overkomme problemene med signaler og delte ressurser. Resultatet ble SCI-spesifikasjonen, standardisert i 1992 (IEEE Std. 1596-1992)*[78]*. SCI er en relativt stor og komplisert spesifikasjon. Store deler av spesifikasjonen er formelt spesifisert og simulert i programmeringsspråket C.

SCI-konseptet utgjør et globalt distribuert adresse område på 64 bit og kan adressere opptil 64k noder innenfor samme domene. SCI ser på I/O som en minne-til-minne operasjon. Dette tillater prosessorene å gjøre send og mottak som enkle *load* og *store* instruksjoner mot adresser i deres virtuelle adresseområde. All kommunikasjon kan dermed skje på *user-level*, slik at overhead (o) gjennom operativsystemet blir unngått. Det benyttes enkle protokoller for å sikre levering av data og for å unngå *deadlock* og *starvation*.



Figur 3.2: **Skjematisk oversikt over en SCI-torus.** Skjematisk oversikt over SCI-linkene i et 2-dimensjonalt SCI-torus. Linkene danner ringer i begge dimensjonene.

SCI er basert på ensidig trafikk over lukkede ringer (eller *ringlets*[64]). Ruting mellom ringer gjøres enten ved svitsjer eller i adapteret selv. Inkludering av alle noder i en ring er tilstrekkelig for mindre system, men for å oppnå god skalering benyttes svitsjer eller flerdimensjonale toruser[89, 88]. Figur 3.2 viser hvordan et 16 noders system kan kobles som en 2-dimensjonal torus. I slike tilfeller opptrer SCI-adapterene selv som en distribuert svitsj (kontrollerbar av programvare) og eliminerer nødvendigheten av sentraliserte svitsjer.

For å kunne gjøre høyhastighetskommunikasjon blir feilsjekking gjort i adapter-hardwaren basert på en 16-bit CRC kode som beskytter hver SCI-datapakke. Overføringene kan gjøres ved DMA eller PIO.

SCI-spesifikasjonen inneholder også et valgfritt cache koherens lag som spesifiserer hardware-protokoller og konsepter som tillater prosessorer å cache ikke-lokalt minne. Siden et SCI-nettverk ikke har en sentralisert ressurs (for eksempel en minnebuss) som kan lyttes på (snoopes) av tilkoblede prosessorer, er SCI-basert på en katalogbasert løsningsmodell.

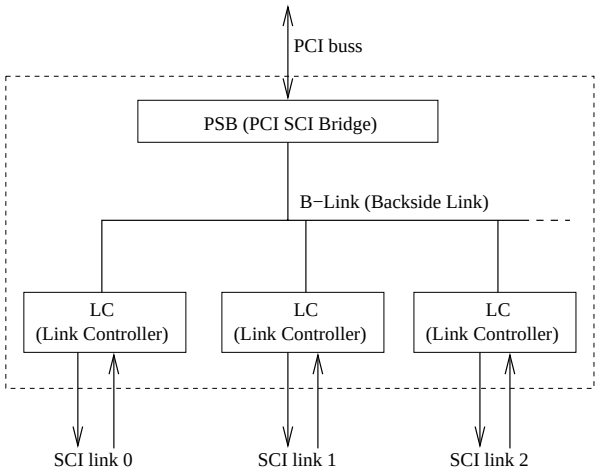
En nøkkelfunksjonalitet i SCI er at det tilbys rettferdig levering av data og en garanti om at sendersiden kan detektere hvorvidt dataene kom frem eller ikke. De innebygde feilsjekkingsprotokollene i SCI garanterer at de data som kommer frem er riktige. SCI-standarden garanterer derimot ikke rekkefølge på pakker og legger dermed ansvaret for implementasjon av en konsistensmodell[4] over på biblioteker og mellomvare.

Dolphins SCI-implementasjon

SCI-adaptere[87] fra Dolphin Interconnect Solutions[28] har vært på markedet siden 1993. Adapterene tilkobles nodens I/O-buss. Utviklingen startet med enkle SBus-adaptere[31, 1] og utviklet seg til dagens avanserte PCI-adaptere[37]. Dolphins adaptere tillater direkte *mapping* av minneaksesser fra fra en maskin til en annen målmaskin.

SCI-adapterene støtter kun ikke-koherente SCI-transaksjoner. SCI benyttes i utgangspunktet kun som en sikker kanal med høy båndbredde og lav forsinkelse. D33x adapterene har en link hastighet på 666 MByte/s. For 32 bit buss systemer mappes maskinens fysiske adresseområdet fra 32 bit til 64 bit SCI-adresser. Adapterene kan både benytte SCI-svitsjer og ruting i selve adapteret.

Dolphin adapterene har en tredelt arkitektur (som vist i figur 3.3):



Figur 3.3: **Skjematisk oversikt over Dolphin SCI-adaptere.** PSB-en står for oversettelsen mellom maskinens I/O-buss (PCI eller SBus). LC enhetene (en for hver dimensjon) står for ruting og selve pakkehåndteringen på SCI-linkene.

- B-link (Backside link)[29]: PCI-SCI-adapteret er fokusert rundt B-link bussen. Denne opererer som en *backbone* link mellom linkkontrolleren (LC-en) og PSB-en.

- PSB (PCI to SCI Bridge)[36, 34, 32]: er en bridge mellom PCI-bussen og B-link bussen. Denne modulen konverterer PCI-transaksjoner til SCI-transaksjoner og omvendt. PSB-en overfører data mellom bussene i begge retninger.

- LC (Link Controller)[35, 33, 30]: Denne enheten håndtere dataoverføringer på det fysiske SCI-laget ved, å sende og motta pakker på eksterne SCI-linker. Den har også et grensesnitt til B-linken og benytter denne for å overføre data mellom de eksterne grensesnittene og PCI-bussen (via PSB-en). LC-en inkluderer rutingfunksjonalitet som bestemmer hvorvidt de innkommende dataene har destinasjon for “denne” noden. Hvis så ikke er tilfellet blir dataene plassert i en FIFO (First In First Out) kø i PSB-en som videresender dataene direkte til output-linken (via tilhørende LC).

En PCI-enhet (master) kan følgelig lese og skrive til og fra minne/registre på en annen PCI-buss via SCI-interconnectet. Det benyttes to typer operasjoner for dataoverføring:

PIO PIO-operasjoner er enkle *load* og *store* operasjoner fra CPU-en mot PCI-SCI bridgen. Bridgen vil videresende requester og responser mellom PCI-bussene. En PCI-read/-write request vil oversettes til SCI-read/-write requester og sendes til mottakende PCI-buss. Den mottakende PCI-bussen vil utføre de tilsvarende operasjonene. Responsen oversettes til en SCI-responspakke og sendes til PCI-bussen hvor requesten kom fra.

DMA DMA-operasjoner vil benytte DMA-maskinen på adapteret. Når denne er startet vil den kjøre fra en allerede bygget eksekveringskø i sendernodens minne. Dette betyr at PCI-SCI-bridgen nå har full kontroll over operasjonen. Data hentes fra minnet og dyttes ut på SCI-nettet terminert med en *word-count* i kontrollblokken. Ferdigstillelse av operasjonen signaliseres.

Interruptfunksjonen behandles noe spesielt ettersom SCI-adapterene normalt ikke benytter denne funksjonen for normal kommunikasjon (som for eksempel Ethernet adaptere). Interrupter sendes ved hjelp av eksplisitte driveroperasjoner. Det er også mulig å videresende et interrupt fra en PCI-buss til en annen. Interrupt-funksjonen utføres som en eksplisitt *lock-modify-write* operasjon. En slik SCI-lock-request overføres til en normal *lock* operasjon på den mottakende PCI-bussen.

Write gathering

Det finnes flere generasjoner av SCI-adaptere fra Dolphin. En av forskjellene mellom disse kortene er størrelsen på det interne *stream-buffere* i PSB-en. Dagens D33x-adaptere har 16 write-streamer og 16 read-streamer på 128 byte hver.

For å oppnå optimal båndbredde kan *write-gathering* benyttes når senderenheten på PCI-bussen ikke kan overføre mer enn 64 byte i en burst. Ved bruk blir dataene samlet i stream-bufferne inntil 128 byte grensen er nådd. Stream-bufferet er således fullt og tømmes (dataene sendes over SCI). Et buffer kan også eksplisitt tømmes ved å skrive til et kontrollregister eller ved et respons-interrupt.

Hvor mye data og hvor lenge det har ligget i streamen er med på å bestemme hvor mange SCI-transaksjoner som genereres. Bruk av write-gathering kan bestemmes ved kall til driveren (se avsnitt 6.1.1) med argumenter som RMAP_IO eller RMAP_GATHERING for henholdsvis *write-through* eller write-gathering.

Address Translation Table

PCI-adapteret aksepterer både 32 bit og 64 bit PCI-adressering. En 64 bit PCI-adresse blir i mindre grad modifisert. En 32 bit PCI-adresse derimot, blir oversatt til en 64 bit SCI-adresse ved å benytte en Address Translation Table (ATT). ATT-en ligger i en SRAM på kortet og bygges opp av driveren ved initiering og mapping av minneområder. Elementer i tabellen (også kalt *Page Descriptors*) caches i PSB-en. ATT tabellen har i utgangspunktet 16K rader på 64 bit hver. På nyere adaptere er dette programmerbart.

ATT tabellen benyttes altså for å mappe PCI-adresser til SCI-adresser og omvendt. Et viktig poeng i så måte er at entrier i ATT tabellen ikke kan overlappe hverande.

3.3.4 Kommende teknologier

To nye og lovende interconnect-teknologier på vei inn på markedet i dag er InfiniBand og StarFabric. Hvorvidt disse egner seg for cluster-interconnect med hensyn på skalerbarhet og da spesielt ytelse/pris faktoren gjenstår å se.

InfiniBand

InfiniBand-arkitekturen(IBA)[75, 23, 48] er et resultat av to industriinitiativer; Next Generation I/O(NGIO) og Future I/O. IBA er for øyeblikket en industristandard, og mange forventer å se den som en *de facto* standard om noen år.

IBA likner Myrinet med punkt-til-punkt linker, et hierarki av svitsjer og en aktiv bruk av DMA. Til forskjell fra Myrinet kan IBA benytte flere kanaler pr. adapter. IBA skiller seg også fra flere andre teknologier i at den kan skalere i flere dimensjoner. Skalering i antall noder er begrenset av et adresseområde på 64 bit. Med reservasjon av noen adresser for *multicast*-funksjoner hevdes det at IBA skalerer opptil 48000 noder. Ved bruk av IBA-rutere (det er ruting headere i alle IBA pakker, liknende de i IPv6) er den teoretiske grensen 2^{80} antall noder. Båndbredden kan skaleres opp ved å legge til

flere adaptere i nodene eller benytte flere av kanalene på samme adapter til samme punkt-til-punkt-forbindelse.

IBA benytter full-duplex serielle linker. Disse linkene er har hastigheter opp mot 2.5 Gbps, ca. dobbelt så fort som en 32-bit 33MHz PCI-buss. I tillegg definerer IBA parallelle linker; 1X, 4X og 12X. Ved å koble opptil 12 linker sammen kan man oppnå båndbredde opp mot 30Gbps (aggregert båndbredde begge veier).

StarFabric

StarFabric fra StarGen[133] er en litt spesiell, men meget interessant løsning på interconnect utfordringen. StarFabric er i likhet med InfiniBand basert på punkt-til-punkt linker og et hierarki av svitsjer. StarGen har tatt utgangspunkt i overheaden forbundet med protokoller i for eksempel GbE og laget en løsning som prøver å omgå dette. Til forskjell fra både Myrinet, SCI og InfiniBand tar StarFabric utgangspunkt i eksisterende kategori 5 Twisted Pair kabler (benyttes for GbE).

StarFabric løsningen kan minne mye om PCI-SCI. Grensesnittet mot adapteret består av to 2.5Gbps serielle linker. Etter initiering fungerer adapteret som en PCI-bridge mellom den lokale PCI-bussen og de andre PCI-bussene. Adapteret oversetter altså PCI-transaksjoner til StarFabric pakker for i andre enden å oversette StarFabric pakker til PCI-transaksjoner på den nye noden. Dette medfører minimalt av overhead siden ingen BIOS, driver, operativsystem eller annen programvare er nødvendig for kommunikasjon. Adapteret utfører adresseoversetting mellom lokale og ikke-lokale busser. Med andre ord fungerer StarGen som en forlengelse av nodens PCI-buss.

3.3.5 Oppsummering av interconnect

Den høye tilgjengeligheten og lave prisen på dagens GbE komponenter gjør teknologien til en god kandidat for cluster-interconnect. Ved hjelp av teknikker som packet bursting og jumbo frames oppnår man ofte tilfredstillende båndbredde for mange applikasjoner.

Myrinet tilbyr en både høyere båndbredde lavere forsinkelse (sammenliknet med Ethernet-familien) og et avansert DMA-basert adapter. Disse egenskapene har gjort Myrinet til et populært cluster-interconnect i de tilfeller GbE faller blir en flaskehals.

SCI-implementasjonen fra Dolphin tilbyr både høyere link-hastighet, lavere forsinkelse og høyere båndbredde enn både GbE og Myrinet. Til forskjell fra både GbE og Myrinet adapterene kan SCI benytte både PIO- og DMA-moduser ved overføring. Likevel benyttes for det meste PIO da dette gir lavere forsinkelse og høyere båndbredde (se figur 3.4).

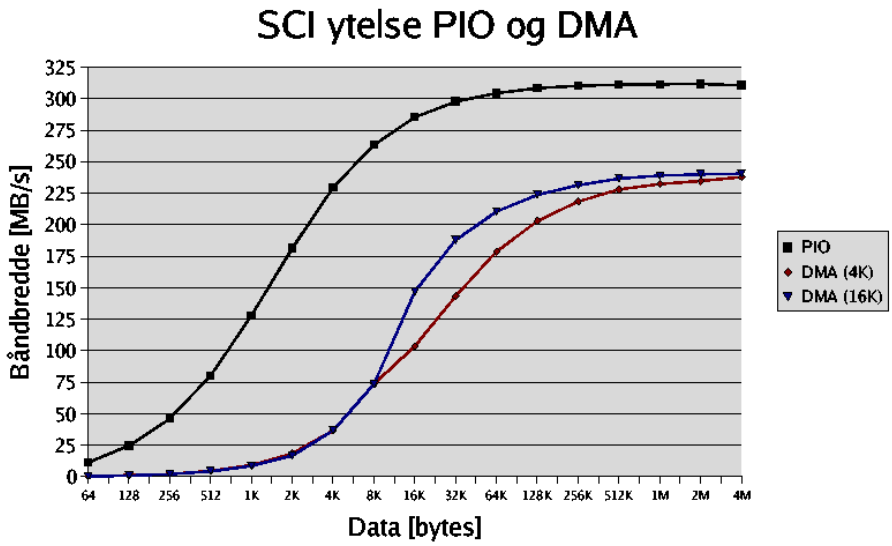
Problemene relatert til GbE som cluster-interconnect er hovedsaklig forsinkelse(L og g) og overhead(o). Som man kan se av tabell 3.3.5 vil forsinkelsen variere med antallet svitsjer (ved bruk av svitsjhierarki) pakken går gjennom og antall meter kabel. Dette gjelder også for Myrinet-nettverket.

SCI-toruser fungerer derimot som en distribuert svitsjressurs fremfor GbE og Myrinet sine sentraliserte svisjhierarkier. Ruting gjennom en LC fra en ring til en annen introduserer en noe høyere forsinkelse enn ruting innenfor samme ring. Likevel kan både antallet LC-er og dimensjoner som byttes minimaliseres ved utvikling av gode rutingalgoritmer. SCI-toruser er er altså mer stabile ved kabelfeil og gir en bedre skalerbarhet ved alle-til-alle kommunikasjon. For detaljer rundt distribuert SCI-ruting refereres det til [64] og [18].

For å ikke la mottakermaskinen kun stå å behandle interrupt, benytter de fleste GbE adaptere en teknikk med å samle opp flere innkommende pakker i et buffer før interruptet genereres. Hvis ikke

	GbE	**Myrinet**	**SCI (torus topologi)**
L	signalhastighet pr. meter * meter + svitsj-forsinkelse * antall svitsjer	signalhastighet pr. meter * meter + svitsj-forsinkelse * antall svitsjer	signalhastighet pr. meter * meter + forsinkelse i LC for samme ring * adaptere + forsinkelse LC ved bytte av ring * adaptere
o	interrupt-tid + protokollhåndtering + kopiering til og fra applikasjonsbuffer	DMA-initiering	SCI-checkpointing
g	interrupt-intervall + inter-packet gap 8 KByte + sendetid for forrige pakke	Denne faktoren avhenger av hvorvidt det benyttes såkalt *DMA-chaining* eller ikke. Konkrete tall er ikke kjent.	SCI-standarden spesifiserer 1 klokkesykel (en transmisjon) som såkalt *idle-byte* mellom pakker. For Dolphin har det ikke latt seg gjøre å implementere dette slik at de benytter 4 sykler mellom pakker i sin implementasjon.
G	Ongz og Farelly[112] viser til målinger med en TCP båndbredde på 76.6 MByte/s (MTU 9000) og en LAM/MPI båndbredde på 35.7 MByte/s (MTU 9000). Forsinkelsen (ping-pong-half) ble målt til 66 μs (MTU 1500). Disse tallene er sterkt implementasjonsavhengig, både når det kommer til programvaren og adapterene. Maksimal teoretisk hastighet er 125 MByte/s.	Myricom rapporterer på sin webside[101] (for PCII64C adapterene) en MPI-båndbredde på 245 MByte/s og en forsinkelse (ping-pong-half) på 7 μs.	Scali AS oppgir på sin webside[127] en MPI-båndbredde på > 320 MByte/s og en forsinkelse (ping-pong-half) på < 4 μs.
P	Uttrykkene over er oppgitt med et utgangspunkt om 2 prosessorer/noder. Teoretisk sett vil svitsjene være begrensende for størrelsen på et totalsystem.	Uttrykkene over er oppgitt med et utgangspunkt om 2 prosessorer/noder. Teoretisk sett vil svitsjene være begrensende for størrelsen på et totalsystem.	Det har teoretisk blitt vist at SCI skalerer linjert opp til 512 noder (med 3D torus) [18].

Sammenlikning av GbE, Myrinet og SCI i LogGP.



Figur 3.4: **SCI-ytelse med PIO og DMA på Itanium-arkitektur.** *(Log x)* Figuren viser båndbredde ved bruk av PIO og DMA for Dolphin ICS sine SCI-adaptere på Itanium-arkitektur. Som man kan se av figuren gir PIO en betydelig høyere båndbredde enn DMA (både med 4KByte og 16KByte pager).

flere pakker mottas, gjøres interruptet etter et tidsintervall. Dette har innvirkning på gap-faktoren, altså avstanden mellom behandling av to etterfølgende transmissjoner eller pakker på mottakersiden. Som overhead blir både tiden det tar for hardware og operativsystem å behandle et interrupt utslagsgivende. Siden pakker kan forsvinne på nettverket må også protokoller (for eksempel TCP/IP[27]) for å håndtere dette prosesseres før dataene kan kopieres til maskinens internminnet.

For Myrinet blir bruk av protokoller og hvorvidt interrupter benyttes, styrt av MCP programmet. Likefullt vil overhead-parameteren også være avhengig av kommunikasjonsbibliotek og protokoller. Den mest benyttede meldingsutvekslingsprotokollen for Myrinet er Myricoms GM[104] API. GM tilbyr en sikker og sekvensiell levering av meldinger og benyttes derfor som basis for grensesnitt som VIA[139] og MPI[11, (mpich, version 1.2.4)].

Overhead for SCI-basert kommunikasjon er i utgangspunktet meget lav (ca 2-4μs uavhengig av datastørrelse). SCI-checkpoint på sendersiden tømmer stream-bufferne i adapteret og sjekker at overføringen gikk greit. Effektiv bruk av denne teknikken er beskrevet av Huse i [66]. Som en konsekvens av dette opereres det ofte med en LogGP overhead for SCI nær 0.

InfiniBand-standarden har fokusert på å forbedre flere av manglene ved eksisterende interconnect. Som eksempler kan man nevne minimal bruk av interrupter og CPU. Dette sammen med en fokus på lav kostnad og meget god evne til skalerbarhet gjør IBA til en meget spennende teknologi i clustersammenheng.

StarFabric er en ny og meget spennende teknologi. StarGen påstår å kunne skalere opptil flere hundre noder og er fullt bakoverkompatibel med PCI. Hvorvidt StarFabric vil slå igjennom gjenstår å se.

3.4 Kommunikasjonsbibliotek

Overhead (o) parameteren i LogGP modellen prøver å klassifisere nødvendige meta-operasjoner som utføres av bibliotek/mellomvare ved kommunikasjon. Slike operasjoner kan for eksempel være protokollhåndtering, datakonvertering og pakking, synkroniseringer, kopieringer og sjekksumming. Ende-til-ende-ytelse måles derfor som tiden dataene bruker fra minnet til senderprosessen til datane befinner seg i minnet til mottakerprosessen.

Generelt vil et grensesnitt eller API tilby disse servicene gjennom et sett biblioteksfunksjoner. Et ideelt API har klar semantikk, er uavhengig av plattform og presenterer et høyt nivå av abstraksjon mot brukeren.

Det har blitt laget mange mer eller mindre standardiserte bibliotek. Det er følgelig fire punkter som er viktige å merke seg ved kommunikasjonsbibliotek:

- *Abstraksjon av API mot bruker:* Abstraksjonen API-et bestemmer hvilken programmeringsmodell brukeren av kommunikasjonsbiblioteket må forholde seg til. For eksempel vil en biblioteksimplementasjon av MPI-standarden tvinge brukeren til å forholde seg til meldingsutvekslingsmodellen og tosidig kommunikasjon. Et SHMEM-bibliotek derimot lar derimot brukeren forholde seg mer til en shared memory-liknende modell og ensidig kommunikasjon.

- *Funksjonalitet:* Bibliotekets funksjonalitet i form av nivå av generell databehandling, kan ha stor påvirkning for applikasjonskoden. Hvor mye applikasjonen eksplisitt må behandle dataene før sending og mottak kan være avgjørende for størrelsen på applikasjonskoden. Slik funksjonalitet kan for eksempel være sjekksummering, implisitte variable som adresser til andre noder og feilhåndteringer.

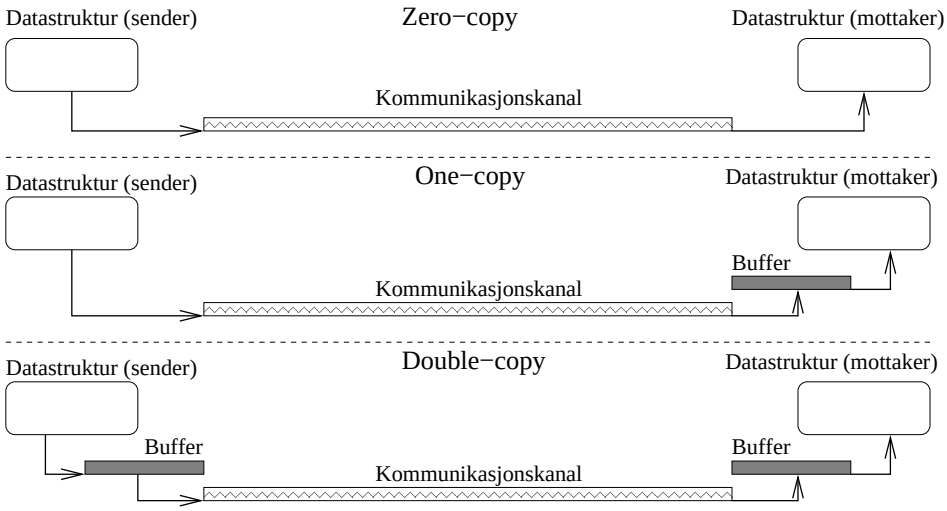
- *Overhead:* Prosesseringstiden nødvendig ved benyttelse av biblioteket har stor innvirkning på forsinkelsen, og i visse tilfeller også båndbredden. Et høyt abstraksjonsnivå rettferdiggjør ikke nødvendigvis høy overhead i biblioteket. Man ønsker derfor en minimal overhead ved bruk av biblioteket, men samtidig en høy funksjonalitet og et høyt abstraksjonsnivå.
  Tre viktige begreper i denne sammenheng er *zero-copy*, *one-copy* og *double-copy* (se avsnitt 3.4.1).

- *Underliggende programmeringsmodell og portabilitet:* Generaliteten av underliggende hardware og avhengige bibliotek har er viktig for portabiliteten av biblioteket. En applikasjon som for eksempel forholder seg til en konkret adapterdriver eller interconnect med et proprietært API, er mindre portabel. Drivere har i tillegg en begrenset funksjonalitet. Kommunikasjonsbiblioteket kan også være implementert spesielt med hensyn på en spesifikk maskin- eller prosessorarkitektur.

Jeg vil ikke prøve å forklare forskjeller, fordeler og ulemper på alle varianter av kommunikasjonsbibliotek som eksisterer i dag. Det fokuseres her på de bibliotek og standarder med størst relasjon til problemstillingen i denne rapporten.

3.4.1 Dataflyt og buffering

Bruk av buffering og en effektiv dataflyt er essensielt for alle kommunikasjonsbibliotek. Hvilken modell som velges er avhengig blant annet av ytelseskrav, datastrukturer, protokoller og hvorvidt nettverket kan regnes som en sikker kommunikasjonskanal eller ikke. Modellene kan klassifiseres etter hvor mange buffere dataene må passere gjennom (se figur 3.5):



Figur 3.5: **Minnekopierings- og kommunikasjonsmodeller.** Kopieringsmodellene deles inn etter bruk av buffere. Modellene er uavhengige av arkitektur og interconnect. Den mest effektive modellen for et system er således ikke nødvendigvis den mest effektivitet for et annet. Zero-copy modellen regnes likefullt som den modellen med generelt minst overhead.

Zero-copy protokollen kjennetegnes ved minimal kopiering av data og dermed minimal overhead. Dataene kopieres direkte fra senderprosessens datastruktur og inn i datastrukturen på mottakerprosessen. Det forutsettes at senderprosessen kjenner den virtuelle adressen hvor dataene skal legges hos mottakeren. Det meste av kommunikasjonslogikken ligger på sendersiden. Denne protokollen benyttes ofte på shared memory-system.

One-copy begrepet reflekterer bruken av et buffer hos sender eller mottaker. Som oftest ligger dette bufferet hos mottaker. Sender prosessen kopierer data fra sin datastruktur og inn i et buffer hos mottakeren. Mottaker bufferet er ofte implementert som et ringbuffer. Ettersom dataene blir overført vil mottakeren kopiere dataene over i sin datastruktur. Protokollen forutsetter at sender kjenner den virtuelle adressen og *offset* for bufferet hos mottakeren. Denne protokollen er hyppig benyttet i bibliotek for løst koblede system (eks. MPI over SCI og Myrinet).

Double-copy protokollen likner one-copy protokollen, men introduserer i tillegg et buffer på sendersiden. Senderprosessen kopierer dermed sine data først inn i sitt senderbuffer før dataene kopieres til mottakeres buffer. Man får således to ekstra datakopieringer gjennom buffere i forhold til zero-copy protokollen. Denne protokollen benyttes ofte ved bruk av større protokollstakker som for eksempel Sockets over TCP/IP, som referansemodell for optimaliseringer eller ved kopiering av kompliserte datastrukturer.

Teoretisk sett er zero-copy den ideelle modellen, men i praktiske tilfeller kan en slik modell være uhensiktsmessig eller rett og slett umulig å implementere. Grunner til dette kan være usikre forbindelser, avanserte datastrukturer eller manglende evne til å bestemme mottakers virtuelle adresse. Interconnectets egenskaper er også med på å bestemme effekten av forskjellige kopieringsmetoder; G og g parameterene er således avgjørende.

3.4.2 BSD Sockets

Berkeley Sockets API-et[134, 16] ble opprinnelig utviklet for Berkeley Unix (BSD) for kommunikasjon mellom prosesser (IPC). I dag støttes API-et av de fleste operativsystem, inkludert Linux og benyttes primært for kommunikasjon over Ethernet. Det eksisterer implementasjoner av både MPI[11] og PVM[45] over Sockets.

En Sockets-forbindelse identifiseres med en fildeskriptor hos hver av prosessene (et heltall). Etter at en slik forbindelse er etablert, benyttes vanlige systemkall som *read()* og *write()* med fildeskriptoren som argument for å sende og motta meldinger. Forbindelsesløs kommunikasjon støttes; dette medfører at en prosess må kunne ta imot data/meldinger fra andre prosesser uten at en forbindelse er opprettet. API-et er dessuten ikke-blokkerende, en funksjonalitet som vanskeliggjør en zero-copy implementasjon av Sockets-biblioteket.

I nettverk som Ethernet og ATM[85, 122] benyttes TCP/IP og UDP/IP som underliggende protokoller for Sockets-implementasjoner. Dette er tunge protokollstakker på operativsystemnivå designet for WAN og usikre nettverk. Disse fører til høy overhead, gir relativt dårlig båndbredde og høy forsinkelse. Slike protokollstakker utgjør som sagt en stor del av overhead-parameteren i LogGP modellen.

User-space Sockets-implementasjoner eksisterer både for Myrinet[101] og SCI[120, 125, 59]. Hovedproblemet med Sockets over usikre forbindelser (for eksempel Ethernet) er at funksjonaliteten må kobles tett sammen med operativsystemet. For eksempel blir protokollhåndtering og skriving til og fra adapterbuffere gjort i operativsystemkjernen. Videre må alle sender og mottakerbuffere låses i kjernen. Sockets eller TCP kommunikasjon er følgelig vanskelig å legge utelukkende i *user-space* for usikre kommunikasjonskanaler uten å introdusere en høy overhead.

Oppsummering Sockets

For kort å oppsummere kan man si at Sockets har et høyt nivå av abstraksjon samtidig som lavnivåkontroll støttes. API-et abstraherer vekk både interconnect-teknologi og underliggende plattform. Bibliotekets funksjonalitet er rik med hensyn på eventuelle underliggende protokoller og nettverksarkitektur. Sockets gir også muligheten for dynamisk opp- og nedkobling av forbindelser. Overheaden i biblioteket varierer med underliggende nettverksteknologi, men er i ofte relativt høy. I utgangspunktet fungerer Sockets mellom prosesser i både tett og løst koblede system. Dette gir implementasjoner over dette API-et et høyt nivå av portabilitet. Desverre kan likvel noe av funksjonaliteten og semantikken være vanskelig å implementere på enkelte interconnect og/eller over spesielle drivere.

3.4.3 Message Passing Interface (MPI)

MPI-standarden[100] er et grensesnitt, ikke en implementasjon. MPI-standarden definerer brukergrensesnitt og funksjonalitet for et bredt område av meldingsutvekslingsfunksjonalitet. Både syntaks og semantikk er definert. Duato [38, s.387] forteller hvordan designerene har måttet balansere spesifikasjonen mellom portabilitet, effektivitet og brukervennlighet. For eksempel er datatyper inkludert i grensesnittet og tillater støtte over heterogene miljø hvor operander har forskjellige representasjoner på forskjellige maskiner.

Standarden går ikke inn på hvordan operasjonene skal utføres, men fokuserer heller på den logiske funksjonaliteten som skal utføres. Man unngår spesifikasjoner av adresseringssystem, systemspesifikke operasjoner som I/O, oppgavehåndtering og andre operativsystem spesifikke egenskaper. Standarden unngår også eksplisitte hensyn til shared memory-operasjoner, selv om MPI kan (og har blitt[131]) implementert på slike maskiner.

Et MPI-program består av autonome prosesser som kjører sin egen kode i et MIMD-miljø. Hver prosess eksekveres vanligvis i sitt eget adresseområde og gjør mellomprosesskommunikasjon (IPC) ved hjelp av MPI-primitiver. Den grunnleggende kommunikasjonsmekanismen i MPI er punkt-til-punkt kommunikasjon mellom par av prosesser. Modellen antar bruk av en konsistent meldingsprotokoll på bruker nivå. Videre sikres rekkefølgen på meldingene mellom par av sender og mottaker.

MPI er et relativt stort API med ca 120 funksjonskall. I tillegg til de typiske punkt-til-punkt funksjonene:

- *MPI_Send(buf, count, datatype, dest, tag, comm)*
- *MPI_Recv(buf, count, datatype, source, tag, comm, status)*

definerer MPI følgende kollektive kommunikasjonsoperasjoner:

- *Broadcast:* En kildeprosess sender identiske data til alle andre prosesser i en gruppe.
- *Scatter:* En kildeprosess sender distinkte meldinger til alle andre prosesser i en gruppe.
- *Gather:* Det motsatte av Scatter.
- *All-to-all broadcast:* Hver prosess sender samme data til hver av de andre prosessene innenfor en gruppe.
- *All-to-all personalized exchange:* Hver prosess sender unike data til hver av de andre prosessene i gruppen.

Forskjellige moduser eksisterer på de fleste funksjoner for både blokkerende og ikke-blokkerende funksjonalitet. Brukeren kan således bestemme hvorvidt synkron eller asynkron meldingsutveksling skal benyttes.

MPI-versjoner

Foranledningen for utarbeidelsen av MPI var problemet med at forskjellige MPP-produsenter hver hadde sitt eget proprietære meldingsutvekslings-API. Dette gjorde det nær umulig å skrive portable programmer. Gjennom en åpen prosess ble derfor den første MPI-versjonen utviklet, og lansert i juni 1994[93]. En ny revisjon ble lansert i juni 1995 som MPI versjon 1.1[94]. I 1997 ble MPI 2.0[95] lansert sammen med en utvidelse av MPI 1.1 til MPI 1.2. MPI 1.2 inneholder klareringer og korreksjoner i forhold til MPI 1.1 standarden, mens MPI 2.0 definerte et sett utvidelser til MPI 1.x standardene. De gamle MPI 1.x standardene ble navngitt MPI-1 (refereres til som MPI her), mens MPI 2.0 ble navngitt MPI-2.

Hwang[70, s.682] lister opp de viktigste forskjellene mellom MPI og MPI-2:

- *Dynamiske prosesser:* Prosesser i et MPI-program er statiske: programmet starter eksekvering med et gitt antall prosesser, og ingen prosess kan legges til eller fjernes mens programmet kjører. MPI-2 støtter dynamiske prosesser som gir følgende fordeler:
  * MPI spesifiserer ikke hvordan prosesser opprettes eller hvordan de initierer kommunikasjon. MPI trenger derfor en underliggende plattform for slik funksjonalitet (for eksempel Unix *rsh* forbindelse i et cluster). De dynamiske prosessmekanismene i MPI-2 tilbyr denne funksjonaliteten på en portabel plattformuavhengig måte.
  * Dynamiske prosesser gir mulighet for porting av PVM[51, 135] kode til MPI. Dette er viktig

for blant annet klient-server-baserte applikasjoner.
* Dynamiske prosesser tillater mer effektiv bruk av ressurser samt lastbalansering, for eksempel kan antallet noder øke og minske ettersom dette trengs.
* Feiltoleranse kan støttes. Når en node feiler kan prosessene opprettes på en ny node og fortsette kjøringen der. En såkalt *failover* løsning.

- *Ensidig kommunikasjon:* MPI-2 inkluderer en ny punkt-til-punkt kommunikasjonsmodell basert på RMA - Remote Memory Access. Denne funksjonaliteten tillater prosesser å utføre ensidig kommunikasjon. En prosess kan sende eller hente data hos andre prosesser "uten" deres deltagelse. Dette er i kontrast til MPI hvor all punkt-til-punkt kommunikasjon et tosidig.

Den nye funksjonaliteten til MPI-2 gir muligheter for både mer dynamiske, skalerbare og kompliserte applikasjoner. Dessverre viser det seg at en full implementasjon av MPI-2 standarden er meget vanskelig, og det finnes derfor kun noen få implementasjoner[138, 99, 54]. For øyeblikket holder de fleste MPI-implementasjoner og -applikasjoner seg til MPI-1-standarden.

Oppsummering MPI

MPI har som nevnt et API som abstraherer både protokoller og underliggende interconnect- og maskinarkitektur vekk fra brukeren. Likefullt tvinger MPI-standarden brukeren til å forholde seg til en tosidig meldingsutvekslingsmodell. MPI-2 har utvidet modellen til også å omfatte ensidig kommunikasjon, men denne standarden er foreløpig lite brukt.

MPI-spesifikasjonene oppgir operasjoner i logisk forstand, noe som gjør at den overhead som produseres i biblioteket i stor grad er avhengig av den enkelte implementasjon. Samtidig gir MPI ingen føringer på maskinarkitektur. Dette gjør at MPI kan implementeres over flere underliggende maskinarkitekturer og programmeringsmodeller. Mye av selve funksjonaliteten i MPI er relativt enkel, og samtidig spredt over et stort antall biblioteksfunksjoner. Dette gir brukeren god kontroll over funksjonaliteten utført av biblioteket. Det viktigste med MPI-standarden er at den definerer en sikker forbindelse mellom prosesser.

3.4.4 Shared Memory Access Library (SHMEM)

De eksplisitte programmeringsegenskapene til MPI og PVM[51, 135] har blitt videreutviklet til Shared Memory Access Library (SHMEM)[22]. SHMEM har blitt en viktig del av Cray inc. og SGI inc. sine MPT (Message Passing Toolkit) bibliotek, men er ikke en definert standard.

SHMEM er et API for kollektive operasjoner og RMA fordelt på ca 140 Fortran og 150 C funksjoner. API-et baserer seg på bruk av ensidig kommunikasjon. Biblioteket ble i utgangspunktet laget for å gi effektiv kommunikasjon for Cray T3D, men er også implementert for Cray T3E, Cray SV1, SGI Origin og Compaq AlphaServer SC serien.

SHMEM inkluderer funksjonalitet for remote read (get) og write (put), atomiske operasjoner (for eksempel *compare-and-swap* og *fetch-and-add*), samt kollektiv funksjonalitet (for eksempel *barrier, broadcast, gather, reduce* og *reduce-all*). SHMEM benytter ikke et globalt shared memory, men refererer isteden til ikke-lokale data ved å oppgi adressen til de korrisponderende lokale dataene samt en prosess *rank*. Minne kan allokeres implisitt (statisk) ved oppstart eller eksplisitt (dynamisk) ved runtime ved hjelp av spesielle funksjonskall. Minneallokering er en kollektiv operasjon, og alle prosesser må allokere samme mengde minne symmetrisk. I utgangspunktet er alle SHMEM-datatyper 64 bit, men 32 bit data er også tilgjengelig. Til forskjell fra MPI (som gjør bruk av *MPI_Init()*), benyttes det ingen eksplisitt initiering og/eller terminering av biblioteket.

ScaShMem

Huses ScaShMem[65] er et kompatibilitetsbibliotek for SHMEM, laget over ScaMPI. ScaMPI(se avsnitt 5.1) er Scali AS sin MPI-implementasjon over SCI. Som beskrevet tidligere benyttes en tosidig kommunikasjon ved hjelp av eksplisitte *send* og *receive* kall ved bruk av meldingsutveksling og MPI. ScaShMem har som funksjon å maskere vekk denne tosidig kommunikasjonen ved bruk av ensidig RMA kommunikasjonen i SHMEM.

ScaShMem ser på operasjoner som *shmem_put()* og *shmem_get()* som meldinger inneholdende data og en peker til den assosierte operasjonen (med parametere). Disse sendes til mottakerprosessen for eksekvering. Slike meldinger blir referert til som såkalte Active Messages [140]. Alt minne i SHMEM-applikasjoner må derfor, direkte eller indirekte, være globalt synlig. En Cray T3D og T3E prosess opererer direkte på fysisk minne. SHMEM kan dermed direkte aksessere ikke-lokale prosessers minne, gjennom spesielle E-registre, i en en-til-en mapping. Disse maskinene opererer altså kun på fysisk minne. Clustere har i utgangspunktet ikke denne muligheten, da alle prosesser kjører i hvert sitt separate virtuelle adresseområde. Her må denne en-til-en mappingen mellom de virtuelle adresseområdene være implementert i programvare. En av ScaShMem sine viktigste funksjoner er således å benytte en indirekte adressering for å oppnå denne funksjonaliteten.

ScaShMem benytter en egen *server-tråd* alle requester går gjennom. Alle SHMEM-prosesser på en node kobler seg til denne tråden og gir denne tilgang til sitt minneområde. Alle SHMEM-kall i applikasjonsprosessene blir av underliggende lag oversatt til MPI-meldinger. Server-tråden ligger å venter i en *MPI_Recv()* for mottak av meldinger/requester. Server-tråden har dermed ansvaret for å motta og utføre eventuelle operasjoner/requester sendt av andre prosesser, uten å involvere selve SHMEM-prosessene på den lokale noden. ScaShMem oppnår dermed ensidig kommunikasjon over en tosidig meldingsutvekslingsmodell (MPI) ved hjelp av indirekte adressering.

En bakdel med ScaShMem er at den introduserer en viss overhead i kjøringen av sin server-tråd. ScaShMems server-tråd ligger i en polling-receive (som en følge av ScaMPI-implementasjonen) og vil også kreve CPU-tid ved behandling av requester. Resultatet er at server-tråden bør kjøre på en separat prosessor for ikke å stjele CPU-sykler fra applikasjonsprosessene.

Oppsummering SHMEM

I likhet med MPI er interconnect og kommunikasjonsprotokoller abstrahert vekk fra brukeren i SHMEM-API-et. Likefullt er SHMEM i utgangspunktet designet for høyhastighets kommunikasjon på MPP-maskiner og er basert på en ensidig kommunikasjonsmodell. Som sagt er SHMEM ingen standard og i utgangspunktet utviklet for bruk på spesielle maskinarkitekturer og en shared memory programmeringsmodell. Implementasjonen av ScaShMem viser likevel at det er mulig å implementere dette biblioteket over en tosidig kommunikasjonsmodell, da ikke uten økt overhead. Effektiviteten av implementasjoner som ScaShMem vil dermed være lavere pr. CPU enn for maskiner som Cray T3D og T3E. Dette indikerer en lav portabilitet for SHMEM og samtidig en sterk avhengighet til underliggende arkitektur. I likhet med MPI er funksjonaliteten fordelt utover et sett funksjoner og skaper en sikker kommunikasjonskanal.

3.4.5 Global Arrays (GA)

Meldingsutvekslingsmodellen benyttes ofte grunnet sin enkle portabilitet. Nieplocha et. al. [109] fremhever likevel at noen applikasjoner er for kompliserte å implementere i en slik programmeringsmodell. Spesielt når man vil balansere last og unngå redundante beregninger. Shared memory-

modellen forenkler implementasjoner, men er ikke portabel og gir sjelden kontroll over kommunikasjonskostnader mellom prosesser.

Nieplocha beskriver videre i sin artikkel en ny fremgangsmåte kalt Global Arrays (GA)[53]. GA benytter de bedre egenskapene av begge overnevnte modeller resulterende i både enkel kode og effektiv eksekvering. Nøkkel konseptet i GA-modellen er at det tilbys et portabelt grensesnitt hvor prosesser i et MIMD-parallelt program asynkront kan aksessere logiske blokker av fysisk distribuerte matriser. Dette skjer uten eksplisitt samarbeid av andre prosesser. Denne funksjonaliteten likner både programmeringsmodellen for shared memory og SHMEM. Hovedforskjellen ligger i at GA-modellen innser at ikke-lokale data har lengre aksesstid enn lokale data og tillater datalokalitet å bli eksplisitt spesifisert og benyttet. Denne funksjonaliteten likner mer en meldingsutvekslingsmodell.

GA ble utviklet ved Pacific Northwest National Laboratory (PNNL)[119] for kjemisk forskning på PNNL sitt Environmental and Molecular Science Laboratory (EMSL)[41] og er frigitt som Open Source. GA-biblioteket har blitt implementert med støtte for flere systemer, for eksempel Intel DELTA og Paragon, IBM-SP-1, Kendall Square KSR-2 og clustere av Unix/Linux arbeidsstasjoner.

Applikasjoner

Nieplocha lister opp følgende generelle applikasjonskarakterestikker som motivasjon for GA:

- Krav til oppgaveparallellisme (MIMD), muligens i tillegg til dataparallellisme.
- Aksess av relativt små blokker av matriser som er for store til å ha i minnet for en enkelt prosessor (vil kreve blokkvis fysisk distribusjon).
- Stor variasjon i oppgavenes eksekveringstid (vil kreve dynamisk last balansering).
- Ha et relativt stort forhold mellom beregning og dataflytting (gjør det mulig å ha høy effektivitet mens data aksesseres når nødvendig).

Mer generelt ønsket de å beregne elektroniske strukturer av molekyler og andre små eller krystalliske kjemiske system. Dette er beregninger brukt for å prediktere mange kjemiske egenskaper og spiller en dominerende rolle i antall CPU-sykler benyttet innenfor *computational chemistry*.

Typiske matematiske metoder som beregnes av slike applikasjoner er den iterative Self Consistent Field (SCF) metoden, annen ordens Møller-Plesset Perturbation metode og Multi-Reference Configuration Interaction (MRCI). Applikasjonene NWChem[110], GAMESS-UK[50] og Colombus[20] er eksempler på applikasjoner som gjør slike beregninger ved hjelp av GA.

Programmeringsmodell

Global Arrays programmeringsmodellen blir karakterisert av Nieplocha [109] som følger:

- MIMD-parallellisme blir tilbudt ved hjelp av flere prosesser hvor alle ikke-GA-data, fildeskriptorer og annen prosessinformasjon blir kopiert, eller en unike for hver prosess.
- Prosesser kan kommunisere med hverandre ved å opprette og aksessere GA-distribuerte matriser, og også (hvis ønskelig) gjennom konvensjonell meldingsutveksling.
- Matriser blir fysisk distribuert blokkvis enten regulært eller som det kartesiske produktet av irregulære distribusjoner for hver akse.

- Hver prosess kan uavhengig og asynkront aksessere enhver n-dimensjonal bit av en GA-distribuert matrise, uten samarbeid fra applikasjonskoden i andre prosesser.

- Flere typer aksess støttes, inkludert *get*, *put*, *accumulate(acc)* og *get and increment*.

- Hver prosess antas å ha rask aksess til en del av en distribuert matrise og lengre aksess til den gjenværende delen. Hastighetsforskjellen definere dataene som *local*(lokale) eller *remote*(ikke-lokale). Tidsforskjellen mellom *local* og *remote* er uspesifisert.

- Hver prosess kan bestemme hvilken del av den distribuerte matrisen som er lagret lokalt *(locally)*. Hvert element i en distribuert matrise er garantert å være lokal for en og bare en prosess.

GA krever eksplisitte bibliotekskall for å aksessere data, men unngår overhead fra operativsystemet assosiert med minnekoherens og virtuelle *page faults*. Implementasjonen garanterer likevel at alle nødvendige data for en array-del kan overføres samtidig. GA-biblioteket har grensesnitt for både C og Fortran-77 applikasjoner.

Operasjoner

Følgende primitive GA-operasjoner kjøres synkront på alle prosesser:

- Opprette en array, kontroll av *alignment* og distribusjon.

- Opprette en array basert på en gitt mal (eksisterende array).

- Slette en array.

- Synkronisere alle prosesser.

Følgene primitive operasjoner kan utføres i sann MIMD-stil uten betydning for og/eller synkronisering med andre prosesser. Operasjonene har ingen garanti for atomisitet. Applikasjonsprogrammereren må således benytte GA-låser for å sikre atomisitet ved operasjoner.

- *Fetch*, *store* og atomisk *accumulate* til en regulær del av en to-dimensjonal array.

- *Gather* og *scatter* array-elementer.

- Atomisk read *and increment* av et array-element.

- Spørring av lokasjon og distribusjon av dataene.

- Direkte aksess til lokale elementer av en distribuert array for å støtte og/eller forbedre ytelsen av applikasjonsspesifikke dataparallelle operasjoner.

I tillegg finnes vektor operasjoner som dot-produkt, og matrise multiplikasjon med fler.

3.5 Oppsummering

LogGP modellen kan være nyttig for klart definert uttrykk som forsinkelse, båndbredde og overhead. Jeg har i denne presentasjonen forsøkt å peke på hvilke parametere som gjør seg gjeldene for GbE, Myrinet og SCI. Et viktig poeng å understreke er at kommunikasjonsbibliotek og mellomvare står ofte for det største bidraget til reduksjon av båndbredde, økt forsinkelse eller økt CPU-bruk ved kommunikasjon, med andre ord overhead.

Som en konklusjon kan man si at ved et fastsatt interconnect er det kun denne overheaden som kan reduseres. Ved endring av interconnect må man passe på at overheaden i eventuelle nye (eller gamle) kommunikasjonsbibliotek ikke fører til mer overhead i forhold til interconnectets ytelse. Ved implementering av et raskere interconnect skal den totale kommunikasjonsytelsen (applikasjonsbuffer til applikasjonsbuffer) øke tilsvarende. Samtidig må man passe på at grensesnitt, funksjonalitet og portabilitet passer med applikasjonen og interconnectet som benyttes.

Det finnes i dag en rekke tilgjengelige kommunikasjonsbibliotek. Enkelte bibliotek og grensesnitt er generelle i sin funksjonalitet, er portable og kan benyttes på flere nettverksarkitekturer. Ved bruk av slike bibliotek innenfor HPC-segmentet setter man likefullt andre og tøffere krav til ytelse enn for systemer til bruk innenfor for eksempel internet- og/eller databaseløsninger. Bibliotek benyttet i clustere har en større fokus på minimal overhead og maksimal utnyttelse av nettverksarkitekturen. Dette kan komme i konflikt med den programmeringsmodell som er mest hensiktsmessig for en aktuell applikasjon. Valg av interconnect, grensesnitt og biblioteksimplementasjon må således sees i forhold til den aktuelle applikasjonen som skal utvikles/kjøres og kan være en komplisert avgjørelse.

Kapittel 4

Global Arrays over SCI

Selv om Global Arrays (GA) har primitiver som get, put og accumulate, implementerer ikke GA denne funksjonaliteten direkte. Av portabilitets- og vedlikeholdshensyn har biblioteket blitt splittet i GA og det selvstendige biblioteket ARMCI (se avsnitt 4.1). De overnevnte kommunikasjonsprimitivene i GA mappes nærmest direkte til ARMCI-kall. På denne måten har man effektivt lagt denne funksjonaliteten i et eget bibliotek.

I utgangspunktet ble ARMCI implementert for SMP-maskiner og clustere ved hjelp av shared memory og bruk av TCP/IP. Implementasjonen over TCP/IP er basert på Sockets-grensesnittet. Nieplocha et. al.[105] påpeker senere at dette ikke var tilfredsstillende for flere applikasjoner. Ytelsen over usikre forbindelser som Ethernet og ATM ble for dårlig og gav dårlig skalering for enkelte applikasjoner. Utvidelser av ARMCI til å støtte kommunikasjon over blant annet Myrinet[105] ble derfor utviklet. ARMCI ble således et portabelt frittstående bibliotek for ensidig kommunikasjon. Per i dag støtter derfor ARMCI interconnect som Ethernet gjennom Sockets-protokollen, QsNet gjennom Elan/SHMEM, Myrinet gjennom GM og Giganet cLAN gjennom VIA[15] grensesnittet.

ARMCI har i tillegg til GA blitt benyttet som basis for en SHMEM-implementasjon (GPSHMEM [108, avsnitt 5]) og runtimebiblioteket Adlib[19]. Nieplocha fremhever ARMCI sin evne til å erstatte både funksjonalitet og ytelse i grensesnitt som SHMEM og LAPI[71].

I dette kapittelet beskrives ARMCI-bibliotekets semantikk, funksjonalitet og design. Til slutt beskrives en sammenlikning av ARMCI over eksisterende TCP/IP-baserte gigabitløsninger.

4.1 ARMCI

ARMCI (Aggregate Remote Memory Copy Interface)[106, 107, 105, 12][1] ble utviklet for å støtte RMA-operasjoner i sammenheng med distribuerte array-bibliotek (da først og fremst GA) og kompilator runtime-system (for eksempel Adlib). Til forskjell fra bibliotek som GA, MPI og SHMEM er ARMCI i utgangspunktet ikke beregnet for direkte bruk i applikasjoner, men som et verktøy for biblioteksutviklere.

ARMCI-grensesnittet er portabelt og MPI-kompatibelt. På enkelte plattformer kan det også benyttes sammen med meldingsutvekslingsbibliotek for PVM og TCGMSG grensesnittet. Det er viktig

[1] Det eksisterer ikke noe dokument som fullt beskriver design- og Implementasjonsdetaljer for ARMCI. Full forståelse av biblioteket (utenfor det av en normal bruker) mener jeg derfor at kun oppnås ved gjennomgang av alle de relaterte artikler, samt kildekoden. Beskrivelsen av ARMCI-systemet er basert på flere av artiklene. Enkeltreferanser i dette avsnitten er derfor benyttet i mindre utstrekning.

å merke seg at ARMCI trenger et meldingsutvekslingsbibliotek for oppstart og initiering. ARMCI er tilgjengelig på både clustere, shared memory og MPP-systemer.

ARMCI likner på mange måter SHMEM-grensesnittet for ensidig kommunikasjon. Forskjellen ligger i at ARMCI fokuserer på overføring av ikke-sekvensielle (*strided*) datastrukturer. Slike strukturer er mye benyttet i vitenskapelige applikasjoner (for eksempel deler av flerdimensjonale arrayer). Effektiv kommunikasjon av slike overføringer er mulig på grunn av ARMCI sitt grensesnitt for beskrivelse av datamønstere.

4.1.1 Grensesnitt og bruk

ARMCI-biblioteket støtter tre hovedklasser av operasjoner:

- Dataoperasjoner som *put*, *get* og *accumulate* (ikke atomisk).
- Synkroniseringsoperasjoner som lokal og global *fence*, atomisk *read-modify-write* samt ikke-lokale mutex- og låsoperasjoner.
- Verktøysoperasjoner for allokering og frigjøring av minne, samt feilhåndtering.

Det kan kun kommuniseres via datastrukturer allokert via minneallokeringsrutinen *ARMCI_Malloc()*, som likner *MPI_Win_malloc()* i MPI-2. På shared memory-systemer mappes minneallokeringen til delte variable. På denne måten oppnås full minnebåndbredde og minimal forsinkelse ved ensidige kommunikasjonsoperasjoner. Tabell 4.1 lister de viktigste ARMCI-operasjonene med en kort beskrivelse.

4.1.2 Ensidig kommunikasjon

ARMCI-semantikken er noe forskjellig fra den ensidige “aktive” kommunikasjonsmodellen i MPI-2 i at den erklærer alle sine operasjoner til å være fullstendig *unilaterale*. Dette vil si at operasjonen gjennomføres uavhengig av operasjoner gjennomført av den ikke-lokale prosessen. Spesielt er polling eller eksplisitte bibliotekskall i applikasjonens målprosess i utgangspunktet unødvendig for gjennomføring av kommunikasjonen. ARMCI-operasjonene er altså ekte ensidig kommunikasjon sett fra applikasjonens (eller programmeringsmodellens) nivå.

Operasjonene har garantert rekkefølge ved at de gjennomføres i den rekkefølgen de ble kalt, så lenge det refereres til samme målprosess. Operasjoner sendt til forskjellige prosesser utføres i en udefinert rekkefølge. Nieplocha fremhever at denne semantikken forenkler implementasjonen av enkelte applikasjoner hvor rekkefølge er nødvendig.

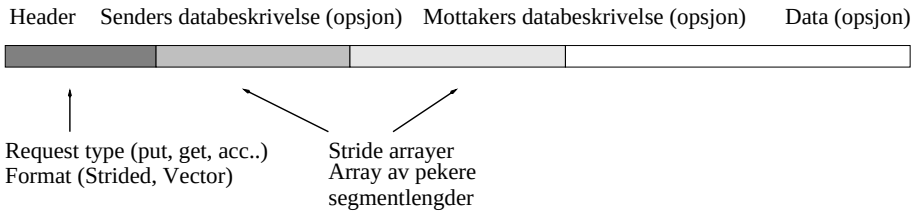
Atomisitet og datakonsistens

Get-operasjoner er typiske blokkerende kall og er garantert gjennomført ved godkjent returstatus. Put- og accumulate-operasjoner derimot ikke garantert å være gjennomført, eller kommet frem til målprosessen før etter en *fence*-operasjon. Operasjonene returnerer hos sender etter en lokal ferdigstilling av operasjonen. *Fence*-operasjonene returnerer kun når alle put-/accumulate-operasjoner er gjennomført. Operasjonen garanterer dermed global (mellom to eller flere prosesser) ferdigstilling.

Atomisiteten for accumulate gjelder for datatypene *integer*, *float*, *double* og *singel*- og *double-precision complex*. ARMCI-spesifikasjonen sier ingen ting om hvilken prosess som skal utføre operasjonen. Dette åpner for implementasjon gjennom eventuelle operasjoner i interconnectets eller arkitekturens hardware. For *read-modify-write* er atomisitet spesifisert kun for datatypene *integer* eller

Operasjon	Beskrivelse
ARMCI_Put, ARMCI_PutV, ARMCI_PutS	kontinuerlig, vektor og ikke-kontinuerlig(strided) versjoner av put
ARMCI_Get, ARMCI_GetV, ARMCI_GetS	kontinuerlig, vektor og ikke-kontinuerlig(strided) versjoner av get
ARMCI_Acc, ARMCI_AccV, ARMCI_AccS	kontinuerlig, vektor og ikke-kontinuerlig(strided) versjoner av atomisk reduksjon
ARMCI_Fence	blokkerer til utestående operasjoner på en spesifisert prosess er gjennomført. Selv om en prosess returnerer fra sitt ARMCI_Put eller ARMCI_Acc kall er man ikke garantert at oppgaven er utført for den fjerntliggende prosessen.
ARMCI_AllFence	blokkerer til utestående operasjoner på alle prosesser er gjennomført (for eksempel put-requester på flere noder).
ARMCI_Rmw	atomisk read-modify-write
ARMCI_Malloc	minne allokator, returnerer array av adresser for minnet allokert av alle prosesser. Operasjonen må kjøres synkront og på alle prosesser.
ARMCI_Free	frigir minne allokert av ARMCI_Malloc
ARMCI_Lock, ARMCI_Unlock	mutex operasjoner

Tabell 4.1: **De viktigste ARMCI-operasjonene.** Tabellen viser ARMCI-grensesnittets viktigste operasjoner. Legg merke til at det finnes tre forskjellige former for put-, accumulate- og get-operasjonene.



Figur 4.1: **ARMCI-request-format.** En request melding fra klient til server initierer alle operasjoner. Etter request-meldingen følger rådataene som en kontinuerlig datastrøm eller som en sammenhengende melding. Request-meldingene er minimalt på 28 byte, men kan være større avhengig av operasjon.

long. Det garanteres en serialisering av eksterne requester. Låser og mutexer må benyttes for å sikre konsistens i datamodellen da atomisitet ikke garanteres for sekvenser av datatypene. Disse låsene må eksplisitt settes opp av applikasjonen. Låsene kan settes på lokale såvel som ikke-lokale data.

Buffering og kopieringsmodell

Ettersom ARMCI-grensesnittet gir en detaljert beskrivelse av dataene som blir sendt (ved hjelp av flere versjoner av put, get og accumulate), er grunnlaget i utgangspunktet lagt for en zero-copy implementasjon. Nieplocha viser til at enkelte API-implementasjoner (for eksempel LAPI[71]) sender slike data som flere meldinger, med dertil økt overhead fremfor en stor melding. Hvorvidt zero-, singel- eller double-copy benyttes for kommunikasjon, avhenger av implementasjonen over den enkelte maskin- og/eller interconnect-arkitekturen.

ARMCI-implementasjonen benytter i utgangspunktet egne send og receive buffere (double-copy). Ved kommunikasjon på shared memory-systemer benyttes zero-copy modellen. Under kommunikasjon av sekvensielle datastrukturer går dataene uten modifikasjon gjennom henholdsvis send- og receive-bufferne på vei inn og ut av datastrukturene på de forskjellige nodene. For kommunikasjon av ikke-sekvensielle data pakkes disse sammen i sendbufferet for å skape en stor melding fremfor flere små. På denne måten utnyttes båndbredden og overhead minimaliseres. Strukturen på de pakkede dataene er beskrevet i ARMCI-request formatet vist i figur 4.1 . Mottakende prosess pakker ut dataene basert på beskrivelsen fra sitt receive buffer og inn i datastrukturen. For implementasjonene over Myrinet (GM) og Sockets benyttes henholdvis one- og zero-copy[2] av dataene ved større meldinger. Myrinet benytter såkalt *pipelined DMA* mens Sockets benytter en *write()* for hver sekvensielle del av datastrukturen.

4.1.3 Dataformater

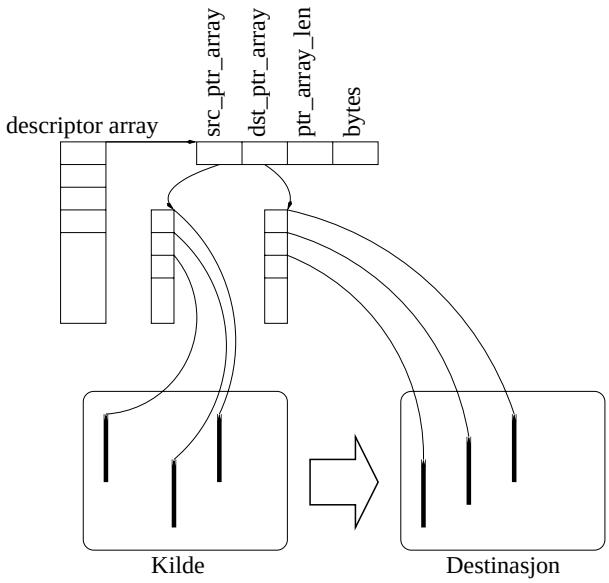
ARMCI støtter to ikke-kontinuerlige dataformater. Sekvensielle data sendes som et spesialtilfelle av disse:

Generalized I/O vector: Dette er det mest generelle formatet for beskrivelse av flere sett av datasegmenter av samme størrelse. Formatet utvider formatet benyttet i Unix *readv-/writev*-operasjoner ved å minimalisere lagringskravene hvor flere datasegmenter har samme størrelse. Figur 4.2 viser strukturen som beskriver formatet. Figur 4.3 viser hvordan dataflyten er definert. Dette

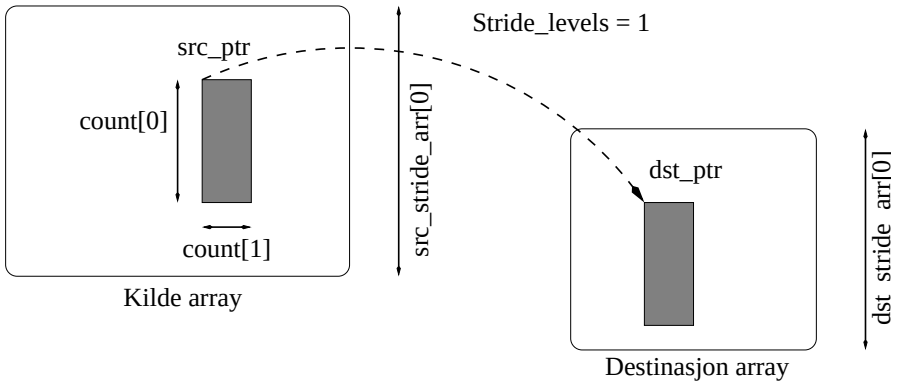
[2]Dette er sett ut fra Sockets API-et. Merk at Sockets-implementasjonene benytter intern buffering. Dette kan ikke overstyres av brukerapplikasjonen.

```
/* Generalized I/O format */

typedef struct {
void **src_ptr_array;
void **dst_ptr_array;
int    ptr_array_len;
int    bytes;
} armci_giov_t

/* Strided format */

void *src_ptr;
void *dst_ptr;
int stride_levels;
int src_stride_arr[stride_levels];
int dst_stride_arr[stride_levels];
int count[stride_levels+1];
```

Figur 4.2: **Beskrivelse for requester med Generalized I/O og Strided formatene.** Det videre formatet for en request bestemmes på bakgrunn av requestens header. Merk at kilde- og mottakerformatet på dataene kan være forskjellig. For strided (ikke-sekvensielle) formater følger det med informasjon for oversetting av en sekvensiell datastrøm til en ikke-sekvensiell datastrøm og omvendt.



Figur 4.3: **Generalized I/O vector formatet.** Figuren viser en skisse av formatet for Generalized I/O definert i figur 4.2.



Figur 4.4: **Strided formatet.** Figuren viser en skisse av Strided formatet definert i figur 4.2.

formatet benyttes ved kall til *ARMCI_PutV/GetV/AccV()*.

Strided: Dette formatet er en optimalisering av Generalized I/O vector formatet for flerdimensjonale arrayer. I steden for å inkludere adresser for alle segmenter spesifiseres kun adressen til det første segmentet i settet. De andre segmentene blir utledet fra den såkalte *stride*-informasjonen vist i figur 4.2. Figur 4.4 viser dataformatet i en Fortran datastruktur.

4.1.4 ARMCI på clustere

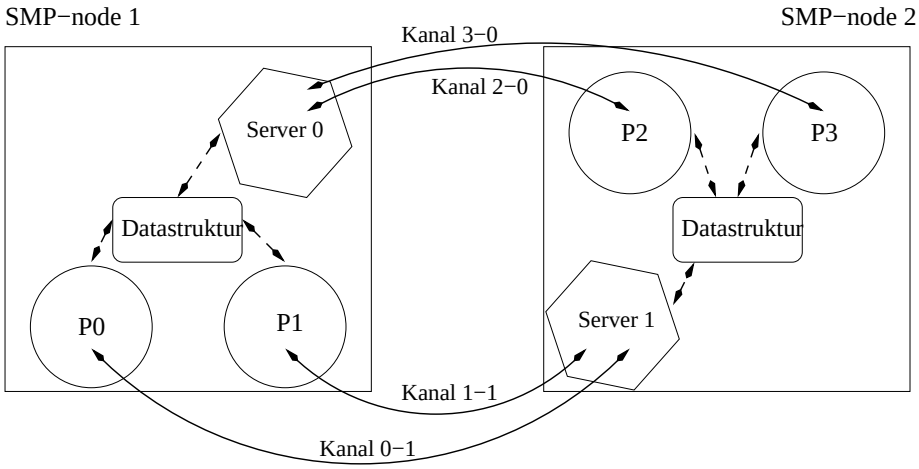
ARMCI-spesifikasjonen hverken beskriver eller antar noen spesiell implementasjonsmodell (for eksempel bruk av tråder). Det blir derimot oppfordret til å utnytte de mest effektive mekanismer som er tilgjengelig på den gitte plattform. Dette kan inkludere Active Messages[91], innebygde put/get, shared memory og/eller tråder.

Klient-server arkitektur

Som kjent har hver enkelt cluster-node separate adresserom. For å støtte et fullt sett av RMA-operasjoner på clustere benytter derfor eksisterende ARMCI-implementasjoner en strategi basert på en klient-server arkitektur. Dette implementeres ved å starte en eller flere server-tråd(er) på hver node. Disse er dedikert til behandling av operasjoner fra prosesser på andre noder. Hvis implementasjonen av nettverksprotokollene ikke er *thread-safe*, kan en dedikert prosess benyttes. Ved bruk av ARMCI på Linux-system benyttes tråder (eller såkalte *lightweight prosesses*). Server-tråden eksekverer operasjoner basert på requester den mottar, og om nødvendig (for eksempel ved en get-operasjon) sender data tilbake til sender prosessen. Klientprosessene er selve applikasjonsprosessene som utfører beregninger og kall til ARMCI for put, get, accumulate og synkroniseringskall.

Det optimale antallet av server-tråder avhenger av flere faktorer; antallet prosessorer og oppgaver/prosesser per node, nettverksytelse, kommunikasjonslast og kommunikasjonsmønster i applikasjonen. Av ytelsesgrunner benyttes det på dagens interconnect og hardware, med et lavt antall prosessorer pr. SMP-node (vanligvis 2), en enkelt server-tråd.

Figur 4.5 viser kommunikasjonskanalene for fire applikasjonsprosesser og to server-tråder på to SMP-noder. Det er viktig å merke seg at applikasjonsprosessene ikke kommuniserer med andre applikasjonsprosesser direkte eller sin lokale server-tråd. Server-trådene kommuniserer heller aldri med hverandre.



Figur 4.5: **Kommunikasjonskanaler i ARMCI.** Figuren skisserer ARMCI sine kommunikasjonskanaler i et program kjørt over to noder og med fire prosesser. Hver node er en 2-CPU SMP-node. Server-tråden, P0 og P1 (P2 og P3) kommuniserer gjennom sin felles datastruktur ved bruk av shared memory. Requester sendes kun fra P*i*-prosessene til server-tråden på den andre noden. Server-tråden svarer ved eventuelle get- eller synkroniseringsoperasjoner. Server-trådene på de forskjellige nodene kommuniserer ikke med hverandre.

Ressursbruk

ARMCI fokuserer på å la server-trådene utgjøre minimal ressursbruk både når det kommer til minne, nettverksbåndbredde og CPU-bruk, til fordel for applikasjonsprosessene. Med andre ord kan server-trådene defineres som en del av overheaden (o i LogGP).

For å hindre trådene i å benytte ressurser når de ikke har requester å prosessere fokuserer Nieplocha på bruk av blokkerende (sovende) *wait* fremfor aktiv polling av interconnectets adapter/grensesnittet. ARMCI-implementasjoner over blant annet VIA og Sockets benytter blokkerende kommunikasjonskall som effektivt lar tråden vente på en request før den våkner (interrupt). For eksempel benyttes Sockets-kallet *select()* for implementasjonen over TCP/IP. Nieplocha understreker den resulterende interrupt-håndteringen og dermed økningen i forsinkelsen ved bruk av denne løsningen, men rettferdiggjør dette ved mindre CPU-bruk.

4.2 Den perfekte løsning?

SCI-arkitekturen gir i utgangspunktet mulighet for et shared memory eller DSM mellom noder. Det er lett og se at et slikt system er meget nærliggende det ARMCI faktisk prøver å emulere/tilby som programmeringsmodell. Jeg kan derfor enkelt tenke meg en løsning hvor man ved load/store instruksjoner fra enkelte virtuelle adresser, utfører load/store i andre noder sitt ved bruk av SCI, såkalt direkte aksess. SCI-operasjonene read-and-increment kan benyttes for accumulate-operasjoner. En slik løsning ville ha både minimal overhead og høy effektivitet. Bruk av en eller flere server-tråder ville være unødvendig og implementasjonen ville i prinsippet være meget enkelt; rett og slett kun en mapping mellom ARMCI-funksjoner og SCI-minnekopieringsfunksjoner. Alle prosesser ville dermed ha direkte tilgang til de andre prosessenes minne, både lokale (innenfor samme SMP-node) og ikke-lokale.

Huse[65] vurderte en liknende fremgangsmåte for sin ScaShMem. Desverre finnes det en del

praktiske begrensninger ved en slik løsning. Huse påpeker at kommunikasjons adaptere tilkoblet I/O-busser vanligvis ikke støtter virtuelt minne. Det er nemlig få arkitekturer som tilbyr en Memory Management Unit (MMU) for oversetting mellom fysiske og virtuelle adresser mot I/O-bussen. Som et resultat må I/O-adaptere alltid operere direkte på fysiske adresser.

Bruk av direkte aksess i GA-sammenheng ville derfor involvere å *pinne* ned hele applikasjonens minne/datastruktur og eksportere en komplett mapping til alle andre prosesser. Som en følge av dette kan man tenke seg alvorlige skaleringsbegrensninger på 32 bit system ettersom kun 4GByte kan adresseres. I tillegg vil dette være meget ressurskrevende ettersom ingen prosesser hvor minnet er pinnet kan swappes ut av operativsystemet. Som et resultat likner Huses løsning av ScaShMem ARMCI ved bruk av en dedikert server-tråd.

Dynamisk pinning

En liknende, men alternativ, løsning til å pinne ned hele applikasjonsminnet ved oppstart, er dynamisk pinning. Dynamisk pinning går ut på å dynamisk mappe de delene av datastrukturen man ønsker å lese/skrive kun i den korte perioden operasjonen pågår. På denne måten kommer man rundt adresseringsbegrensningen og prosesser som for øyeblikket ikke har pinnet deler av sitt minne kan swappes ut. En server-tråd er nødvendig for administrasjon av en slik løsning. Likefullt vil denne løsningen, på samme måte som løsningen skissert over kunne basere seg på en direkte aksessmodell med liten overhead i selve overføringen.

Tanken om en liknende løsning basert på Myrinets DMA-maskiner er skissert av Nieplochas utvidelse av ARMCI over Myrinet[105]. Nieplocha gikk tidlig vekk fra denne løsningen da han oppdaget at dynamisk pinning av minne (spesielt ved bruk av Linux) er en meget dyr operasjon. Fordelen med bruk av dynamisk pinning fremfor eksplisitt bruk av kommunikasjonsbuffere ble spist opp av overheaden involvert i den dynamiske pinningen. Nieplocha gikk dermed over til bruk av eksplisitt pinnede buffere for sender og mottaker ved blokkerende transmissjoner.

4.2.1 Oppsummering

Som en følge av blant annet begrensningene ved bruk av direkte aksess og SCI beskrevet i 4.2, ble løsningen beskrevet i kapittel 4.3, 5 og 6 hovedsaklig basert på ARMCI sitt eksisterende design over TCP/IP (Sockets).

For kort å oppsummere ble følgende design lagt til grunn:

- ScaIP, ScaMPI eller ScaFun som underliggende bibliotek/mellomvare.
- En server-tråd pr. node (to applikasjonsprosesser på en SMP-node deler en server-tråd). Shared memory benyttes internt i noden.
- Eksplisitte kommunikasjonsbuffere hos både sender og mottaker (double-copy).
- Eksplisitt pakking av ikke-kontinuerlige data før sending (med påfølgende utpakking på mottakersiden).

Eventuelle optimaliseringer eller endringer i dette grunnleggende designet er beskrevet for den enkelte implementasjon.

4.3 ARMCI over GbE og ScaIP

Det første jeg ønsket å se på var ytelse over eksisterende teknologier og bibliotek. Den mest nærliggende fremgangsmåten var derfor å se på muligheten for å benytte den allerede eksisterende Sockets-implementasjonen (TCP/IP) av ARMCI over SCI. Nieplocha[105] viser til liknende prosjekter og tester gjort for ARMCI over Myrinet via Sockets-grensesnittet.

For SCI har det blitt gjort flere prosjekter med det mål for øyet å kunne utnytte SCI med allerede eksisterende TCP/IP-baserte implementasjoner. Jeg kan her nevne prosjekter som SciFS og SciOS[77] og Hellwagners Sockets-bibliotek for SCI[59] basert på SISCI-driveren.

Scali AS utviklet i denne sammenheng en full implementasjon av IP kalt ScaIP for SCI. Scali AS har optimaliser implementasjonen for sine cluster-konfigurasjoner og for å effektivt kunne samarbeide med andre bibliotek som benytter seg av SCI-nettet; for eksempel ScaMPI. Dette åpner for muligheten til å kunne kjøre den Sockets-baserte MPICH eller ARMCI over SCI uten modifikasjoner av kildekoden.

Sockets-implementasjonen er designet primært for å fungere på en plattform bestående av flere homogene noder koblet sammen til et cluster. ARMCI benytter designet med en server-tråd (som beskrevet i 4.1.4). Sockets-forbindelsene initieres ved oppstart og settes opp alle-til-alle mellom server-tråder og klientprosesser (det settes ikke opp forbindelse til lokal server-tråd).

4.3.1 ScaIP - IP over SCI

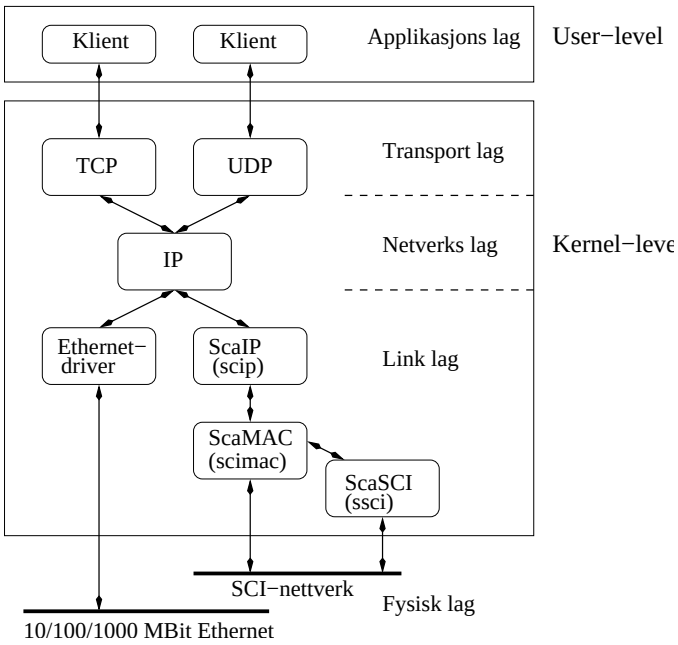
ScaIP tilbyr båndbredde i størrelsesorden 118MB/s for TCP og 116MB/s for UDP (testet med Netperf[76] på x86-arkitektur). Likevel bærer ScaIPs ytelse preg av at dette er et relativt tungt bibliotekslag introdusert mellom applikasjonen og SCI-adapteret. Forsinkelsen for mindre pakker er i størrelsesorden det dobbelte av den for GbE. Dette blir likevel borte ved større pakker ettersom SCI har en høyere båndbredde og man kan operere med høyere MTU-størrelser. I utgangspunktet har ScaIP en MTU (Maximum Transmission Unit) størrelsen på 16K (kan justeres).

Som det fremgår av figur 4.6 har ScaIP følgende laginndeling/moduler:

- *ScaIP:* Implementerer et drivergrensesnitt for Solaris/Linux kjernen. Modulen lar kjernen henvende seg til SCI-adapteren som et Ethernet device.
- *ScaMAC:* Selve kjernen i ScaIP. Modulen benytter ScaSCI for opprettelse av SCI-forbindelsene. ScaMAC gjør initielle kall til ScaSCI, før den så tar over all kommunikasjon over SCI-devicene gjennom minnekopieringrutiner. Denne modulen introduserer et ekstra lag i forhold til standard Ethernet kommunikasjon.
- *ScaSCI:* Selve SCI-driveren. Denne modulen er kun aktiv ved initialisering og avslutning av ScaIP kommunikasjonen. ScaSCI setter opp minneområder (SCI-fellesminne) for alle nodene og klargjør nodene for SCI-kommunikasjon. ScaSCI er nærmere beskrevet i 6.1.

Disse tre modulene er alle såkalte *kernel-modules*, eller kjerne moduler for Solaris/Linux. Det betyr at ScaIP kjører i *kernel-space* og at disse modulene dermed er tett integrert med operativsystemets kjerne. Modulene har en rekke parametere for optimalisering og tilpasning til brukerenes eventuelle individuelle behov. På denne måten kan ScaIP konfigureres for den enkelte applikasjonens måte å kommunisere på (for eksempel små eller store pakker).

Når alle ScaIP sine moduler er lastet, vil grensesnittet mot IP-kommunikasjon være det samme som for Ethernet. Forskjellen ligger i at ekte Ethernet devicer vil hete “*eth<n>*”, hvor n er device



Figur 4.6: **ScaIP lagdeling.** Figuren viser hvordan ScaIP-implementasjonen passer inn i operativsystemets protokollstakk og behandling av TCP/IP. ScaIP står i praksis for den samme oppgaven som Ethernet-driveren, nemlig å tilby et Ethernet grensesnitt til operativsystemet. ScaMAC emulerer et Ethernet-adapter og kommuniserer over SCI ved hjelp av ScaSCI og minnekopieringsrutiner.

nummer (begynner på 0), og ScaIP devicer vil hete "*scip<n>*". Valg av device gjøres via standard oppsett av IP-segment og ruting.

4.3.2 ARMCI-Sockets over gigabit-forbindelser

Sockets-implementasjonen av ARMCI er i utgangspunktet optimalisert for 10/100 Mbit/s Ethernet. En slik forutsetning gjør at kommunikasjonsmønsteret (pakkestørrelser, forventet båndbredde og antagelser om forsinkelse) ikke er optimal i forhold til hverken ScaIP eller GbE. Jeg forventet derfor en høyere båndbredde, men også høyere forsinkelse ved bruk av ScaIP fremfor Fast Ethernet.

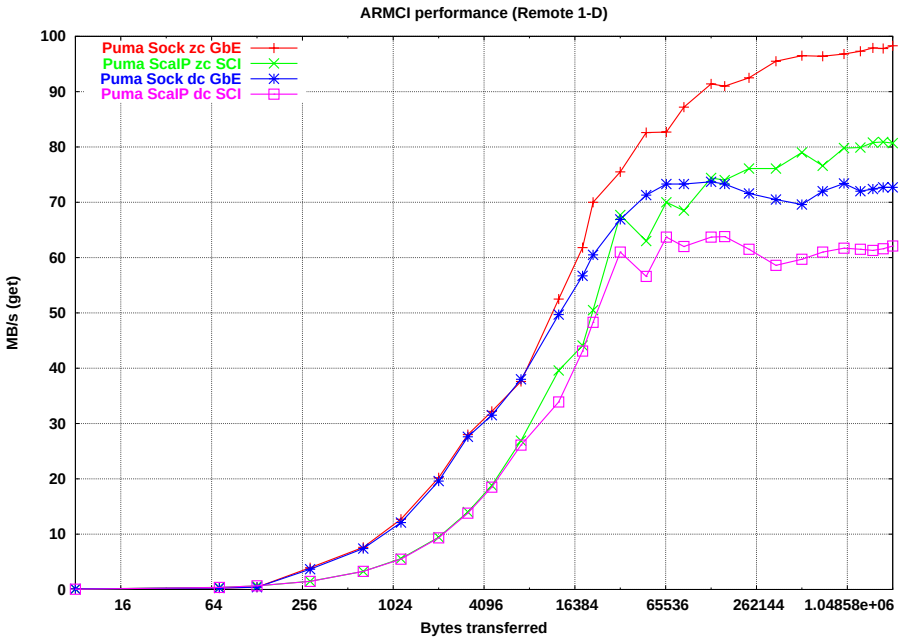
Et noe mer åpent spørsmål var ytelse sammenliknet med GbE. Et problem i så måte er at selv om GbE introduserer høyere båndbredde, introduseres også en høyere forsinkelse sammenliknet med Fast Ethernet.

I utgangspunktet ønsket jeg å se hvordan ARMCI over Sockets kunne benyttes over GbE og ScaIP og finne eventuelle forskjeller i ytelse. I tillegg til testing med standardkonfigurasjon, kjørte jeg tester med endringer i to parametere:

1. MTU (Maximum Transfer Unit) størrelse i ScaIP.

2. Bruk av zero-copy eller double-copy for både ScaIP og GbE.

Jeg kom til at en direkte sammenlikning av ytelse for samme MTU-størrelse for ScaIP og GbE er lite hensiktsmessig av flere grunner; Bruk av midlertidig buffering har både design- og størrelsesforskjeller i kjernemodulen for GbE og ScaIP (ScaMAC). Dessuten er MTU-størrelsene for for de to



Figur 4.7: **1-D båndbredde (get) GbE og ScaIP.** *(Log x)* Figuren viser båndbredde ved bruk av ARMCI over Sockets for sekvensielle data (get-operasjon). Ved sammenlikning av GbE og ScaIP ser man tydelig at ScaIP har en dårligere båndbredde enn ved bruk av GbE. Som forventet har zero-copy implementasjonene den beste ytelsen.

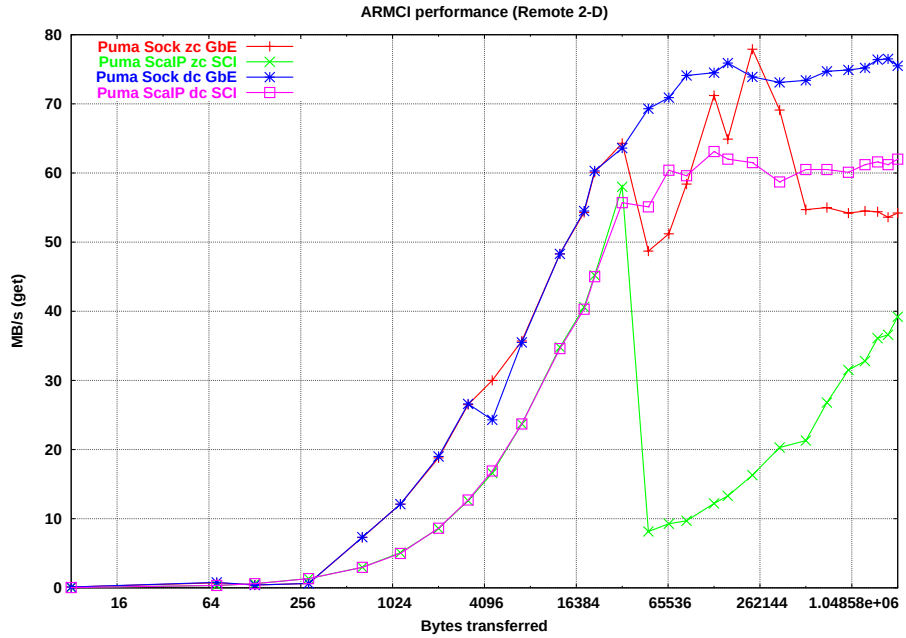
modulene i utgangspunktet veldig forskjellig (1500 byte for GbE og 16KByte for ScaIP). Ytelsestester med TCP-benchmarken Netperf har allerede vist at ScaIP har høyere båndbredde enn GbE, likefullt er ikke dette nødvendigvis tilfellet for get-requester i ARMCI. Testene (testbeskrivelser i avsnitt 7.4) ble kjørt på Puma-clusteret (se avsnitt 7.1.1) uten bruk av jumbo frames eller svitsj for GbE. Den mest effektive MTU størrelse for både GbE og ScaIP kan ikke generelt fastsettes. Jeg valgte derfor å benytte standard størrelser både for GbE og ScaIP.

Båndbredde

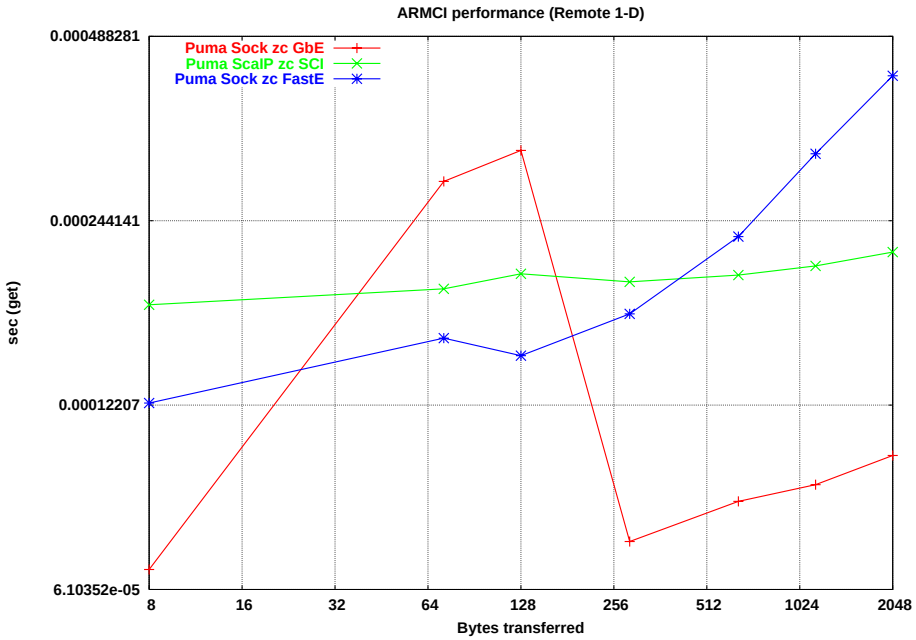
Figur 4.7 (resultater fra ARMCI_perf se avsnitt 7.4) viser ytelsen for ScaIP og GbE for sekvensielle ARMCI-get-requester. Her ser man en stødig og jevn båndbreddeøkning hvor ytelsen er høyest ved bruk av GbE og ARMCI sin zero-copy implementasjon. ScaIP yter også bra, men har ca 10-20 MByte/s dårligere båndbredde enn samme ARMCI-implementasjon over GbE. En annen observasjon verdt å merke seg er at zero-copy implementasjonen av ARMCI yter bedre på både GbE og ScaIP enn double-copy implementasjonen.

Figur 4.8 viser derimot hvordan ARMCI-Sockets kan være uheldig for kommunikasjon av ikke-sekvensielle data over gigabit-forbindelser. GbE har fortsatt den høyeste ytelsen, men nå har zero-copy implementasjonen en dårligere ytelse enn double-copy implementasjonen. Dette fremkommer enda sterkere for ScaIP. Årsaken til denne effekten har bakgrunn i det man kan se på som g parameteren i LogGP modellen. Parameteren defineres som tiden fra en melding kan sendes/mottas til neste kan sendes/mottas.

ARMCI-Sockets sin zero-copy implementasjon benytter som nevnt flere mindre *write()* kall mot



Figur 4.8: **2-D båndbredde (get) GbE og ScaIP.** *(Log x)* Figuren viser båndbredde ved bruk av ARMCI over Sockets for ikke-sekvensielle data (get-operasjon). Som for sekvensielle data, yter GbE best også her. Legg likevel merke til hvordan zero-copy implementasjonen faktisk har negativ effekt på både GbE kjøringen og ScaIP kjøringen.



Figur 4.9: **Forsinkelse (get) GbE og ScaIP.** *(Log y)* Figuren viser forsinkelse ved bruk av ARMCI over Sockets for sekvensielle data (get-operasjon). Som det kommer frem av figuren yter zero-copy implementasjonene best for 8 byte data. Likevel kan man av figuren se hvordan forsinkelsen kan variere sterkt for GbE på grunn av buffering og/eller interrupter. GbE mangedobler sin forsinkelse ved 128 byte sammenliknet med 256 byte. Den konkrete årsaken tid denne effekten er ukjent, men har trolig sammenheng med chipset og GbE adaptere (se figur 7.4 på side 89 og tabell 7.1 på side 82).

Socket-en for å unngå eksplisitt buffering. Dette har en uheldig effekt på både GbE og ScaIP ettersom disse implementasjonene/interconnectene i større grad fokuserer på båndbredde for større meldinger. Både for ScaIP og GbE er altså en stor *write()* bedre enn flere små da båndbredden i interconnectet kan utnyttes bedre. Selv om man ved bruk av double-copy gjør intern kopiering i minnet (i server-tråden) er slik kopiering i størrelsesorden 18 ganger raskere enn både GbE og ScaIPs båndbredde. Med andre ord blir overheaden (o) ved å gjøre en minnekopiering liten sammenliknet med tiden det tar å sende dataene over interconnectet. Ved bruk av eksplisitt buffering vil man følgelig alltid skrive sekvensielle data til interconnectet og også få en bedre utnyttelse av CPU-cache sammenliknet med ikke-sekvensielle kopieringer.

Forsinkelse

Figur 4.9 viser forsinkelsen ($2 * (o + L)$) ved get-requester for mindre meldinger (applikasjonsbuffer til applikasjonsbuffer). Som forventet har ScaIP en høyere forsinkelse enn både GbE og Fast Ethernet. En interessant ting å merke seg er at GbE faktisk har den laveste forsinkelse på requester mellom 8 og 16 byte. For request-meldinger > 32 byte økes forsinkelsen betraktelig før den igjen synker ved 256 byte requester. Denne oppførselen er noe vanskelig å forklare, men har trolig sammenheng med bufferstørrelser og GbE sin bruk av forsinket interrupt (pakkesamling) i adapteret. Som forventet har ScaIP klart høyest forsinkelse ved 8-16 byte requester.

4.4 Oppsummering

ARMCI er et et selvstendig bibliotek som implementerer ensidig kommunikasjon i clustere. Biblioteket har et begrenset antall funksjoner og benytter et enkelt, men kraftig design. Selv om jeg kan, ved hjelp av ScaIP, benytte ARMCI over SCI utnytter ikke et slikt mellomvarelag SCI til sitt fulle potensiale. Som man kan se av testene yter faktisk GbE bedre enn ScaIP både når det kommer til båndbredde og forsinkelse. Som en følge av dette valgte jeg å fokusere på GbE fremfor ScaIP ved senere ytelsestester.

Resultatene understreker nødvendigheten av mer spesialdesignede implementasjoner av ARMCI for SCI. Undersøkelsene viser at jeg både må se på mer effektiv kommunikasjon av dataene, men de setter også spørsmålstegn ved effektiviteten av zero-copy implementasjoner over interconnect med høy (Gbps) båndbredde.

Kapittel 5

ARMCI over ScaMPI

Som nevnt er ARMCI avhengig av et meldingsutvekslingsbibliotek for initiering og avslutning samt enkelte operasjoner (eks. fence). Man antar derfor at alle programmer/bibliotek benytter PVM, TCGMSG eller MPI i tillegg til ARMCI. Applikasjonene er dermed basert på en hybrid programmeringsmodell som blander ensidig og tosidig kommunikasjon.

Ved kjøring av slike applikasjoner (over for eksempel et nytt interconnect) må man ha muligheten til å kunne benytte både meldingsutvekslingsmodellen og ARMCI over det nye nettverket. De fleste levrandører tilbyr implementasjoner av MPI for sine system da dette er et standardisert API. ARMCI derimot er ikke standardisert, og man er således avhengig av å enten utføre en utvidelse av ARMCI selv, eller kun benytte interconnect allerede støttet av ARMCI. I tillegg må man sikre at ARMCI-implementasjonen og en eventuell ny MPI-implementasjon er kompatible. I verste fall kan man se seg nødt til å benytte to forskjellige interconnect-arkitekturer for en og samme applikasjon. Dette er tilfellet for bruk av ARMCI-baserte applikasjoner for SCI-clustere i dag; MPI-kommunikasjonen benytter SCI mens ARMCI-kommunikasjonen benytter Fast/Gigabit Ethernet.

Slike hensyn skaper problemer ved bruk av nye interconnect. I dette tilfellet har jeg fokusert på SCI, men problematikken gjelder også for kommende interconnect som InfiniBand og StarFabric. Et mål er derfor å gjøre ARMCI og dermed alle applikasjoner/bibliotek som benytter ARMCI så portable som mulig. Jeg gjorde derfor et forsøk på å la ARMCI utelukkende benytte MPI. En slik implementasjon vil således gjøre alle eksisterende ARMCI- og GA-applikasjoner portable til ethvert cluster med et implementert MPI-bibliotek uten modifikasjoner av kildekoden. Jeg vil således kunne optimalisere ARMCI mot MPI, slippe kompatibilitetproblemer og utnytte nye interconnect.

I denne oppgaven har jeg fokusert på SCI og valgt å benytte ScaMPI fra Scali AS som underliggende MPI-implementasjon. Målsettingen for ARMCI-utvidelsen var likevel å kunne kjøre over enhver MPI-implementasjon og ikke på noen måte være ScaMPI-spesifikk. Et viktig krav var å introdusere en minimal overhead selv om et ekstra biblioteklag introduseres.

5.1 ScaMPI

ScaMPI[67] er Scali AS sin implementasjon av MPI-standarden over SCI. ScaMPI er basert på ScaFun (se avsnitt 6.1) over de PCI-baserte SCI-adapterene fra Dolphin. Sammenlignet med andre kjente MPI-implementasjoner (eks. MPICH over Ethernet) har ScaMPI en formidabel ytelse, mye på grunn av SCI-nettverket. Med en effektiv båndbredde på over 320MByte/s for Intel Itanium2-maskiner[3, 83] og 160MByte/s for x86-baserte arkitekturer. Prosess-til-prosess forsinkelse ($o + L$ i LogGP) er anslagsvis mindre enn 4 μs.

ScaMPI benytter Sockets for oppstart og initiering av MPI-prosessene på de enkelte nodene. Videre går man over til enkle minnekopieringsoperasjoner mot SCI-adapteren ved sending og mottak av MPI-meldinger.

De viktigste fordelene med ScaMPI kan oppsummeres som følger:

- Skalerbarhet: System fra en til hundrevis av noder støttes.
- Lav forsinkelse: I et kommunikasjonssystem basert på høyhastighetslinker med meget lav forsinkelse (som SCI), vil programvaren utgjøre den største delen av forsinkelsen ved en applikasjon-til-applikasjon kommunikasjon. ScaMPI øker denne forsinkelsen med mindre enn $4\mu s_{total} - 1.46\mu s_{SCI} = 2.54\mu s_{ScaMPI}$.
- Høy båndbredde: ScaMPIs båndbredde er veldig nære det teoretisk maksimale for PCI-bussen. Båndbredden er konstant og reduseres ikke ved økt systemstørrelse.
- Thread-safe: MPI-applikasjoner som er såkalt flertrådet kan komme i situasjoner hvor flere tråder forsøker å gjøre MPI-kall uavhengig av hverandre. ScaMPI støtter slike parallellisering av applikasjonskoden mens effektiv ressurshåntering fortsatt sikres.
- Feil tolerant: SCI-kommunikasjon kan forstyrres ved CRC feil eller frakobling av kabler. ScaMPI håndterer dette og tillater applikasjonen å fortsette sin prosessering uten avbrudd. Clusterets rekonfigurering er fullstendig gjennomsiktig for applikasjonen.
- Automatisk valg av medium: ScaMPI vil velge den beste fysiske kommunikasjonsmetoden for intern kommunikasjon i SMP-noder (shared memory benyttes) mot mellomnodekommunikasjon (hvor SCI benyttes).
- MIMD støtte: ScaMPI støtter ekte MIMD (Multiple Instruction - Multiple Data) eksekvering ved at flere MPI-programmer kan startes samtidig, hvor disse utgjør hele MPI-applikasjonen.

ScaMPI benyttes en såkalt *Remote-Write-Local-Read* politikk (Omang [111]). Dette gjøres da en *remote-read* ikke er like rask som en *remote-write* (se også avsnitt 6.1.2). Det benyttes en one-copy protokoll basert på PIO.

5.2 ARMCI over MPI

For implementasjonen av ARMCI over MPI tok jeg utgangspunkt i den eksisterende Sockets-implementasjonene. Utvidelsen av ARMCI ble implementert som en enkel *compile-time plug-in*-modul som aktiveres istedenfor Sockets/Myrinet eller andre ARMCI-utvidelser.

5.2.1 Requester og rekkefølge

Jeg benyttet i utgangspunktet en enkel fremgangsmåte ved å erstatte alle *read()* og *write()* kall med tilsvarende *MPI_Send()* og *MPI_Recv()* (vist i figur 5.1). Et problem i så måte var at Sockets-implementasjonen benyttet forskjellige kanaler (Sockets) for de forskjellige prosessene, og således unngikk problemer med at requester og etterfølgende data ble blandet. Hver Socket er videre en strøm slik at rekkefølge på dataene garanteres. Den opprinnelige Sockets-implementasjonen sjekker således alle de forskjellige Socket-ene etter “round-robin” prinsippet og behandler requester etter tur.

```
/* SOCKET version */
int armci_send_req_msg(int proc, void *buf, int bytes)
{
int cluster = armci_clus_id(proc);
    if(armci_WriteToSocket(SRV_sock[cluster], buf, bytes) <0) return 1;
    else return 0;
}

/* ------------------------------------------------------------- */

/* MPI version */
int armci_send_req_msg(int proc, void *buf, int bytes) {

  request_header_t* tmp = buf;
  int status;
  /* Setting message tag
     (request tag in header should never be SERVER_REQUEST_TAG) */
  do { tmp->tag = ++sm_current_tag; }
  while(tmp->tag == SERVER_REQUEST_TAG);

  /* Sending request header and descriptor/bytes(all) by MPI */
  /* All initial requests have tag=SERVER_REQUEST_TAG*/
  status = MPI_Send(buf, bytes, MPI_CHAR, proc,
    SERVER_REQUEST_TAG, sm_armci_world);
}
```

Figur 5.1: **MPI-utvidelse basert på Sockets.** I MPI-implementasjonen benyttes *MPI_Tags* for å skille request-meldinger og meldinger inneholdende rådata. Arrayet *SRV_sock[]* inneholder en mapping mellom rank-nummer og hvilken Socket som skal skrives til. Funksjonen *armci_WriteToSocket()* gjør en enkel *write()* mot Socket-en den får som argument og behandler eventuelle unntakstilfeller (detekterer om Socket-en er nede).

ARMCI har lagt opp en strategi hvor det for put- og accumulate-requester gjøres med to *write()* kall fra senderprosessen. Først sendes selve requesten, deretter dataene tilhørende operasjonen. Dette gjøres for å kunne optimalisere eventuelle vektor eller flerdimensjonale formater. For get-requester sendes kun en request-melding som beskriver operasjonen og hva man ønsker returnert.

I MPI-implementasjonen benyttet jeg *MPI_Tags* for å opprettholde samme funksjonalitet. Hver request fikk tildelt en definert *request-tag* (tag 0) for å skille request-pakker fra datapakker. Datapakkene ble tildelt en periodisk tag fra $1 - 2^{32}$. MPI-standarden garanterer rekkefølge på pakker mellom to prosesser. Sjekking av etterfølgende datapakker fra samme prosess var derfor unødvendig, så snart jeg hadde etabelert hvilken prosess request-pakken kom fra. Serveren lytter altså etter requester for å bestemme operasjon, format på etterfølgende data og kilde til requesten.

5.2.2 Buffering og pakking

Sockets-implementasjonen benytter pakking og buffering av mindre datastørrelser (double-copy). For større datastørrelser (større enn 32768 byte), benyttes en zero-copy modell.

MPI-implementasjonen benytter utelukkende pakking og buffering (double-copy) av data uavhengig av datastørrelse. Det er tre hovedgrunner til at jeg i utgangspunktet gikk bort fra bruk av en zero-copy modell:

1. Det ødelegger den enkle plug-in arkitekturen av MPI-modulen for ARMCI da flere høynivåfunksjoner må modifiseres.

2. De mest optimaliserte funksjonene i MPI-implementasjoner er vanligvis *MPI_Send()* og *MPI_Recv()* på byte nivå. Implementasjon av en zero-copy modell krever bygging av MPI-datastrukturer som *MPI_Type_hindexed* eller tilsvarende ved hver ikke-sekvensiell request. Dette fører til overhead i både indeksering og bygging av beskrivelses-arrayer, samt behandling av MPI-implementasjonen både ved sending og mottak.

3. Mangel på kompatibilitet i intern representasjon av ARMCI-datatyper og MPI-datatyper.

Jeg bestemte meg derfor for å, i utgangspunktet, benytte en double-copy modell som ved hjelp av ARMCI-funksjoner pakker ikke-sekvensielle data. Dataene pakkes i sendbuffere og sendes som rene byte sekvenser før tilsvarende inverse funksjon pakker dataene ut av mottakerbufferet.

MPI-datatyper og zero-copy

Generelt sett er flytting av data dyrt og skaper overhead. En ideell løsning vil således kopiere dataene direkte fra ARMCI-datastukturen og ut på nettverket og omvendt.

I praktisk sammenheng er derimot MPI-implementasjonens behandling av data et viktig poeng. Et MPI-bibliotek vil muligens kun ha optimalisert sending og mottak av sekvensielle data, mens ikke-sekvensielle data pakkes og legges i buffere. Dette er for eksempel tilfellet for ScaMPI. Jeg er således ikke garantert høyere ytelse ved bruk av MPI-datatyper for ikke-sekvensielle data. I slike tilfeller kan applikasjonen selv (i dette tilfellet ARMCI) kunne gjøre en bedre jobb i å pakke dataene inn og ut av send- og receive-buffere. Et annet poeng i så måte er at intern kopiering til og fra buffere har en båndbredde i størrelsesorden 4-5 ganger SCI-linkhastighet. Overheaden ved denne kopieringen blir derfor liten totalt sett.

For MPI-implementasjoner som har god støtte for kommunikasjon av ikke-sekvensielle data bør man derimot se på bruken av MPI-datatyper for ikke-sekvensielle data. I utgangspunktet kan MPI-datatyper som *MPI_Type_hindexed* optimaliseres mot zero-copy protokollen.

```
/* strided put */
extern int ARMCI_PutS(
   void *src_ptr,        /* pointer to 1st segment at source*/
   int src_stride_arr[], /* array of strides at source */
   void* dst_ptr,        /* pointer to 1st segment at destination*/
   int dst_stride_arr[], /* array of strides at destination */
   int count[],          /* number of units at each */
                         /* stride level count[0]=bytes */
   int stride_levels,    /* number of stride levels */
   int proc              /* remote process(or) ID */
   );
```

Figur 5.2: **ARMCI-put-grensesnittet.** Som vist i figur 4.2 på side 49, og som man også kan se av denne figuren, kan sender og mottaker ha en vidt forskjellig oppfatning av datamønsteret. Dette er et problem i forhold til MPI-standarden, da den i utgangspunktet forventer samme datatype for både sender og mottaker.

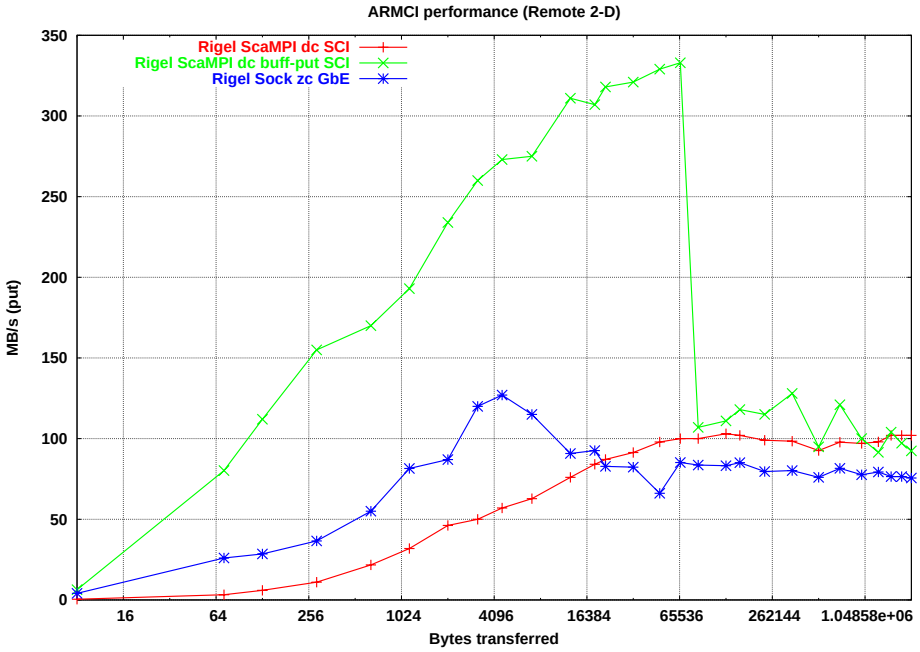
Kompleksiteten ligger i at ARMCI-grensesnittet oppfordrer til bruk av forskjellige beskrivelser av datamønsteret for sender og mottaker. Som vist på figur 5.2 består for eksempel put-requesten av en beskrivelse for mottaker og en for sender. Disse beskrivelsene kan følgelig være forskjellige. Dette er et problem i forhold til MPI-standarden. MPI forventer nemlig samme datatype både for sender og mottaker. Bruk av forskjellige datatyper for sender og mottaker for samme melding er således udefinerte og avhengig av MPI-implementasjonen. Jeg kan derfor risikere at bruk av MPI-datatyper for ikke-sekvensielle data kun fungerer på enkelte MPI-implementasjoner og således bryter med intensjonen om portabilitet.

En annen mulighet er å benytte samme prinsipp som er implementert over Sockets. En zero-copy implementasjon kan implementeres ved at hvert sekvensielle segment av et ikke-sekvensielt datasett sendes som en MPI-melding. Med andre ord flere mindre meldinger. Dette vil skape både større overhead og et problem i forhold til konvertering på mottakersiden. Sockets-implementasjonen unngår dette ettersom dataene der sendes og mottas som en stream. Ved bruk av MPI-meldinger vil man ikke kunne oppnå den samme effekten uten buffering. Dette betyr at de sekvensiell delene fra sender kan måtte bli konvertert til flere/færre sekvensielle deler hos mottaker. Dette gir igjen en overhead gjennom typekonvertering. Man er rett og slett ikke garantert høyere kommunikasjonsytelse med disse løsningene.

5.2.3 Buffering av put- og accumulate-requester

Som nevnt er get-operasjonen i ARMCI blokkerende, mens operasjoner som put og accumulate er ikke-blokkerende. I så måte er det derfor ønskelig å kunne returnere fra slike kall så fort som mulig, for å la klientprosesser utføre videre beregninger. Man er logisk sett ikke er avhengig av data man sender til andre prosesser. Ettersom jeg således kan forvente sending av et større antall put- og/eller accumulate-requester på rad bør jeg ta hensyn til dette. For Sockets-implementasjonen blir slike serier effektivt håndtert av Sockets-bufferne i kjernen, som fra applikasjonens synspunkt vil gjøre dette i bakgrunnen.

MPI har ingen slik tilsvarende funksjonalitet da implementasjonen utelukkende kjører i user-space. ScaMPI har mulighet for justering av diverse buffer hvor meldinger som venter på å bli sendt kan legges, men administrasjonen og bruken av disse kan ikke kontrolleres via MPI-kall fra ARMCI. Man kan således risikere å måtte vente unødvendig lenge på et put-/accumulate-kall fordi MPI-



Figur 5.3: **Oppsamling av put-requester.** *(Log x)* Oppsamling av put-requester gir en bedre utnyttelse av interconnectets båndbredde og introduserer mindre overhead. Som man kan se av kurven (merket *buff-put*) gir pakkeoppsamling en meget høy båndbredde sett fra klientens synspunkt. Forsinkelsen som oppfattes fra sendersiden er kun den tiden det tar å kopiere dataene over i et lokalt midlertidig buffer. Når bufferet fylles opp eller en synkronisering utføres tømmes bufferet ved sending av en stor melding til mottakeren fremfor flere små. Likefullt krever disse bufferene en lite skalerbar mengde minne (avhenger av nodeantallet).

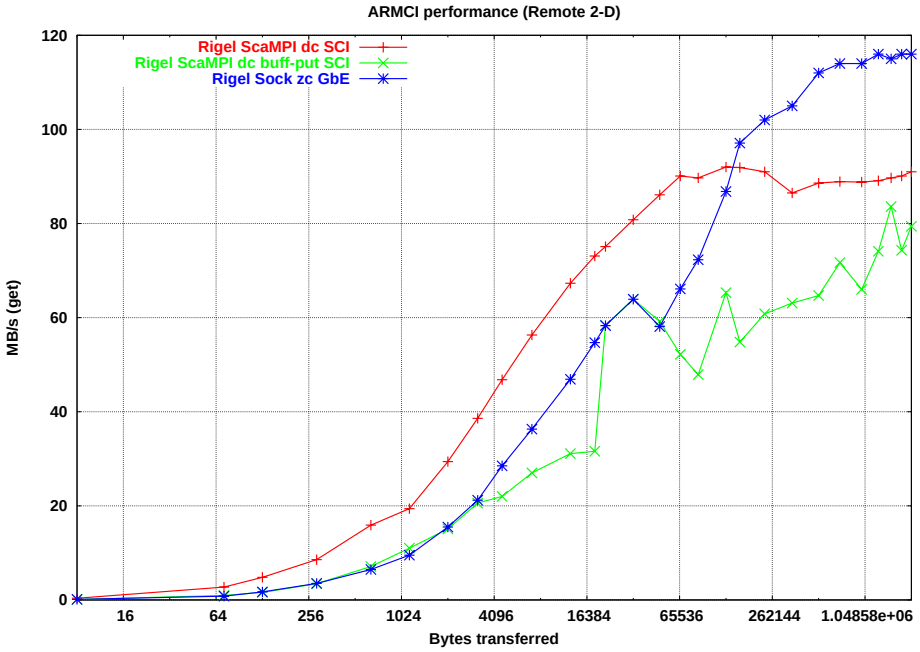
implementasjonen ikke får sendt avgårde meldingen (for eksempel fordi mottaker tråden er opptatt med en annen request).

Jeg utviklet derfor derfor en egen versjon av ARMCI over MPI som implementerte oppsamling av mindre pakker for senere sending i et eget buffer. Tanken bak denne ideen var å unytte interconnect med høy båndbredde og således kunne returnere fortere fra ikke-blokkerende kall. Den resulterende forsinkelsen sett fra applikasjonsprosessen blir derfor kun den for bygging av requesten og en minnekopiering av eventuelle data inn i et samlebuffer.

Implementasjonen overholder ARMCI-semantikk i henhold til request-rekkefølge, men sender flere requester som en stor melding fremfor flere små. For å opprettholde konsistens i datastrukturer og sikre fremdrift ble bufferet tømt ved ARMCI sine get-, *fence*- og *lock*-operasjoner. Som et resultat forventet jeg en noe større forsinkelse ved operasjoner som fører til tømming av bufferet. Figur[1] 5.3 viser forskjellen i applikasjonens oppfattelse av båndbredde når man sammenlikner oppsamling av put-requester og direkte send av slike requester.

Bruk av denne pakkesamlingsfunksjonaliteten vil kreve noe mer minnebruk av ARMCI. Et moment i så måte er å finne den "ideelle" størrelsen på bufferet basert på størrelsen og antallet meldinger en vil samle opp. Større put-/accumulate-requester vil resultere i en tømming av bufferet og deretter sendes direkte. Hvorvidt en slik algoritme og implementasjon er effektiv avhenger helt og fullt av ap-

[1] Testene er her kjørt på Rigel-clustere da Puma-clusteret ble utilgjengelig av eksterne årsaker. Relevansen av testen er fortsatt tilstedet, da det er forholdet mellom kjøringene som er viktig.



Figur 5.4: **Oppsamling av put-requester og effekten på get-requester.** *(Log x)* Oppsamlingen av put-requester kan føre til en økt forsinkelse, større overhead og dermed lavere båndbredde for get-requester (kurven merket *buff-put*). Både forsinkelsen og båndbredden for get-requestene blir nå avhengig av antallet utestående put-requester. Dette fordi (semantiske regler i ARMCI) alle operasjoner i ARMCI må komme frem til serveren i den rekkefølgen de ble sendt. En serie med put-requester og deretter en enkelt get-request fører dermed til at alle put-requestene må utføres før get-requesten kan utføres.

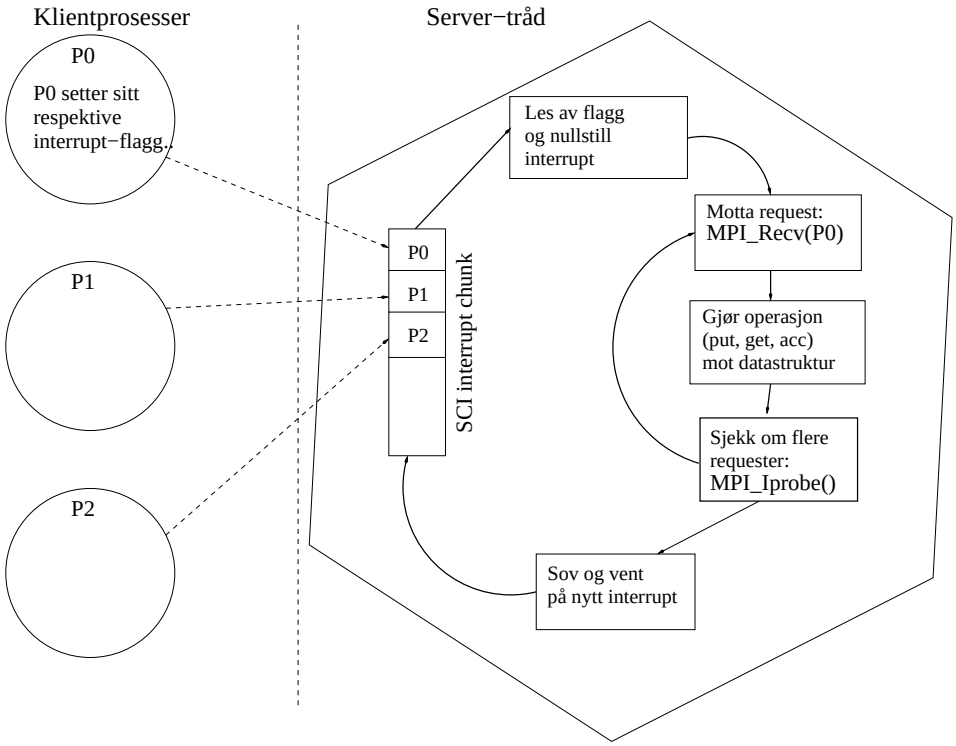
plikasjonens kommunikasjonsmønster. Denne funksjonaliteten kun vil være utslagsgivende i spesielle applikasjoner med en stor andel put- fremfor get-requester og kjører løst synkronisert. Ettersom jeg ikke har funnet en slik applikasjon og (som man kan se av figur 5.4) dette fører til lavere båndbredde for get-requester, har valgte jeg ikke å forfølge denne modellen videre.

5.2.4 SCI-interrupter

Nieplocha legger i sine artikler mye vekt på at server-tråden/-prosessen i ARMCI ikke skal polle eller periodisk sjekke etter innkommende requester. Han tegner dermed opp et design med bruk av interrupt-basert mottak. MPI-spesifikasjonen sier ingenting om bruk av slike interrupter, men lar slik funksjonalitet være opp til selve implementasjonen. Fra MPI-nivå har man altså ingen kontroll over bruk av interrupter, eller mangel på sådan.

ARMCI har således satt opp et rammeverk som krever et mer lavnivå API enn MPI, for eksempel driverfunksjonalitet. Implementasjoner på drivernivå krever betydelig mer utvikling, blant annet eget design av kommunikasjonsprotokoller (se avsnitt 6).

For å sikre at server-tråden ikke benytter unødvendig mye ressurser i perioder med lite eller ingen trafikk, har jeg derfor også implementert en interrupt-basert versjon av ARMCI over MPI. Jeg har for denne prototypen fokusert på SCI, men likefullt lagt vekt på å la interrupt-funksjonaliteten være uavhengig av selve MPI-kommunikasjonen. Jeg ser således for meg en ARMCI-implementasjon



Figur 5.5: **SCI-interrupter i ARMCI.** Figuren viser hvordan hver enkelt ekstern klientprosess får tildelt et eget (32 bit) minneområde hvor interrupter genereres ved skriving til dette. Basert på hvilket område (hvilken variabel) det skrives til kan server-tråden detektere hvilken klientprosess som er i ferd med å sende en melding. Server-tråden nullstiller så interruptflagget og tar i mot meldingen fra klienten. Server-tråden sjekker så hvorvidt det finnes andre utestående requester med *MPI_Iprobe()*. Hvis slik er tilfellet behandles disse før tråden igjen legger seg til å sove i påvente av nye interrupter.

hvor all kommunikasjon benytter MPI for sending og mottak av requester og data, mens interrupt-funksjonaliteten er avhengig underliggende interconnect-arkitektur. Valg av interrupt-funksjonalitet kan således velges *compile-time* eller run-time. Selv om en slik løsning til en viss grad strider med ideen om portabilitet, vil kun implementasjon av interrupt-funksjonalitet være en betydelig mindre jobb enn en full implementasjon av ARMCI over samme driver.

Hovedstrukturen av den interrupt-baserte versjonen av ARMCI over MPI er vist i figur 5.5 . Server-tråden benytter *MPI_Iprobe()* for først å sjekke hvorvidt den har utestående requester. Hvis så ikke er tilfellet benyttes ScaSCI-driverkallet *SciWaitForInterrupt()* for å få server-tråden til å sove og våkne på et eventuelt interrupt. Et SCI-minneområde ble avsatt med word for hver prosess (basert på *MPI_Comm_size()*). Klientprosessene generer et interrupt hos server-tråden ved å skrive til sin respektive adresse i dette minneområdet. Server-tråden kan således bestemme hvilken prosess som genererte interruptet og kalle en tilsvarende *MPI_Recv()*.

Desverre oppsto det enkelte problemer med driveren ved bruk av denne funksjonaliteten. Nærmere beskrivelse av ScaSCI-driveren og problemene rundt denne er lagt til avsnitt 6.2.3.

5.3 ScaMPI: fordeler og ulemper

Selv om MPI-grensesnittet er standardisert, gir standarden store rom for forskjeller i selve implementasjonen. En optimalisering av ARMCI rettet mot en enkelt implementasjon (for eksempel ScaMPI) vil således kunne føre til betydelige ytelsesendringer for samme applikasjon over en annen MPI-implementasjon. Jeg her her valgt å beskrive to hovedpunkter ved ved ScaMPI (i tillegg til bruk av SCI) avgjørende for MPI-utvidelsen av ARMCI.

5.3.1 Thread-safeness

Server-tråden i ARMCI kjører uavhengig av klientprosessene på samme node. Både klientprosesser og server-tråden vil med relativt stor sannsynlighet utføre MPI-kall samtidig. Et slikt design og kjøremønster krever at MPI-implementasjonen er thread-safe[2]. En slik funksjonalitet er anbefalt i MPI-spesifikasjonen. I motsetning til ScaMPI har likevel ikke alle implementasjoner denne funksjonaliteten (for eksempel SCI-MPICH og MPICH).

5.3.2 Polling-receive

ScaMPI benytter en polling-implementasjon av *MPI_Recv()*. En slik implementasjon fører til meget lave forsinkelser ved MPI-kommunikasjon. Likevel fordrer dette en godt synkronisert applikasjon hvor senderprosessen helst allerede har postet en *MPI_Send()* før mottakerprosessen kaller *MPI_Recv()*. Nieplocha[105] peker på hvordan et slikt design passer dårlig med ensidig kommunikasjon ettersom send- og receive-kall ikke er synkronisert.

I tilfellet med ARMCI fører dette til at server-tråden vil “stjele” CPU-sykler fra applikasjonsprosessen(e) i perioder hvor det er lite eller ingen trafikk. Dette kan i føre til lavere ytelse for applikasjoner hvor andelen CPU-tid i forhold til kommunikasjonstid er høyt. Man kan således se seg nødt til å reservere en egen CPU i SMP-maskiner til bruk for server-tråden.

ScaMPI har mulighet for å benytte seg av såkalt *eksponential backoff*. Denne funksjonaliteten går ut på å senke polling-frekvensen mer og mer desto lenger man venter på en melding. Problemet med å benytte dette er at man får samme effekten på alle kall til *MPI_Recv()*, ikke bare for kallet som venter på requester. Forsøk gjort med denne opsjonen har vist seg å senke den totale ytelsen for hele applikasjonen. Ideelt sett kunne jeg tenke meg en MPI-implementasjon som etter en tid gikk over til en interrupt-receive etter en tids polling. Det er således et definisjonsspørsmål hvorvidt ARMCI eller MPI-biblioteket er ansvarlig for en interrupt-funksjonalitet i server-tråden.

Selv om Nieplocha har fremhevet bruk av interrupter som fordelaktig er det ikke til å se bort fra at et slikt design skaper stor overhead. Interrupter er dyre å håndtere og skaper høyere forsinkelser for mindre meldinger. I den sammenheng har jeg sett nærmere på bruk av Hyper-Threading i Intel Pentium 4 sammen med ARMCI sin trådstruktur og polling-receive. Resultater fra disse testene er presentert i avsnitt 7.4.

5.4 Oppsummering

Det har her blitt benyttet ScaMPI for implementasjon og testing. Desverre har jeg ikke hatt tilgang på andre MPI-implementasjoner å kjøre over. Implementasjoner som MPICH og SCI-MPICH faller på thread-safe-funksjonalitet og er således uegnet.

[2] Også for andre applikasjoner, for eksempel applikasjoner som benytter en blanding av OpenMP[113] og/eller Pthreads[79] sammen med MPI, er slik funksjonalitet nødvendig.

ARMCI over ScaMPI har vist seg å fungere utmerket både på ytelsestester og verifikasjonstester. ScaMPI benytter en polling-receive som jeg forsøkte å eliminere ved bruk av SCI-interrupter. Dessverre har enkelte mangler i ScaSCI-driveren begrenset muligheten for å teste denne implementasjonen i større skala (se avsnitt 6.2.3). Det understrekes at hvorvidt interrupt-funksjonaliteten er et ARMCI-problem eller et MPI-problem er et definisjonsspørsmål.

Bruk av pakkesamling har kun vist seg å gi gode resultater for ytelsestester og har gitt liten effekt for applikasjoners totale kjøretid.

Det er viktig å påpeke at når jeg prater om zero/one/double-copy forholder jeg meg til underliggende API. Jeg har altså ikke tatt hensyn til MPI-implementasjonens eventuelle bruk av buffere eller bruk av zero-copy protokoll. En ideell implementasjon av ARMCI over MPI ville likefult hatt zero-copy både gjennom ARMCI og MPI-biblioteket. På grunn av kompleksitet og problemer med konvertering av ARMCI-datatyper til MPI-datatyper har jeg ikke funnet en løsning som utnytter MPI-datatyper. Det er også viktig å presisere at en zero-copy modell for ikke-sekvensielle data ikke nødvendigvis vil oppnå hverken bedre båndbredde eller forsinkelse, men at dette bør undersøkes. Slike implementasjoner må sees i sammenheng med den høye interne båndbredden i maskinen, sammenliknet med interconnect-hastigheten samt CPU-ens bruk av cache.

Kapittel 6

ARMCI over lavnivå SCI-driver

ARMCI-implementasjonen over MPI-API-et har høy portabilitet. Dessverre innfører slike API et ekstra bibliotekslag. Dette betyr at kommunikasjonsprotokoller og implementasjonen av biblioteket er utviklet med hensyn på et mer generelt tilfelle enn ARMCI-kommunikasjon. Full kontroll over protokoller og overhead involvert i kommunikasjon kan dermed kun oppnås med et mer lavnivå API.

Et problem i så måte er portabilitet. Drivere for forskjellige nettverksarkitekturer og adaptere har ofte egne API. Forskjeller i slike grensesnitt forekommer både mellom forskjellige nettverksarkitekturer og ved forskjellige implementasjoner over samme arkitektur. For eksempel har SISCI- og ScaSCI-driverene for SCI både forskjellige API og fokus for sitt design. Det har blitt fremmet forskjellige felles grensesnitt for drivere, for eksempel VIA[15] og DAT (Direct Access Transport)[137], men ingen slike generelle grensesnitt foreløpig er fullt implementert for SCI[1].

Jeg så meg derfor nødt til å implementere ARMCI direkte over et SCI driver-API. Dette medfører relativt lav portabilitet, men jeg forventet mer kontroll over kommunikasjonen og dermed en høyere potensiell ytelse.

Jeg valgte å benytte ScaSCI-driveren sammen med minnekopieringsbiblioteket ScaMemfra Scali AS, sammen kalt ScaFun. Målsettingen for implementasjonen var å få ARMCI til å kjøre over SCI ved hjelp av minnekopieringsrutiner, uten bruk av mellomliggende standardprotokoller og bibliotek.

6.1 ScaFun

ScaFun[66] tar sikte på å være et effektivt, intuitivt og sikkert kommunikasjonslag for shared memory og SCI-kommunikasjon. ScaFun-implementasjonen er delt i to selvstendige bibliotek: ScaSCI og ScaMem.

6.1.1 ScaSCI, driverdesign og API

ScaSCI-driveren er basert på API-et beskrevet i av Ryan[124] og bruk av de PCI-baserte SCI-adapterene fra Dolphin. Det tilbys funksjonalitet for bruk i både *user-level* og *kernel-level*. Fordi kernel-level-klienter er "sikrere" enn user-level-klienter har driveren to separate grensesnitt. User-level-grensesnittet (USRAPI) er implementert på toppen av kernel-level-grensesnittet og utfører parametersjekking og validering av funksjonskallene. Den største delen av driveren, også referert til som Interconnect Mana-

[1] Ghouas[52] viser i sin hovedfagsoppgave en eksempelimplementasjon av VIA over SCI. Dessverre er ikke denne implementasjonen videreutviklet til en kommersiell kvalitet.

ger (ICM) er en implementasjonen av kernel-level grensesnittet. ARMCI er et brukerbibliotek og må følgelig benytte det mer begrensede user-level grensesnittet.

Gjennom USRAPI-et kan en user-level klient indirekte benytte ICM servicene for å sette opp SCI-minne områder (kalt *chunker*). Driveren er kun involvert i oppsett og fjerning av disse områdene. Når disse er satt opp, går all kommunikasjon direkte gjennom shared memory-mappinger ved hjelp av PIO (ScaMem) eller DMA (initiering gjøres av ScaSCI).

Oppsett av SCI-chunker gjennom USRAPI kan deles inn i følgende 4 steg:

1. En node allokerer et område (chunk) av sitt lokale minne som den ønsker å tilby andre noder på interconnectet. Minneområdet blir automatisk initiert til null. I så måte er det viktig å innse at dette minnet vil bli pinned (kan ikke flyttes på eller swappes ut). I tillegg krever ScaSCI-driveren at dette minnet er *alignet* 4 byte og sekvensielt allokert.

2. Noden tilbyr deretter chunken til andre noder gjennom SCI-adapteret. Chunken gis et unikt navn i form av *NodeID*, *ChunkID* og *ModuleID*.

3. Prosesser på andre noder kan nå koble seg til denne chunken ved å benytte dens unike navn (beskrevet i punkt 2).

4. Når ikke-lokale prosesser har koblet seg til chunken kan prosessene mappe (hele eller deler av) chunken inn i sitt virtuelle adresseområde. De kan således aksessere minnet på andre noder ved hjelp av load/store instruksjoner mot sine lokale virtuelle adresser. Det er viktig å merke seg at dette minnet ikke kan caches på andre noder enn der minnet fysisk er allokert.

Interrupter

Interrupt-funksjonen ved bruk av ScaSCI-driveren har i utgangspunktet samme initiering som ved normalt chunk-oppsett. Hovedforskjellen ligger i at den lokale prosessen ikke kan lese og skrive til dette minneområdet uten bruk av eksplisitte funksjonskall.

Interrupt uføres med en *fetch-and-increment* funksjon i SCI. Deteksjon og nullstilling av interrupt er ikke en atomisk operasjon (se avsnitt 6.2.3). Dette betyr at man står i fare for å miste interrupter. ScaSCI-driveren vil ikke køe interrupter eller holde styr på hvor interruptet kom fra. Slik funksjonalitet må følgelig implementeres av brukerapplikasjonen.

6.1.2 ScaMem

ScaMem biblioteket er i utgangspunktet enkle minnekopieringsrutiner optimalisert for SCI. Disse rutinene forutsetter ingen spesielle kommunikasjonsprotokoller eller liknende, men effektiviserer minnekopiering i forhold til SCI-adaptere og chipset.

Introduksjonen av nye arkitekturer, chipset og prosessorer har ført til at raske minnekopieringsoperasjoner ikke lenger er trivielle. Nye prosessorer implementerer stadig mer avanserte prefetch-heuristikk i hardware og nye sett med registre (for eksempel MMX og SSE). I tillegg må man ta hensyn til varierende bufferstørrelser og koherensprotokoller i nye arkitekturer av chipset og PCI-SCI-adaptere for å utnytte disse maksimalt.

Unix kall som *memcpy()* implementerer i liten grad slik funksjonalitet. Shared memory-system blir følgelig ofte implementert med egne minnekopieringsrutiner som tar sikte på å fullt unytte de enkelte arkitekturer. Dette er blant annet tilfellet for interne operasjoner i ARMCI og implementeres også av ScaMem. ARMCI har derimot ikke støtte for minnekopieringsrutiner over SCI.

SCI: Remote-write-local-read

Effektivitet av PCI-SCI-kommunikasjon kompliseres ved at lesing av data (over nettet) er betydelig tregere enn lokale read (Omang [111]). Dette har sammenheng med manglende evne for prefetching ved remote read. Dette problemet gjør seg gjeldende for de fleste NUMA-maskiner (non-uniform memory access) som implementerer distribuert minne.

Nettverksadapterene har til forskjell fra CPU-er ikke innebygget logikk for prefetching og cache koherens. Følgelig benytter lokale read prefetch, mens remote read ikke kan utnytte dette. Kommunikasjon over SCI-nettverk er dermed mer effektiv for lokale read og remote read enn motsatt. Read av minnelokasjoner gjennom SCI-adapteret med påfølgende write til lokalt minne har altså liten nytte av prefetching og er derfor betydelig mindre effektiv.

ScaFun legger derfor opp til bruk av denne protokollen ved kommunikasjon over SCI. Dette har innvirkning på utvikling av kommunikasjonsprotokoller for ARMCI (og andre program/bibliotek).

Verifisering av overføringer

Vanligvis, når data sendes fra en node til en annen over SCI-nettverket, ankommer dataene sin destinasjon raskt og korrekt. Likevel er det, i et flernode SCI-/shared memory-system, alltid en mulighet for at noder er midlertidig utilgjengelig, for eksempel på grunn av høyprioritets operativsystemkall eller feil oppdaget av CRC sjekker.

Checkpointing er en vanlig programmeringsteknikk benyttet i systemer hvor dynamiske feil kan oppstå, for eksempel i shared memory- eller databasesystemer. Teknikken består i å samle nødvendige statusinformasjon før og etter operasjonen, for så å i etterkant sjekke om alt gikk bra. Hvis operasjonen feilet må operasjonen gjentas. Ved bruk av SCI er denne teknikken nødvendig for å kunne sikre en semantisk riktig datakopiering over SCI-nettverket.

For SCI initieres checkpoint-prosedyren ved først å tømme alle adapteres stream-buffere til nettverket. Checkpoint-tilstanden utledes deretter av driveren og en interrupt-teller. Hvis disse dataene endres etter operasjonen må den gjentas. Senderprosessen kan følgelig bestemme hvorvidt operasjonen gikk bra eller ikke.

Ved bruk av remote-write operasjoner som involverer prefetching er det også viktig å sikre at alle data har blitt sendt til kortet fra eventuelle cacher. Dette gjøres ved en minnebarriere (også kalt *CPU-flush*) på sendernoden. Denne operasjonen sikrer konsistens mellom dataene som sendes og dataene i cache på sendernoden. Ved bruk av ScaFun blir dette utført av ScaMem. ScaMem er også thread-safe.

6.2 ARMCI og ScaFun

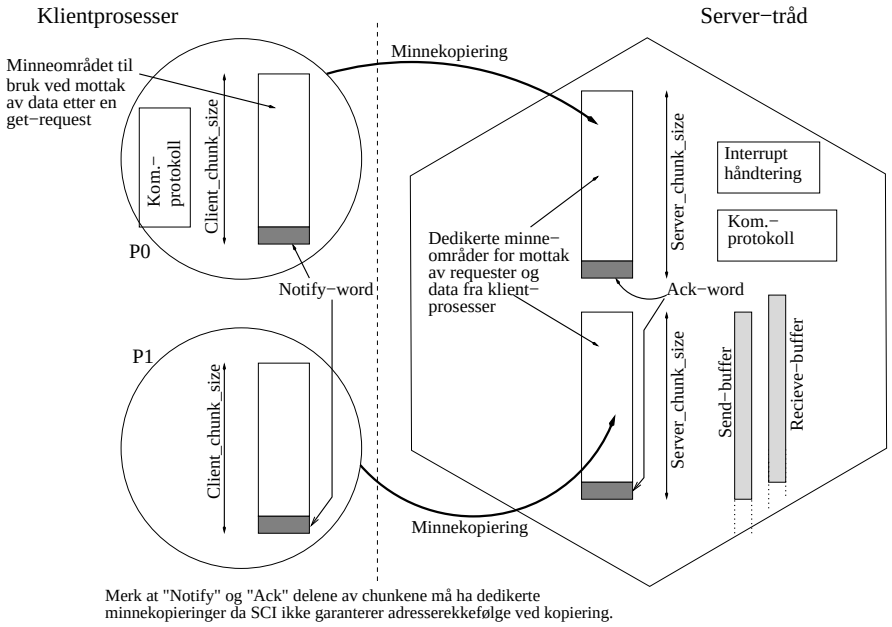
Eksempelimplementasjonen tok utgangspunkt i implementasjonen av ARMCI over MPI. Det ble primært satt en måletting om å bedre ytelsen til denne ved i utgangspunktet bruk av enkle, men effektive kommunikasjonsprotokoller.

For ARMCI over MPI vil MPI-implementasjonen ha ansvaret for å implementere en effektiv og sikker kommunikasjonsprotokoll. Denne funksjonaliteten måtte i implementasjonen over ScaFun eksplisitt utvikles og implementeres.

Det er tre hovedpunkter som er viktige ved ARMCI-implementasjonen over SCI:

1. Oppsett, størrelse og distribusjon av chunker og chunk-informasjon.

2. En effektiv kommunikasjonsprotokoll.



Figur 6.1: **ARMCI-SCI-chunker.** Hver klientprosess har opprettet en chunk for mottak av data som en følge av get-requester. Disse områdene mappes av alle de andre server-trådene (bortsett fra lokal tråd). Hver enkelt server-tråd oppretter en chunk for hver eksterne klientprosess (størrelsen varierer med antallet prosesser). Disse mappes en-til-en av klientene (en chunk pr. klient pr. server-tråd). SCI-kommunikasjonen skjer (etter initiering) ved minnekopieringsfunksjoner og bruk av notify/ack-protokollen skissert i avsnitt 6.2.2.

3. Effektive minnekopieringsrutiner.

I tillegg kommer oppsettet og viktigheten av effektive interrupt-funksjoner (se avsnitt 5.2.4). Det ble i utgangspunktet benytte samme implementasjon av interrupter som for ARMCI over ScaMPI (avsnitt 5.2.4).

6.2.1 SCI-chunker og -initiering

Allokeringsoppsettet for SCI-chunker i ARMCI ble implementert for å være enkelt, skalerbart og med mulighet for en viss grad av parallellitet. Oppsettet er skissert i figur 6.1 .

I første omgang allokerer hver klientprosess hver sin chunk. Denne chunken benyttes for mottak av data som følge av get-requester mot andre server-tråder. Hver server-tråd oppretter så en request-chunk for hver klientprosess som ikke er lokale for noden den selv befinner seg på. Størrelsen på både server- og klient-chunkene kan settes både runtime og compile-time. Størrelsene vil således avhenge av antall prosesser som benyttes og andelen kommunikasjons-overhead. Mindre chunker skaper typisk mer overhead en større for store request-størrelser (se protokoll i avsnitt 6.2.2).

Chunk-adresser (*NodeID, Module ID og Chunk ID*) blir deretter distribuert til prosessene i et alle-til-alle mønster. Denne informasjonen er nødvendig for at prosessene skal kunne mappe hverandres minneområder. Dette kan sammenliknes med IP-adresse og portnummer for Sockets-prosesser.

Et problem i så måte er utvekslingen av denne informasjonen. Det finnes protokoller for dette i SCI, men dette er komplisert. Jeg fant det derfor unødvendig å benytte disse, ettersom dette faller uten-

for fokus av denne rapporten. For eksempelimplementasjonen valgte jeg derfor enkelt nok å benytte MPI.

På bakgrunn av adresseinformasjonen fra de andre nodene mapper dermed prosessene hverandres minne. Flere server-tråder mapper samme klient-(get)-chunk, server-trådenes chunker er unike for hver klientprosess. Dette som en direkte følge av at en get-request er blokkerende og dermed kun kan motta data fra en server-tråd av gangen. Server-trådene derimot, kan motta requester fra flere klientprosesser i parallell (for eksempel flere put).

6.2.2 Protokoll

Jeg la i første omgang opp til en enkel kommunikasjonsprotokoll med sikte på å garantere sikker overføring. Mer ytelseseffektive protokoller for SCI-kommunikasjon er allerede utviklet og implementert. I denne sammenheng kan man nevne Valid-bit ideen fremsatt av Omang[111] og forfinet av Scali AS for bruk i ScaMPI. Denne protokollen muliggjør kopiering av hittil mottatt melding før senderen er ferdig med overføringen. Denne protokollen er basert på beskrivende headere og historiebit som genereres i mottakerens buffer.

For protokollen utviklet for ARMCI blir siste word (*ack/notify*) i hver chunk (både for klient og server) benyttet for synkronisering. Selv om ScaFun kan garantere en sikker overføring og checkpointing på sendersiden, har ikke mottakeren noen direkte måte å finne ut hvorvidt overføringen er ferdig. SCI garanterer ikke at overføringen skjer i sekvens (at buffer[i]skrives først og buffer[i+1] skrives sist). En dedikert notify er således nødvendig for å informere mottakeren om at dataene er overført. På samme måte kan ikke avsenderen vite når mottakeren har tømt sin chunk og er klare for nye data. Jeg må altså sikre at mottatte data er overført til andre buffere/datastrukturen før bufferet igjen kan fylles.

En melding fra A til B får dermed følgende steg:

1. Hvis A er en klientprosess sendes et interrupt (eller tilsvarende) for å la B vite at det kommer en melding. Ved generering av interrupt hos mottaker trenger ikke A vente på en bekreftelse på denne, men kan direkte sette i gang med overføringen av requesten. B vil på bakgrunn av polling eller interrupt på et tidspunkt forvente en melding fra A. Først nullstiller A sitt lokale *ack*-word. A kopierer så hele eller deler av request-dataene inn i B sin respektive chunk. A sjekker at overføringen gikk greit (checkpoint) og setter deretter *notify*-wordet nederst i B sin chunk (også her utføres det checkpoint). A poller på sitt ack-word og venter på at B skal bekrefte meldingen.

2. B som nå vet at det ankommer en melding fra A poller på sitt notify-word og venter på at A skal bli ferdig med overføringen. B tømmer deretter sin chunk og overfører deretter dataene til sitt receive-buffer. B nullstiller så sitt notify-word og bekrefter meldingen ved å sette ack-wordet hos A.

3. Hvis mer data skal overføres gjentas prosessen, hvis ikke kan send-/receive-funksjonene returnere.

En grafisk beskrivelse av protokollen er å finne i tillegg C.

Effektivitet av PIO mot DMA over SCI

I litteraturen verserer det mange påstander hvorvidt PIO eller DMA er den mest effektive metoden for dataoverføring. I så måte er en spesifisering av hvilken metode som egner seg best for hvilket bruk viktig. Et regnestykke ved bruk av DMA er; antallet CPU-sykler som benyttes for initiering

og sjekking av DMA-status i forhold til å la de samme syklene overføre data via PIO. I tillegg har hardware-implementasjon av både DMA-maskinen og annen omliggende hardware innvirkning på ytelsen.

Generelt sett kan det være flere ytelsesmessige problemer med bruk av DMA:

- *Cache koherens og I/O-bridge-arkitektur:* Ved bruk av PIO og remote write ligger ofte dataene allerede i cache og kan sendes direkte ut på I/O-bussen. CPU-en har også pipelinefunksjonalitet som benyttes ved større overføringer. Avhengig av CPU- og cache-arkitektur vil ikke DMA-maskinen nødvendigvis kunne lese data direkte fra cache. Dette betyr at DMA-maskinen må lese fysiske minneadresser og følgelig synkronisere nodens minne og cache. Dette skaper høyere forsinkelse og dårlig (eller mangel på) prefetching. Chipset-implementasjoner har også stor innvirkning på ytelsen.
  Et annet poeng å merke seg er at en double-copy-implementasjon er dyrere ved bruk av DMA-maskiner (som ikke leser cache) enn for PIO-baserte implementasjoner.

- *Initiering og oppsett av DMA-maskinen:* DMA-maskiner må initieres før bruk og status må sjekkes etter overføringen. Initiering består blant annet av pinning av minne og behandling av eventuelle interrupt eller status ved utført overføring. Disse operasjonene tar et varierende antall CPU-sykler (avhengig av DMA-implementasjon) og kunne mer direkte vært benyttet til PIO-dataoverføringer. Følgelig er DMA-maskiner lite egnet for mindre dataoverføringer.

- *Mangel på sekvensielle buffere og alignment:* Bugge[63] påpeker at ytelsestester for DMA-maskiner ofte ikke er reelle. Disse testene setter opp minneområder og buffere i et bruksmønster som ikke gjenspeiler applikasjoner. Et viktig punkt i så måte er at applikasjoner opererer med virtuelle adresser (dette gjør også PIO). DMA-maskiner derimot opererer med fysiske adresser. Figur 6.2 viser hvordan DMA-maskinen således kan måtte gjøre unødvendig mange overføringer ved applikasjon-til-applikasjon overføringer og bruk av zero-copy protokollen.

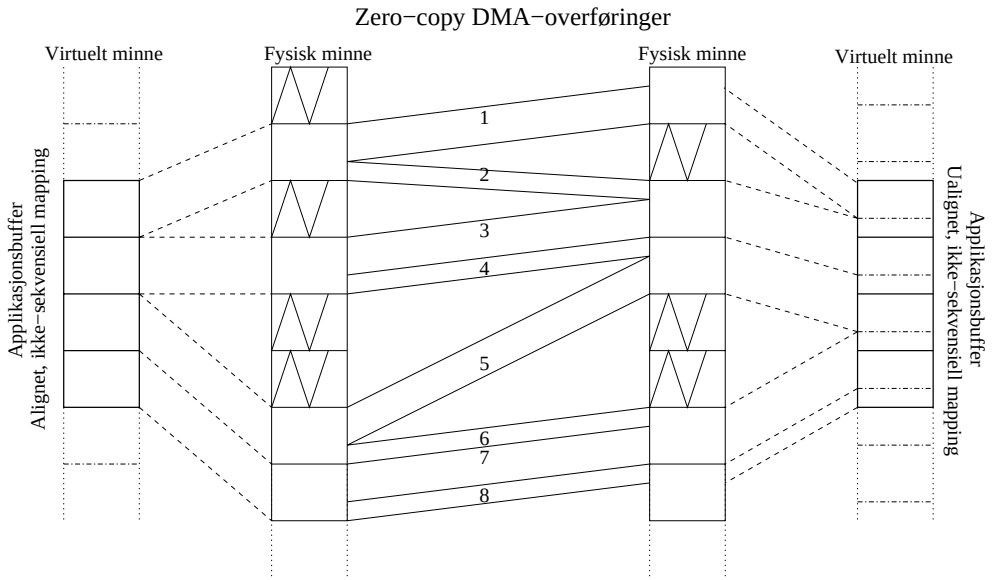
Se forøvrig figur 3.4 på side 35 for eksempler på DMA-ytelse for Dolphin sine SCI kort. Med disse punktene som en utgangspunkt valgte jeg derfor å la ARMCI over SCI i første omgang basere seg på PIO

6.2.3 ScaSCI-interrupter

For å unngå at server-tråden sekvensielt skal sjekke (polle) sine notify-word for innkommende requester, er en interrupt-funksjonalitet nødvendig. Jeg benyttet derfor den samme interrupt-baserte protokollen som for ARMCI over ScaMPI (avsnitt 5.2.4). Problemet i så måte er at jeg i utgangspunktet ikke har tilgang på noen *MPI_Iprobe()* funksjon. En slik funksjonalitet måtte dermed implementeres ved sjekk av notify-word for alle av server-trådens request-chunker.

Desverre kom jeg heller ikke her utenom problemer i forhold til ScaSCI-driveren. Problemet kan deles i fire og gjelder både implementasjonen over ScaFun (ScaSCI) og ScaMPI:

1. *Udetekterte interrupter:* Som sagt holder ikke driveren noen kø over interrupter. Dette er problematisk ved forsøk på å implementere ekte ensidig ikke-blokkerende put- og accumulate-funksjonalitet. Ved to tett etterfulgte put-requester kan driveren følgelig overse interruptet send av request nummer to, mens interruptet fra request nummer en behandles.
   Interrupter genereres ved hjelp av en SCI *fetch-and-increment* på et 32 bit word. Et problem i så måte er at interruptet kun genereres ved forandring av wordets øverste bit. Interruptet nullstilles ved å sette dette wordet til 1 under tallet som vil forandre øverste bit. Ved å da å gjøre en



Figur 6.2: **DMA-operasjoner og alignment.** Figuren skisserer hvordan overføring av et buffer, fordelt på fire *pager* i det fysiske minnet, kan føre til dobbelt så mange DMA-overføringer ved uheldig alignment av buffere. DMA-maskiner kan vanligvis kun overføre fra fysisk adresse til fysisk adresse innenfor en og en page av gangen, både på sender og mottakersiden. DMA-maskiner er ofte optimalisert for sekvensielle overføringer av fysisk alignede databuffere. I applikasjoner er dette sjelden tilfellet. Applikasjonsbuffere er sekvensielle i virtuelt minne, men ikke nødvendigvis i fysisk minne. DMA-maskiner gjør minneoperasjoner mest effektivt på fysisk page nivå. Fysisk fragmenterte applikasjonsbuffere er følgelig ikke optimale for overføring.

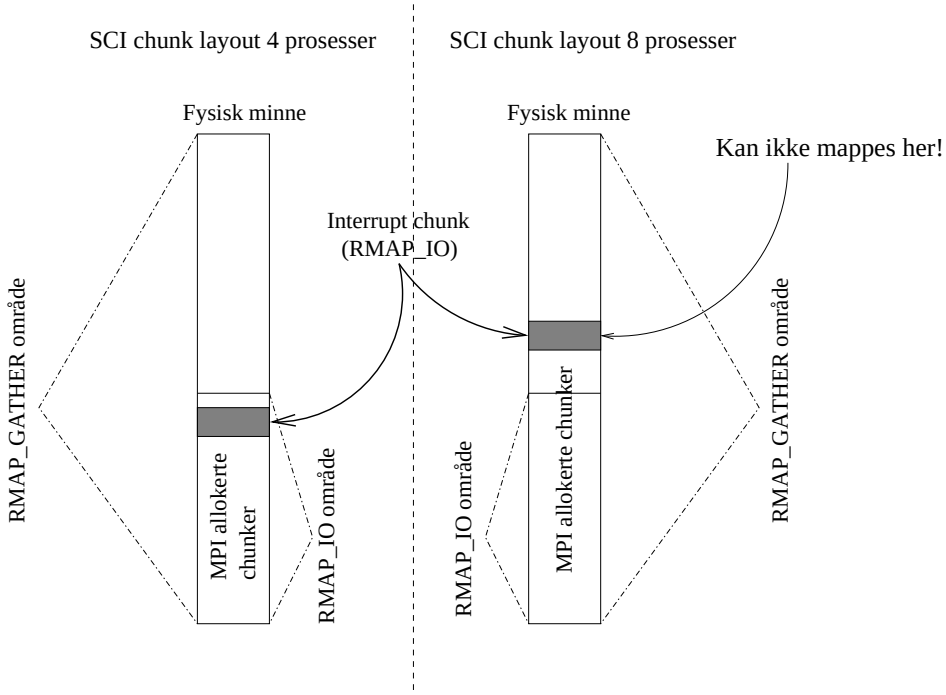
```
/*
  Make thread sleep and wake up on interrupt.
  Flag is disarmed and reset on return
*/
SciWaitForInterrupt(local_server_ints, &uFlag);

/* Find out what node the interrupt came from */
next_process = uFlag;

/*
  Resetting Flag (NB! not all flags)
  We can safely reset flag, node we just received interrupt
  from expects to be served before issuing another
*/
SciResetInterruptFlagToOneShot(local_server_ints, next_process);
```

Figur 6.3: **Interrupt-nullstilling for ScaSCI-driveren.** Funksjonen *SciWaitForInterrupt()* returnerer ved interrupt og gir (ved hjelp av designetet i figur 5.5 på side 66) hvilken klientprosess som genererte interruptet. Funksjonskallet nullstiller likevel ikke interrupt-flagget. Dette må gjøres med funksjonen *SciResetInterruptFlagToOneShot()*, noe som gjør disse operasjonene ikke-atomiske.

fetch-and-increment med 1 vil interruptet genereres. To slike interruptet sendt etterhverandre til samme interrupt-word (uten nullstilling mellom), vil dermed oppdatere variabelen med 2. Dette fører likevel til at det kun detekteres et interrupt hos mottakeren da øverste bit kun skifter en gang. Man kan til en viss grad komme rundt denne problematikken ved å benytte høyere tall ved sending av interrupt (tall som alltid vil forandre øverste bit). Persvold[132] påpeker likevel at dette ikke er en verifisert funksjonalitet og kan medføre at interrupt ikke genereres/detekteres i det hele tatt. I min implementasjon har jeg derfor basert meg på sending av interrupter med 1 som tall for interrupt-generering.

2. *Mangel på atomisitet for deteksjon og nullstilling av interrupter:* Som vist i figur 6.3 er deteksjon og nullstilling av interrupter to eksplisitte ikke-atomiske funksjonskall. Jeg kan altså risikere å nullstille et interruptflagg hvor flere interrupt er generert uten å detektere dette. Kun første interrupt vil oppdages. En kommunikasjonsprotokoll må derfor implementeres som sikrer synkronisering og nullstilling av interruptflagget før neste interrupt genereres.
   For eksempel-implementasjonen av ARMCI over MPI har jeg derfor måtte legge inn en eksplisitt MPI-synkronisering (0 byte meldinger) for å synkronisere sender og mottakerprosessen. En *MPI_Send()* for små datastørrelser kan nemlig returnere før en *MPI_Recv()* er ferdig på mottakersiden.
   Implementasjonen over ScaFun benytter ack og notify-protokollen som sikrer synkronisering av sender og mottaker.

3. *Feil ved allokering av interrupt-chunker:* Ved oppsett av interrupt-chunker benyttes ikke write-gathering (RMAP_GATHER, se avsnitt 3.3.3). Implementasjonsdetaljer i SCI-adapteret gjør at det kun er et begrenset adresseområde som kan benyttes for slik direkte (RMAP_IO) skriving. Allokering av interrupt-chunker er følgelig sårbar for rekkefølge. Kort sagt allokeres chunker (uavhengig av hvorvidt disse er av "vanlige" eller interrupt-type) i den sekvensen allokeringskallene kommer. For ARMCI i sammenheng med en SCI-MPI-implementasjon vil MPI-en allokere sine chunker først (ved *MPI_Init()*). Antallet og størrelsen på disse chunkene vil variere



Figur 6.4: **Fysisk allokering av interrupt-chunker.** Figuren viser hvordan den fysiske allokeringen av interrupt-chunker kun kan skje innenfor et begrenset adresseområde. Desverre tar ikke ScaSCI-driveren hensyn til dette. Ved allokering av "vanlige" (RMAP_GATHER) chunker før initiering av interrupt-chunkene(RMAP_IO) kan man dermed risikere å ikke få benyttet interrupter.

med antallet prosesser i programmet. Når ARMCI så prøver å allokere sine interrupt-chunker (RMAP_IO) vil disse kunne bli forsøkt allokert i områder som ikke tillater slik funksjonalitet og følgelig ikke fungere. Figur 6.4 viser en enkel skisse av problemet. Dette har begrenset min implementasjon av ARMCI med SCI-interrupter, til kun å fungere for 6 prosesser.

4. *Forsinkelse og overhead ved bruk av SCI-interrupter:* Målinger utført viser at forsinkelsen (dvs applikasjon-til-applikasjon) for SCI-interrupter er meget høy. Jeg har målt tiden fra interruptet blir sent til det oppfattes av mottaker prosessen til 90-100μs. Til sammenlikning opererer Ethernet med interrupt-tider på ca 20 μs. Det er vanskelig å peke på konkret hva som forårsaker denne forsinkelsen, men en stor del av denne tiden er overhead gjennom operativsystem og overføring mellom kernel-space og user-space. Med andre ord er både implementasjonen av SCI-driveren, operativsystem og SCI-kortet bidragsytende.

6.3 Potensielle optimaliseringer

Dagens implementasjon over ScaFun benytter en double-copy protokoll og har en enkel kommunikasjonsprotokoll (basert på ack og notify). Dette er langt fra optimalt og det er i hovedsak fire punkter som bør utforskes for videre optimaliseringer:

1. *One-copy modell for ikke-sekvensielle og/eller større datastørrelser:* Jeg ser ved bruk av minnekopieringsfunksjoner over SCI, en bedre mulighet for utviklingen av en one-copy modell

enn for ARMCI over MPI. Spesielt gjør dette seg gjeldende ved større datastørrelser og ikke-sekvensielle data. For ikke-sekvensielle datatyper kan pakking av dataene på sendersiden skje direkte til mottakernes SCI-chunk fremfor å eksplisitt gå gjennom et send-buffer. Det samme kan implementeres for sekvensielle data (da uten å måtte pakke dataene). Dataene kan således kopieres i en *pipelinet* protokoll hvor et og et steg av pakkefunksjonen kopierer til mottakeren. Et viktig poeng i så måte er chunk-størrelsen må mottakersiden. Denne må enten være stor nok til å kunne motta alle dataene, implementeres som et ringbuffer, eller med en senderprotokoll med tilsvarende funksjonalitet. En slik optimalisering er vanskeligere å uføre på ARMCI-versjonen over MPI da en slik protokoll ville føre til generering av flere MPI-meldinger og følgelig mer overhead. Problemet med datatyper (som er reellt for MPI) vil ikke oppstå ettersom jeg hele tiden forholder meg til ARMCI-datatyper og kun flytter dataene uten typekonvertering.

2. *En kommunikasjonsprotokoll som tar hensyn til meldingsstørrelser:* En to eller tredelt kommunikasjonsprotokoll basert på meldingsstørrelser kan senke forsinkelse og øke båndbredden. For eksempel ScaMPI benytter en slik tredeling av kommunikasjonsprotokollene for små, mellomstore og store meldinger.

3. *Ikke-blokkerende send for put- og accumulate-requester:* For ikke-blokkerende operasjoner bør man vurdere bruk av protokoller som enten a) benytter en write-and-forget, og deretter sjekker status, eller b) benytter DMA. Selv om båndbredden ved bruk av SCI og DMA ikke er optimal kan man anse den for “god nok”. Hovedgrunnen for dette ønsket er å ha muligheten for å kunne returnere tilbake til applikasjonen så fort som mulig og la det underliggende ARMCI-laget håndtere kommunikasjonen. For å sikre koherens i datastrukturen er slik funksjonalitet vanskelig å implementere ved hjelp av en one-copy protokoll. Man må her muligens se seg nødt til å buffere på sendersiden og eventuelt avgjøre hvorvidt slik buffering skal utføres på bakgrunn av datastørrelsen.

4. *Interrupt-basert mottak i server-tråden:* Som nevnt har interrupt-funksjonalitetens problemer i hovedsak vært avhengig av implementasjonen og API-et for ScaSCI-driveren. Likevel er denne funksjonaliteten kritisk for å kunne utnytte alle prosessorene i en SMP-node når Hyper-Threading ikke er tatt i bruk. Forandring av API-er for atomisitet av lesing og nullstiling av interrupter er ikke heldig ettersom andre applikasjoner allerede er basert på eksisterende grensesnitt.
   En relativt enkel omskriving av ScaSCI-driveren vil likevel kunne håndtere problemet med allokering av interrupt-chunker ved å sette av dedikerte områder for RMAP_IO write.
   Det finnes også en annen metode for generering av interrupter på PCI-SCI-adapteret. Interruptfunksjonaliteten jeg i dag har forsøkt å benytte kalles *local interrupter* i ScaSCI-driveren. Alternativt kan man benytte *remote interrupter* som er basert på skriving til mottaker-adapterets CSR register i PSB-en. Denne funksjonaliteten benyttes av for eksempel ScaMAC i ScaIP-implementasjonen. Dessverre er ligger denne funksjonaliteten foreløpig kun i kernel API-et. Jeg har likevel fått indikasjoner på at Scali AS er i ferd med å endre denne delen av ScaSCI-API-et for således å kunne tilby denne funksjonaliteten i fremtiden. Man vil likefullt være utsatt for overheaden forbundet med at interruptet må flyttes fra kernel-space til user-space.

6.4 Oppsummering

Som forventet er implementasjonen av ARMCI over ScaFun noe mer komplisert enn implementasjonen over MPI. Likefullt gir denne implementasjonen mer kontroll over både dataflyt og kommuni-

kasjonsprotokoller. Implementasjonen er således mer i tråd med Nieplochas[105] tanke om ARMCI med optimal, lavnivå og skreddersydd kommunikasjon.

Implementasjonen av ARMCI med ScaFun har vist at jeg selv med enkle kommunikasjonsprotokoller kan oppnå en relativt god ytelse over SCI (testresultater i 7.4). Ytelsen er likevel fundamentalt avhengig av minnekopieringsrutinene i ScaMem og er betydelig dårligere ved bruk av standard Unix *memcpy()*. Jeg har av ytelseshensyn valgt utelukkende å benytte kopiering via PIO og remote-write-read-local protokoll, men ser for meg bruk av DMA ved put- og accumulate-funksjonalitet ved videre utvikling.

Ettersom chunk-størrelser kan variere med både antall noder og antall prosesser lar løsningen seg, i likhet med ScaMPI, skalere. En punkt å undersøke i så tilfellet er optimal chunk-størrelse for de individuelle applikasjoner/kommunikasjonsmønster.

Selv om denne løsningen har vist en meget bra ytelse, er det fortsatt rom for forbedring. I så henseende er minimalisering av bufferbruk og en effektiv kommunikasjonsprotokoll de viktigst punktene.

Kapittel 7

Verifiserings- og ytelsesresultater

Jeg har lagt liten vekt på en statistisk korrekt presentasjon av de empiriske målinger i denne oppgaven. Slike analyser og presentasjoner krever mye tid og en grundigere analyse av de forskjellige testene. Testresultatene i denne oppgaven er således ment som en indikasjon på ytelsen som kan oppnås ved hjelp av prototypeimplementasjonene beskrevet i avsnitt 4.3 samt kapittel 5 og 6.

Tester som anses som spesielt sårbare for sideeffekter ved *timere* og/eller annen varians, ble kjørt et antall ganger for å bekrefte målingene. Resultatene er likefullt presentert ved en instans av disse målingene og ikke som et maksimal, minimal eller gjennomsnittsresultat. Resultatene bør derfor leses i lys av dette. Jeg la likefullt vekt på å la testene følge Hennessy og Patterson[61, s.32] sitt prinsipp om reproduserbarhet av testresultater.

7.1 Testbeskrivelser

Avsnitt 7.1.1 beskriver test-clusterene benyttet i denne oppgaven. Endring av hardware-komponenter som CPU og/eller chipset, SCI-adaptere og svitsjer kan ha stor innvirkning på kjøringer. Forventede resultater ved bruk av annen hardware og/eller operativsystem er således vanskelig å bestemme.

Avsnitt 7.1.2 beskriver målsettingen med testene.

7.1.1 Testplattformer

Det ble benyttet to forskjellige clustere fra Scali AS for testing. Begge systemer hadde samme system-programvare: Red Hat Linux 8.0 med 2.4.18-10smp kjerne, Scali SSP 3.1 med ScaMPI versjon 1.13.9 og ScaSCI versjon 2.4.14-1. Det ble benyttet Global Arrays versjon 3.1.8 med og uten modifisert ARMCI-bibliotek. For kompilering av bibliotek og testapplikasjoner ble GCC 2.91.66 benyttet. ScaMPI over SCI ble benyttet også for alle tester (også der hvor ARMCI benytter GbE).

Clusterene, heretter referert til som “Puma” og “Rigel”, varierer noe i sine hardware-konfigurasjoner, se tabell 7.1 .

7.1.2 Målsetting med testene

Målsettingen med testene kjørt over de nye ARMCI-implementasjonene kan kort beskrives i to hovedpunkt:

1. Verifisere at de nye implementasjonene fungerer semantisk likt som den originale implementasjonen. Da med hovedvekt på API og funksjonalitet.

Konfigurasjon	Puma	Rigel
Nodetype	Supermicro SuperServer 6012P-6	Dell PowerEdge 2650
#Noder	2 (SMP 2)	8 (SMP 2)
#CPU-er	4	16
CPU	Intel Xeon 1.80GHz (512KB cache)	Intel Xeon 2.40GHz (512KB cache)
Chipset	Intel Corp. e7500 [Plumas]	ServerWorks Grand Champion LE
Dolphin SCI-adaptere	PSB66 SCI-Adapter D33x	PSB66 SCI-Adapter D33x
SCI-topologi	1D ring	2D torus
Gigabit Eth. adapter	Intel Corp. 82544GC	BROADCOM Corp. NetXtreme BCM5701
Gigabit Eth. topologi	Punkt-til-punkt	Svitsj (Netgear GS524T, støtter ikke jumbo frames)

Tabell 7.1: **Hardware-konfigurasjon for test-clusterene.**

2. Forsøke å måle (ved hjelp av flere metoder) eventuelle ytelsesendringer for de nye implementasjonene sammenliknet med den originale Sockets-implementasjonen over GbE. Da med hensyn på forsinkelse (overhead), båndbredde, CPU-bruk og eksekveringstid.

Som referanse ble ARMCI sin Sockets-implementasjon over GbE benyttet både for ytelses- og verifikasjonstestene.

7.2 Verifkasjonstester

Både GA- og ARMCI-distribusjonene inneholder flere mindre verifikasjonstester. Resultatet av disse kan tolkes som enten "bestått" eller "ikke-bestått". Det samme gjelder verifikasjonstester i GAMESS-UK- og NWCHEM-distribusjonene. Verifikasjonstestene ble kjørt på et varierende antall noder (>= 2). Alle testene ble bestått av samtlige prototypeimplementasjoner, noe som er essensielt for relevansen av videre ytelsestester.

7.3 Ytelse og målestokker

7.3.1 Forsinkelse

Forsinkelse er et omsluttende begrep, og kan omfatte enten hardware, programvare eller begge og måles i tid (gjerne mikrosekunder). I hardware er man for eksempel interesset i å måle forsinkelsen av utførelsen av en instruksjon, henting av n byte fra minnet, flytting av data i minnet eller skriving og lesing til diverse I/O-enheter. Hardware-forsinkelse ved bruk av interconnect er definert som L parameteren i LogGP. For programvare er bibliotek og konkrete funksjonskall typiske målepunkter. Et viktig poeng i så måte er å konkretisere nøyaktig hva som måles og hvilke operasjoner som utføres.

Et typisk problem når det kommer til slike målinger er når man skal definere start og stopp av tidsmålingen. Hvorvidt operasjonen kan måles ved hjelp av gjennomsnittet av flere like operasjoner må også bestemmes. Ved for eksempel minneoperasjoner spiller cache fort en viktig rolle for ytelsen. Om man ønsker å kun måle selve minnehastigheten uten involvering av cache må benchmarken spesifiseres og designes deretter. Et annet poeng i så måte, som forøvrig også gjelder for måling av forsinkelse i interconnect, er størrelsen på dataelementene som sendes og hvorvidt resultatene har noe fotfeste i den virkelige verden. Man kan godt måle minneforsinkelse uten bruk av cache, eller nettverksforsinkelse uten bruk av kommunikasjonsbibliotek. Men alle maskiner benytter cache og er avhengig av et kommunikasjonsbibliotek for å sende og motta data i applikasjoner.

Huse[68, s.14] forteller at den eneste målemetoden de alle er enige om i MPI-sammenheng, er måling av såkalt *ping-pong-half (ping-pong/2)* forsinkelse. Denne testen uføres ved å ta tiden på en såkalt *round-trip*, for deretter å dele denne tiden på 2. En round-trip er definert som: t_0 =ved utføring av n byte *MPI_Send()* fra A til B, når B har mottatt meldingen fra A sender den en melding tilbake til A, t_1 =når A har mottatt meldingen. Round-trip blir dermed $T_{round} = t_1 - t_0$ og ping pong half som $T_{ping-pong-half} = \frac{T_{round}}{2}$. Antallet byte n kan være fra 0 til N avhengig av hvilke resultater man ønsker å sammenlikne med. For andre tester understreker Huse viktigheten av å oppgi hva man måler og hvordan man har kommet frem til resultatet.

For en ensidig kommunikasjonsmodell har man tilsvarende problem med når tiden skal måles. En get-request i ARMCI kan sammenliknes med en round-trip i tosidig kommunikasjon (en ping-pong test). Likevel må man være klar over at størrelsen på en request ikke nødvendigvis er den samme som størrelsen på dataene som returneres. Med andre ord er selve request-meldingen ofte følsom for forsinkelse, mens dataene som returneres er følsomme for båndbredde. En ARMCI-put-request kan heller ikke direkte sammenliknes med en ping-pong half:

1. En put-request på 0 byte gir ingen mening. Hvor mange byte skal så benyttes?

2. Hvordan skal man måle tiden? Mottakerprosessen vil i ARMCI-semantikken ikke en gang være klar over at den har mottatt en melding! Hvis man måler på sender siden, vil tiden avhenge kun av hvor raskt send-/put-kallet returnerer. Dette betyr at man ikke tar med tiden for å behandle meldingen på mottaker siden, selv om dette strengt tatt er en del av operasjonen. Denne effekten fremkommer ved bruk av oppsamling av put-requester i ARMCI over ScaMPI.
   Hvis man innfører en hanshake (eller fence-operasjon) fra mottaker prosessen for å bekrefte at hver melding er mottatt og behandlet, har man effektivt skapt en get-request. Samtidig har man introdusert unødvendig og urealistisk overhead og følsomhet for forsinkelse.

Litteraturen jeg har kommet over ser ikke ut til å adressere disse problemene. De fleste benchmarker måler forsinkelsen av en put-request med et variabelt antall byte og tar tiden på sendersiden. Slike tester favoriserer enkelte kommunikasjonsbibliotek og/eller interconnect som for eksempel Sockets (som benytter operativsystembuffere) eller DMA-baserte adaptere (ved større datastørrelser). Disse systemene returnerer fra send kallet sitt allerede før alle dataene er sendt av noden og gir dermed et urealistisk bilde av ytelsen.

7.3.2 Båndbredde

Båndbredde er definert som: $\frac{Antall\ byte}{Forsinkelse + Sende\ tid}$. Typisk vil båndbredden variere med antall byte som overføres. For et lavt antall sendte byte vil forsinkelsen være den dominerende faktoren i nevneren, mens sende tid vil dominere ettersom antall byte øker. For å sammenlikne i LogGP vil små datastørrelse være mest avhengige av $o + L$ mens større datastørrelser vil avhenge av g, G og o.

Båndbredde og forsinkelse har mange felles problemer når det kommer til målemetode og operasjoner. For eksempel ser ikke MPI-benchmarker ut til å være standardisert på dette punktet heller (i likhet med måling av forsinkelse). Viktigheten av å spesifisere metoder og operasjoner gjelder derfor også her.

Et annet viktig poeng når det kommer til måling av båndbredde, er bruken av PIO eller DMA-maskiner. Bruk av PIO krever nødvendigvis mer CPU-tid ved sending av større meldinger, mens DMA krever forholdsvis mye initiering (blant annet kopiering til DMA-buffere) og skaper høyere forsinkelse ved mindre meldinger. Dette kan skaper problemer ved realismen av målinger.

For ensidig kommunikasjon vil man dermed få de samme problemene som ved forsinkelse, både ved bruk av DMA og ved bruk av PIO.

7.3.3 Tid

Hennessy og Patterson[60, s. 40] skriver følgende:

> The author's posision is that the only consistent and reliable measure of performance is the execution time of real programs, and that all proposed alternatives to time as the metric or to real programs, as the items measured have eventually led to misleading claims or even mistakes in computer design.

7.4 Ytelsestester

ARMCI-distribusjonen inneholder et mindre utvalg ytelsestester for måling av forsinkelse og båndbredde. De viktigste testene er kort beskrevet nedenfor:

ARMCI_perf Programmet tester ARMCI-put, -get og -accumulate. Denne testen er i utgangspunktet beregnet på kjøring mellom to prosesser, hvorav en er aktiv og en er sovende. Den aktive prosessen utfører operasjoner mot den sovende. Operasjonene utføres på forskjellige datastørrelser og tidsmålingen skjer på sendersiden.

SPLASH-2 LU Dette programmet er en kjerne implementasjon av SPLASH-2[143] LU faktorisering basert på ARMCI-API-et. Testen benytter utelukkende get-operasjoner og måler *wall-time* av den totale eksekveringen. Programmet kan konfigureres for større eller mindre datasett og kan kjøres på vilkårlig antall noder.

ARMCI_perf er en syntetiske ytelsestest. SPLASH-2 LU testen er derimot en implementasjon av en reell applikasjonskjerne. Siden dette programmet benytter tid og kun get-operasjoner, vurderes denne testen som den beste av de to mikrobenchmarkene.

For å presentere ytelse ble det kjørt fire forskjellige applikasjoner: en båndbredde- og forsikelsestest mellom to noder (ARMCI_perf), en mikro-benchmark basert på en applikasjonskjerne (SPLASH2 LU factorization) og DFT og SCF analyse i NWChem. Resultater fra kjøringer med GAMESS-UK (Generalised Atomic and Molecular Electronic Structure System)[50], presenteres ikke her da resultatene sammenfaller med de gitt av NWChem[1].

Testen for båndbredde og forsinkelse (ARMCI_perf) ble kjørt på Puma-clusteret da disse maskinene har et av de bedre chipsetene for SCI-ytelse på markedet i dag. Dessverre hadde jeg ikke tilgang

[1] NWChem og GAMESS-UK er i stor grad basert på de samme algoritmene for DFT- og SCF-analyse. Ytelsesforskjellene er således minimale for kjøringene foretett her.

til et større cluster med tilsvarende noder, slik at skaleringstestene ble kjørt på Rigel (med et annet chipset), men med kraftigere prosessorer. Skaleringstestene ble gjort både med og uten bruk av Hyper-Threading.

7.4.1 Båndbredde og forsinkelse

Som en følge av vanskeligheten ved å måle båndbredde og forsinkelse for put- og accumulate-requester (se avsnitt 7.3), har jeg valgt kun å referer resultatene for get-requester her. Jeg anser altså målingene for get-requestene som mer nøyaktige og realistiske enn for put og accumulate. Resultatene for disse ensidige operasjonene er likevel å finne i tillegg B.

Figur 7.1 viser båndbredde ytelsen ved henholdsvis 1-dimensjonale og 2-dimensjonale get-operasjoner for ARMCI over Fast og Gigabit Ethernet (Sockets), ScaIP (Sockets), ScaMPI og ScaFun. Som man kan se av figurene gir ARMCI-implementasjonen over MPI (ved bruk av ScaMPI over SCI) den høyeste båndbredden tett etterfulgt av implementasjonen over ScaFun. Som referanse er også ytelsen for Sockets over GbE og ScaIP (da uten bruk av zero-copy) tegnet inn (se avsnitt 4.3.2).

Ettersom ARMCI tar sikte på å være best på ikke-sekvensielle data er grafen for 2-dimensjonale data mest interessant. Ved bruk av ScaMPI oppnår jeg således en *peak* båndbredde på 137 MB/s over GbE sin peak ytelse på 78 MB/s ved overføring av ikke-sekvensielle data. Med andre ord en forbedring på 76%. Implementasjonen over ScaFun kan vise til en peak båndbredde på 120 MB/s. Jeg har ikke kommet over noen rene MPI-tester (som kommuniserer med ikke-sekvensielle data) resultatene kan sammenliknes med for å indikere overhead gjennom ARMCI-biblioteket.

Figur 7.2 viser forsinkelsen ved mindre request-størrelser. Også her yter implementasjonen over MPI best med en forsinkelse for 8 byte get-requester på < 30 μs ved kommunikasjon av sekvensielle data. Implementasjonen over ScaFun er noe dårligere med sine 54.7 μs, men stadig bedre enn GbE sine 65.8 μs forsinkelse. ARMCI over ScaMPI kan altså skilte med en halvering av forsinkelsen for 8 byte meldinger sammenliknet med Sockets over GbE. Til sammenlikning viser *bandwidth*-testen fra Scali AS en round-trip forsinkelse på ca. 10.6 μs for ren MPI-kommunikasjon. Dette betyr at ARMCI-biblioteket kun introduserer en overhead (o) på ca 20 μs for 8 byte meldinger[2].

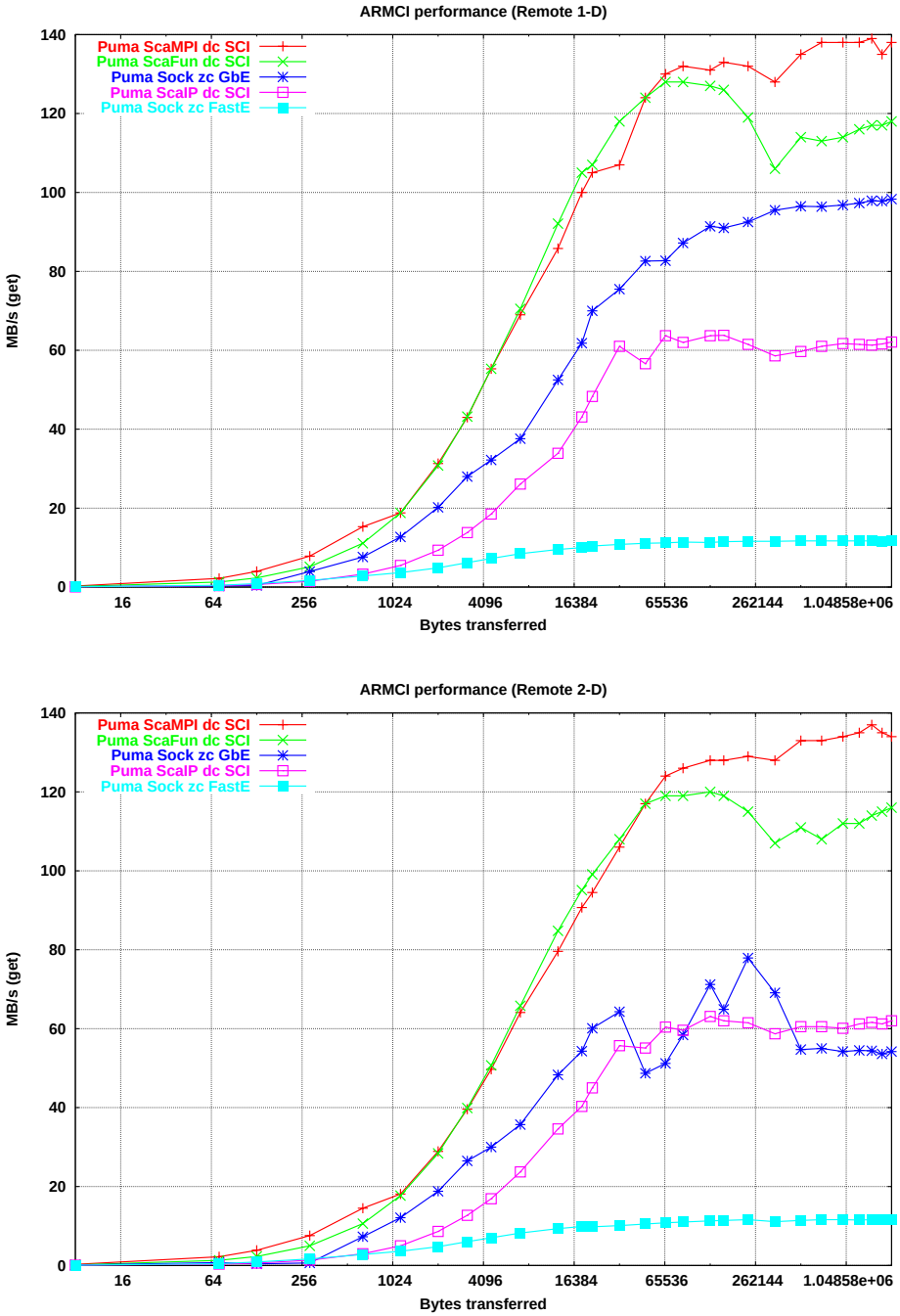
Båndbreddeforskjellen mellom Puma og Rigel maskinene er vist i figur 7.3 og viser tydelig forskjellene i SCI-ytelse for ulike chipset-implementasjoner. Legg også merke til at Rigel-clusteret har en høyere båndbredde enn Puma-clusteret ved bruk av GbE. Dette har sammenheng med blant annet CPU-en og dens evne til å prosessere TCP/IP-headerene. Figur 7.4 viser en sammenlikning av forsinkelsen forbundet med get-requester på de to clustrene. Testresultatene i neste avsnitt må derfor sees i lys av disse forskjellene ettersom disse har blitt kjørt på Rigel-clusteret.

7.4.2 Applikasjoner og skalerbarhet

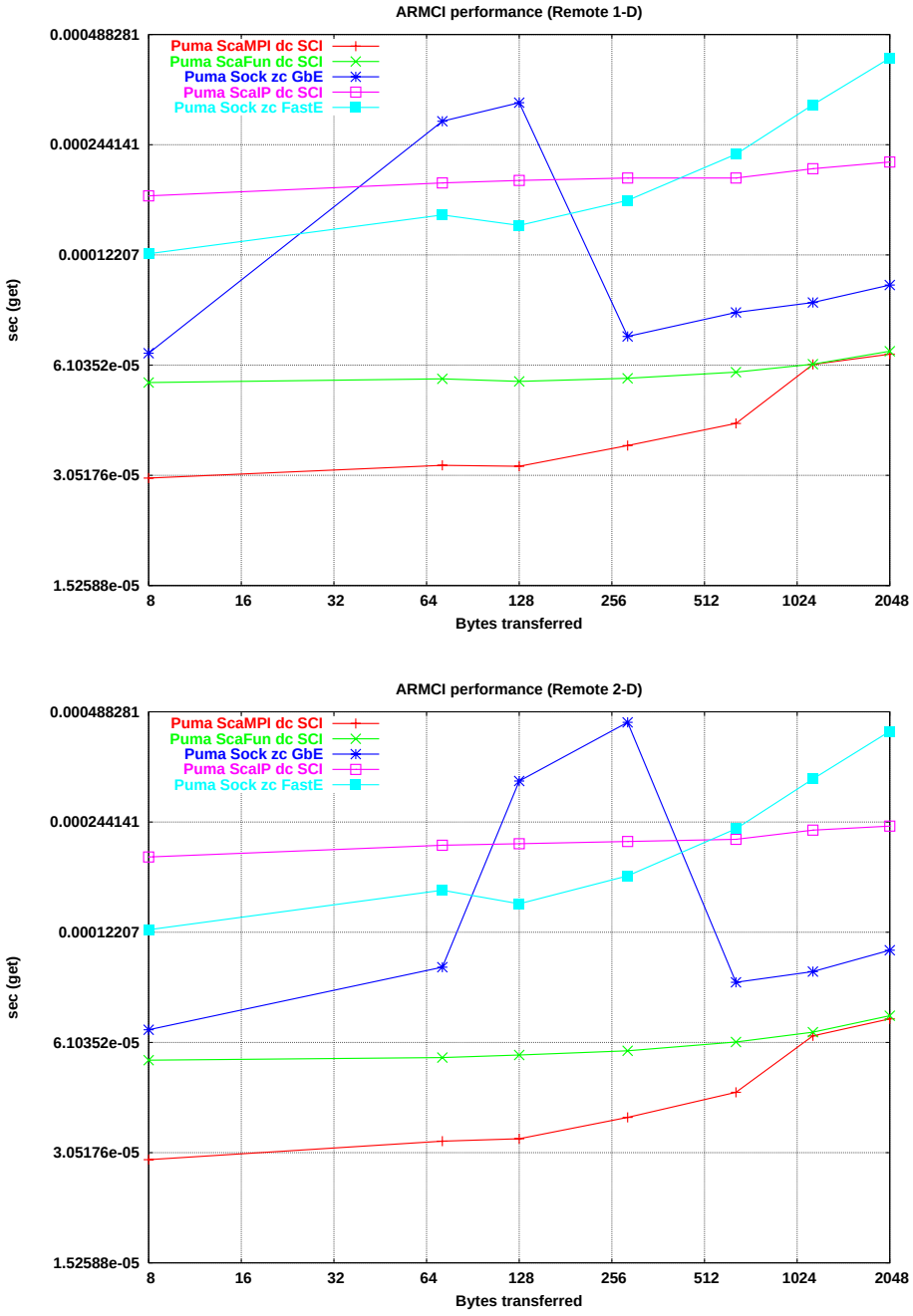
På grunn av de uheldige interrupt-begrensningene i ScaSCI-driveren (beskrevet i avsnitt 6.2.3) kunne jeg ikke kjøre med SCI-interrupter for et større antall prosessorer. SPLASH-2 LU benchmarken ble derfor kjørt med kun 6 prosessorer (på 6 noder) for ARMCI over GbE, ScaMPI (med og uten interrupt) og ScaFun-versjonen (med interrupt). Jeg har valgt å ikke vise resultatene for ARMCI over ScaIP og Fast Ethernet da disse (som indikert i 7.4.1) har en dårligere eller like god ytelse som GbE.

For NWChem ble testene kun kjørt med ARMCI over GbE og ScaMPI (uten interrupter). Her ble det kjørt med 2-16 prosessorer. Siden ARMCI benytter en egen server-tråd ble disse benchmarkene

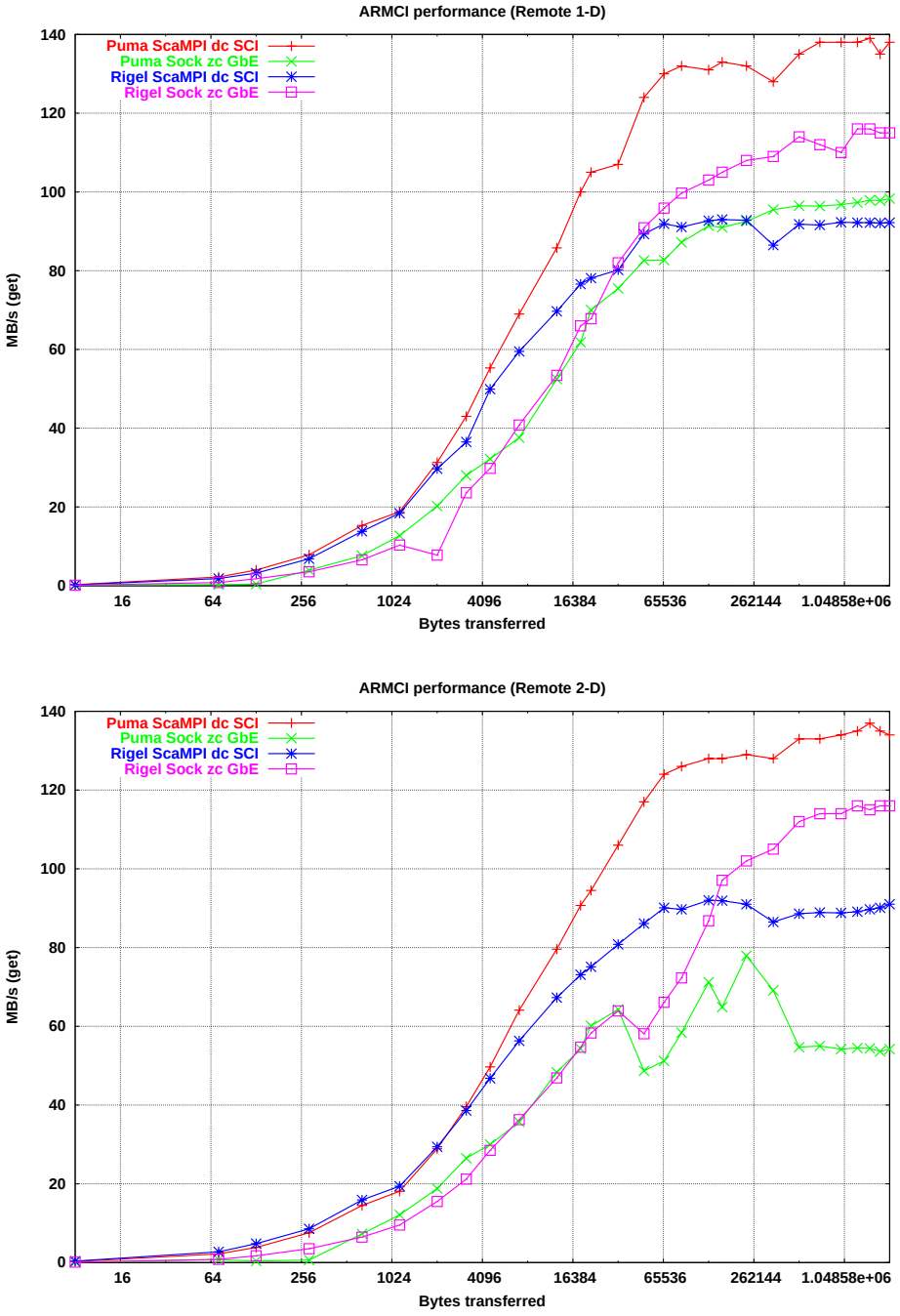
[2]Det understrekes at disse testene ikke er direkte sammenliknbare ettersom en get-request ikke nødvendigvis sender like store pakker begge veier. Det er også viktig å merke seg at disse testene oppgir resultatet som et gjennomsnitt av et antall iterasjoner. Resultatene er således også influert av LogGP parameteren g.



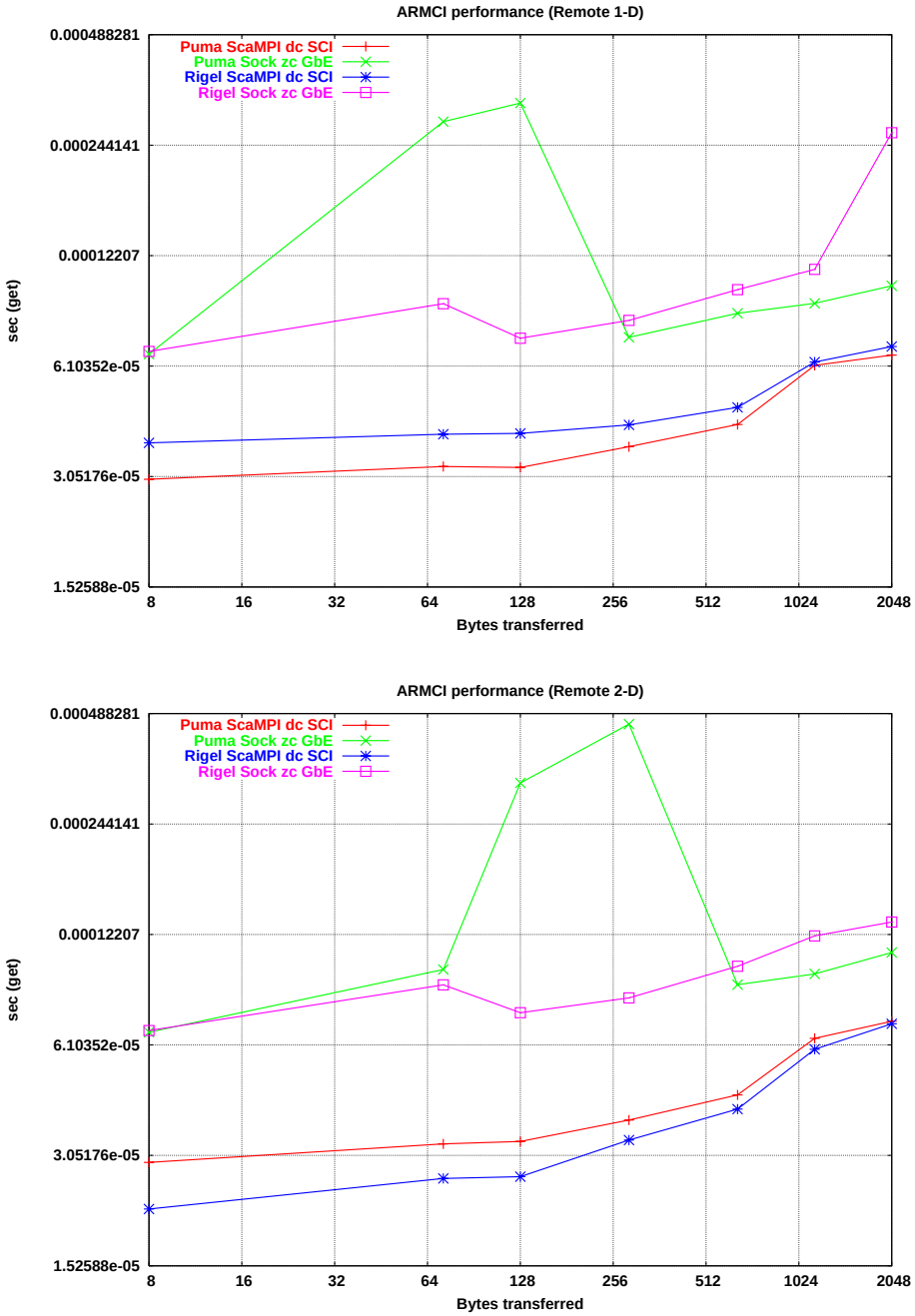
Figur 7.1: **Båndbredde for get-requester.** *(Log x)* Sammenlikning av båndbredde for sekvensielle og ikke-sekvensielle dataoverføringer (get-operasjoner) ved bruk av ARMCI over ScaMPI, ScaFun, GbE, ScaIP og Fast Ethernet. Som man kan se av figuren gir bruk av ScaMPI den beste ytelsen, tett etterfulgt av implementasjonen over ScaFun. Legg spesielt merke til ytelsesreduksjonen for GbE ved introduksjon av ikke-sekvensielle data.



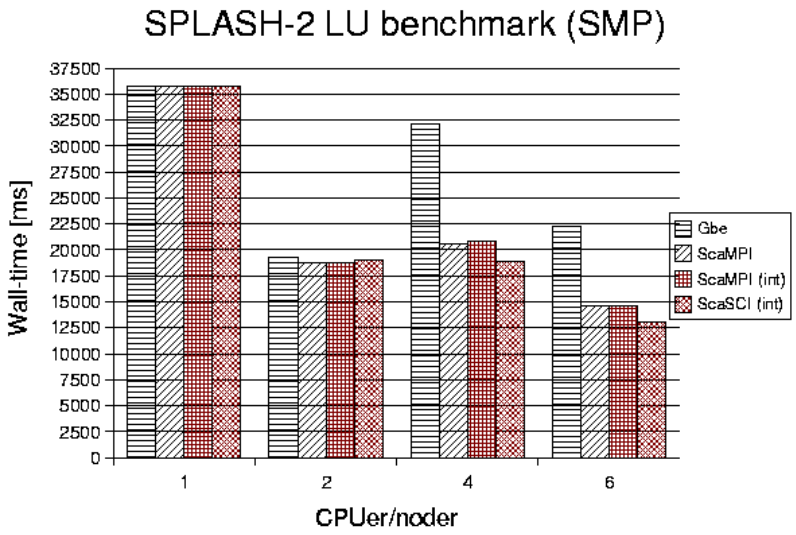
Figur 7.2: **Forsinkelse for get-requester.** *(Log y)* Figurene sammenlikner forsinkelse ved overføringer av mindre mengder sekvensielle og ikke-sekvensielle data (get-operasjoner). Lavest forsinkelse oppnår ARMCI-implementasjonen over ScaMPI og SCI. Høyest forsinkelse (og overhead) introduseres av ScaIP.



Figur 7.3: **Sammenlikning av båndbredde på Puma og Rigel.** *(Log x)* Figurene viser en sammenlikning av ARMCI-båndbredden (sekvensielle og ikke-sekvensielle data) på Puma og Rigel clustrene (se tabell 7.1 på side 82 for konfigurasjonsdetaljer). Som man ser av figuren gir Puma-clusteret best ytelse over SCI (ScaMPI) mens Rigel gir best GbE ytelse.



Figur 7.4: **Sammenlikning av forsinkelse på Puma og Rigel.** *(Log y)* Figurene sammenlikner forsinkelsen forbundet med get-requester for Rigel og Puma clustrene ved bruk av ScaMPI over SCI og GbE. Puma-maskinene gir lavest forsinkelse ved bruk av ScaMPI, mens Rigel-maskinen yter best for GbE. Legg merke til forskjellen mellom resultatene for GbE for de to clustrene.



Figur 7.5: **SPLASH-2 LU, dedikerte prosessorer for kommunikasjon.** Figuren viser kjøretiden (wall-time) for SPLASH-2 LU benchmarken ved bruk av henholdsvis 1, 2, 4 og 6 2-CPU SMP-noder. Implementasjonene som benytter SCI-interrupter er merket med *(int)*. Implementasjonen over ScaFun er merker *ScaSCI*. Det ble kjørt en arbeidsprosess pr. node, den andre CPU-en sto således fritt til å bli benyttet av ARMCI sin server-tråd for kommunikasjon. Økningen i kjøretid for samtlige implementasjoner ved bruk av 4 fremfor 2 prosessorer har trolig med en uheldig økning av applikasjonens (algoritmens) behov for kommunikasjon å gjøre.

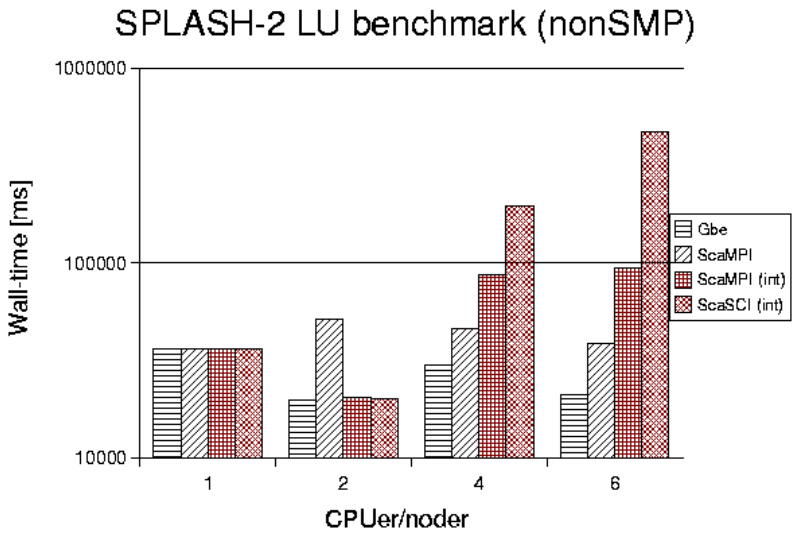
også kjørt med og uten Hyper-Threading (HT).

SPLASH-2 LU

SPLASH-2 LU benchmarken ble kjørt etter samme retningslinjer som beskrevet av Nieplocha i [105]. Følgelig ble det benyttet en matrisestørrelse på 3072 elementer og en bokkstørrelse på 32 elementer. Matrisestørrelsen reflekterer størrelsen på problemet (matrisestørrelse^2), mens blokkstørrelsen indirekte angir størrelsen på requestene/meldingene som utveksles mellom prosessene (blokkstørrelse^2). Benchmarken ble kjørt med fast problemstørrelse da jeg ønsket å se skalering etter Amdahls lov.

Figur 7.5 viser SPLASH-2 LU kjøringer for ARMCI over GbE, og MPI (ScaMPI med polling-receive og med SCI-interrupter). Implementasjonen over ScaFun ble utelukkende kjørt med SCI-interrupter. Benchmarken ble kjørt på 6 SMP-noder med en regneprosess pr. node. Den andre prosessoren ble effektivt benyttet for kommunikasjon (*schedulert* av operativsystem). Ettersom prosessantallet øker kan man se hvordan både implementasjonene over ScaMPI og ScaFun skalerer bedre enn implementasjonen over GbE, når en ekstra kommunikasjonsprosessor stilles til disposisjon.

Figur 7.6 viser samme benchmarken kjørt, men uten bruk av en egen prosessor for kommunikasjon. SMP-funksjonaliteten i nodene ble ugyldiggjord ved å benytte en ikke-SMP Linux kjerne. Jeg så meg nødt til å gjøre testen på denne mindre realistiske måten for å kunne generere maksimal kommunikasjon (over flere noder) på grunn av interrupt-begrensningene i ScaSCI-driveren. Som det fremgår av figuren gir dette et mindre heldig bilde av skaleringen for de nye ARMCI-implementasjonene. I denne testen kommer poenget om hvordan server-tråden kan stjele CPU-sykler fra applikasjonen godt frem i resultatene for implementasjonen over ScaMPI (med polling-receive). Implementasjonen skale-



Figur 7.6: **SPLASH-2, en enkelt prosessor pr. node.** *(Log y)* Figuren viser kjøretiden (wall-time) for SPLASH-2 LU benchmarken ved bruk av henholdsvis 1, 2, 4 og 6 1-CPU noder. Implementasjonene som benytter SCI-interrupter er merket med *(int)*. Implementasjonen over ScaFun er merker *ScaSCI*. Både arbeidsprosessen og server-tråden benyttet samme CPU.

rer fortsatt brukbart, men den totale kjøretiden er langt dårligere enn implementasjonen sover Sockets og GbE.

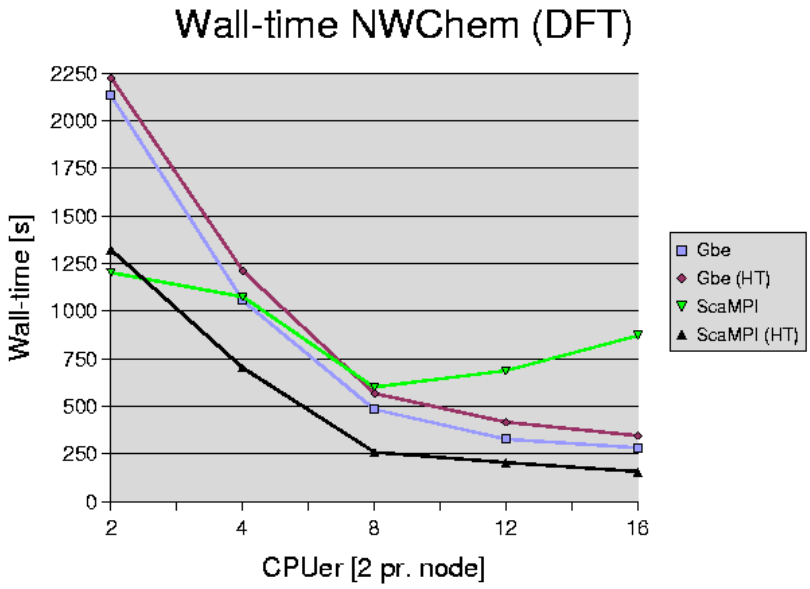
For ARMCI-implementasjonene som benytter SCI-interrupter ser man en annen interessant utvikling. Begge implementasjoner yter godt på to noder, men skalerer "feil" vei ved introduksjon av flere noder. Jeg har god grunn til å tro at dette har med den store overheaden forbundet med prosessering av SCI-interrupter og følger av disse. Interruptene forårsaker prosessbytte (*context-swithes*) i operativsystemet og følgelig dårlig synkronisering i applikasjonen. Som en følge av dette valgte jeg å forlate løsningene basert på SCI-interrupter ved videre testing. Jeg konsentrere meg heller om den både mer stabile og portable løsningen utelukkende basert på MPI, uten bruk av interrupter.

NWChem

For testing av NWChem-applikasjonen valgte jeg å benytte kun ARMCI-Sockets over GbE og ARMCI over MPI ved hjelp av ScaMPI. Som en følge av de dårlige skaleringsresultatene fra SPLASH-2 LU benchmarken (figur 7.6) ønsket jeg også å se på den alternative prosessormodusen Hyper-Threading(HT, se avsnitt 2.3.3) for å se om denne kunne hjelpe til å minimalisere overheaden forbundet med ScaMPI sin polling.

NWChem[110] beskrives som en såkalt computational chemistry pakke designet for å kjøre på høyytelses superdatamaskiner og konvensjonelle clustere. Programmet sikter på å være skalerbart både med hensyn på effektiviteten ved løsing av større problem, samt å utnytte mest mulig tilgjengelige parallelle ressurser. NWChem er utviklet av Environmental Molecular Sciences Laboratory (EMSL)[41] ved Pacific Northwest National Laboratory (PNNL)[119].

NWChem er et stort og komplisert program estimert til over en million linjer kode. Applikasjonen tilbyr mange metoder for å beregne egenskaper ved molekulære og periodiske system ved hjelp av kvantemekaniske beskrivelser av elektroniske bølgefunksjoner og tetthet. I tillegg har NWChem



Figur 7.7: **Wall-time for NWChem (DFT).** Figuren viser kjøretid (wall-time) for NWChem og DFT algoritmen. Kjøringene sammenlikner bruk av ARMCI over GbE og SCI (gjennom ScaMPI) og også bruk av Hyper-Threading (merket HT). Av figuren fremgår det at beste ytelse oppnås ved bruk av ScaMPI over SCI og HT.

muligheten til å foreta klassisk molekylær dynamikk og frie energisimuleringer. Typiske algoritmer benyttet er; Self Consistent Field (SCF [84, s.57]) eller Hartree Fock (RHF, UHF) og Gaussian Density Functional Theory (DFT [84, s.177]).

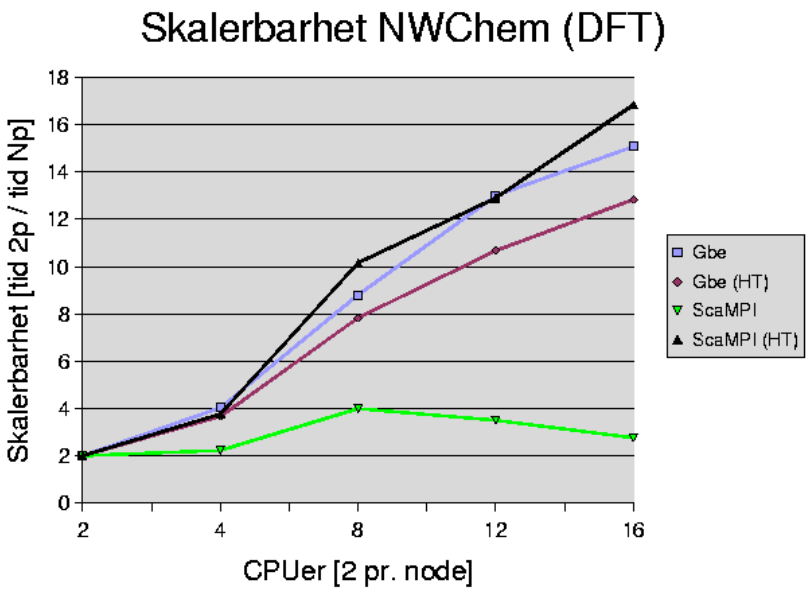
NWChem (og GAMESS-UK) har i hovedsak to typer av algoritmer som benyttes, DFT og SCF. Disse metodene sikter på å løse samme type problem, men har store forskjeller i fremgangsmåte og underliggende matematisk teori. Forskjellene gjør seg blant annet gjeldende i aksessmønster og frekvens i de distribuerte GA-matrisene.

Ved test av DFT benchmarken ble molekylstrukturen $SIOSI_3$ benyttet da NWChem sin dokumentasjon [110] selv benytter dette datasettet for kjøring av benchmarker.

Figur 7.7 viser wall-time for DFT benchmarken. Kjøringer både med og uten Hyper-Threading (HT) er plottet. Et interessant resultat av disse kjøringene er at ARMCI-Sockets-implementasjonen går betydelig dårligere på to prosessorer på samme SMP-node enn MPI-implementasjonen. Jeg har ikke klart å finne en god forklaring på denne effekten da hverken Sockets- eller MPI-biblioteket er involvert i disse kjøringene, kun ARMCI sine egne shared memory-operasjoner. Jeg har derfor valgt å fokusere på resultatene ved bruk av 8 til 16 prosessorer.

Som man kan se av figuren gir ARMCI over ScaMPI klart lavest kjøretid ved bruk av HT. Uten bruk av denne funksjonaliteten ser man hvordan økt overhead gjennom ScaMPI-biblioteket fører til en ineffektiv eksekvering og følgelig økning i kjøretid. En interessant observasjon er videre at bruk av HT for Sockets-implementasjonen kun fører til en mindre økning i kjøretid, mens det for MPI-versjonen fører til en kraftig forbedring. For 16 prosessorer gir dermed ARMCI over ScaMPI en kjøretid (wall) på 156.9s ved hjelp av HT mot GbE sine 300.2s uten bruk av HT. En nær halvering av kjøretiden!

Figur 7.8 viser skalerbarheten av DFT kjøringene. Foruten kjøringen av ARMCI over ScaMPI



Figur 7.8: **Skalerbarhet for NWChem (DFT).** Figuren viser skalerbarheten for NWChem og DFT algoritmen (regnet ut fra tallene i figur 7.7). Legg merke til skaleringsforskjellen for ARMCI over ScaMPI med og uten bruk av Hyper-Threading (merket HT). Bruk av ScaMPI, SCI og HT fører til nær perfekt skalering.

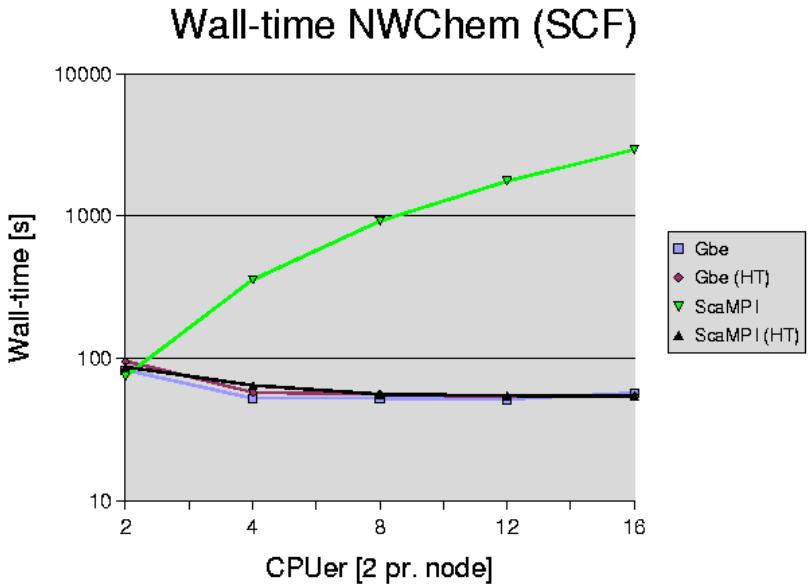
uten bruk av HT, ser man en nær perfekt (etter Amdahls lov) skalerbarhet for NWChem.

For kompletthets skyld har jeg også valgt å ta med resultatene for NWChem kjøringer ved bruk av SCF-algoritmen. Jeg har her valgt å benytte datasettet $Feco5$. Det ble ikke forventet en særlig ytelsesøkning ved disse kjøringene da denne algoritmen ikke benytter på langt nær like mye mellomprosesskommunikasjon som DFT-algoritmen. Figur 7.9 viser wall-time for SCF benchmarken. Som man ser av figuren har jeg minimalt å hente ved bruk av ARMCI over ScaMPI i forhold til Sockets og GbE. For 16 prosessorer har ARMCI over ScaMPI en kjøretid på 54.5s mens GbE kjøringen avslutter etter 57.4s (54.8 med HT). De mest interresante observasjonene med disse kjøringene er den enorme forbedringen av resultater ved bruk av HT for ARMCI over ScaMPI. Samtidig er forskjellen mellom bruk og ikke bruk av HT for kjøringene over GbE er minimale. Dette har med mengden kommunikasjon benyttet av SCF algoritmen i forhold til DFT algoritmen å gjøre.

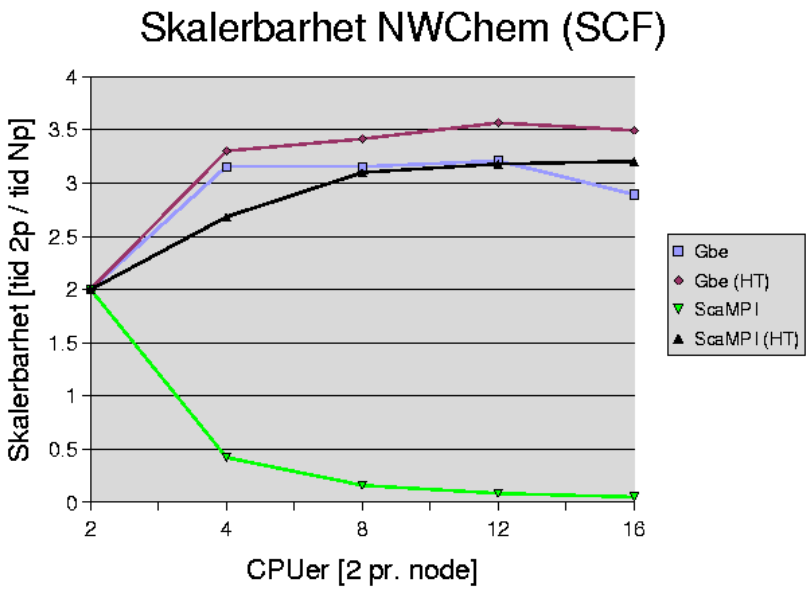
Figur 7.10 viser skalerbarhet for SCF-algoritmen. Som man kan se av grafen har man minimalt å oppnå ved økning av ressurser. Dette gjelder både mine prototypeimplementasjoner og bruk av den originale Sockets-implementasjonen over GbE. Jeg har derfor grunn til å tro at SCF-algoritmen og applikasjoner basert på denne, i liten grad er skalerbare.

7.5 Diskusjon

ARMCI-kommunikasjonen følger en LogGP-modell ved at kommunikasjonen kan analyseres basert på modellens parametere (se avsnitt 3.1). De fire ARMCI-implementasjonene over GbE (Sockets), ScaIP (Sockets), ScaMPI (SCI) og ScaFun (SCI) har alle styrker og svakheter i forhold til denne



Figur 7.9: **Wall-time for NWChem (SCF).** *(Log y)* Figuren viser kjøretid (wall-time) for NWChem og SCF algoritmen. Kjøringene sammenlikner bruk av ARMCI over GbE og SCI (gjennom ScaMPI) og også bruk av Hyper-Threading (merket HT). Av figuren fremkommer det hvordan bruk av ScaMPI uten HT fører til en økning i kjøretiden ved økt antall noder. Figuren indikerer at SCF-algoritmen benytter lite kommunikasjon og har en dårlig skalering.



Figur 7.10: **Skalerbarhet for NWChem (SCF).** Figuren viser skalerbarheten for NWChem og SCF algoritmen (regnet ut fra tallene i figur 7.9). Legg merke til skaleringsforskjellen for ARMCI over ScaMPI med og uten bruk av Hyper-Threading (merket HT). Som det også ble indikert i figur 7.9 har SCF-algoritmen en dårlig skaleringsevne.

modellen.

Implementasjonen over GbE bærer preg av en høy L introdusert av nettverket og betydelig o introdusert gjennom bruk av TCP/IP-protokollen. Versjonen over ScaIP har en lav L ved bruk av SCI, men har en betydelig større o introdusert både av TCP/IP-protokoller og ScaIP-/ScaMAC-modulene. Selv om både GbE og ScaIP tilbyr en høy G, er fortsatt g lav. Med andre ord er den rene båndbredden høy for begge løsninger, men evnen til å sende flere tett etterfølgende pakker er dårlig. Dette kommer av en utvidet bruk av buffering, time-out og interrupter. Svakhetene ved løsningene kommer tydelig frem ved transmisjon av ikke-sekvensielle datamønstere. Selv om ScaIP benytter SCI gir likevel løsningen over GbE den beste ytelsen.

Bruk av ScaMPI over SCI har en lav L og en i utgangspunktet lav o direkte forbundet med sending og mottak. Likefullt lider denne løsningen av å introdusere en betydelig implisitt o gjennom bruk av polling. Det har blitt vist at bruk av ScaMPI over SCI gir den høyeste G og g av implementasjonene testet i denne rapporten.

ARMCI over ScaFun har i likhet med ScaMPI versjonen en meget lav L. Likevel er det to hovedpunkt som avgjør denne implementasjonens ytelse; interrupter og kommunikasjonsprotokoll. Implementasjonen gir en høy G, men introduserer en altfor høy g (og o) ved bruk av SCI-interrupter. Selv om kommunikasjonsprotokollen og dens bruk av synkronisering har innvirkning på o-parameteren, kan dette i stor grad optimaliseres. Når det kommer til SCI-interruptene forblir dette et problem som introduserer en høy o. Et alternativ i så tilfellet er å benytte polling i likhet med ScaMPI.

Jeg har altså avdekket fire hovedproblemer med implementasjoner av ARMCI (og dermed GA) over SCI:

- Høy overhead gjennom bruk av SCI-Interrupt eller polling.
- Kompliserte datatyper (ARMCI-datatyper) og kopieringsprotokoller (zero-, one- eller double-copy).
- Implementasjon av ikke-blokkerende kommunikasjon.
- Krav om thread-safeness.

Bruk av SCI-interrupter ser ut til å introdusere en utolererbar høy overhead ettersom disse behandles både i kernel-space og user-space. Det må også utføres endringer av ScaSCI-driveren for å få bruk av disse til å skalere. En alternativ bruk av polling åpner for kommunikasjon med lave forsinkelser, men også introduserer en økt implisitt overhead. Det har blitt vist (figur 7.6 på side 91) hvordan denne overheaden kan ha stor innvirkning på applikasjonsytelsen ved at den stjeler CPU-sykler fra applikasjonen. Likefullt har det også blitt vist (figur 7.7 på side 92) hvordan denne overheaden i stor grad kan elimineres ved bruk av Hyper-Threading-teknologien.

Som beskrevet tidligere fokuserer ARMCI på kommunikasjon av ikke-sekvensielle datatyper. Dette fører til nødvendigheten av et mer komplisert samarbeide mellom kommunikasjons- og kopieringsprotokoller. ARMCI-Sockets har forsøkt å benytte Sockets-biblioteket for pakking av ikke-sekvensielle data ved hjelp av flere *write()*-kall. Ettersom Sockets benytter buffering ikke kontrollerbar av bruker, har jeg vist hvordan dette kan slå uheldig ut over Gbps-forbindelser (se avsnitt 4.3).

Med en antagelse at MPI-datatyper er implementert som zero-copy har jeg forsøkt å utarbeide en oversettelse mellom ARMCI- og MPI-datatyper (se tillegg D). Dette har vist seg å være vanskelig ettersom (til forskjell fra ARMCI-datatyper) MPI-datatyper må ha samme type både for sender og mottaker. Jeg vil således stille spørsmål ved hvorvidt konvertering (med påfølgende overhead) av disse datatypene rettferdiggjøres med en resulterende zero-copy implementasjon.

Ettersom SCI-adapterene kan benytte write-gathering i hardware, mener jeg implementasjonen med ScaFun har det største potensialet. Data kan her effektivt pakkes direkte mot nettverket på en effektiv måte. Likevel vil en ren zero-copy implementasjon bli vanskelig og/eller lite skalerbar. Dette kommer av at man kun kan adressere en begrenset del av de enkelte nodenes minne og at dynamisk pinning av minne er dyrt (se avsnitt 4.2). Jeg ser således for meg en one-copy-løsning.

Selv om Dolphins SCI-adaptere har en dårligere ytelse ved bruk av DMA fremfor PIO (figur 3.4 på side 35), er dette et alternativ man bør vurdere for ARMCI sine ikke-blokkerende operasjoner. For implementasjonen over MPI har man ikke mulighet til å direkte styre kommunikasjonsmetodene. Jeg har forsøkt å benytte MPI-funksjoner som *MPI_Isend()* og *MPI_Bsend()* med ScaMPI, men med liten suksess. Dette fordi ScaMPI over SCI utelukkende benytter PIO (se avsnitt 5.1). For andre MPI-implementasjoner kan dette muligens være et alternativ. En nærmere styring av kommunikasjonsmetoden er mulig for implementasjoner over ScaFun. I så tilfelle bør man benytte flere forskjellige kommunikasjonsprotokoller som varierer for både datastørrelser og operasjoner.

På grunn av den trådbaserte klient-server-konstruksjonen i ARMCI (se avsnitt 4.1.4), er thread-safeness av kommunikasjonsbibliotekene nødvendig. Dette betyr at Sockets, MPI og ScaFun-implementasjonene må kunne håndtere tilfeller hvor både klientprosesser og server-tråd kommuniserer samtidig. Implementasjoner av kommunikasjonsbibliotek som ikke støtter slik funksjonalitet (for eksempel MPICH) er følgelig ubrukelige.

7.6 Oppsummering

Gjennom verifikasjonstestene har jeg vist at de nye implementasjonene av ARMCI utfører sine operasjoner semantisk riktig. Disse verifikasjonen ble utført for flere mindre applikasjoner og for større applikasjoner som NWChem og GAMESS-UK. Jeg har således dannet et grunnlag for relevante ytelsestester.

Jeg har valgt å konsentrere meg om et mindre antall ytelsestester. Måling av ensidig kommunikasjon ved hjelp av mikrobenchmarker har flere usikkerhetsmomenter og gir derfor kun indikasjoner på ytelsen. For eksempel ser man liten forskjell på ytelsen for ARMCI over ScaMPI med og uten bruk av interrupter ved bruk av ARMCI_perf benchmarken. Forskjellene kommer mer tydelig frem i SPLASH-2 LU benchmarken. Denne kjører likefullt et relativt enkelt og statisk kommunikasjonsmønster og gir liten realisme av den totale ytelsen. Jeg anser derfor NWChem og DFT-algoritmen med wall-time som målestokk som den beste ytelsestesten. Dette er den mest realistiske og grundigste benchmarken av de tre.

Jeg har altså valgt kun å benytte eksisterende benchmarker fremfor utvikling av egne. Dette er gjort da det er vanskeligere å få aksept for egenutviklede benchmarker.

Dessverre har jeg ikke hatt mulighet til å kjøre referansetester mot Myrinet-implementasjonen da jeg ikke hadde tilgang på et slikt cluster. Det understrekes at en sammenlikning med eksterne resultater fra kjøringer på andre maskiner ikke er heldig. Som vist ved sammenlikningen av Rigel og Puma clustrene har både CPU og chipset stor innflytelse nettverksytelsen (både for Myrinet, SCI og GbE).

Det viktigste punktet i dette kapittelet er likefullt at jeg oppnådde den beste NWChem-ytelsen ved bruk av ScaMPI over SCI ved hjelp av Hyper-Threading. Merk at dette ble gjort på maskiner med et chipset som ikke er optimalt for SCI. En endring av ytelsen kan således forventes ved bruk av maskiner med annet chipset.

Kapittel 8

Konklusjon og videre arbeide

Ensidig kommunikasjon i clustere er foreløpig lite beskrevet i akademiske artikler. Likefullt har man sett nødvendigheten av slike programmeringsabstraksjoner, noe som er understreket med MPI-2 standarden. Jeg vil i dette kapittelet forsøke å peke på noen av de prosjektene og implementasjonene relatert til ensidig kommunikasjon i clustere og ARMCI.

Videre legges konklusjonen frem før det avsluttes med en avsnitt om erfaringer og videre arbeide.

8.1 Relatert arbeid

ARMCI skiller seg fra andre liknende system i på flere nøkkelpunkter. Ulikt implementasjoner som primært støtter RMA-operasjoner med kontinuerlige data (for eksempel Cray SHMEM[22], LAPI[71] eller Active Message run-time-system for Split-C, som inkluderer put-/get-operasjoner[25]), prøver ARMCI å effektivisere ikke-kontinuerlige dataoverføringer. Cray SHMEM-biblioteket, tilgjengelig på Cray og SGI plattformer og clustere gjennom HPVM[2] og Huses ScaShMem[65], støtter kun put-/get-operasjoner for kontinuerlige data.

Spesifikasjonen av MPI-2[95] ensidet kommunikasjon inkluderer RMA-operasjoner som put og get. Ikke-kontinuerlige dataoverføringer er fullt støttet gjennom MPI-datatypene. MPI-2 spesifikasjonen baseres på to modeller for ensidig kommunikasjon; *active-target* og *passive-target*. Active-target modellen er avledet fra meldingsutvekslingsprinsipper og dens semantikk inkluderer relativt strenge regler. Passive-target modellen benytter enklere og mer avslappede regler, men kan introdusere potensielle ytelsesbegrensninger på grunn av låser og forbud mot aksess av overlappende destinasjonsområder i minnet. ARMCI har ingen liknende restriksjoner og tilbyr dermed en enklere programmeringsmodell enn MPI-2.

Som beskrevet i avsnitt 3.4.4 har Huse utviklet et bibliotek som muliggjør kjøring av SHMEM-baserte applikasjoner over tosidig MPI. Denne implementasjonen likner mye på grunndesignet i implementasjonen beskrevet i denne oppgaven (ARMCI over MPI). Huse benytter i stor grad zero-copy for sin kommunikasjon, men har sluppet å bry seg om ikke-sekvensielle overføringer. ScaShMem er også laget mer utfra et portabilitetshensyn, fremfor et ytelseshensyn, i at den tar sikte på å benytte en egen dedikert prosessor i SMP-noden for kommunikasjon.

En liknende implementasjon har blitt gjort av Nieplocha et. al. for SHMEM over ARMCI kalt GPSHMEM[108]. Denne implementasjonen benytter ARMCI for SHMEM sine ensidige operasjoner og MPI, PVM[45] eller TCGMSG for kollektive operasjoner.

Som nevnt har Nieplocha et. al.[105] også utviklet en utvidelse av ARMCI som benytter Myrinet-interconnectet gjennom GM-protokollen[104]. Her utnyttes Myrinet-adapterens DMA-maskin blant

annet gjennom ikke-blokkerende put-/accumulate-operasjoner og en interrupt-basert server-tråd. Implementasjonen kan vise til resultater som indikerer både effektivitet og skalerbarhet. Dessverre er denne implementasjonen både komplisert og lite portabel sammenliknet med implementasjoner over MPI- og Sockets-bibliotekene. Det ble også målt en betydelig dårligere ytelse ved bruk av TCP/IP over Myrinet.

Scali AS har den siste tiden forsøkt å videreutvikle sitt ScaFun bibliotek til å inkludere minnekopieringsoperasjoner for RMA over SCI. Disse operasjonene inkluderer funksjoner som *localPut(), remotePut, localGet() og remoteGet()*. Jeg har desverre ikke hatt mulighet til å teste disse implementasjonenene enda. Likefullt kan slike funksjoner være interessante både i ARMCI- og MPI-2-sammenheng.

De overnevnte API og implementasjoner av ensidig kommunikasjon er basert på mer eksplisitte programmeringsmodeller. Det finnes også prosjekter som tar sikte på et mer implisitt grensesnitt til distribuerte datastrukturer og ensidig kommunikasjon. I denne sammenheng kan jeg for eksempel nevne TreadMarks[86]. TreadMarks-implementasjonen forsøker å i større grad maskere clusterarkitekturen ved hjelp av mappinger mellom lokale memory-pager og ikke-lokalt minne. Applikasjonen kjører får dermed en programmeringsmodell nærmere generell shared memory- og SMP-arkitektur fremfor en mer distribuert cluster eller MPP-arkitektur.

8.2 Konklusjon

Lavkostnads-clustere har i dag blitt akseptert som et godt alternativ til store SMP- og MPP-maskiner. Den dominerende programmeringsmodellen i slike clustere er en tosidig meldingsutvekslingsmodell fremfor en shared memory-modell. Likevel er enkelte algoritmer vanskelig å implementere i en slik modell. Global Arrays har gjennom ARMCI gjort det mulig å også benytte en ensidig programmeringsmodell for cluster, i tillegg til den standardiserte tosidig modellen.

Jeg har i denne rapporten vist at implementasjoner av Global Arrays fungerer over SCI. Gjennom benchmarking har jeg likefullt vist at overhead gjennom underliggende kommunikasjonsbibliotek er avgjørende for ytelsen. Jeg har vist at bruk av TCP/IP-implementasjoner som ScaIP over SCI introduserer en høy overhead og gir dårligere ytelse enn allerede eksisterende ARMCI-implementasjoner over GbE.

Jeg har videre utviklet en ARMCI-implementasjon utelukkende over den tosidige meldingsutvekslingsstandarden, MPI. Denne kan benyttes over SCI ved hjelp av MPI-implementasjonen ScaMPI. Selv om denne implementasjonen ikke er optimal med hensyn på minimal datakopiering og benytter en polling-receive, har jeg vist en betydelig ytelsesøkning for både båndbredde og forsinkelse. Jeg har gjennom benchmarking fremsatt resultater som viser en høyere ytelse sammenliknet med tidligere løsninger. For enkelte applikasjoner har jeg også sett en nær perfekt skalerbarhet. Jeg har også vist hvordan prosessorteknologi som Hyper-Threading sammen med polling-receive kan være et effektivt alternativ til interrupt-baserte mottak av requester.

Jeg har også implementert ARMCI direkte over SCI ved hjelp av ScaFun. Denne implementasjonen ser ut til å ha størst potensiale når det kommer til selve datakommunikasjonen og bruk av zero- eller one-copy protokoller. Likefullt er faller implementasjonen i denne oppgaven på den høye overheaden forbundet med SCI-interrupter. Høy prosesseringstid for disse introduserer både økt forsinkelse og en uhensiktsmessig skalerbarhet. Det har tydelig blitt understreket at en slik implementasjon hverken er enkel eller effektiv uten tredjepartsdrivere og minnekopieringsrutiner for SCI-kommunikasjon.

Ved bruk av Hyper-Threading-teknologien og en nødvendig thread-safe MPI-implementasjon har jeg vist en minimal introduksjon av overhead gjennom bruk av MPI, men samtidig fått en løsning med

høy portabilitet. Selv om den rene SCI-løsningen har større ytelsesmessig potensiale, konkluderes det med at gevinsten ved en slik implementasjon ikke kan rettferdiggjøres med tanke på ytelse og portabilitetsaspektene ved en MPI-implementasjon av ARMCI.

8.3 Erfaringer og videre arbeide

I denne siste del av oppgaven oppsummeres de problemer jeg har støtt på gjennom arbeidet med denne oppgaven. Videre peker jeg på fordeler med prototypeimplementasjonene utført i forbindelse med denne rapporten og hvilke deler som bør endres eller sees nærmere på.

8.3.1 SCI-grensesnittet

SCI sine lave overhead-parametere, øye båndbredde, lave hardware-forsinkelse i LogGP kan sammen med sitt, i utgangspunktet enkle, brukergrensesnitt fremstå som et nær perfekt interconnet for clustere. Likevel understreker Huse gjennom hele sin doktoravhandling[68] at effektiv kommunikasjon over SCI ikke er en enkel affære. Hensyn må tas til hardware-komponenter som prosessorer, chipset, SCI-adaptere og topologi. Disse faktorene fører med seg nødvendigheten av spesialdesignede minnekopieringsrutiner, kommunikasjonsprotokoller og tunge driverimplementasjoner.

At man for å oppnå effektiv kommunikasjon også bør bruke PIO fremfor DMA og samtidig en remote-write-read-local politikk for SCI øker kompleksiteten av SCI-kommunikasjon. Ved første øyekast kan enveis kommunikasjon over SCI virke enkelt ettersom man på sendersiden kan sjekke hvorvidt dataene kom frem og dermed også garantere deres riktighet. Likefullt har SCI ikke garantert rekkefølge på sine overføringer og man må således fortelle mottakeren at dataene har kommet frem enten ved interrupter eller ved hjelp av periodisk polling og dermed en tosidig kommunikasjonsprotokoll. One-copy implementasjoner blir vanskeliggjort ved at SCI-adaptere tilkoblet I/O-bussen kun kan aksessere fysisk minne, har begrenset adresseområde og bruk av dynamisk pinning skaper stor overhead.

8.3.2 Portabilitet av ARMCI

ARMCI-biblioteket blir av Nieplocha[106] fremsatt som et lavnivå bibliotek, hvor intensjonen har vært å lage utvidelser over andre lavnivå interconnect-API og -drivere. I tillegg understreker han bruk av skreddersøm ved bruk av zero-/one-copy implementasjoner for det enkelte interconnect. For SCI gir et slikt rammeverk muligheter for en effektiv og høyytelses implementasjon, men samtidig med en betydelig utviklingstid. Skal man ta hensyn til alle faktorer for SCI uten bruk av eksterne bibliotek (annet enn SCI-driver), vil kompleksiteten bli høy og portabiliteten lav. Samtidig vil lite av denne implementasjonen direkte kunne dras nytte av ved implementasjoner over nye interconnect som for eksempel InfiniBand.

Som en følge av disse problemene ser jeg derfor for meg en noe større fokus på portabilitet ved videreutvikling av ARMCI-biblioteket. Implementasjonen over MPI presentert i denne oppgaven er et eksempel på dette. Samtidig er MPI relativt abstrakt og gir kun en begrenset kontroll av kommunikasjonen. Et mer lavnivå grensesnitt passende for ARMCI kan derfor være DAT (Direct Access Transport), utarbeidet av The DAT Collaborative[137]. DAT er et i utgangspunktet tynt API implementert rett over eller i driveren for interconnectet. DAT er et forsøk på en standardisering av drivergrensesnitt og RDMA-operasjoner (Remote Direct Memory Access). Grensesnittet kan dermed tilby ARMCI full kontroll over interconnectet samtidig som portabiliteten opprettholdes. En implementasjon av ARMCI over DAT bør derfor implementeres og optimaliseres med hensyn på dette grensesnittet. Et problem

i så måte er thread-safeness. På samme måte som for ARMCI over MPI forutsetter ARMCI sitt u-avhengige tråd-design et thread-safe kommunikasjonsbibliotek. Desverre er DAT-spesifikasjonen noe uklar på dette punktet.

Et annet argument for mer generelle API som DAT eller MPI er introduksjonen av såkalt *Grid computing*. Konseptet med Grid computing er å kunne utnytte flere kraftige parallelle maskiner som en. Med andre ord dynamisk skape et slags meta-cluster av parallelle datamaskiner. Konseptet åpner således for bruk av forskjellige interconnect innenfor deler av samme meta-cluster og forutsetter således at kommunikasjonsbibliotekene kan forholde seg til dette. Implementasjonen av ARMCI over MPI vil for eksempel kunne kjøre på et slikt meta-cluster. Det finnes flere MPI-implementasjoner for Grid miljø (*meta-computing*) for eksempel Stampi[74], MPI-Glue[121], MPICH-G2[47], PACX-MPI (Extending MPI for Distributed Computing)[49] og MPI_Connect[43].

Selv om det foreløpig kun finnes et begrenset antall MPI-2 implementasjoner, bør man også se nærmere på muligheten for bruk av MPI-2 sitt API for ensidig kommunikasjon som basis for ARMCI. En slik implementasjon vil i mindre grad være avhengig av synkroniseringen nødvendig i MPI. MPI-2 vil således kunne være mer effektiv som underliggende bibliotek, samtidig som portabiliteten opprettholdes.

8.3.3 Dataflytting og kommunikasjonsprotokoller

Som nevnt er det tre hovedmåter å flytte data fra lokalt minne til ikke-lokalt minne gjennom et interconnect; zero-copy, one-copy og double-copy. Det er flere faktorer som er med på å bestemme både hva som er mest effektivt og hva som er enklest å implementere.

Zero-copy regnes teoretisk som den mest effektive metoden. Likevel garanterer ikke denne metoden best ytelse i alle tilfeller. Ved kommunikasjon av ikke-sekvensielle data begynner flere faktorer å gjøre seg gjeldende; CPU-cache, datamønster og g, G og o parameterene i LogGP for det underliggende interconnect. For ikke-sekvensielle data kan en zero-copy implementasjon for eksempel implementeres ved hjelp av flere mindre pakker som sendes i sekvens. Dette benyttes i Sockets-implementasjonen av ARMCI. Man må altså beregne hvorvidt tiden og overheaden ved å sende flere små meldinger er mindre enn den ved å kopiere de dataene til et sekvensielt buffer for så å sende dette som en stor melding. Bruk av småmeldinger vil avhenge av systemets o og g mens de større meldingene i større grad vil avhenge av G.

En annen faktor som gjør seg gjeldende er cache-bruk. En zero-copy implementasjon som leser fra flere steder i minnet vil i mindre grad utnytte cachen effektivt enn en sekvensiell kopiering. For eksempel vil man ved bruk av send-buffere i praksis skrive direkte fra cache til interconnectet. Ved bruk av receive-buffere vil man nye godt av den samme effekten og kunne lese inne bufferet i cache før det kopieres rett fra cache og ut i datastrukturen. Effektiviteten av CPU-ens cache vil være avhengig av dens størrelse.

Nytten av zero-copy protokollen kommer således til syne først ved større meldinger og får først full effekt ved store meldinger. Bruk av one- og double-copy protokoller kan således rettferdiggjøres ved små og mellomstore meldinger, spesielt når man tar hensyn til implementasjonstid. Zero- eller one-copy protokoller bør likevel gi økt båndbredde og muligens mindre forsinkelse ved sekvensielle dataoverføringer. Slike protokoller bør derfor implementeres i ARMCI. Man bør også ta hensyn til at put- og accumulate-requester i ARMCI ikke er blokkerende operasjoner. En buffering på sendersiden kan således være mer effektivt for ikke-blokkerende requester.

Som Huse[66] allerede understreker, er gode kopieringsrutiner for både intern og ekstern SCI-kopiering avgjørende for ytelsen. For videre å kunne oppnå god ytelse i ARMCI bør derfor slike rutiner være biblioteksbasert. En eksplisitt implementasjon som tar hensyn til forskjellige arkitekturer,

chipset og prosessorer i ARMCI vil rett og slett bli for omfattende.

Selv om ARMCI inviterer til spesialdesign av kommunikasjonsalgoritmer kan implementasjon og vedlikehold bli omfattende. Get-operasjoner er relativt enkle å implementere ettersom dette krever en synkronisering av server og klient. Noe verre er det med implementasjon av ikke-blokkerende operasjoner som put og accumulate. Slike operasjoner bør fortrinnsvis implementeres etter *write-and-forget* prinsippet. For get-operasjoner er altså både forsinkelse og båndbredde kritiske for ytelsen, mens dette er mindre viktig ved put- og accumulate-operasjoner. Kommunikasjonsprotokoller og selve sendingen bør således variere med operasjonen ARMCI utfører. Et forslag i så henseende er bruk av DMA for put og accumulate mens PIO bør benyttes for get-requester, ettersom dette er et blokkerende kall. Implementasjon av slike kommunikasjonsprotokoller blir vanskeliggjort ved bruk av API som MPI, men man bør se nærmere på alternative MPI-kall som *MPI_Isend()* og *MPI_Bsend()*. Effektiviteten av disse vil avhenge sterkt av underliggende MPI-implementasjon.

8.3.4 Hyper-Threading, et alternativ til interrupter?

Nieplocha et. al.[106, 105] understreker i flere av sine artikler hvordan ARMCI bør unngå polling på nettverksadapter eller minneområder. Selv om Nieplocha anerkjenner gevinsten med lav forsinkelse ved bruk av slik polling og at interrupter er relativt dyre, understreker han at slike implementasjoner stjeler CPU-sykler fra selve applikasjonen.

For enkelte interconnect er interrupter ansett som billig og i enkelte tilfeller en nødvendighet for kommunikasjon (for eksempel Ethernet). Denne teknikken blir mye brukt i de implementasjoner hvor operativsystemet inngår i deler av kommunikasjonssekvensen. For interconnect som SCI foregår all prosessering i forhold til interconnectet i user-space. Dette betyr at interrupt som genereres av hardware først må behandles av operativsystemet i kernel-space og (i motsetning til Ethernet) deretter signaleres ut i user-space. Dette skaper økt overhead og kan for SCI resultere i en behandlingstid av interrupt på rundt 100 μs.

Ved bruk av polling vil det i ARMCI oppstå en situasjon hvor operativsystemet er ansvarlig for å schedulere og dermed bytte mellom den pollende server-tråden og de beregnede klientprosessene. Denne sheduleringen skaper overhead gjennom prosessbytter og operativsystem-overhead. Ved bruk av den nye Hyper-Threading-teknologien[90] fra Intel minimaliseres denne overheaden ved at prosessoren indirekte gjør denne byttingen på hardware-nivå. Operativsystemet vil i liten grad avbryte prosessene/trådene mens prosessoren eksekverer begge trådenes instruksjoner samtidig eller i en flettet orden.

Som en konklusjon bør man således revurdere eventuelle holdninger om at interrupter er det absolutt riktige. Man bør i enkelte tilfeller snu på flisa og argumentere for å unngå interrupter og heller benytte polling og Hyper-Threading. Dette kan nemlig føre til kommunikasjon med lav forsinkelse og liten detekterbar overhead. Den bestemmende faktoren ligger i applikasjonens kommunikasjonsmønster og kommunikasjonsbibliotekenes implementasjon. En kompromiss basert på en viss tids polling for deretter å benytte interrupter kan således være mest hensiktsmessig. Slike implementasjoner bør derfor vurderes både for ARMCI og andre kommunikasjonsbibliotek.

8.3.5 Neste generasjon ARMCI

Følgende punkt bør altså være i fokus ved utvikling av neste genrasjon ARMCI:

- Bruk av både MPI, MPI-2 og DAT som underliggende bibliotek.

- En *trade-off* mellom polling- og interrupt-baserte mottak av meldinger i server-tråden, samt videre forskning rundt Hyper-Threading og liknende teknologier.
- Mulighet og tilrettelegging for bruk av eksterne minnekopieringsbibliotek.
- Mer generell zero- og/eller one-copy protokoller med hensyn på DAT og MPI.
- Valg av DMA- og PIO-basert kommunikasjon etter operasjon (hvis mulig).

Jeg mener altså et en løsning med sikte på portabilitet, som kan benyttes over SCI, i større grad kan rettferdiggjøres fremfor en skreddersydd implementasjon for SCI. Selv om en lik implementasjon muligens vil ha noe dårligere ytelse enn en ren SCI-løsning, vil levetiden på implementasjonen være desto lenger.

Bibliografi

[1] *SBus-to-SCI Adapter User's Guide*, 1995.

[2] Chen A, S. Pakin, M. Lauria, M. Buchanan, K. Hane, L. Giannini, and J. Prusakova. High performance virtual machines HPVM: Clusters with supercomputing API and performance. *In Proceedings 8th SIAM Conference on Parallel Processing in Scientific Computing*, 1997.

[3] a Hewlett Packard Technical White Paper. Inside the Intel Itanium 2 Processor: an Itanium Processor Family member for balanced performance over a wide range of applications. Technical report, Hewlett Packard, Juli 2002.

[4] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[5] Albert Alexandrov, Mihai F. Ionescu, Klaus e. Schauser, and Chris Scheiman. LogGP: Incorporating Long Messages into the LogP Model - One step closer towards a realistic model for parallel computation. *Proc. of the 7th Annual ACM Symp. on Parallel Algorithms and Architectures*, June 1995.

[6] Gigabit Alliance. Gigabit Ethernet - accelerating the standard for speed. Whitepaper, avaliable at http://www.gigabit-ethernet.org/technology/whitepapers, 1997.

[7] Alteon Networks. Extended Frame Sizes for Next Generation Ethernets. Whitepaper avaliable at http://www.alteonwebsystems.com, 1999.

[8] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of AFIPS Spring Join Computer Conference*, volume 30, pages 483–485. International Business Machines Corporation, 1967.

[9] Gene M. Amdahl. Storage and I/O Parameters and System Potential. In *IEEE Computer Group Conference*, pages 371–372, June 1970.

[10] ANSI/VITA. *ANSI/VITA 26-1998, Myrinet-on-VME*, 1998.

[11] Argonne National Laboratory. *MPICH: Portable MPI Model Implementation. Version 1.2.1*, 2000. http://www.mcs.anl.gov/mpi/mpich.

[12] ARMCI - Aggregate Remote Memory Copy Interface. http://www.emsl.pnl.gov:2080/docs/parsoft/armci/index.html, January 2003.

[13] Scali AS. *Scali System Guide - Scali Library Guide, version 3*. Scali AS, 2002.

[14] The ATM (Asynchronous Transfer Mode) forum. Online at http://www.atmforum.com.

[15] Berkeley VIA project. http://www.cs.berkeley.edu/ philipb/via.

[16] Lawrence Besaw. *BSD Socket Reference - Berkeley UNIX System Calls and Interprocess Communication*, January 1987. Revised, September 1987, January 1991 by Marvin Solomon.

[17] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles E. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, pages 29–36, February 1996.

[18] Håkon Bugge and Knut Omang. *Affordable Scalability using Multi-Cubes*, volume 1734 of *Lecture Notes in Computer Science*, chapter 8, pages 167–165. Springer-Verlag, September 1999.

[19] B. Carpenter, G. Zhang, and Y. Wen. NPAC PCRC runtime kernel definition. Technical report, Center for Research on Parallel Computation, 1997.

[20] The Colombus Parallel CI Program Project. http://www.itc.univie.ac.at/ hans/Columbus/, January 2003.

[21] Myrinet Experiences Exchange Site: PCI Performance. http://www.conservativecomputer.com/myrinet/perf.html, January 2003.

[22] Cray Inc. *Application Programmer's Library Reference Manual. Document number : 004-2165-002*.

[23] Crossroads, Compaq, Dell, Intel, and LSI Logic. *InfiniBand Technology Prototypes White Paper*. ftp://download.intel.com/design/servers/future/_server_io/documents/Final_whitepaperxx.pdf, 2000.

[24] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorstein von Eicken. LogP: Towards a Realistic Model of Parallel Computation. *Proc. ACM Symp. on Principlesand Practice of Parallel Programming*, 1993.

[25] David E. Culler, Andrea Dusseau, Seth Copen GoldStein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel Programming in Split-C. In *Proceedings of the International Conference on Supercomputing*, November 1993.

[26] Dalton QCP. http://www.kjemi.uio.no/software/dalton/dalton.html, January 2003.

[27] Daryl Banttari. Daryl's TCP/IP Primer - Addressing and Subnetting on the Near Side of the 'Net . Online at http://www.tcpipprimer.com, 2001.

[28] Dolphin Interconnect Solutions. Dolphin Interconnect Solutions Web site. Online at http://www.dolphinics.com.

[29] Dolphin Interconnect Solutions. *A Backside Link (B-Link) for Scalable Coherent Interface (SCI) Nodes*, draft 2.4 edition, September 1995.

[30] Dolphin Interconnect Solutions. *Link Controller LC-1 Specification*, revision 1.06 edition, July 1995.

[31] Dolphin Interconnect Solutions. *SBus-to-SCI Adapter User's Guide, DIS303 SBus-2*, 1995.

[32] Dolphin Interconnect Solutions. *PCI-SCI Bridge Functional Specification*, version 3.01. edition, November 1996.

[33] Dolphin Interconnect Solutions. *Link Controller LC-2 Specification*, version 1.03 edition, October 1997. Preliminary version.

[34] Dolphin Interconnect Solutions. *PSB64 Specification - D665*, version 1.30 edition, September 1999.

[35] Dolphin Interconnect Solutions. *Link Controller LC-3 Specification*, version 0.83. edition, April 2000. Preliminary version.

[36] Dolphin Interconnect Solutions. *PSB66 Specification - D667*, version 0.90. edition, February 2000.

[37] Dolphin Interconnect Solutions Inc. *PCI-SCI Cluster Adapter Specification*, version 1.1 edition, May 1996.

[38] Jose Duato, Sudhakar Yalamanchili, and Lionel Ni. *Interconnection Networks: An Engineering Approach*. IEEE Press, 1997.

[39] Earth Simulation Center. http://www.es.jamstec.go.jp/, January 2003.

[40] Michael Eberl, Hermann Hellwagner, Martin Schulz, and Bjarne G. Herland. SISCI – Implementing a Standard Software Infrastructure on an SCI Cluster. In *Workshop Cluster-Computing, Chemnitz, Germany*, November 1997. In Chemnitzer Informatik-Berichte, Band CSR-97-05, ISSN 0947-5125.

[41] PNL's Environmental and Molecular Science Laboratory (EMSL). http://www.emsl.pnl.gov/, January 2003.

[42] Emulex Inc. (formerly Giganet, Inc.). cLAN products. Online at http://www.emulex.com.

[43] Graham E. Fagg, Kevin S. London, and Jack Dongarra. MPI_Connect Managing Heterogeneous MPI Applications Interoperation and Process Control. In *Proceedings of 5th PVM & MPI European Users Meeting (EuroPVM/MPI)*, pages 93–96, September 1998.

[44] Marco Fillo and Richard B. Gillett. Architecture and Implementation of Memory Channel 2. *Digital Technical Journal*, 9(1), 1997.

[45] Markus Fischer. PVM Message Passing Environments. Online at http://www.markus-fischer.de.

[46] Michael J. Flynn. Some Computer Organizations and Their Effectivenes. *IEEE Transactions on Computers*, c-21(9):948–960, September 1972.

[47] Ian Foster and Nicholas T. Karonis. Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems. *Proceedings of the International Conference on Supercomputing*, November 1998.

[48] William T. Futral. *InfiniBand Architecture Development and Deployment*. Intel Press, 2001.

[49] Edgar Gabriel, Michael Resch, Thomas Beisel, and Rainer Keller. Distributed computing in a heterogenous computing environment. In *Proceedings of 5th PVM & MPI European Users Meeting (EuroPVM/MPI)*, pages 180–187, September 1998.

[50] GAMESS-UK - A Generalised Atomic and Molecular Electronic Structure System. http://www.dl.ac.uk/TCSC/QuantumChem/Codes/GAMESS-UK/, January 2003.

[51] G.A. Geist and V.S. Sunderam. The Evolution of the PVM Concurrent Computing System. In *Proceedings of COMPCON spring'93*, pages 549–557, February 1993.

[52] Karim Ghouas. VIA i klyngenettverk - eksisterende teknologier og eksempelimplementasjon over SCI. Master's thesis, Universitetet i Oslo, Institutt for Informatikk, 2001.

[53] Global Arrays. http://www.emsl.pnl.gov:2080/docs/global/ga.html, January 2003.

[54] William Gropp. MPICH2000 - A High-Performance Implementation of MPI-2. In *Proceedings of 3rd Austrian-Hungarian workshop on distributed and parallel systems (DAPSYS)*, September 2000.

[55] J. L. Gustavson. Reevaluating Amdahl's Law. *Communication of the ACM, 31(5) 532-533*, 1988.

[56] Alan Heirich, David Garcia, Michael Knowles, and Robert Horst. ServerNet: A Reliable Interconnect for Scalable High Performance Cluster Computing. http://www.servernet.com, september 1998.

[57] Hermann Hellwagner. *The SCI Standard and Applications of SCI*, chapter 1, pages 3–34. Volume 1734 of Hellwagner and Reinefeld [58], September 1999.

[58] Hermann Hellwagner and Alexander Reinefeld, editors. *SCI: Scalable Coherent Interface, Architecture and Software for High-Performance Compute Clusters*, volume 1734 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1999.

[59] Hermann Hellwagner and Josef Weidendorfer. *SCI Sockets Library*, chapter 11, pages 209–229. Volume 1734 of Hellwagner and Reinefeld [58], September 1999.

[60] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann Publishers, first edition, 1990.

[61] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann Publishers, third edition, 2003.

[62] High-Performance Parallel Interface (HIPPI) Standards Activities. http://www.hippi.org/.

[63] Håkon O. Bugge, Vice President Product Development, Scali AS. Personal communication, November 2002.

[64] Geir Horn. *Scalability of SCI Ringlets*, chapter 7, pages 151–166. Volume 1734 of Hellwagner and Reinefeld [58], September 1999.

[65] Lars P. Huse. Layering SHMEM on top of MPI. In *Proceedings of 8th PVM & MPI European Users Meeting (EuroPVM/MPI)*, pages 44–51, September 2001.

[66] Lars P. Huse, Knut Omang, and Håkon Bugge. ScaFun – a fundament for process communication. In *Proceedings of 2nd SCI Europe at Euro-Par'99*, pages 13–20, August 1999.

[67] Lars P. Huse, Knut Omang, Håkon Bugge, Arnt-Tore Haugsdal, and Einar Rustad. *ScaMPI – Design and Implementation*, volume 1734 of *Lecture Notes in Computer Science*, chapter 14, pages 249–260. Springer-Verlag, September 1999.

[68] Lars Paul Huse. *A comparative study of different programming paradigms in an SCI based multiprosessor evironment*. PhD thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, 2002.

[69] Kai Hwang. *Advanced Computer Architecture (Parallelism, Scalability, Programmability)*. Computer Science. McGraw-Hill, 1993.

[70] Kai Hwang and Zhiwei Xu. *Scalable Parallel Computing (Technology, Architecture, Programming)*. Computer Engineering. McGraw-Hill, 1998.

[71] IBM LAPI. http://www.research.ibm.com/actc/Opt_Lib/LAPI_Intro.htm, January 2003.

[72] IEEE 802.3 CSMA/CD (ETHERNET). http://grouper.ieee.org/groups/802/3/, January 2003.

[73] IEEE P802.3ae 10Gb/s Ethernet Task Force. http://grouper.ieee.org/groups/802/3/ae/, January 2003.

[74] Toshiyuki Imamura, Yuichi Tsujita, Hiroshi Koide, and Hiroshi Takemiya. An Architecture of Stampi: MPI Library on a Cluster of Parallel Computers. In *Proceedings of 7th PVM & MPI European Users Meeting (EuroPVM/MPI)*, pages 200–207, September 2000.

[75] InfiniBand trade assosiation. IBTA 1.0 Specifications. Available from http://www.InfiniBandta.org/home.html.

[76] Information Networks Division, Hewlett-Packard. *Netperf: A Network Performance Benchmark*, revision 2.0 edition, February 1995. See also Netperf public Web page at http://www.netperf.org.

[77] INRIA Rhône-Alpes. SCI cluster research at INRIA Rhône-Alpes. Online at http://sci-serv.inrialpes.fr, 2001.

[78] Institute of Electrical and Electronics Engineers, Inc. *IEEE Standard for Scalable Coherent Interface (SCI), IEEE Std 1596-1992*, August 1993.

[79] Institute of Electrical and Electronics Engineers, Inc. *POSIX System Application Program Interface IEEE Std 1003.1c1995*, 1995.

[80] Institute of Electrical and Electronics Engineers, Inc. *IEEE Standard for Gigabit Ethernet, ANSI/IEEE Std 802.3z-1998*, June 1999.

[81] Intel Corporation. *Intel Pentium 4 Processor Optimization, Reference Manual*, 2002. http://developer.intel.com, Order Number: 248966.

[82] ISSD Technology Communications. PCI-X: An Evolution of the PCI Bus. Available from http://www.compaq.com, 1999. White Paper.

[83] Intel Itanium 2. http://www.intel.com/products/server/processors/server/itanium2/, January 2003.

[84] Frank Jensen. *Introduction to Computational Chemistry*. John Wiley & Sons, 1999.

[85] Moldeklev K., Klovning E., and Kure Ø. TCP/IP behavior in a high-speed local ATM network environment. In *Proceedings of 19th IEEE conference on local computer networks, Minneapolis*, pages 176–185, October 1994. ISBN 0-8186-6680-3.

[86] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenenpoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of USENIX Winter 1994 Conference*, pages 115–131. USENIX Association, January 1994.

[87] Marius Christian Liaaen and Hugo Kohmann. *Dolphin SCI Adapter Cards*, chapter 3. Volume 1734 of Hellwagner and Reinefeld [58], September 1999.

[88] Manfred Liebhart. Performance Aspects of Switched SCI Systems. In *Proceedings of 6th International Symposium on High Performance Distributed Computing*. IEEE, August 1997.

[89] Manfred Liebhart, André Bogaerts, and Eugen Brenner. A Study of an SCI Switch Fabric. In *Proceedings of IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, Haifa*, January 1997.

[90] Deborah T. Marr, Frank Binns, David L Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-Threading Technology Architecture and Microarchitecture. Technical report, Intel Corporation, 2002.

[91] Richard P. Martin. HPAM: An Active Message layer for a Network of HP Workstations. In *Proceedings of Hot Interconnects II*, 1994.

[92] Richard P. Martin, Amin M. Vahdat, David E. Culler, and Thomas E. Adnerson. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. *Proc. of the 24th Int'l. symp. Computer Architecture*, June 1997.

[93] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, May 1994. Version 1.0.

[94] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, June 1995. Version 1.1.

[95] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, July 1997. Version 1.2 and 2.0.

[96] MM5 Home Page. http://www.mmm.ucar.edu/mm5/, January 2003.

[97] Gordon E. Moore. Cramming More Components Onto Integrated Circuits. *Electronics*, 38(8):114–117, 1965.

[98] Gordon E. Moore. Progress in digital integrated electronics. In *Proceedings of IEEE Integrated Electron Devices Meeting*, pages 11–13, 1975.

[99] F.E. Mourão and J.G. Silva. Implementing MPI-2 One-Sided Communications for WMPI. In *Proceedings of 6th PVM & MPI European Users Meeting (EuroPVM/MPI)*, pages 231–240, September 1999.

[100] MPI Forum. http://www.mpi-forum.org, January 2003.

[101] Myricom. The Myricom Web site. Online at http://www.myri.com.

[102] Myricom. *Open Specifications and Documentation*, 2000. http://www.myri.com/open-specs/index.html.

[103] Myricom. http://www.myricom.com, January 2003.

[104] Myricom Inc. *The GM Message Passing System*, 2000. http://www.myri.com/scs/GM/doc/gm.html.

[105] Jarek Nieplocha and Jialin Ju Edoardo Apra. One-sided Communication on Myrinet-based SMP Clusters using the GM Message-Passing Library. *Proc. CAC01 Workshop of IPDPS'01, San Francisco*, 2001.

[106] Jarek Nieplocha and Bryan Carpenter. ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems. *Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) of International Parallel Processing Symposium IPPS/SPDP '99, San Juan, Puerto Ric*, April 1999.

[107] Jarek Nieplocha and Jialin Ju. ARMCI: A Portable Aggregate Remote Memory Copy Interface. Technical report, Pacific Northwest National Laboratory (PNNL), 30 October 2000.

[108] Jarek Nieplocha, K. Parzyszek, and R. Kendall. A Generalized Portable SHMEM Library for High Performance Computing (GPSHMEM). *Computing and Information Sciences 2000 Annual Report*, 2000.

[109] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global Arrays: A Portable Shared-Memory Programming Model for Distributed Memory Computers. Technical report, Pacific Northwest National Laboratory (PNNL), 1994.

[110] NWChem - High Performance Computational Chemistry Software. http://www.emsl.pnl.gov:2080/docs/nwchem/, January 2003.

[111] Knut Omang. Synchronization Support in I/O Adapter Based SCI Clusters. In *Proceedings of Workshop on Communication and Architectural Support for Network-based Parallel Computing, San Antonio, Texas*, volume 1199 of *Lecture Notes in Computer Science*, pages 158–172. Springer-Verlag, February 1997.

[112] Hong Ongz and Paul A. Farrelly. Performance Comparison of LAM/MPI, MPICH, and M-VICH on a Linux Cluster connected by a Gigabit Ethernet Network. In *Proceedings of the 4th Annual Linux Showcase and Conference*, October 2000.

[113] OpenMP Architecture Review Board. OpenMP Application Program Interface home page. Online at http://www.openmp.org.

[114] OpenMP Architecture Review Board. OpenMP Specifications - C/C++ version 1.0. Online at http://www.openmp.org, October 1998.

[115] OpenMP Architecture Review Board. OpenMP Specifications - FORTRAN version 2.0. Online at http://www.openmp.org, November 2000.

[116] PCI Special Interest Group. *PCI Local Bus Specification, Revision 2.1*.

[117] Fabrizio Petrini, Adolfy Hoisie, Wu chun Feng, and Richard Graham. Performance Evaluation of the Quadrics Interconnection Network. In *Proceedings of Workshop on Communication Architecture for Clusters (CAC '01)*, April 2001.

[118] Gregory F. Pfister. *In Search of Clusters - The Ongoing Battle in Lowly Parallel Computing*. Prentice Hall PTR, first edition, 1995. ISBN 0-13-437625-0.

[119] Pacific Northwest National Laboratory (PNNL). http://www.pnl.gov/, January 2003.

[120] S. Pope, S.J. Hodges, G.E. Mapp, D.E. Roberts, and A. Hopper. Enhancing Distributed Systems with Low Latency Networking. In *Proceedings of Parallel and Distributed Computing and Networks*, December 1998.

[121] Rolf Rabenseifner. MPI-GLUE: Interoperable high-performance MPI combining different vendor's MPI worlds. In *Proceedings of Euro-Par '98*, volume 1470 of *Lecture Notes in Computer Science*, pages 563–569. Springer-Verlag, 1998.

[122] Allyn Romanov and Sally Floyd. Dynamics of TCP Traffic over ATM. *IEEE Journal on Selected Areas in Communications*, May 1995.

[123] RWTH aachen. http://www.lfbs.rwth-aachen.de/, January 2003.

[124] Stein Jørgen Ryan. The Design and Implementation of a Portable Driver for Shared Memory Cluster Adapters. Research Report 255, Department of Informatics, University of Oslo, Norway, December 1997.

[125] Stein Jørgen Ryan and Haakon Bryhni. Eliminating the Protocol Stack for Socket Based Communication in Shared Memory Interconnects. In *Proceedings of International Workshop on Personal Computer based Networks Of Workstations (at IPPS'98)*, April 1998.

[126] Stein Jørgen Ryan, Arne Maus, and Stein Gjessing. The Design of an Efficient Portable Driver for Shared Memory Cluster Adapters. In *Proceedings of 7th International Workshop on SCI-based High-Performance Low-Cost Computing*, March 1997.

[127] Scali - Scalable Linux Systems. http://www.scali.com, January 2003.

[128] Michael S. Schalansker and B. Ramakrishna. EPIC: Explicitly Parallel Instruction Computing. Technical report, Hewlett-Packard Laboratories, February 2000.

[129] SCI-MPICH. http://www.lfbs.rwth-aachen.de/users/joachim/SCI-MPICH/, January 2003.

[130] SISCI. http://www.dolphinics.com/products/software/sisci_devkit.html, January 2003.

[131] Steve Sistare, Erica Dorenkamp, Nick Nevin, and Eugene Loh. Optimization of MPI Collectives on Clusters of Large-Scale SMP's . In *Proceedings of the International Conference on Supercomputing*. Sun Microsystems, Inc., November 1999.

[132] Systems Developer, Scali AS. Personal communication, January 2003.

[133] StarGen. http://www.stargen.com/, January 2003.

[134] W. Richard Stevens. *UNIX Network Programming; Networking APIs: Sockets and XTI, publisher = Prentice Hall*, volume 1 of *Proffesional Technical Reference*. 1998.

[135] V.S. Sunderam, G.A. Geist, J. Dongarra, and R. Manchek. The PVM Concurrent Computing System: Evolution, Experiences and Trends. *Journal of Parallel Computing*, 20(4):531–546, 1994.

[136] Surendranath Talla. *Adaptive Explicitly Parallel Instruction Computing*. PhD thesis, Department of Computer Science, New York University, 2000.

[137] The DAT Collaborative. http://www.datcollaborative.org/, January 2003.

[138] Jesper Larsson Träff, Hubert Ritzdorf, and Rolf Hempel. The Implementation of MPI-2 One-Sided Communication for the NEC SX-5. In *Proceedings of the International Conference on Supercomputing*, November 2000.

[139] VIA over GM. Kan lastes ned fra ftp://ftp.myri.com.

[140] Thorsten von Eicken. *Active Messages: an Efficient Communication Architecture for Multiprocessors* . PhD thesis, University of California at Berkeley, 1993.

[141] John von Neumann. First Draft of a Report on the EDVAC. Technical Report Contract No. W-670-ORD-4926, Moore School of Electrical Engineering, University of Pennsylvania, June 1945. An edited and corrected by Michael D. Godfrey appeared in IEEE Annals of the History of Computing, Vol. 15, No. 4, 1993.

[142] Wolf-Dietrich Weber, Stephen Gold, Pat Helland, Takeshi Shimizu, Thomas Wicki, and Winfried Wilcke. The Mercury Interconnect Architecture: A cost-effective infrastructure for high-performance servers. *International Symposium on Computer Architecture*, pages 98–107, jun 1997.

[143] Steve Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of 22th International Symposium on Computer Architecture*, pages 24–36, June 1995.

[144] J. Worringen and M. Dormanns. *SMI - Shared Memory Interface: User & Reference Manual*. Lehrstuhl für Betriebssysteme, RWTH Aachen, September 2000.

[145] Joachim Worringen. SCI-MPICH: The Second Generation. In *Proceedings of 3rd SCI Europe at Euro-Par'00*, pages 10–20, August 2000.

Tillegg A

Terminologi og ordforklaringer

- **Benchmark:** Også kalt test. Både test- og benchmarkbegrepet blir benyttet i rapporten.
- **Bit/Byte:** Ved forkortelser av *bit* benyttes b mens forkortelser av *byte* forkortes B.
- **Bridge:** På norsk bro, men benyttes ikke i dagligtalen. Betegner elektronikk/hardware mellom to andre hardware-system/-moduler.
- **Cluster:** Det norske ordet er klynge, men dette benyttes ikke i dagligtalen.
- **COTS** - Commercial Off The Shelf: på norsk kalt hyllevare.
- **Hardware:** Refererer til fysisk maskinvare og elektronikk i en datamaskin. Jeg har her valgt å benytte den termen som benyttes i dagligtale.
- **Interconnect:** Klassifiserer nettverket mellom de enkelte nodene i et cluster. Begrepet er en begrensning (spesialdel) av klassifiseringen LAN.
- **I/O:** Forkortelse for Input/Output. Refereres ofte til i sammeneng med datamaskiners kommunikasjon med perifere enheter; tastatur, mus, nettverk, lagringsenheter og liknende.
- **Kernel-level/space:** se *User-level/space* under.
- **Klient-Server:** En *klient-server* arkitektur benyttes som et begrep i dagligtalen. Jeg har derfor valgt å benytte de engelske uttrykket for tjener, *server*, i denne rapporten. Det norske ordet klient benyttes fremfor *client*. Videre benyttes *server-tråd* fremfor tjener-tråd.
- **Mapping:** Termen beskriver funksjonen en prosess utfører for å oversette et adresseområde, tilhørende en annen modul, inn i lokalt virtuelt adresseområde. I denne oppgaven benyttes begrepet hovedsaklig i forbindelse med oversetting av SCI-relaterte minneområder.
- **Open Source:** Refererer til programvare fritt tilgjengelig med frigitt kildekode. Velkjente Open Source prosjekter er for eksempel Linux og GNU.
- **Overhead:** Kan sammenliknes med ekstraarbeide som må gjøres for å uføre en funksjon, men samtidig ikke er en del av funksjonen. Ved kommunikasjon regnes for eksempel protokollinformasjon som sendes sammen med data som ekstra informasjon som ikke har direkte sammenheng med dataene.

- **Request:** Termen beskriver en forespørsel. En request-melding er en forespørsel implementert som en melding.
- **Shared/Remote memory:** Shared Memory eller delt minne refererer til minne direkte aksesserbart av to prosessorer. Remote memory eller fjerntliggende minne refererer til minne tilhørende en annen CPU og som kun kan aksesseres via mellomliggende hardware og/eller programvare (for eksempel SCI).
- **Send/Receive:** Jeg har i enkelte sammenhenger valgt å benytte *send* og *receive* fremfor send og motta da dette blir mest benyttet i dagligtalen; for eksempel benyttes *receive-kall* fremfor mottaks-kall når det refereres til funksjoner.
- **User-level/space:** *User* refererer til det miljø normale brukerprogrammer/applikasjoner kjører i. *Kernel*, eller kjerne på norsk, refererer til miljøet i operativsystemets kjerne. Level refererer til grensesnitt og API. Kernel-level ligger under user-level i bibliotekstakken. Space refererer til instansen av funksjonen, hvorvidt denne ligger i operativsystemet (kernel) eller i brukerområdet (user).
- **Word:** På norsk ord. Refererer til informatikkbegrepet for 4 byte eller 32 bit. Jeg har valgt å benytte den engelske termen da dette benyttes i dagligtalen.

A.1 Parallelle maskinarkitekturer

Hwang og Xu[70] klassifiserer parallelle datasystemer i seks klasser. SIMD-datamaskiner benyttes kun på spesielle applikasjoner. De resterende klassene går under klassifiseringen MIMD (Multiple Instruction Multiple data):

PVP *Parallel Vector Processors*. PVP-maskiner er systemer bygget opp av et mindre antall kraftige spesial designede *vector prosessorer* (VP). Et spesial designet *crossbar*-svitsjet nettverk knytter disse vektor prosessorene til et antall shared memory (SM)-moduler som gir en høyhastighets dataaksess.

SMP *Symmetric Multiprocessor*. Ulikt PVP-maskiner består SMP-maskiner typisk av COTS mikroprosessorer med “on-chip” eller “off-chip” cache. Disse prosessorene er knyttet til et shared memory gjennom en høyhastighets *snooping-bus* (eller tilsvarende teknologier for cache koherens). Begrensningene i størrelse er hovedsaklig grunnet bruken av et felles delt minne gjennom en buss og/eller et *crossbar* interconnect.

MPP *Massively Parallel Processor*. MPP-maskiner kan sees på som en mellomting mellom Clustere og SMP-maskiner. Som i SMP-maskiner benyttes vanligvis COTS mikroprosessorer, men minnet er fysisk distribuert over prosesseringsnodene. Et høyhastighets interconnect, som oftest implementert som en backplane buss eller liknende, knytter nodene sammen. Prosessorene har ikke shared memory og kommuniserer gjennom meldingsutveksling.

Cluster/COW *Cluster Of Workstations*. Grensen mellom clustere og MPP-maskiner kan virke uklar. For eksempel regnes IBM sin SP2 for å være en MPP-maskin selv om den har en clusterarkitektur. Forskjellen ligger i at MPP-nodene er knyttet sammen med et proprietært høyhastighets crossbar-svitsj og ikke COTS interconnect.

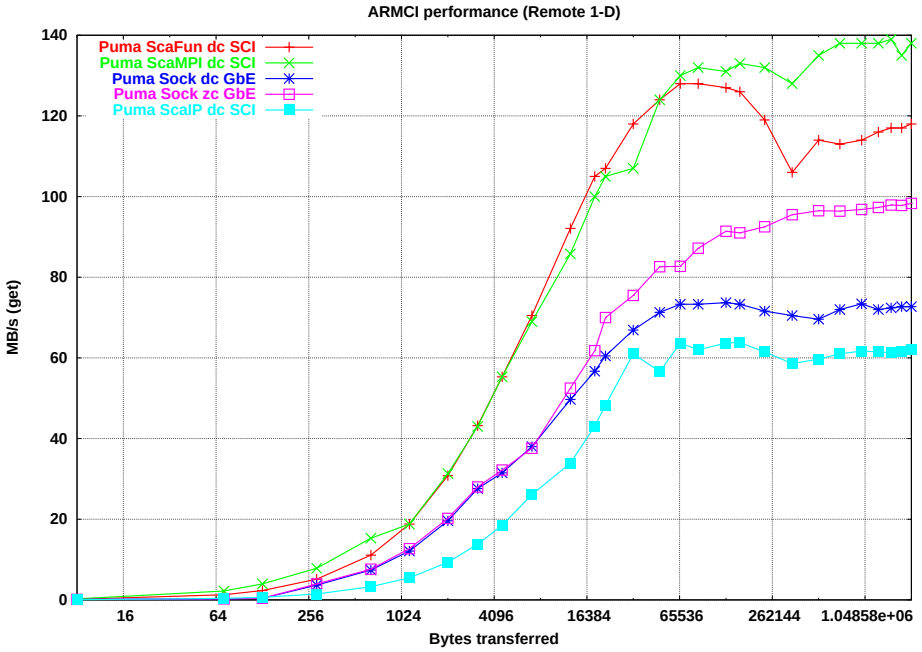
DSM *Distributed Shared Memory*. I motsetning til SMP-maskiner er fellesminnet fysisk distribuert mellom de forskjellige prosessorene/nodene i et DSM-system. System-hardware og/eller -programvare skaper illusjonen om et enkelt adresserom ovenfor brukeren. En DSM-maskin kan i sin helhet implementeres ved hjelp av programvare, som for eksempel TreadMarks[86] på både clustere og MPP-maskiner.

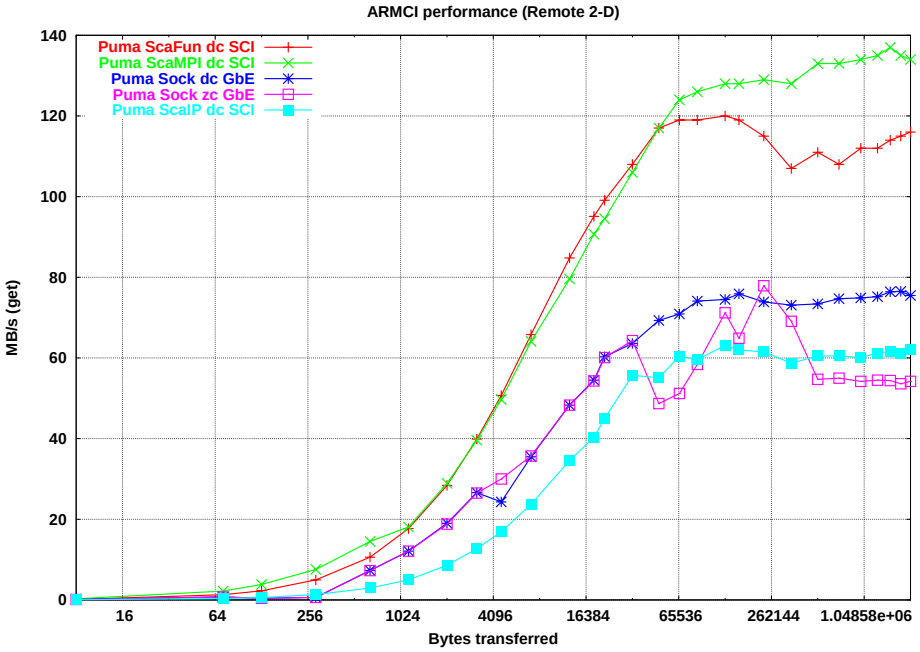
Tillegg B

Utfyllende testresultater

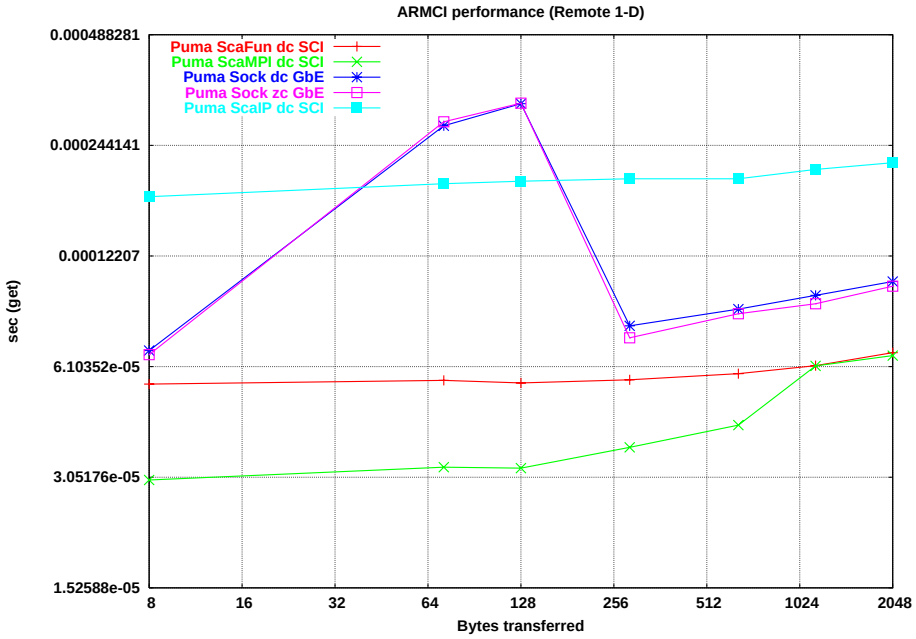
Følgende figurer viser en sammenlikning av båndbredde og forsinkelse for ARMCI over GbE (Sockets), ScaIP (Sockets), ScaMPI (SCI) og ScaFun (SCI). Resultatene er tegnet inn for GbE både med zero-copy (zc) og double-copy (dc). De andre implementasjonene benytter utelukkende double-copy. Kjøringene er gjort på Puma-clusteret. Alle målinger er målt på sendersiden (også put- og accumulate-båndbredde). Ved bruk av buffering på sendersiden (for eksempel ved bruk av Sockets) blir derfor forsinkelsen (kunstig) lav. Målingene for put og get må således sees på som tiden det tar for prosessen kan fortsette etter en put-/accumulate-operasjon.

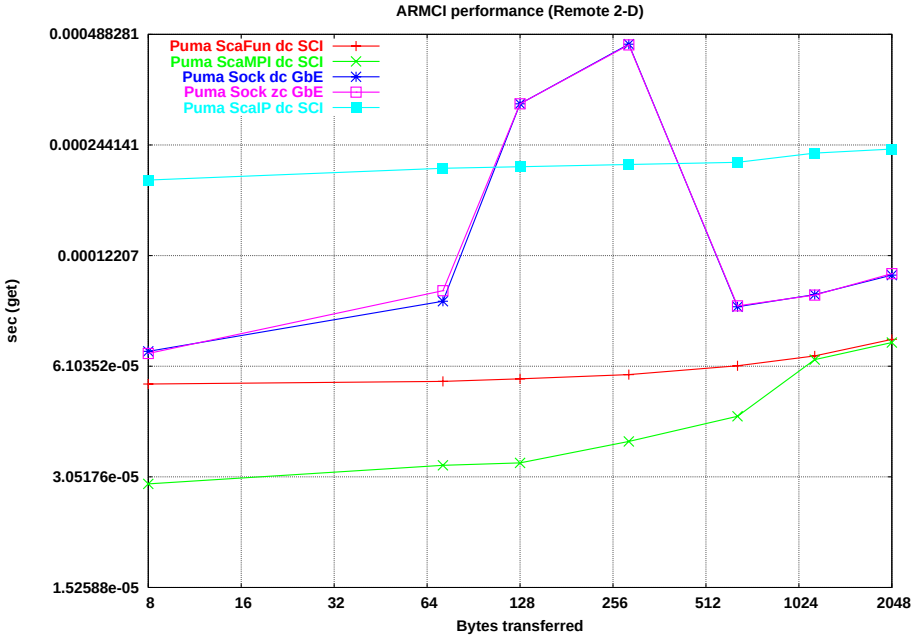
B.1 Båndbredde for ARMCI get-operasjoner



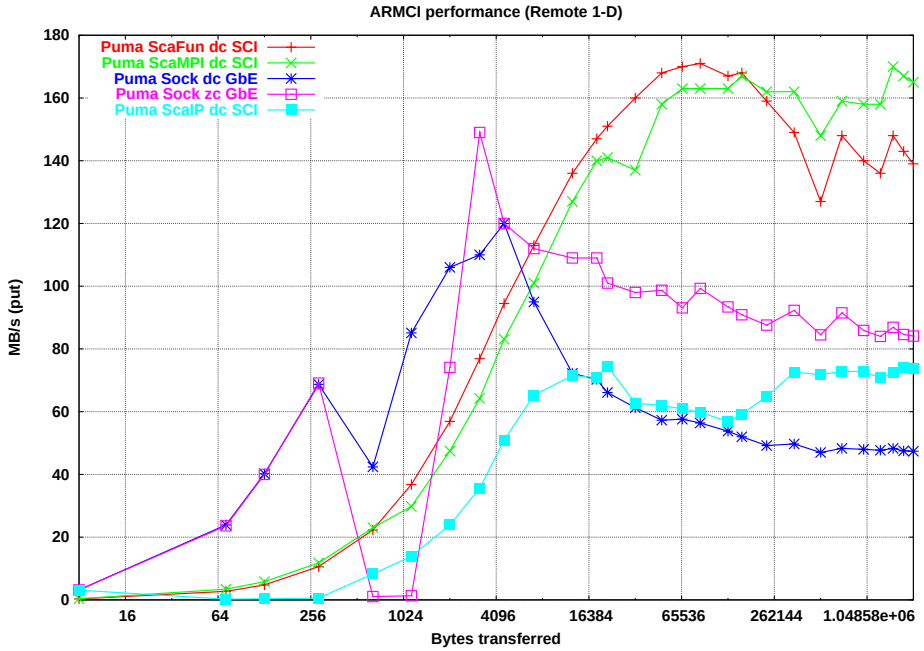


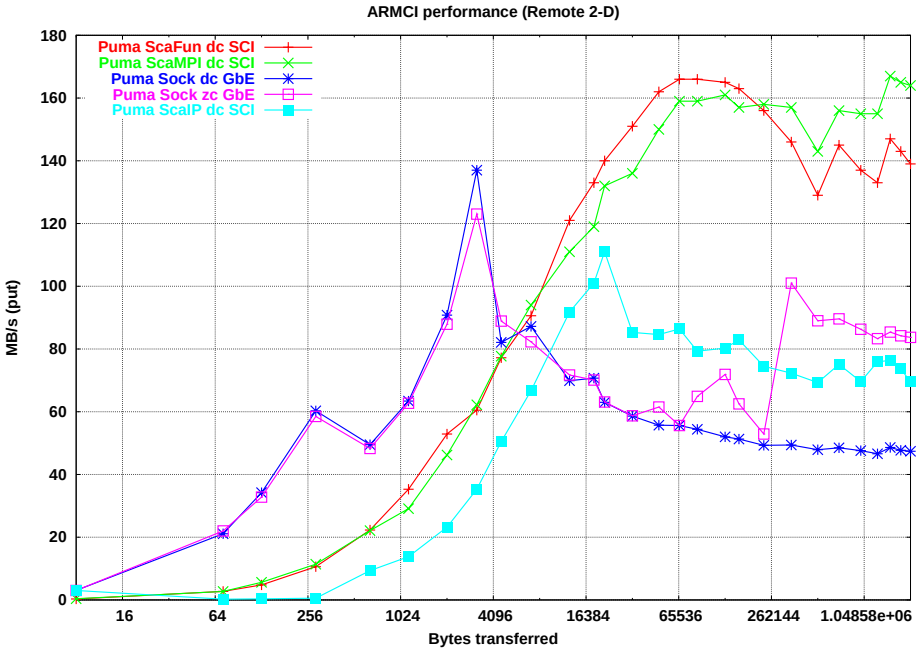
B.2 Forsinkelse for ARMCI get-operasjoner



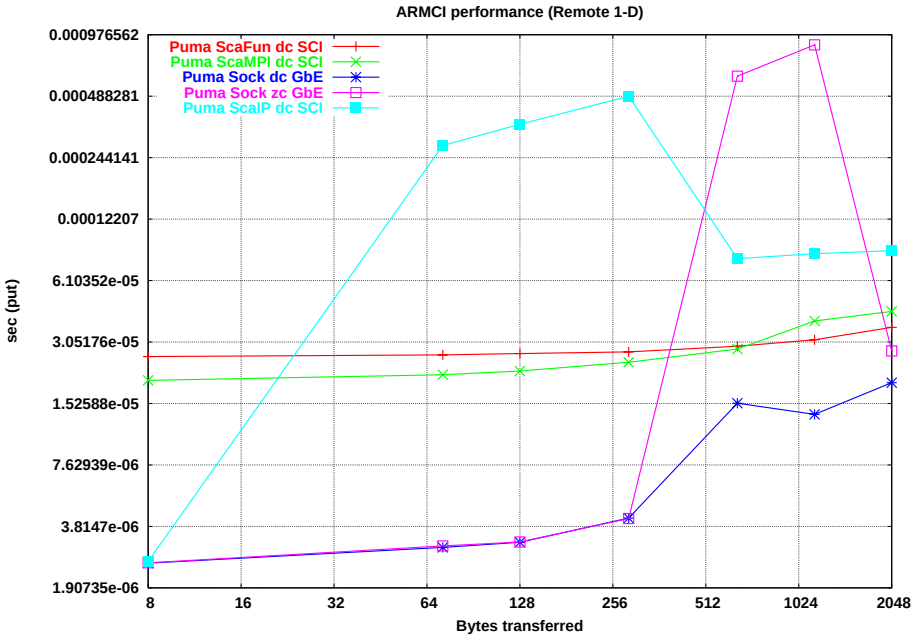


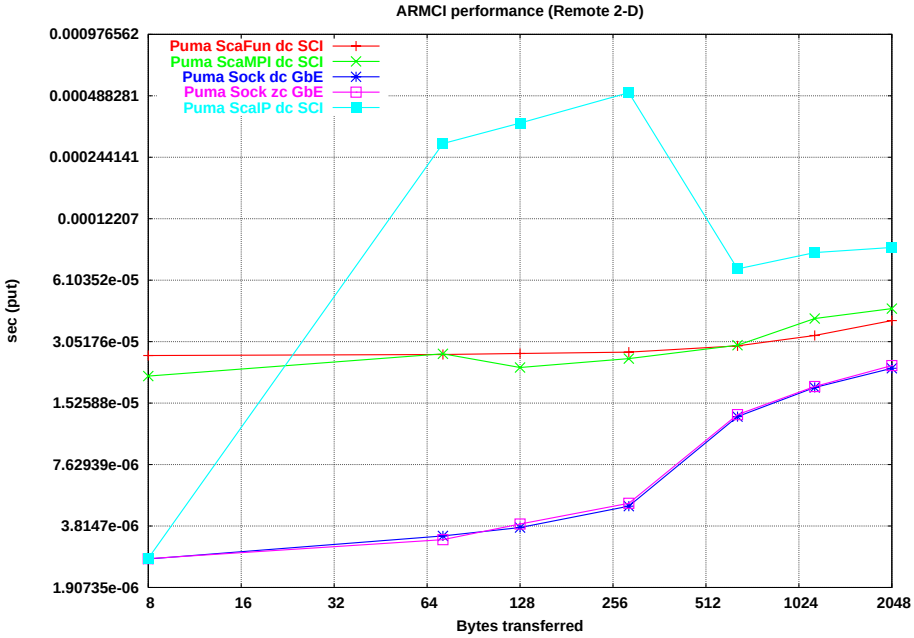
B.3 Båndbredde for ARMCI-put-operasjoner



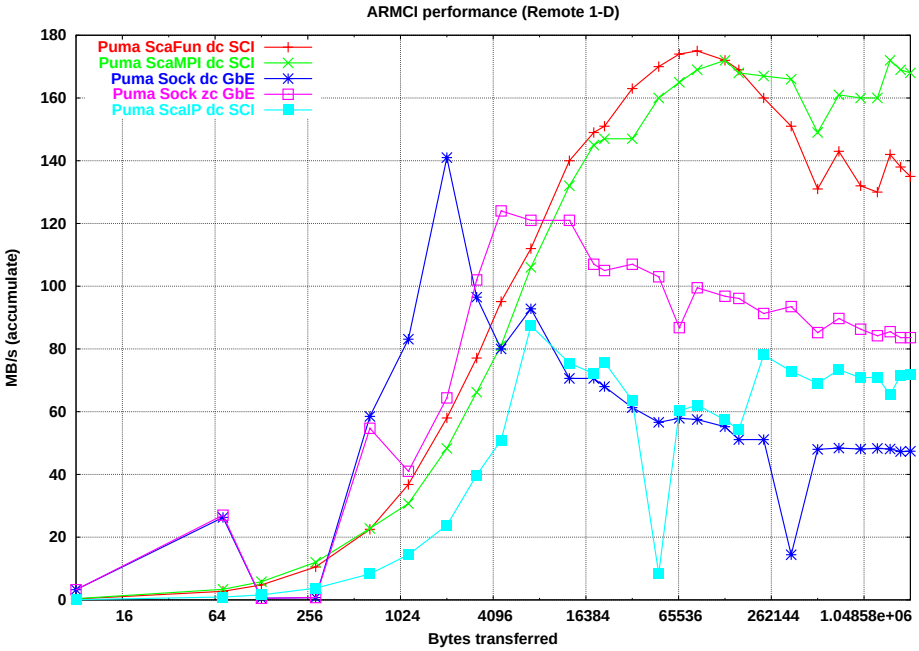


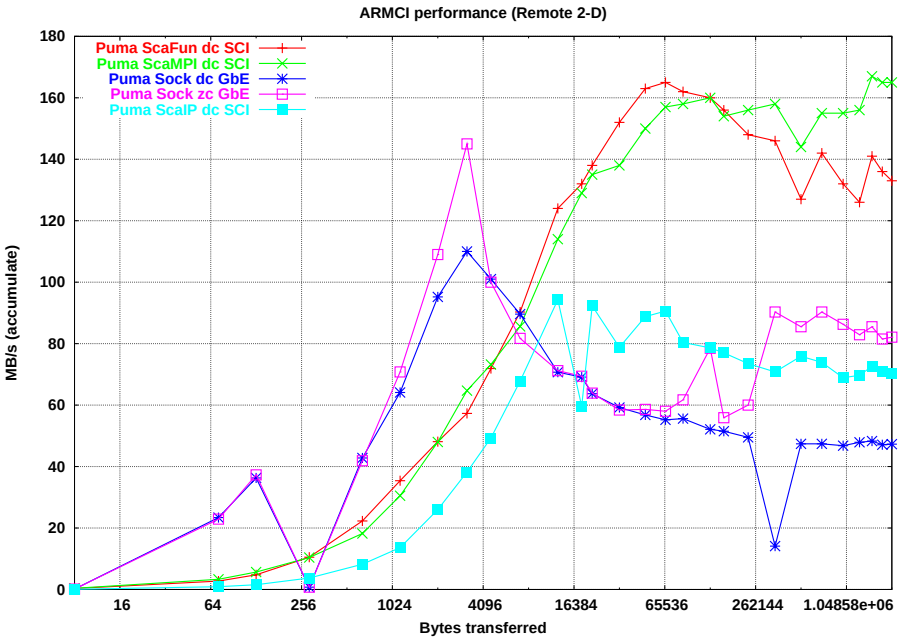
B.4 Forsinkese for ARMCI-put-operasjoner



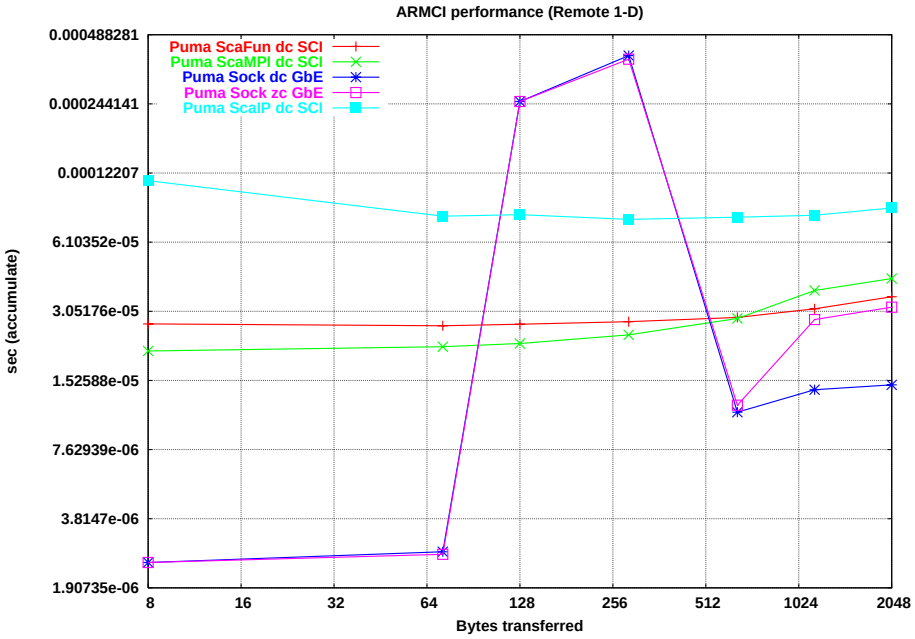


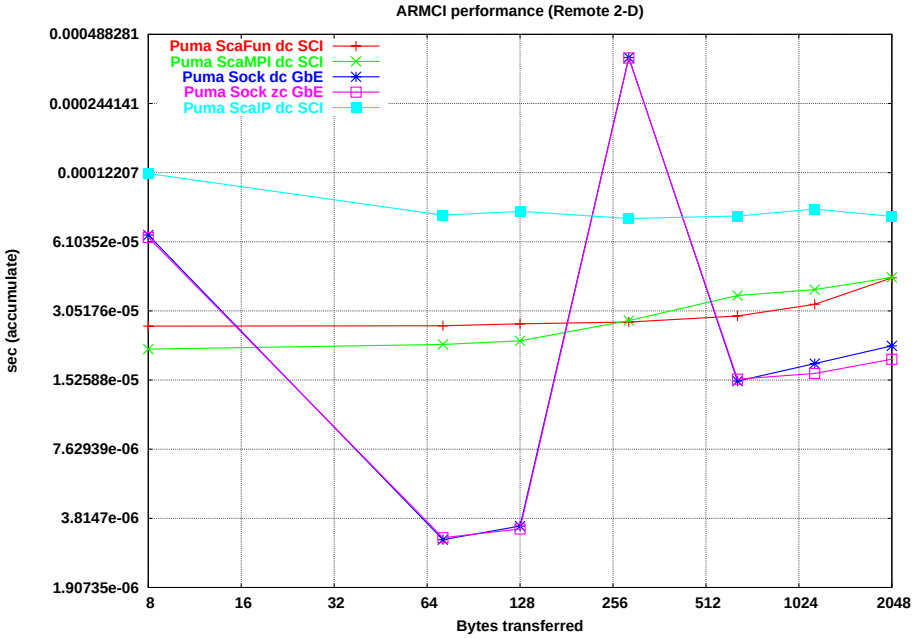
B.5 Båndbredde for ARMCI-accumulate-operasjoner





B.6 Forsinkelse for ARMCI-accumulate-operasjoner

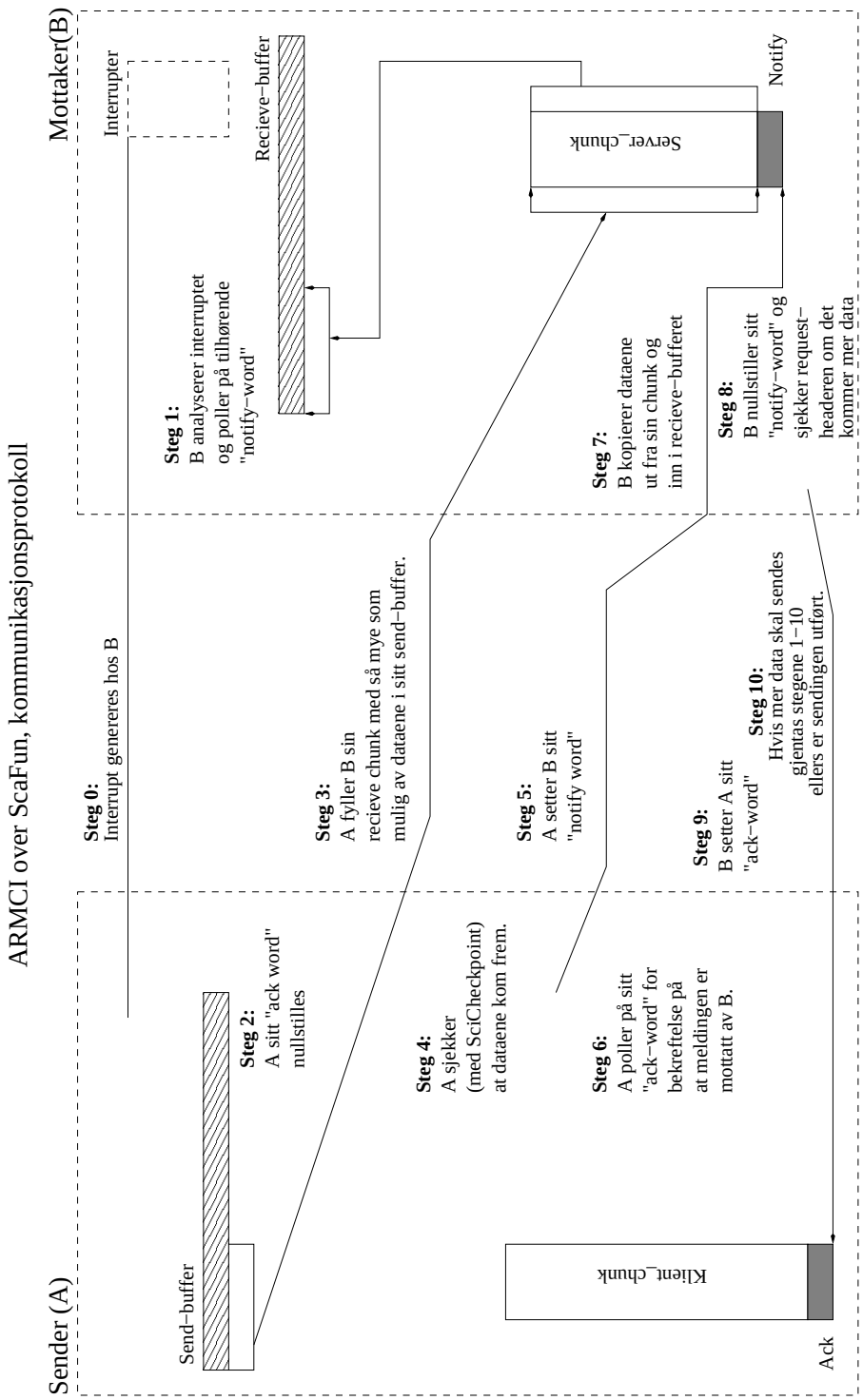



ARMCI performance (Remote 2-D)
Puma ScaFun dc SCI
Puma ScaMPI dc SCI
Puma Sock dc GbE
Puma Sock zc GbE
Puma ScaIP dc SCI
0.000488281
0.000244141
0.00012207
6.10352e-05
3.05176e-05
1.52588e-05
7.62939e-06
3.8147e-06
1.90735e-06
sec (accumulate)
8
16
32
64
128
256
512
1024
2048
Bytes transferred

Tillegg C

ARMCI over ScaFun, kommunikasjonsprotokoll

Figuren forklarer kommunikasjonsprotokollen benyttet for ARMCI over ScaFun. Protokollen foretar ingen *remote-read* over SCI.


ARMCI over ScaFun, kommunikasjonsprotokoll
Sender (A)
Mottaker(B)
Interrupter
Recieve−buffer
Server_chunk
Notify
Send−buffer
Klient_chunk
Ack
Steg 0:
Interrupt genereres hos B
Steg 1:
B analyserer interruptet
og poller på tilhørende
"notify−word"
Steg 2:
A sitt "ack word"
nullstilles
Steg 3:
A fyller B sin
recieve chunk med så mye som
mulig av dataene i sitt send−buffer.
Steg 4:
A sjekker
(med SciCheckpoint)
at dataene kom frem.
Steg 5:
A setter B sitt
"notify word"
Steg 6:
A poller på sitt
"ack−word" for
bekreftelse på
at meldingen er
mottatt av B.
Steg 7:
B kopierer dataene
ut fra sin chunk og
inn i recieve−bufferet
Steg 8:
B nullstiller sitt
"notify−word" og
sjekker request−
headeren om det
kommer mer data
Steg 9:
B setter A sitt
"ack−word"
Steg 10:
Hvis mer data skal sendes
gjentas stegene 1−10
ellers er sendingen utført.

Tillegg D

Konvertering mellom ARMCI- og MPI-datatyper

Konvertering mellom ARMCI- og MPI-datatyper:

```
/*
  Argument description:
  ptr:           Pointer to start of memory area of the n-dim array
  stride_levels: Dimension of input array
  stride_arr[]:  Array has stride_levels levels of stride.
  Source array of stride distance
  count[]:       count[0] is sizeof(datatype in array)*num of datatype
  count[1]-count[stride_levels] is number of elements to
  transfer in that dimension.
  proc:          MPI-Process to sent to / recieve from
  tag:           Tag of message to send / recieve
  op:            ARMCI operation (possible later use)
  scale:         ARMCI ACC scale  (possible later use)
*/

void armci_write_strided_mpi(void *ptr, int stride_levels, int stride_arr[],
     int count[], int op, void *scale, int proc,
     int tag) {

  int i, j;
  long idx;
  int n1dim;
  int bvalue[MAX_STRIDE_LEVEL], bunit[MAX_STRIDE_LEVEL];
  MPI_Datatype sm_write_slice;
  int *array_of_blocklengths;
  long *array_of_displacements;

  /* Number of elements when viewing array in the first dimension */
  n1dim = 1;
  for(i=1; i<=stride_levels; i++)
    n1dim *= count[i];

  if(count[0] >= MPI_DATATYPE_LIMIT) {
    /* Creating array of blocklengths */
    array_of_blocklengths = (int*)calloc(sizeof(int),n1dim);
    /* Creating array of displacement */
    array_of_displacements = (long*)calloc(sizeof(long),n1dim);
```

```
  }
  /* Calculate the destination indices */
  bvalue[0] = 0; bvalue[1] = 0; bunit[0] = 1; bunit[1] = 1;

  for(i=2; i<=stride_levels; i++) {
    /* array of bvalue is set to 0 */
    bvalue[i] = 0;
    /* array of bunit is set to value of previous cell
       times value of previous cell count */
    bunit[i] = bunit[i-1] * count[i-1];
  }
  for(i=0; i<n1dim; i++) {
    idx = 0;
    for(j=1; j<=stride_levels; j++) {
      idx += bvalue[j] * stride_arr[j-1];
      if((i+1) % bunit[j] == 0) bvalue[j]++;
      if(bvalue[j] > (count[j]-1)) bvalue[j] = 0;
    }
    /* We are sending several short MPI-messages rather than
       one big MPI-datatype message */
    if(count[0] < MPI_DATATYPE_LIMIT) {
      MPI_Send((char*)ptr+idx, count[0], MPI_CHAR, proc, tag, sm_armci_world);
    }
    else {
      /* Setting blocksize */
      array_of_blocklengths[i] = count[0];
      /* Setting displacement */
      array_of_displacements[i] = idx;
    }
  }
  if(count[0] >= MPI_DATATYPE_LIMIT) {
    /* Making new MPI Strided datatype */
    MPI_Type_hindexed(n1dim, array_of_blocklengths,
      array_of_displacements, MPI_CHAR,
      &sm_write_slice);
    /* Commiting datatype for later use */
    MPI_Type_commit(&sm_write_slice);
    MPI_Send(((char*)ptr), 1, sm_write_slice, proc, tag, sm_armci_world);
    /* Releasing datatype (will not be used later) */
    MPI_Type_free(&sm_write_slice);
    free(array_of_blocklengths);
    free(array_of_displacements);
  }
}
```