UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Uma biblioteca para programação paralela por troca de mensagens de *clusters* baseados na tecnologia SCI

por

FÁBIO ABREU DIAS DE OLIVEIRA

Dissertação submetida à avaliação,
como requisito parcial para a obtenção do grau de Mestre
em Ciência da Computação

Prof. Dr. Philippe Olivier Alexandre Navaux
Orientador

Porto Alegre, abril de 2001.

CIP – Catalogação na Publicação

Oliveira, Fábio Abreu Dias de

Uma biblioteca para programação paralela por troca de mensagens de clusters baseados na tecnologia SCI / por Fábio Abreu Dias de Oliveira. – Porto Alegre: PPGC da UFRGS, 2001.

131 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2001. Orientador: Navaux, Philippe Olivier Alexandre.

1. Troca de mensagens. 2. Redes de alto desempenho. 3. SCI. 4. Computação baseada em agregados. 5. DECK. I. Navaux, Philippe Olivier Alexandre. II. Título.

Agradecimentos

Aqui manifesto meus sinceros agradecimentos ao Prof. Philippe Navaux, pelo constante apoio, pela oportunidade que me concedeu de realizar este Trabalho e por proporcionar, sob sua liderança, um ambiente propício à pesquisa de excelência e ao saudável convívio social.

Agradeço aos colegas Rafael Bohrer Ávila, Marcos Ennes Barreto e Élgio Schlemer, pela acolhida no Grupo de Pesquisa, pela aprazível convivência e pelas frutíferas conversações técnicas (e não-técnicas!) que entabulamos.

Agradeço à CAPES, pelo fundamental apoio financeiro dispensado à execução desta pesquisa.

Externo, nesta oportunidade, um especial agradecimento aos meus pais, pela garantia de uma fonte inesgotável de carinho e compreensão, e pelos incessantes estímulo e incentivo em todas as etapas de minha vida acadêmica.

Sumário

Lista de Abreviaturas

ADI	Abstract Device Interface
API	Application Programming Interface
BIP	Basic Interface for Parallelism
CC-NUMA	Cache Coherent - Non-Uniform Memory Access
CML	Common Messaging Layer
CRC	Cyclic Redundancy Code
DECK	Distributed Execution Communication Kernel
DMA	Direct Memory Access
DSM	Distributed Shared Memory
FPU	Floating Point Unit
GPPD	Grupo de Processamento Paralelo e Distribuído
LAMP	Local Area Multiprocessor
MMU	Memory Management Unit
MMX	Math Matrix Extensions
MPI	Message Passing Interface
MPP	Massively Parallel Processors
NUMA	Non-Uniform Memory Access
POSIX	Portable Operating System Interface
PVM	Parallel Virtual Machine
RRB	Receive Ring Buffer
SCI	Scalable Coherent Interface
SCI-VM	SCI - Virtual Memory
SIMD	Single Instruction Multiple Data

SISCI	Standard Software Infrastructure for SCI-based Parallel Systems
SMI	Shared Memory Interface
SMP	Symmetric Multiprocessor
SPMD	Single Program Multiple Data
SSP	Scali Software Platform
VIA	Virtual Interface Architecture
YASMIN	Yet Another Shared Memory Interface

Lista de Figuras

Lista de Tabelas

Resumo

A presente Dissertação propõe uma biblioteca de comunicação de alto desempenho, baseada em troca de mensagens, especificamente projetada para explorar eficientemente as potencialidades da tecnologia SCI (*Scalable Coherent Interface*). No âmago da referida biblioteca, a qual se denominou DECK/SCI, acham-se três protocolos de comunicação distintos: um protocolo de baixa latência e mínimo *overhead*, especializado na troca de mensagens pequenas; um protocolo de propósito geral; e um protocolo de comunicação que emprega uma técnica de *zero-copy*, também idealizada neste Trabalho, no intuito de elevar a máxima largura de banda alcançável durante a transmissão de mensagens grandes. As pesquisas desenvolvidas no decurso da Dissertação que se lhe apresenta têm por mister proporcionar um ambiente para o desenvolvimento de aplicações paralelas, que demandam alto desempenho computacional, em *clusters* que se utilizam da tecnologia SCI como rede de comunicação. A grande motivação para os esforços envidados reside na consolidação dos *clusters* como arquiteturas, a um só tempo, tecnologicamente comparáveis às máquinas paralelas dedicadas, e economicamente viáveis.

A interface de programação exportada pelo DECK/SCI aos usuários abarca o mesmo conjunto de primitivas da biblioteca DECK (*Distributed Execution Communication Kernel*), concebida originalmente com vistas à consecução de alto desempenho sobre a tecnologia Myrinet.

Os resultados auferidos com o uso do DECK/SCI revelam a eficiência dos mecanismos projetados, e a utilização profícua das características de alto desempenho intrínsecas da rede SCI, haja visto que se obteve uma performance muito próxima dos limites tecnológicos impostos pela arquitetura subjacente. Outrossim, a execução de uma clássica aplicação paralela, para fins de validação, testemunha que as primitivas e abstrações fornecidas pelo DECK/SCI mantêm estritamente a mesma semântica da interface de programação do original DECK.

Palavras-chave: troca de mensagens, redes de alto desempenho, SCI, computação baseada em agregados, DECK

TITLE: “A LIBRARY FOR MESSAGE-PASSING PARALLEL PROGRAMMING OF CLUSTERS BASED ON THE SCI TECHNOLOGY”

Abstract

This Thesis proposes a high-performance communication library based on message-passing, which was specifically designed to efficiently explore the advanced capabilities of the SCI (Scalable Coherent Interface) technology. The main components of such library, named DECK/SCI, are three distinct communication protocols: a minimal overhead and low-latency protocol, optimized for exchanging small messages; a general-purpose protocol; and a protocol which makes use of a zero-copy communication technique that was also developed in this work in order to increase the maximum achievable bandwidth for large messages. The research activities carried out during this work aimed at providing an environment for the development of CPU-demanding parallel applications in clusters of PCs whose interconnect is SCI. The main motivation for all efforts made to accomplish the mentioned aim is the fact that nowadays’ clusters undoubtedly comprise a computer architecture at the same time economically viable, and technologically comparable to the expensive dedicated parallel machines.

The API provided by DECK/SCI has the same primitives as the programming interface of DECK (Distributed Execution Communication Kernel) library, which was originally conceived to obtain high-performance over Myrinet-based clusters.

The obtained results with DECK/SCI reveal the designed mechanisms efficiency and the SCI high-performance characteristics useful exploitation, once the achieved performance is near to hardware limits imposed by technological issues. Also, the DECK/SCI validation, by executing a classical parallel application, shows that the primitives and abstractions provided by DECK/SCI present exactly the same semantics as those of the original DECK library.

Keywords: message-passing, high-performance networks, SCI, cluster computing, DECK

1 Introdução

Na diversidade de tópicos de pesquisa que compõem o arcabouço da área de Processamento Paralelo e Distribuído, merecem destaque os incessantes esforços envidados na busca de alto desempenho computacional. Vários projetos de máquinas paralelas dedicadas testemunham a preocupação em se levar a termo um sistema de computação realmente capaz de suprir a demanda por poder computacional, suscitada por aplicações que se tornaram tão habituais quanto imprescindíveis no mundo moderno, permeando os mais diversos campos do conhecimento, como por exemplo, previsão de tempo, mecânica dos fluidos, física de partículas, dinâmica molecular, etc.

Um dos fatores que impedem a disseminação das máquinas paralelas dedicadas é o seu elevado custo, inevitavelmente limitando a percepção dos resultados de pesquisa a instituições economicamente favorecidas. À luz desta barreira econômica e do crescente clamor por sistemas de computação eficientes, soluções alternativas passaram a ser vislumbradas e novos horizontes abriram-se para a pesquisa. O marco inicial desta mudança de direção foi a concepção da idéia de agregados de computadores ou, simplesmente, *clusters*.

O termo *cluster* é habitualmente empregado para designar um conjunto de *nodos* homogêneos — PCs ou estações de trabalho —, conectados por uma rede de comunicação de alto desempenho, compondo uma arquitetura que pode ser utilizada para a execução de aplicações paralelas com performance próxima a que se pode esperar de uma máquina paralela dedicada. O grande atrativo dos *clusters* é, pois, a ótima relação custo–performance, a qual os torna uma opção que se contrapõe de forma competitiva às dispendiosas máquinas ditas “massivamente paralelas” — MPPs.

O que torna os *clusters* economicamente viáveis é a ampla disponibilidade de seus componentes vitais, os quais, em verdade, são tão-somente computadores pessoais. Por outro lado, o que os faz tecnologicamente comparáveis às máquinas paralelas dedicadas são as avançadas redes de comunicação que interligam os computadores que os compõem.

As repercussões e possibilidades de uso que advieram da concepção de *cluster* foram tamanhas que, atualmente, a computação baseada em agregados — ou *cluster computing* — constitui-se em uma área de pesquisa por si só. Corroboram esta observação infindos projetos, publicações e eventos neste campo [AHM 2000][BUY 99][HIP 97][PFI 98][STE 99].

A mola propulsora dos *clusters* está, justamente, nos ininterruptos aprimoramentos experimentados pela tecnologia de redes de comunicação; destarte, o foco das pesquisas atuais está direcionado tanto ao desenvolvimento de hardware de interconexão, quanto à formulação de métodos para assegurar a profícua utilização do mesmo, o que implica na especificação e implementação de protocolos de comunicação *ad hoc*, haja visto que a simples presença de redes de alto desempenho em *clusters*, tais como Myrinet [BOD 95], SCI (*Scalable Coherent Interface*) [IEE 93] e Gigabit Ethernet [IEE 97a], não impede a subutilização do hardware de comunicação. Em outras palavras, as redes de alto desempenho trazem consigo a necessidade de uma mudança abrupta na forma como os protocolos de comunicação são implementados, a fim de que

as aplicações possam fazer uso efetivo da elevada largura de banda e da baixa latência que o hardware disponibiliza. Especificamente falando, soluções baseadas na tradicional pilha de protocolos TCP/IP não mais são aplicáveis.

O grande mérito dos protocolos de comunicação específicos para as redes de alto desempenho é o acesso direto aos recursos do hardware — interfaces de rede —, em nível de usuário, evitando, durante operações de comunicação, a ingerência do sistema operacional e excessivas cópias de dados entre diferentes regiões da memória, de sorte que se possa garantir às aplicações paralelas as baixas latências suportadas pelo hardware especial.

Pode-se asserir que Myrinet e SCI são duas tecnologias de comunicação de destaque no cenário de *cluster computing*. A primeira, indubitavelmente, auferiu maior aceitação e popularidade, estando presente na maioria dos *clusters* do mundo inteiro; SCI, por sua vez, encontra seu nicho mormente em países europeus. Embora tenham o mesmo propósito, trata-se de duas tecnologias totalmente diferentes. Ao passo que Myrinet é uma rede fundamentalmente de troca de mensagens, SCI tem por fulcro um espaço de endereçamento global permeando todos os *nodos* do *cluster*, de modo a propiciar a comunicação através de compartilhamento de memória. Em última análise, SCI proporciona um mecanismo de DSM — *Distributed Shared Memory* — implementado em hardware.

Embora cada *nodo* do *cluster* tenha sua própria memória local, a comunicação em SCI é efetuada através do espaço de endereçamento de 64 bits instaurado pelo hardware, sendo que os 16 bits mais significativos especificam um determinado *nodo*, e os 48 bits restantes referem-se à memória local do *nodo* designado na outra porção do endereço. Em virtude disso, posto que se tenha estabelecido, com o auxílio do driver SCI, segmentos de memória compartilhada, a comunicação entre *nodos* dá-se, em nível de usuário, por simples acessos à memória, sem a ingerência do sistema operacional e de qualquer protocolo adicional, o que redunda em latências tão baixas quanto 2,3 μs.

À primeira vista, o paradigma de comunicação mais intuitivo a ser empregado em *clusters* SCI parece ser o compartilhamento de memória; contudo, não se pode ignorar o fato de que a baixíssima latência, inerente a esta tecnologia, dá ensejo à formulação e implementação de protocolos de comunicação, por troca de mensagens, extremamente eficientes.

Constatam-se diversos grupos de pesquisa trabalhando no desenvolvimento de bibliotecas de comunicação baseadas em troca de mensagens, especificamente para *clusters* SCI, havendo disponíveis, inclusive, implementações dos padrões MPI [MPI 94] e PVM [GEI 94]. Observa-se, no entanto, que alguns protocolos de comunicação propostos subutilizam de forma exacerbada as potencialidades da tecnologia SCI, pela utilização de mecanismos inadequados, sob a ótica de desempenho; outros, apesar de eficientes e inteligentemente formulados, são impedidos de proporcionar às aplicações paralelas desempenhos mais próximos do limite do hardware SCI, devido a restrições impostas no nível da interface de programação, como é o caso das implementações do padrão MPI que, por natureza, torna impossível a efetivação da comunicação através de esquemas do tipo *zero-copy*, visto que exige, necessariamente, a cópia explícita da mensagem para um buffer fornecido pelo usuário.

No contexto de *cluster computing*, foi concebido, no GPPD — Grupo de Processamento Paralelo e Distribuído — do Instituto de Informática da UFRGS, o DECK/Myrinet [BAR 2000], uma biblioteca que fornece serviços de comunicação e multiprogramação, originalmente desenvolvida com base no protocolo de baixo nível BIP [PRY 98], com dois propósitos: consecução de alto desempenho, e fornecimento de adequadas abstrações para a programação de *clusters* Myrinet por troca de mensagens.

Ao DECK/Myrinet sobreveio o recente projeto MultiCluster [BAR 2000a], também no GPPD da UFRGS, com o objetivo de integrar *clusters* Myrinet e SCI, de modo que esta arquitetura integrada possa vir a ser usada como uma máquina paralela única. Para tanto, planeja-se, futuramente, disponibilizar aos usuários um ambiente de programação único, que seja capaz de entremear a comunicação entre os diferentes *clusters*.

Face ao projeto MultiCluster, e à relativa ineficiência dos protocolos de comunicação atualmente disponíveis para *clusters* SCI, propõe-se, nesta Dissertação de Mestrado, o DECK/SCI: uma biblioteca para programação paralela por troca de mensagens de *clusters* baseados na tecnologia SCI. Este Trabalho propugna pela formulação e implementação de protocolos de comunicação de alto desempenho, específicos para SCI, promovendo baixa latência e mecanismos de *zero-copy*, com vistas à obtenção de performance próxima dos limites impostos pela arquitetura subjacente. Além de alto desempenho, o DECK/SCI tem por meta fornecer aos seus usuários a mesma API do DECK/Myrinet, o que é vital para a futura integração dos *clusters*.

O texto a seguir está organizado de tal forma que a leitura de cada capítulo é essencial para a compreensão do capítulo subseqüente. O capítulo 2 versa sobre a tecnologia SCI em um lato sentido, fornecendo todo o embasamento teórico para o restante do texto. O capítulo 3 apresenta as interfaces de programação, baseadas em memória compartilhada, desenvolvidas para a programação de *clusters* SCI. Discutem-se APIs de baixo e alto nível, desde o driver, até sofisticadas bibliotecas. O conhecimento destas interfaces de programação é importante para respaldar decisões a serem tomadas quando do desenvolvimento de qualquer software básico para *clusters* SCI. O capítulo 4, por sua vez, discorre sobre os trabalhos acerca de troca de mensagens em *clusters* SCI, mostrando as técnicas que têm sido adotadas para a elaboração de protocolos de comunicação. O capítulo 5 formaliza a proposta do DECK/SCI, detalhando as primitivas da sua interface de programação e as abstrações fornecidas. Por sua vez, o capítulo 6 complementa seu antecessor, explicando, no detalhe, o projeto e a implementação propriamente dita do DECK/SCI, os protocolos de comunicação, técnicas e mecanismos propostos. Constam também do capítulo 6 as justificativas de cada importante decisão tomada. O capítulo 7 analisa o desempenho dos protocolos de comunicação do DECK/SCI, tanto em relação à performance máxima da rede SCI, quanto em comparação com outras bibliotecas semelhantes. Por fim, o capítulo 8 dá conta das conclusões que deste Trabalho decorreram, e tece os últimos comentários.

2 SCI: *Scalable Coherent Interface*

SCI não é uma rede de interconexão de alto desempenho usada em *clusters*; antes, assume uma perspectiva muito mais ampla: trata-se de um padrão [IEE 93] especificado para conectar até 64k dispositivos — não somente computadores —, cujo âmago é um conjunto de protocolos, em nível de hardware, que trazem consigo soluções para as limitações inauferíveis dos barramentos, limitações estas decorrentes da centralização que lhes é peculiar. A distribuição é, pois, a tônica dos protocolos e mecanismos especificados no padrão SCI, e o meio pelo qual esta tecnologia relativamente recente disponibiliza, aos dispositivos que conecta, largura de banda elevada e incremental, bem como reduzida latência de comunicação.

Este capítulo dedica-se a uma apresentação da tecnologia SCI em um lato sentido, no intuito de evitar qualquer interpretação errônea ou concepções simplistas acerca da referida sigla. No entanto, enfatiza-se a aplicação de SCI atinente ao contexto deste trabalho, qual seja, seu emprego como uma rede de alto desempenho que interliga *nodos* de um *cluster*, sendo abordados as particularidades e eventuais inconvenientes que advêm deste uso.

Com esta postura, o presente capítulo fornece bases teóricas para o restante do texto.

2.1 Histórico

A tecnologia SCI originou-se dos esforços de projetistas de barramentos de alto desempenho durante pesquisas realizadas no projeto dos barramentos Fastbus (IEEE 960) e Futurebus+ (IEEE 896.x). Aos referidos projetistas fora lançada a incumbência de aprimorar as técnicas de comunicação usadas em barramentos para multiprocessadores, de forma a possibilitar a profícua utilização da largura de banda com o mínimo de contenção. Contudo, percebeu-se que a velocidade dos microprocessadores não tardaria a superar a capacidade de qualquer barramento suportar multiprocessamento significativo, fato que faria sucumbirem os esforços de pesquisa despendidos em tais barramentos de alto desempenho.

Em assim sendo, foi organizado um grupo de estudo, em 1987, para analisar o impacto das implicações que se afiguravam. De novembro de 1987 a julho de 1988, o assim denominado *Superbus Study Group* examinou propostas alternativas para prover serviços à maneira de barramentos sem, entretanto, incutir-lhes as limitações que lhes são peculiares. Em julho de 1988, formou-se um grupo de trabalho oficial do IEEE. Inicialmente, os membros do grupo tinham um ponto-de-vista voltado estritamente a barramento mas, ao depararem-se com os seus entraves, paulatinamente aclaravam-se os moldes da solução e evoluía-se para uma proposta mais robusta. O resultado foi o padrão IEEE SCI.

As origens de SCI tornam claro por que tal tecnologia de interconexão está centrada na comunicação por memória compartilhada. Gustavson e Li [GUS 96] utilizam a sigla LAMP — Local Area Multiprocessor — para designar redes como SCI,

posto que se tem um ambiente semelhante a multiprocessadores — em que se evidencia um espaço de endereçamento global de memória —, porém, podendo assumir dimensões de uma rede local.

2.2 Filosofia e objetivos

Em suma, SCI descreve hardware e protocolos que proporcionam aos processadores conectados a visão de memória compartilhada típica dos barramentos; por conseguinte, são também especificadas as transações correlatas do tipo *read*, *write* e *lock*, aplicadas sobre regiões do espaço de endereçamento global de memória, sem qualquer intervenção de software.

Ao contrário de outros sistemas de interconexão, no entanto, a rede SCI e seus protocolos subjacentes são totalmente distribuídos, o que implicou na adoção de *links* unidirecionais ponto-a-ponto e na implementação de DSM — Distributed Shared Memory — em hardware, isto é, embora seja provido um espaço de endereçamento global, acessível por todos os *nodos*, a memória é, em verdade, fisicamente distribuída.

No decurso do processo de especificação do padrão, vários objetivos ambiciosos foram levados em consideração, alguns dos quais não foram atingidos a contento. Dentre as principais metas traçadas, acham-se:

- **Alto desempenho**: Este é tido como o objetivo primordial da tecnologia SCI, o qual está voltado à consecução de alta largura de banda, baixa latência e mínimo *overhead* à CPU durante operações de comunicação.

- **Escalabilidade**: A escalabilidade encerra várias facetas — performance, endereçamento, economia, tecnologia, etc. [GUS 94] Apud [HEL 99] —; porém, não raro, a única conotação considerada é a escalabilidade de performance, ou seja, permitir que a largura de banda total cresça à medida que aumente o número de *nodos* da rede. O projeto SCI esteve abalizado no lato sentido do termo escalabilidade, conquanto a escalabilidade de performance seja a mais destacável.

- **Suporte à coerência de *cache***: Visto que SCI é, em última análise, um mecanismo de DSM implementado em hardware, acessos a endereços de memória remotos são mais lentos do que acessos locais; portanto, é desejável a utilização de *caches* para que se evitem, sempre que possível, acessos remotos. Destarte, um eficiente suporte à coerência de *cache* foi estabelecido como uma das principais metas do padrão SCI.

- **Interface padronizada**: A intenção dos projetistas era a definição de uma interface que permitisse interconectar numerosos dispositivos, ainda que de diferentes fabricantes, em um único sistema de comunicação de alta velocidade. Dito de outra forma, SCI atuaria como um barramento capaz de conectar microprocessadores, módulos de memória e dispositivos de E/S.

2.3 Visão geral

SCI é um padrão de interconexão projetado para conectar até 64k *nodos*. Um *nodo* pode ser um computador completo — e.g. *workstation*, PC ou servidor —, um processador e sua respectiva *cache*, um módulo de memória, um controlador e dispositivo de E/S, ou ainda, um *bridge* para outros barramentos ou redes de interconexão, conforme ilustrado na figura 2.1 [HEL 99].

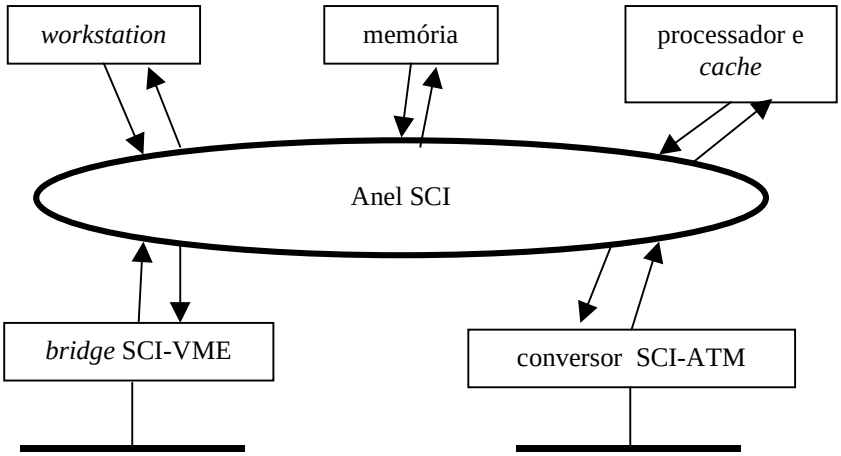

FIGURA 2.1 - *Nodos* conectados em um anel SCI.

O modelo genérico de sistema SCI esboçado na figura 2.1 exige, em cada *nodo*, a presença de uma interface padronizada que permita conectá-lo à rede SCI.

Embora seja possível compor uma rede SCI com topologia complexa e irregular [BUG 99] [RIC 99], incluindo redes de múltiplos estágios, o padrão prevê o emprego de topologias simples, cujo bloco de construção é o anel. Para sistemas de porte limitado, um anel pequeno parece ser mais adequado; por outro lado, sistemas maiores tornam vantajosas topologias como anel de anéis, vários anéis conectados por um *switch*, ou ainda, a conexão toroidal (*torus*). Redes SCI para sistemas de considerável porte encontram-se ilustradas na figura 2.2 [HEL 99].

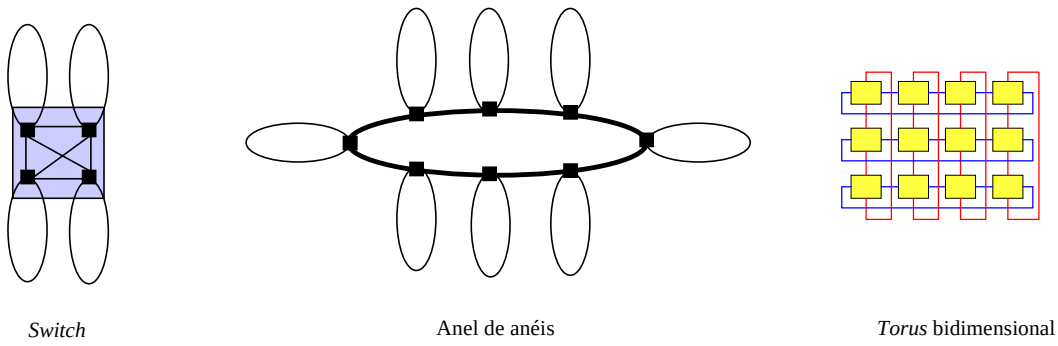

FIGURA 2.2 - Topologias de redes SCI.

2.3.1 Endereçamento

Conforme mencionado na seção 2.2, SCI disponibiliza um espaço global de endereçamento de memória, visível por todos os *nodos*. Os endereços do referido espaço compõem-se de 64 bits, sendo divididos em duas partes, a saber:

- os 16 bits mais significativos especificam a identificação de um *nodo* (eis por que pode haver até 64k *nodos*);
- os 48 bits restantes são usados para endereçar a memória local ao *nodo* designado na outra parte do endereço.

2.4 Camadas do padrão SCI

A fim de agrupar protocolos e mecanismos afins, o padrão SCI foi dividido em três camadas distintas: **camada física**, **camada lógica** e **camada de coerência de *cache***. Sempre que possível, as especificações formais foram escritas na linguagem C, evitando-se assim ambigüidades e eventuais interpretações dúbias. Uma decorrência imediata desta escolha é o fato do padrão tornar-se automaticamente executável, o que é deveras útil para fins de simulação.

2.4.1 Camada física

Na camada física estão especificados os modelos de *links*, bem como as características elétrica, mecânica e térmica dos cabos, conectores e demais componentes SCI. Foram definidos três modelos de *links*, abaixo discriminados:

- *link* paralelo e elétrico, para distâncias pequenas (alguns metros), capaz de suportar 1 Gbyte/s;
- *link* serial e elétrico, para distâncias intermediárias (dezenas de metros), o qual suporta 1 Gbit/s;
- *link* serial e ótico, para distâncias longas (quilômetros), permitindo transmissões à taxa de 1 Gbit/s.

A consideração mais importante acerca da camada física diz respeito à forma como se dão as interconexões: os *links* entre os *nodos* são, necessariamente, unidirecionais e ponto-a-ponto. Esta decisão de projeto adveio da necessidade de evitar a centralização típica dos barramentos, permitindo a efetivação de comunicação concorrente entre os *nodos* partícipes da rede SCI. Ademais, o número de *links* aumenta à medida que são acrescentados *nodos* à rede, o que redunda em uma elevação da largura de banda total (*aggregate bandwidth*); melhor dito, é constante a largura de banda disponível para cada *nodo*. Comunicações concorrentes, por sua vez, garantem a escalabilidade de desempenho.

A escolha de *links* unidirecionais e ponto-a-ponto representou o primeiro passo na definição das soluções totalmente distribuídas apresentadas pelo padrão SCI.

2.4.2 Camada lógica

No nível lógico são especificados protocolos, transações, tipos e formatos de pacote, além da interface-padrão de conexão e dos mecanismos de detecção de erros.

As transações SCI compõem-se de duas subações — *request* e *response* —, envolvendo um par de *nodos*. Um *nodo* requisitante submete um pacote do tipo *request* a um dado *nodo* que, se a transação em questão exigir, retorna um pacote do tipo *response*. Cada subação, por sua vez, segue um protocolo que garante o controle do fluxo dos pacotes. Assim, tanto em requisições (*requests*), como em respostas (*responses*), é necessário o tráfego de dois pacotes: um pacote *send*, que efetivamente corresponde à requisição ou à resposta, e um pacote *echo*, o qual informa se o pacote *send* que o originou foi aceito ou rejeitado, permitindo o reenvio em caso de rejeição. A figura 2.3 retrata todas as nuanças que uma transação SCI encerra.

Este protocolo baseado em *echo*, ao tempo em que garante o controle de fluxo, impede que seja assegurada a entrega ordenada de pacotes. Perceba-se, também, que o *echo* não é uma confirmação fim-a-fim da transação; tal confirmação se dá quando do recebimento efetivo da resposta exigida pela transação. Convém comentar que o padrão SCI estabelece o limite de 64 transações pendentes por *nodo*.

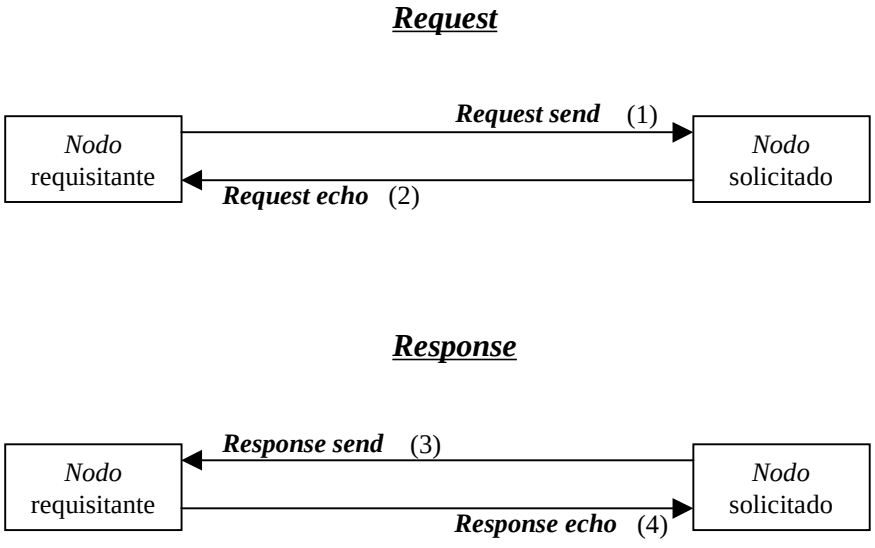

FIGURA 2.3 - Etapas de uma transação SCI.

SCI especifica as transações tipicamente disponibilizadas por barramentos — *read*, *write* e *lock* — e, adicionalmente, as transações *move* e *event*.

Read, *write* e *lock* seguem a semântica habitual. A primeira é usada para ler o conteúdo de uma posição de memória de um *nodo* solicitado, de modo que o pacote

response retorna ao solicitante as informações efetivamente lidas. *Write*, por sua vez, é usada para escrever em uma posição de memória de um *nodo* solicitado; neste caso, o pacote *response* retornado ao solicitante contém informações sobre a operação requisitada, sendo possível determinar a eventual ocorrência de erros durante a escrita. A transação *lock* corresponde a uma operação que realiza, atomicamente, leitura, modificação e escrita em uma posição de memória, permitindo a implementação de semáforos ou *mutexes*, necessários para garantir a exclusão mútua no acesso a dados compartilhados. No caso da transação *lock*, o pacote *response* carrega o antigo conteúdo da região de memória especificada no *nodo* requisitado.

As três transações supradelineadas pertencem à classe das **transações com resposta** que, por assim o serem, provêem naturalmente um mecanismo de confirmação fim-a-fim.

Diferentemente de *write*, a transação *move* não prevê um pacote do tipo *response*, que indica o sucesso ou não da operação; por conseguinte, trata-se de uma escrita mais eficiente e útil nos cenários em que se pode conviver com eventuais erros de comunicação. Observe-se, porém, que mesmo no caso do *move* há ainda controle de fluxo, posto que o pacote *echo* continua presente durante a requisição (*request*).

Finalmente, tem-se a transação *event*. Seu objetivo é distribuir um *time stamp* para fins de sincronização global em um sistema SCI. Ademais, pelo fato de preterir confirmação (*response*) e controle de fluxo (*echo*), pode também ser usada para levar a efeito rápidas transferências de dados, sob certas condições.

A figura 2.4 apresenta uma resenha das propriedades dos cinco tipos de transação definidos no padrão SCI.

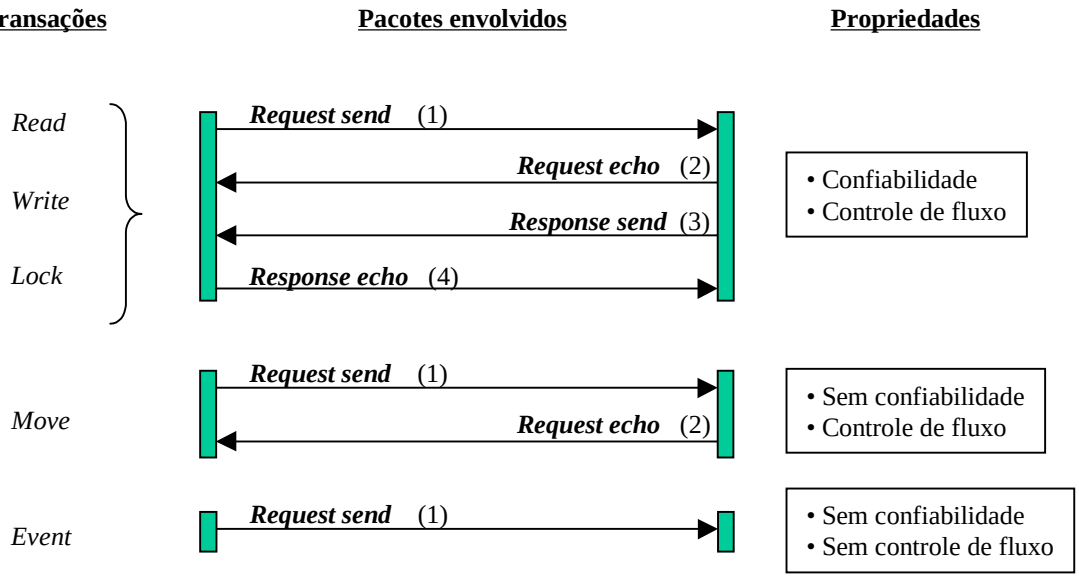

FIGURA 2.4 - Propriedades das cinco transações SCI.

Na camada lógica é também definida a interface-padrão de conexão dos *nodos* à rede SCI. Tal interface lida com dois *links* — um de entrada, e outro de saída —, uma vez que cada *link* é unidirecional (ver seção 2.4.1). Sua principal incumbência é enviar

pacotes ao *link* de saída e, simultaneamente, colher pacotes do *link* de entrada, seja para consumi-los, seja para armazená-los com vistas a um futuro envio, no caso de pacotes que não sejam destinados ao *nodo* em questão. Os pacotes cujo endereço-destino refira-se a outro *nodo* devem ser transmitidos pelo *link* de saída ao próximo *nodo* do anel; para tanto, as interfaces adotam a estratégia de roteamento *cut-through*, permitindo que os pacotes sejam enviados antes que tenham sido integralmente recebidos.

Por fim, a camada lógica do padrão SCI especifica mecanismos de detecção de erros por hardware. Cada pacote é protegido por um CRC de 16 bits; além disso, os pacotes podem conter campos com códigos de erro. Mecanismos de *time-out* são usados para a identificação de pacotes perdidos ou corrompidos. Todavia, o padrão não define estratégias específicas para a recuperação ante um estado errôneo, pois se espera que tal atribuição seja do software, por exemplo, drivers ou APIs de baixo nível.

2.4.3 Camada de coerência de *cache*

Nesta camada são especificados conceitos e protocolos de hardware para garantir a coerência de *cache* entre os processadores conectados pelo padrão SCI. Visto que SCI constitui-se em um sistema inerentemente NUMA — Non-Uniform Memory Access —, o uso de *cache* em segmentos de memória compartilhados é de vital importância para que se evitem desnecessários e custosos acessos remotos à memória.

O protocolo de coerência de *cache* definido pelo padrão SCI adota o modelo de compartilhamento baseado em um “escritor” e vários “leitores”, seguindo o esquema *write-invalidation*. A implementação é necessariamente distribuída, adotando listas associadas a cada segmento compartilhado, listas estas atualizadas cooperativa e concorrentemente pelo controlador de memória e pelos processadores que detêm, em suas *caches* locais, cópias de segmentos compartilhados. Cada lista mantém a relação de todos os processadores que compartilham o segmento a ela associado; a fim de facilitar o gerenciamento, as listas são duplamente encadeadas.

Durante acessos a segmentos de memória compartilhados que integram o esquema de coerência de *cache*, várias transações são necessárias para manter a integridade dos dados e assegurar a correta atualização das listas encadeadas. O protocolo resultante é bastante complexo e de difícil implementação em hardware; ademais, há arquiteturas que tornam impossível a implementação de tal protocolo, conforme pode ser constatado na seção 2.6.2. Por estas razões, a camada de coerência de *cache* do padrão SCI é considerada opcional.

Para uma descrição exaustiva do protocolo de coerência de *cache* especificado pelo padrão SCI, o leitor é instado a consultar o artigo de Gustavson e Li [GUS 96].

2.5 Aplicações do padrão SCI

Conforme suas origens deixam transparecer (ver seção 2.1), SCI foi originalmente concebido como um mecanismo de interconexão para multiprocessadores. Atualmente, existem vários sistemas de computação comerciais que se utilizam do padrão SCI deste modo. Tipicamente, esta classe de sistemas adota SCI

para conectar múltiplos SMPs — Symmetric Multiprocessors —, constituindo uma grande arquitetura de memória compartilhada com coerência de *cache*. Resultam desta modalidade de interconexão sistemas do tipo CC-NUMA — Cache Coherent - Non-Uniform Memory Access. Pode-se citar, como representantes desta categoria, sistemas baseados em múltiplas placas com quatro processadores, placas estas conectadas pela tecnologia SCI. A figura 2.5 esboça tal abordagem de conexão.

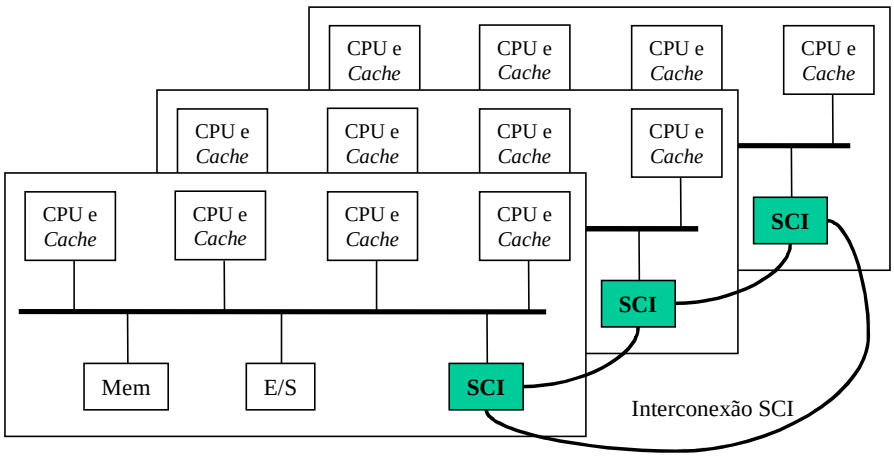

FIGURA 2.5 - Sistema SCI do tipo CC-NUMA baseado em SMPs.

O grande desafio no projeto de sistemas CC-NUMA como os acima citados reside na integração entre o protocolo de coerência de *cache* intraplaca — e.g. *snooping* — e o protocolo de coerência de *cache* SCI. A despeito disso, há implementações exitosas comercialmente disponíveis, tais como o multiprocessador Sequent NUMA-Q [CUL 99] e os servidores AViiON da Data General [DAT 99].

Ao contrário de multiprocessadores convencionais, o número de processadores conectados pela tecnologia SCI não é limitado por fatores como a largura e velocidade do barramento, visto que os *links* SCI são unidirecionais e ponto-a-ponto e, os protocolos, totalmente distribuídos.

Além da aplicação original como sistema de interconexão em multiprocessadores com coerência de *cache*, os projetistas envolvidos na especificação SCI não tardaram em perceber que tal tecnologia poderia também ser utilizada como rede de alto desempenho para *clusters*, porquanto permite que a comunicação entre dois *nodos* seja efetuada sem a ingerência de sistema operacional ou de qualquer protocolo adicional, dependendo sobretudo do hardware subjacente. Isto vai ao encontro da principal motivação para o emprego das redes de alto desempenho em *clusters*, qual seja, reduzir o caminho de comunicação entre dois processos de distintos *nodos*, com o mínimo de intervenção que não a do hardware, a fim de garantir baixa latência de comunicação. É justamente nesta aplicação do padrão SCI que está o cerne desta Dissertação.

2.6 SCI como rede de alto desempenho para *clusters*

Uma aplicação extremamente útil da tecnologia SCI é a construção de *clusters* de PCs ou *workstations*, com vistas à eficiente execução de aplicações paralelas que exigem considerável poder computacional. Neste caso, SCI pode ser considerada uma alternativa que se contrapõe às redes freqüentemente adotadas na área de computação baseada em *clusters*, tais como Myrinet [BOD 95], Fast Ethernet e Gigabit Ethernet. Diferentemente destas, porém, a comunicação entre os *nodos* do *cluster* é levada a efeito, no nível do hardware, através do espaço global de endereçamento de memória proporcionado pela rede SCI, e não por explícita troca de mensagens.

A semelhança entre *clusters* SCI, Myrinet, Fast Ethernet e Gigabit Ethernet está na necessidade de se dispor de uma placa de rede conectada ao barramento de E/S de cada *nodo*. A figura 2.6 ilustra a conexão entre dois *nodos* de um *cluster* SCI.

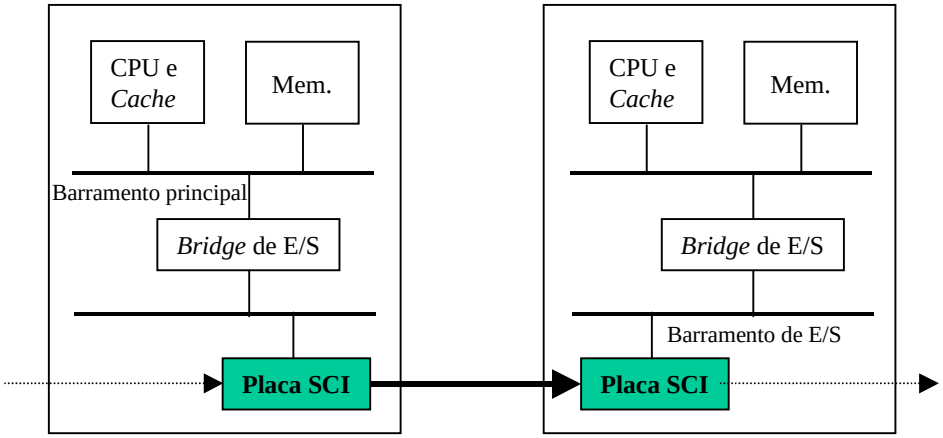

FIGURA 2.6 - Conexão entre dois *nodos* de um *cluster* SCI.

Em *clusters* baseados na tecnologia SCI, a comunicação entre os *nodos* é estabelecida pelo mecanismo de DSM implementado pelo hardware de conexão. As placas de rede SCI, juntamente com os drivers correlatos, originam o espaço global de endereçamento de memória conforme mostrado na figura 2.7.

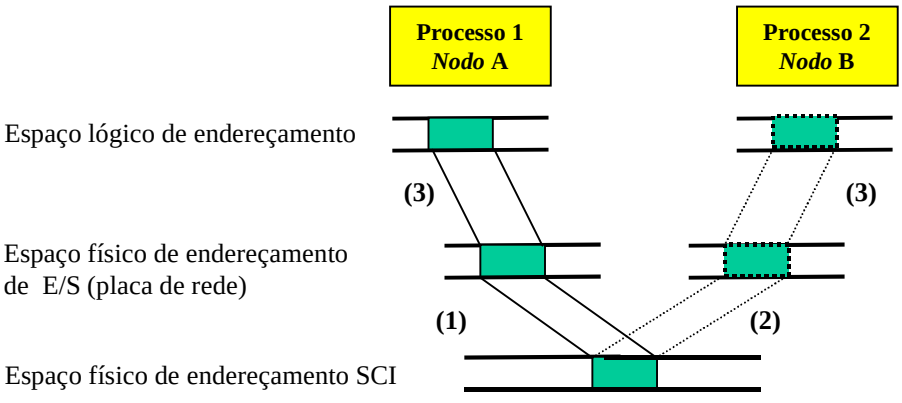

FIGURA 2.7 - Espaço global de endereçamento de memória em *clusters* SCI.

Suponha-se a situação retratada na figura 2.7. O "*nodo* A", primeiramente, cria um segmento de memória no seu espaço físico de endereçamento de E/S, em um endereço situado no intervalo destinado à placa de rede SCI, e o exporta ao espaço de endereçamento SCI, tornando o referido segmento compartilhado **(1)**. O "*nodo* B", por sua vez, importa o segmento DSM previamente exportado; para tanto, a placa de rede SCI mantém uma tabela de tradução de endereços que estabelece o mapeamento entre endereços do espaço SCI e endereços locais pertencentes ao espaço físico de endereçamento de E/S **(2)**. Após, o "processo 1", em execução no "*nodo* A", e o "processo 2", no "*nodo* B", podem efetivar o mapeamento do segmento DSM para seus espaços lógicos de endereçamento **(3)**. Estes últimos mapeamentos são efetuados pelas MMUs — Memory Management Units — dos processadores.

Uma vez que a seqüência de mapeamentos acima descrita tenha sido concluída, a comunicação entre os *nodos* poderá ser efetuada pelos processos, no nível de usuário, através das operações *load*/*store* das CPUs atuando sobre segmentos DSM. Quando as operações *load*/*store* agirem sobre segmentos DSM, as placas de rede SCI traduzirão, de forma transparente, transações do barramento de E/S em transações SCI, e vice-versa. Deste modo, acessos remotos à memória são transparentes aos processos e não há a intervenção do sistema operacional nem de protocolos adicionais, o que redunda em baixas latências de comunicação. Diferentemente de sistemas DSM puramente implementados em *software* — e.g. TreadMarks [AMZ 96] —, a memória é verdadeiramente compartilhada, não havendo a replicação de segmentos; por conseguinte, não existe qualquer *overhead* associado à manutenção da consistência dos segmentos, nem ao problema de falso compartilhamento, característico de sistemas DSM implementados em software.

A rigor, em SCI, tem-se o que se chama de comunicação mapeada em memória, isto é, são feitas leituras e escritas em endereços de memória remotos como se tais acessos fossem locais. O hardware oculta do usuário os mecanismos de comunicação necessários, sendo a única diferença perceptível o tempo de acesso: acessos remotos são mais lentos do que acessos locais. Assim, *clusters* SCI representam uma arquitetura tipicamente NUMA.

2.6.1 Hardware SCI para *clusters*

Atualmente, há apenas uma linha comercial de placas de rede SCI [LIA 99]. A empresa norueguesa *Dolphin Interconnect Solutions* fabrica placas SCI para PCs e para *workstations* da *Sun Microsystems*. Além de placas de rede, são também disponibilizados *switches*. As referidas placas permitem duas formas de comunicação entre os *nodos* de um *cluster*, a saber:

- através de leituras e escritas (*loads* e *stores*) da CPU em endereços remotos de memória, o que garante latências abaixo de 3 μs;

- através de DMA — Direct Memory Access —, por meio do qual os dados são copiados da memória de um *nodo* para a memória de outro, sem qualquer intervenção das CPUs.

Embora a comunicação por DMA libere as CPUs para outras tarefas, sua adoção é apenas indicada para a transferência de mensagens grandes, devido ao elevado tempo despendido pelo driver para inicializar o mecanismo.

As placas SCI provêem alguns recursos adicionais, não especificados no padrão, visando ao aumento do desempenho de comunicação. Por exemplo, várias transações SCI consecutivas, endereçando uma região contígua de memória, podem ser combinadas em uma única transação. Ademais, as placas são capazes de efetuar leitura antecipada (*prefetching*) de dados trazidos de endereços remotos.

Embora as placas possam suportar uma largura de banda de 200, 400 ou 500 Mbytes/s [GON 99], dependendo do *chip* controlador de *link* utilizado, devido a limitações da arquitetura de E/S (barramento) dos *nodos* do *cluster* pode-se esperar, em condições ideais, um *throughput* máximo por volta de 80-90 Mbytes/s.

No que concerne à topologia de *clusters* SCI, conforme mencionou-se na seção 2.3, o anel pode ser considerado o bloco básico de construção. Cada placa SCI possui dois canais unidirecionais — um de entrada e outro de saída —, de modo que o anel é a topologia mais intuitiva. Para um número pequeno de *nodos* — até oito —, não é necessário o uso de *switches*. Algumas placas apresentam dois canais de entrada e dois de saída, a fim de que se possa construir um *torus* bidimensional. Outrossim, vários anéis podem ser conectados através de *switches*, originando topologias variadas (ver figura 2.2).

2.6.2 Inconvenientes de *clusters* baseados na tecnologia SCI

Deve-se atentar para a grande diferença existente entre *clusters* SCI, cuja arquitetura baseia-se no esboço da figura 2.6, e multiprocessadores conectados pela tecnologia SCI, semelhantes ao ilustrado na figura 2.5. No caso dos *clusters*, os *nodos* são computadores completos — PCs ou *workstations* — e, portanto, a interface de conexão SCI é uma placa de rede necessariamente inserida no barramento de E/S de cada *nodo*. Em se tratando de multiprocessadores baseados em SCI (ver figura 2.5), os *nodos* são placas com vários processadores, e a interface de conexão SCI situa-se diretamente no barramento que interliga as CPUs e a memória.

A implicação imediata da referida diferença arquitetural é a impossibilidade de implementação do protocolo de coerência global de *cache*, especificado pelo padrão SCI (ver seção 2.4.3), em *clusters*, posto que as placas de rede SCI, por estarem no barramento de E/S, não percebem as transações entre CPU e memória que trafegam pelo barramento principal (ver figura 2.6). Torna-se impossível a uma dada placa de rede encaminhar as medidas necessárias para assegurar a coerência global de *cache* no âmbito de todo o *cluster*, ou seja, as *caches* locais às CPUs de cada *nodo* operam de forma totalmente independente, exigindo que acessos a segmentos remotos sejam sempre remotamente efetuados. É desnecessário asseverar os prejuízos ao desempenho que decorrem desta restrição arquitetural dos *nodos* dos *clusters*, uma vez que acessos remotos são mais lentos do que acessos locais, estando a diferença entre os tempos na faixa de uma ordem de grandeza.

Levando em consideração a discrepância entre acessos locais e remotos, os projetistas das placas de rede SCI incorporaram algumas técnicas para garantir baixa latência e elevada largura de banda durante operações de comunicação. A utilização de *buffers* de escrita e *buffers* de leitura, associados à idéia de *streaming*, representa um esforço neste sentido. As placas possuem oito *buffers* de escrita e oito de leitura, cada qual com 64 bytes[*]. Desta forma, requisições de escritas remotas em endereços adjacentes são coletadas até que o *stream* (*buffer*) a elas associado tenha sido totalmente preenchido. Somente após é gerada uma única transação de escrita na rede SCI, referente aos dados previamente armazenados. Além do total preenchimento de um *stream*, outros eventos podem causar a geração de uma transação SCI, tais como: requisição explícita do esvaziamento do mesmo (*flush*); escrita em um endereço diferente do daquele esperado pelo *stream*, isto é, fora da seqüência anterior de endereçamento; ou ainda, expiração do limite de tempo decorrido desde o último acesso ao *stream*. No caso de leituras remotas, os *streams* são utilizados para a busca antecipada de dados (*prefetching*).

Há ainda a possibilidade de combinar dois, quatro ou oito *streams* — *stream combining* —, o que acarreta um aumento considerável na largura de banda. A técnica de *streaming* é detalhada por Ryan et al. [RYA 96]. Recentemente, a pesquisa de Gonzáles et al. [GON 99] mostrou os impactos da combinação de *streams* no desempenho de comunicação.

Retomando a questão dos inconvenientes de *clusters* SCI, além da falta de coerência global de *cache*, há que se considerar a não-desprezível taxa de erros de comunicação. Erros de comunicação ocasionais, atribuídos à fragilidade do cabeamento, são detectados e corrigidos transparentemente pelo hardware e pelos drivers SCI, através de CRC e mecanismos de *time-out*, sendo necessário o reenvio de pacotes nestes casos. Portanto, as conseqüências de tais erros refletem-se apenas no desempenho das aplicações.

Entretanto, Butenuth e Heiss [BUT 98] reportam situações que podem suscitar falhas inevitavelmente visíveis às aplicações, comprometendo o resultado final esperado. O problema originador destas situações errôneas não diz respeito ao hardware SCI em si, mas sim, a uma incompatibilidade com arquiteturas de *nodos* baseados no barramento de E/S PCI, como por exemplo, os PCs. Determinados cenários ocasionam o mau funcionamento do *bridge* de E/S — *bridge* PCI — de tais computadores. Considere-se um par de *nodos* do *cluster*. Quando ambos *nodos* escrevem em um endereço de memória do outro, o *bridge* de cada um precisa lidar com requisições em dois sentidos (ver figura 2.6): uma requisição da CPU local submetida à placa SCI local, para gerar uma transação SCI necessária para a escrita remota (CPU–*bridge*–placa SCI); e uma requisição da placa SCI local para escrever em um endereço de memória local (placa SCI–*bridge*–memória), requisição esta originada pela transação SCI submetida pelo outro *nodo*. Se a CPU tentar acessar a placa SCI enquanto esta estiver escrevendo na memória, a CPU deverá ser bloqueada e esperar para obter acesso ao barramento PCI. O *bridge* deveria dar prosseguimento ao acesso da placa SCI à memória, mas isto nem sempre ocorre. Em suma, observa-se um caso típico de *deadlock*.

[*] O modelo mais recente de placas adota 16 *buffers* de escrita e 16 de leitura, cada qual com 128 bytes.

A probabilidade de vir a acontecer um *deadlock* entre a CPU e a placa SCI, em situações como a acima descrita, depende do *chipset* da máquina. Alguns *chipsets* são mais suscetíveis a este inconveniente do que outros. Um estudo detalhado sobre o comportamento de *bridges* PCI de distintos *chipsets*, ante a requisições bidirecionais, é apresentado por Butenuth e Heiss [BUT 98].

Ademais, o desempenho da rede SCI depende fortemente dos *chipsets* dos *nodos*. No trabalho de Gonzáles et al. [GON 99], é avaliada a influência de diferentes *chipsets* na performance da rede SCI, em termos de largura de banda e latência.

2.6.3 Comentários finais

Os problemas, aqui descritos, atinentes a *clusters* baseados em SCI, não devem ser interpretados de forma a causar uma má impressão acerca desta tecnologia. Requisitos de total estabilidade são imprescindíveis apenas quando se deseja construir um *cluster* SCI com uma grande quantidade de *nodos*, tolerando vários usuários simultaneamente, todos submetendo aplicações baseadas em memória compartilhada. Deve ser lembrado que a programação por troca de mensagens é uma possibilidade em *clusters* SCI, e jamais devem ser esquecidos os benefícios de tal tecnologia de interconexão: ótima escalabilidade, alta largura de banda e baixa latência.

O restante do texto considera o emprego da tecnologia SCI como rede de interconexão para *clusters*, que é justamente a arquitetura em que se baseia o presente trabalho.

3 Programação por memória compartilhada em *clusters* SCI

A tecnologia SCI constitui um espaço de endereçamento físico compartilhado entre os *nodos* de um *cluster*, sendo a única restrição a impossibilidade de uso de *cache* para armazenar dados pertencentes a endereços remotos de memória, conforme pôde ser constatado na seção 2.6.2 deste texto.

Por permitir o compartilhamento real de memória, através da consecução de um mecanismo de DSM fisicamente implementado, SCI dá ensejo à concepção de uma série de abstrações e propostas para a programação por memória compartilhada de *clusters*, com a cristalina vantagem de se dispor do suporte direto do hardware. Contudo, o grande desafio para se levar a cabo tais propostas, ainda que o hardware SCI forneça bases sólidas, reside na ocultação do modelo inconsistente do mecanismo de DSM nativo e na implementação de um espaço de endereçamento virtual global, que possa ser acessado conveniente e eficientemente por processos ou *threads* distribuídos no *cluster*.

Não obstante os referidos obstáculos, evidenciam-se vários esforços na direção do suporte a modelos de programação por memória compartilhada em *clusters* SCI, muitos dos quais visando a facilitar a programação paralela em tal arquitetura. Este capítulo arrola, pois, diversos trabalhos neste contexto, descrevendo-os brevemente.

3.1 O driver SCI

Porquanto o paradigma de comunicação seguido por SCI é o compartilhamento de memória, já no nível do hardware, o driver das placas de rede é a primeira camada de software a fornecer uma API baseada em memória compartilhada.

A interface de programação exportada pelo driver [RYA 97] apresenta funções elementares e, como tal, de baixíssimo nível, baseadas no conceito de *chunk* — um bloco de memória que pode ser compartilhado entre diferentes *nodos* do *cluster* SCI. Um *chunk* representa uma região contígua de memória localizada em páginas do espaço físico de endereçamento de determinado *nodo*, sobre as quais não atua o mecanismo de *swapping* associado ao gerenciamento de memória virtual. Esta restrição é imposta devido ao fato de que estas páginas podem ser acessadas por outros *nodos* que não aquele que as abriga, dado o seu caráter de compartilhamento.

O driver oferece, além de primitivas básicas para a inicialização do hardware de comunicação, funções para o tratamento de *chunks*, cujas principais incumbências são:

- alocação e inicialização de um *chunk*;
- exportação de um *chunk* ao espaço de endereçamento global SCI, a fim de que outros *nodos* possam acessá-lo posteriormente;

- conexão a um *chunk* remoto, que tenha sido anteriormente exportado ao espaço de endereçamento global SCI;

- mapeamento de um *chunk* remoto — ao qual se tenha estabelecido previamente uma conexão —, ou de um *chunk* local, para o espaço de endereçamento lógico do processo requisitante, permitindo ao mesmo efetuar leituras e escritas nos endereços pertencentes ao *chunk* em questão.

Além das primitivas que tratam de *chunks*, o driver também fornece funções para a utilização do mecanismo de DMA das placas de rede SCI. A comunicação por DMA pode ser feita tanto para a transferência de dados da memória local para memória remota — que é o caso de escrita remota —, quanto pelo caminho inverso — leitura remota.

Por fim, o driver provê primitivas para o tratamento de interrupções remotas, que é outro recurso existente nas placas de rede SCI. Em um *cluster* SCI é possível a um *nodo* sinalizar uma interrupção em qualquer outro *nodo*. Uma útil aplicação desta facilidade poderia ser a implementação de mecanismos de sincronização, como por exemplo, semáforos.

Atualmente, existem dois drivers SCI: um desenvolvido na Universidade de Oslo [RYA 97] e, outro, pela empresa norueguesa *Dolphin Interconnect Solutions*, a fabricante das placas SCI. Ambos drivers foram portados para o sistema operacional Linux por um grupo de pesquisadores alemães [BUT 99], da Universidade de Paderborn, e, hoje, são utilizados por todos os *clusters* SCI baseados em Linux.

Embora seja possível desenvolver aplicações diretamente sobre o driver SCI, não se recomenda esta prática, pois a interface de programação é de extremo baixo nível. Assim, vários grupos têm investido esforços visando a disponibilizar abstrações de programação mais confortáveis para *clusters* SCI.

3.2 A API de baixo nível SISCI

3.2.1 Contexto

A interface de programação SISCI [GIA 98] é parte integrante do projeto SISCI (*Standard Software Infrastructure for SCI-based Parallel Systems*) [EBE 97], que tem como principal meta proporcionar uma abrangente infra-estrutura para fornecer suporte à programação de *clusters* SCI. Incluem-se no projeto camadas de baixo e alto nível, e a infra-estrutura planejada compromete-se a oferecer, aos programadores de aplicações paralelas, ambientes que permitam a exploração dos paradigmas duais de comunicação: troca de mensagens — MPI e PVM — e memória compartilhada — pacote para *multithreading* conforme o padrão POSIX Pthreads.

As camadas de software que compõem o arcabouço do projeto SISCI acham-se ilustradas na figura 3.1. Os módulos de níveis inferiores fornecem serviços e primitivas utilizados pelos módulos de mais alto nível. Alguns componentes da arquitetura

mostrada na figura estão concluídos; outros, porém, encontram-se em estágio de pesquisa.

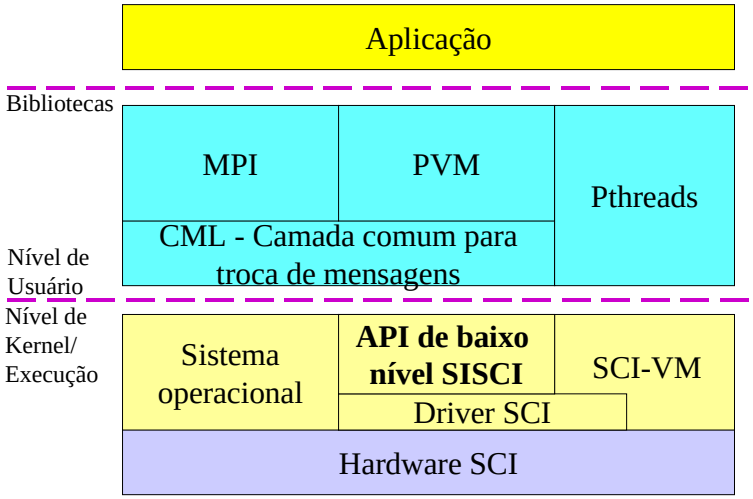

FIGURA 3.1 - Arquitetura do projeto SISCI.

Sobre o driver SCI foi implementada a API de baixo nível SISCI, que oculta dos programadores os detalhes do hardware SCI e do próprio driver, cujas principais funções foram brevemente descritas na seção 3.1. Em suma, a API de baixo nível SISCI pode ser entendida como a primeira camada de software a fornecer uma interface de programação mais confortável do que aquela apresentada pelo driver.

Ainda no âmbito do projeto SISCI (ver figura 3.1), propuseram-se implementações das bibliotecas MPI e PVM, ambas abalizadas em um mesmo núcleo de comunicação — o CML —, responsável pelos serviços elementares de comunicação ponto-a-ponto. No mesmo nível de MPI e PVM está uma proposta de implementação de uma biblioteca para *multithreading* em consonância com o padrão Pthreads, que será desenvolvida com base na abstração de memória virtual global proporcionada pela camada SCI-VM — SCI - Virtual Memory. Estes componentes do projeto SISCI e suas idéias subjacentes serão discutidos oportunamente ao longo do texto.

3.2.2 *SCI Physical Layer API*: a predecessora da API SISCI

Quando do início do projeto SISCI, o único esforço no sentido de especificar uma interface de acesso às funcionalidades do hardware SCI era a proposta do padrão IEEE P1596.9, denominado *SCI Physical Layer API* ou, simplesmente, *SCI PHY-API* [IEE 97]. A especificação *SCI PHY-API* é independente de hardware e sistema operacional, definindo uma camada de software DSM que generaliza a interface de acesso às funcionalidades de hardware, tais como inicialização de mapeamentos de endereço, movimentação de dados por DMA e manipulação de erros e exceções. Embora esteja inserida no contexto do padrão SCI, esta API mostra-se útil em qualquer cenário que envolva DSM. Ambientes com coerência de *cache*, bem como aqueles

desprovidos de tal propriedade — e.g. *clusters* SCI —, são contemplados pela especificação *SCI PHY-API*.

A *SCI Physical Layer API* não pressupõe a existência de drivers, tendo sido concebida como uma camada de abstração de hardware, de *overhead* nulo, com uma interface de programação de propósito geral para DSM projetada para o hardware SCI. Existe uma implementação desta API, a qual foi validada desenvolvendo-se algumas aplicações paralelas que demandam grande poder computacional.

Apesar de sua generalidade, a *SCI PHY-API* apresenta um nível de abstração muitíssimo baixo, encontrando sua maior utilidade no desenvolvimento de drivers, visto que disponibiliza acesso direto às peculiaridades do hardware SCI. Por esta razão, tal interface de programação não é adequada sequer ao desenvolvimento de bibliotecas de comunicação, a exemplo de MPI ou PVM, para cuja implementação requerem-se abstrações à semelhança de segmentos compartilhados, e não o acesso a recursos de hardware.

3.2.3 Características da API SISCI

Face à carência de uma interface de programação que verdadeiramente ocultasse as idiossincrasias do hardware SCI, um grupo multi-institucional de pesquisadores propôs a API SISCI [GIA 98] que, sem dúvida, trouxe aos programadores de *clusters* SCI um conjunto de primitivas que atende aos requisitos mínimos para o desenvolvimento de aplicações — ou software básico — sobre arquiteturas de hardware baseadas em DSM.

Implementada sobre o driver SCI (ver figura 3.1), a API aqui aludida encapsula as funções de baixo nível por ele providas e define estruturas de mais alto nível. Enquanto a interface de programação do driver lida com *chunks*, conforme comentou-se na seção 3.1, a noção de segmento compartilhado é o principal conceito introduzido pela API SISCI.

Existe uma espécie de protocolo a ser seguido pelos *nodos* do *cluster*, antes que possam efetivamente comunicar-se por meio de segmentos de memória compartilhados. O estabelecimento de segmentos compartilhados, em consonância com a semântica da API SISCI, exige a execução de uma seqüência de ações, em que se deve distinguir o *nodo* que cria determinado segmento daqueles que se conectam ao segmento remoto criado. Obviamente, cada *nodo* pode criar mais de um segmento e, também, conectar-se a outros tantos segmentos remotos.

Para a criação de um segmento, determinado *nodo* deve utilizar-se da primitiva `SCICreateSegment`, associando tal segmento a um identificador único, no contexto do *nodo* em questão. Após, o segmento deve ser preparado para tornar-se acessível (`SCIPrepareSegment`) e, por fim, efetivamente disponibilizado a todo o *cluster* (`SCISetSegmentAvailable`). Adicionalmente, o *nodo* criador do segmento pode mapeá-lo (`SCIMapLocalSegment`) para o espaço lógico de endereçamento do processo correlato, permitindo que venha a efetuar leituras e escritas nos endereços de memória compreendidos pelo segmento. As quatro primitivas aqui comentadas

manipulam objetos do tipo `sci_local_segment`. A figura 3.2 ilustra os possíveis estados de um segmento, durante as etapas de criação e disponibilização.

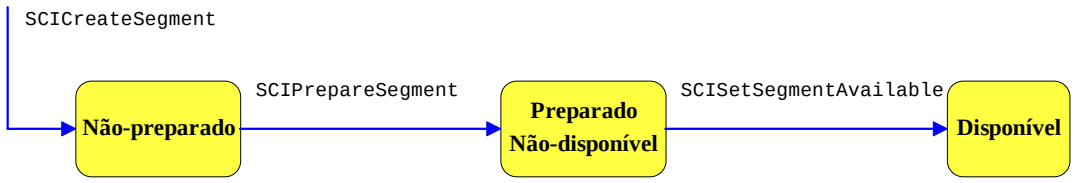

FIGURA 3.2 - Diagrama de estados de um segmento local.

Posto que determinado segmento tenha sido, pelo seu criador, devidamente disponibilizado a todo *cluster*, os demais *nodos* podem obter acesso a ele. Qualquer *nodo* que necessite de efetuar leituras e escritas em um segmento, criado por outro, deve, antes, executar duas ações fundamentais. Primeiramente, faz-se necessário conectar-se ao segmento remoto, por intermédio da primitiva `SCIConnectSegment`. Em seguida, o segmento remoto deve ser mapeado para o espaço de endereçamento lógico do processo invocador (`SCIMapRemoteSegment`).

Após este mapeamento, obtém-se, no espaço de endereçamento lógico, o endereço-base do segmento remoto, o que permite ler e escrever dados simplesmente derreferenciando ponteiros ou, em última análise, por *loads* e *stores* da CPU. Todas as vezes em que o processo, que mapeou o segmento remoto, acessar endereços por ele compreendidos, os mesmos serão, transparentemente, traduzidos em endereços físicos referentes à placa de rede SCI que, por sua vez, traduzi-los-á para endereços de 64 bits do espaço global SCI, efetivando a comunicação.

Note-se que as primitivas `SCIConnectSegment` e `SCIMapRemoteSegment` manipulam objetos do tipo `sci_remote_segment`, e não `sci_local_segment`, cujos objetos são de competência das funções incumbidas da criação e disponibilização de segmentos. Assim, a API SISCI diferencia segmentos locais e segmentos remotos.

Além das funções que dão conta do tratamento de segmentos — inicialização, exportação, conexão e mapeamento —, a API SISCI oferece primitivas para a utilização do mecanismo de DMA das placas de rede SCI e, também, funções que lidam com interrupções remotas.

Outrossim, a API em análise proporciona o mecanismo de “seqüência”, que permite monitorar operações de comunicação na rede SCI, de modo que se possa identificar eventuais falhas. Conceitualmente, uma “seqüência” é uma sucessão de acessos a segmentos remotos. Duas primitivas relacionadas à idéia de “seqüência” são `SCIStartSequence` e `SCICheckSequence`. A primeira realiza um teste preliminar, verificando a existência de erros pendentes, ao passo que a última responsabiliza-se por averiguar a ocorrência de falhas de comunicação desde a última vez em que foi invocada. Um objeto do tipo seqüência (`sci_sequence`) deve estar associado a um, e a somente um, segmento devidamente mapeado, embora um segmento possa estar relacionado a mais de uma “seqüência”. A primitiva `SCICreateMapSequence` produz

um objeto do tipo `sci_sequence`, que pode ser removido através da função `SCIRemoveSequence`.

Primitivas para a transferência síncrona e assíncrona de blocos de dados são também fornecidas pela API aqui discutida.

A especificação da API SISCI foi implementada e é distribuída oficialmente pela empresa fabricante do hardware SCI.

3.3 SMI: *Shared Memory Interface*

Não obstante o fato da API SISCI ser claramente mais confortável do que o driver SCI e, também, do que a interface de programação *SCI Physical Layer API*, pode-se perceber que suas primitivas não proporcionam todas as facilidades esperadas por programadores de aplicações. A API SISCI é apenas e tão-somente uma interface de programação que encapsula as funções elementares fornecidas pelo driver, para o acesso às funcionalidades básicas do hardware SCI: segmentos compartilhados, DMA, interrupções remotas e "seqüências". Conquanto a noção de segmento compartilhado represente a premissa básica para o desenvolvimento de aplicações baseadas em DSM, a API SISCI mostra-se mais útil à construção de software básico, como por exemplo, bibliotecas de comunicação por troca de mensagens.

Diferentemente da API SISCI, SMI [DOR 97] é mais do que uma interface de programação para *clusters* SCI; proporciona um ambiente de execução de aplicações paralelas, baseado no modelo SPMD (*Single Program Multiple Data*), garantindo um espaço de endereçamento uniforme em regiões de memória compartilhada.

Concebida por pesquisadores alemães do RWTH, em Aachen, a biblioteca SMI tem como principal objetivo facilitar o processo de paralelização de aplicações, seguindo o modelo de comunicação por memória compartilhada, em arquiteturas de hardware que o suportem, tais como SCI.

Segundo Butenuth e Heiss [BUT 98], a aprazível programação de *clusters* SCI por memória compartilhada depende fundamentalmente dos seguintes requisitos mínimos:

- mecanismos para inicialização e término do programa distribuído, em todos os *nodos*;
- funções, de fácil uso, para a criação, gerenciamento e destruição de segmentos de memória compartilhados, e;
- primitivas para a sincronização de processos através de barreiras e *mutexes*.

SMI atende a estes requisitos básicos e promove mais facilidades para o desenvolvimento de aplicações paralelas.

3.3.1 Modelo operacional e memória compartilhada

Uma aplicação SMI compõe-se de um conjunto de processos, cada qual, inicialmente, com seu próprio espaço lógico de endereçamento. Por meio de primitivas da biblioteca SMI, os processos podem criar regiões de memória compartilhadas entre eles, visando ao armazenamento de estruturas de dados. Segue-se o modelo SPMD, de modo que os processos executam exatamente o mesmo código, sobre porções distintas dos dados compartilhados.

A criação de uma **região de memória compartilhada —** conforme a nomenclatura sugerida pelos proponentes da biblioteca em estudo — representa um ponto global de sincronização entre os processos SMI; dito de outra forma, todos os processos da aplicação devem invocar a função responsável pelo estabelecimento de uma região compartilhada, que pode ser formada a partir de um ou mais segmentos de memória. Através de argumentos, pode-se especificar a política de distribuição física dos segmentos de cada região compartilhada estabelecida. Três políticas estão correntemente disponíveis aos programadores, a saber:

- `UNDIVIDED`: A região compartilhada compor-se-á de apenas um segmento, o qual estará fisicamente localizado em um único *nodo*, devidamente especificado.

- `BLOCKED`: Os segmentos que compõem a região compartilhada serão distribuídos uniformemente entre os *nodos* da aplicação. A região compartilhada possuirá tantos segmentos quantos forem os *nodos*, sendo cada segmento abrigado por um *nodo* distinto.

- `CUSTOMIZED`: O programador define, à sua própria maneira, a disposição dos segmentos componentes da região compartilhada entre os *nodos*, o número de segmentos que constituirão a região compartilhada, bem como o tamanho de cada segmento.

A característica mais interessante — e útil — acerca do estabelecimento de regiões de memória compartilhadas entre os processos SMI é o espaço lógico de endereçamento uniforme, assegurando que cada região compartilhada, independentemente de sua distribuição física, seja transparentemente mapeada para o mesmo endereço lógico em cada processo. Tal permite, por exemplo, que um ponteiro para um endereço pertencente a uma região compartilhada possa ser usado, indistintamente, por qualquer processo da aplicação. Perceba-se, também, que, embora uma região compartilhada possa estar fisicamente distribuída entre os diferentes *nodos* do *cluster*, cada processo a percebe como uma região de memória contígua no seu espaço lógico de endereçamento, sendo os segmentos que a constituem mapeados exatamente na mesma ordem em todos os processos.

A figura 3.3 [DOR 97] esboça o modelo operacional da biblioteca SMI. Na ilustração, mostram-se dois *nodos* dotados de duas CPUs, e supôs-se uma aplicação SMI composta de quatro processos, cada um dos quais atribuído a uma CPU. Uma região de memória compartilhada foi estabelecida e distribuída fisicamente pela política `BLOCKED`, estando parcial e uniformemente localizada no *nodo* “0” e no *nodo* “1”.

Observe-se que a região compartilhada possui dois segmentos, cada um fisicamente localizado em um *nodo*; além disso, a região compartilhada foi mapeada exatamente no mesmo endereço do espaço lógico de endereçamento de cada processo, e o mapeamento manteve a mesma ordem dos dois segmentos em cada processo.

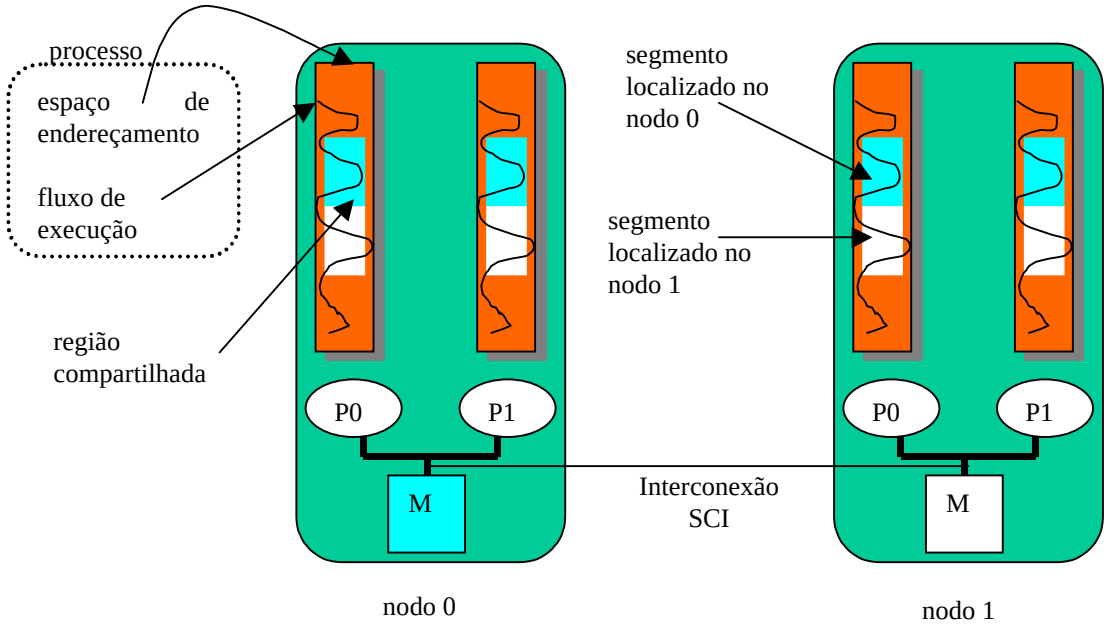

FIGURA 3.3 - Modelo operacional subjacente à biblioteca SMI.

3.3.2 Inicialização e ambiente de execução

Antes de qualquer chamada a funções da biblioteca SMI, é imprescindível que os processos inicializem o ambiente de execução. Tal é feito, à semelhança da biblioteca de comunicação MPI, por meio da função `SMI_Init`, que representa um ponto global de sincronização.

SMI disponibiliza funções, tipicamente encontradas em bibliotecas de comunicação, que permitem aos processos obter as seguintes informações:

- o número total de processos — *P*;
- o identificador único de um processo — entre *0* e *P – 1*;
- o número total de *nodos* — *M*;
- o identificador único de um *nodo* — entre *0* e *M – 1*;

Ao fim de uma aplicação, os processos devem invocar a função `SMI_Finalize` para a liberação dos recursos associados ao ambiente de execução.

3.3.3 Gerenciamento de memória

Deve estar evidente, neste ponto, que a construção de aplicações paralelas com a biblioteca SMI está centrada nas regiões compartilhadas, descritas na seção 3.3.1. Conforme mencionou-se, uma região de memória compartilhada é constituída de um ou mais segmentos de memória, contiguamente endereçáveis por cada um dos processos da aplicação, a partir do mesmo endereço lógico. A primitiva `SCI_Create_shreg` dá conta do estabelecimento de uma região compartilhada, recebendo como argumentos o tamanho da região e a política de distribuição física de seus segmentos componentes, retornando um identificador e o endereço-base, que é o mesmo para todos os processos.

Uma funcionalidade deveras útil, suportada por SMI, é a alocação dinâmica de memória em regiões compartilhadas, ensejando que as mesmas sejam adotadas tanto para o armazenamento de estruturas de dados estáticas, quanto dinâmicas. Para tal, deve-se associar a uma região compartilhada um gerenciador de memória SMI, por intermédio da primitiva `SMI_Init_shregMMU` que, a exemplo da função responsável pelo estabelecimento de regiões compartilhadas, exige sua invocação por todos os processos da aplicação.

Posto que se tenha vinculado um gerenciador de memória a uma determinada região compartilhada, pode-se proceder à alocação dinâmica no intervalo de endereços por ela compreendido, através das funções `SMI_Cmalloc` e `SMI_Imalloc`. Além do argumento requerido pela função `malloc` da biblioteca padrão da linguagem C — quantidade de memória, em bytes, a ser alocada —, estas funções recebem o identificador da região compartilhada sobre a qual devem atuar, retornando o endereço de início da área alocada na região compartilhada em questão.

Existe uma diferença semântica entre as funções `SMI_Cmalloc` e `SMI_Imalloc`. A primeira — *Collective Malloc* — requer sua chamada por todos os processos da aplicação, resultando em um ponto global de sincronização e, por conseguinte, retorna, a todos os processos, exatamente o mesmo endereço de início da área de memória alocada. Ao contrário, a segunda — *Individual Malloc* — não redunda em um ponto global de sincronização, sendo invocada individualmente por um processo; conseqüentemente, o endereço inicial da área de memória alocada somente é conhecido pelo processo invocador. Se, porventura, outros processos necessitarem deste endereço, deverão ser explicitamente informados do mesmo.

Como era de se esperar, fazem-se igualmente presentes as funções de liberação de áreas de memória previamente alocadas, coletiva ou individualmente: `SMI_Cfree` e `SMI_Ifree`.

3.3.4 Modos de consistência de memória

Conforme comentou-se no capítulo 2, não é possível fazer uso de *cache* para armazenar dados contidos em segmentos remotos de memória, no caso de *clusters* SCI; melhor dito, acessos a segmentos remotos devem, necessariamente, ser remotamente efetuados. É desnecessário asseverar os prejuízos ao desempenho decorrentes desta restrição arquitetural.

A biblioteca SMI, levando em consideração a implacável característica NUMA de *clusters* SCI, proporciona aos programadores um mecanismo para mitigar este entrave ao desempenho. Pela observância do fato de que uma aplicação paralela não necessita, efetivamente, de compartilhar uma estrutura de dados durante todo o tempo, SMI promove a capacidade de replicar, temporariamente, uma região compartilhada.

Este modo de replicação, instituído pela primitiva `SMI_Swtich_to_replication` à região compartilhada cujo identificador é passado como argumento, transforma a região, antes compartilhada, em regiões de memória privadas para cada processo da aplicação. As regiões replicadas ocupam, em cada processo, exatamente os mesmos endereços lógicos anteriormente referentes à região compartilhada, de modo que os ponteiros permanecem válidos.

Para restabelecer o modo de compartilhamento, basta uma chamada à primitiva `SMI_Switch_to_sharing`, podendo-se utilizar funções específicas para combinar os dados das regiões privadas.

Além do modo de replicação, o fato de se poder determinar, explicitamente, a política de distribuição física dos segmentos que compõem uma região compartilhada, conforme descrito na seção 3.3.1, mostra a preocupação dos projetistas da biblioteca SMI em facilitar a determinação da localidade dos dados, o que é de vital importância em uma arquitetura DSM desprovida de coerência global de *cache*.

3.3.5 Sincronização

Primitivas de sincronização são essenciais em bibliotecas para programação paralela, especialmente em se tratando do paradigma de comunicação por memória compartilhada. SMI oferece, pois, barreiras, *mutexes* e *progress counters*.

Para a sincronização de todos os processos de uma aplicação SMI, pode-se utilizar a abstração de barreira, representada pela primitiva `SMI_Barrier`, que apresenta a semântica habitual, qual seja, a primitiva somente encerrará sua execução após ter sido invocada por todos os processos.

A fim de sincronizar o acesso dos processos a regiões compartilhadas, os programadores têm ao seu dispor *mutexes*, associados às primitivas `SMI_Mutex_lock`, `SMI_Mutex_unlock` e `SMI_Mutex_trylock`, seguindo a semântica usual.

Adicionalmente, SMI traz a idéia de *progress counters* como uma abordagem alternativa à sincronização de processos. Um *progress counter* consiste em um conjunto de variáveis contadoras, cada uma relativa a um processo. Os processos podem incrementar seus próprios contadores (`SMI_Increment_PC`) e esperar até que o contador de um processo específico (`SMI_Wait_individual_PC`), ou mesmo os contadores de todos os outros processos (`SMI_Wait_collective_PC`), atinjam um determinado valor. Desta sorte, um *progress counter* pode ser interpretado como um mecanismo que possibilita a cada processo informar seu progresso computacional aos demais.

3.3.6 Paralelização e escalonamento de laços

Uma característica que distingue a biblioteca SMI, dentre as propostas sugeridas para programação de *clusters* SCI por memória compartilhada, é o serviço de paralelização e escalonamento de laços. Existem primitivas a serem usadas em conjunto com laços, expressados na linguagem de programação para a qual a biblioteca foi desenvolvida — e.g., a linguagem C —, que permitem a distribuição das iterações entre os processos da aplicação. Ademais, durante o processamento do laço, é feito um balanceamento dinâmico de carga. A estratégia de balanceamento de carga pode ser especificada pelo programador, levando em consideração questões de localidade dos dados das regiões compartilhadas que experimentarão a ação do laço.

3.3.7 Comentários finais acerca da biblioteca SMI

A biblioteca SMI foi implementada diretamente sobre o driver SCI, encontrando-se disponível para *clusters* baseados em Linux, Solaris e Windows-NT. São suportadas as linguagens de programação C, C++ e Fortran. Uma descrição exaustiva dos serviços e primitivas fornecidos acha-se no manual de programação da biblioteca [DOR 98].

3.4 YASMIN

YASMIN (*Yet Another Shared Memory INterface)* [TAS 98] é uma biblioteca de programação paralela para *clusters* SCI, baseada no paradigma de comunicação por memória compartilhada, inspirada na proposta introduzida por SMI — descrita na seção 3.3 —, o que explica a sigla que se lhe atribuiu. Foi projetada por pesquisadores alemães, da Universidade de Paderborn, estando disponível exclusivamente para *clusters* SCI compostos de PCs rodando o sistema operacional Linux.

Tal como a biblioteca SMI, o principal objetivo de YASMIN é simplificar o desenvolvimento e a paralelização de aplicações em *clusters* SCI, fornecendo funcionalidades básicas demandadas por tais tarefas. Pode-se dizer que SMI e YASMIN estão, portanto, no mesmo nível de abstração.

3.4.1 Modelo de programação e memória compartilhada

À semelhança de SMI, YASMIN adota o modelo de programação SPMD. Durante a inicialização, é disparado um número fixo de processos em cada *nodo* — um processo por CPU. Seguindo o mesmo padrão de interfaces de programação congêneres, as primitivas `yasmin_init` e `yasmin_finalize` dão conta, respectivamente, da inicialização do ambiente de execução e do término da aplicação.

YASMIN implementa o conceito de grupos de processos. Quando do início de uma aplicação, todos os processos pertencem ao grupo `sci_all_group`; no decurso da mesma, entretanto, é possível a constituição de subgrupos, mediante a primitiva `sci_create_group`. Adicionalmente, dispõe-se de primitivas responsáveis pelo

fornecimento de informações estruturais — o tamanho de um dado grupo (`sci_get_grpsize`) e a identificação única de um processo em um grupo ao qual pertence (`sci_get_rank`) —, bem como da primitiva cuja função é dissolver um grupo anteriormente constituído (`sci_destroy_group`).

Como era de se esperar, a principal funcionalidade da biblioteca YASMIN reside na capacidade de criação e destruição dinâmicas de segmentos compartilhados. Diferentemente de SMI, no entanto, o compartilhamento de segmentos de memória manifesta-se no âmbito de um grupo de processos, não necessariamente composto de todos os processos da aplicação. As primitivas associadas a esta funcionalidade são `sci_create_distr_seg` e `sci_destroy_distr_seg`.

Cada segmento, compartilhado por um determinado grupo, é mapeado exatamente no mesmo endereço lógico dos processos que o congregam, assegurando-se um espaço de endereçamento uniforme, tal como o faz a biblioteca SMI. Destarte, torna-se possível a troca de ponteiros entre processos de distintos *nodos*; diga-se de passagem, esta propriedade é o grande diferencial dos sistemas de memória compartilhada, em relação àqueles centrados no paradigma de troca de mensagens.

O modelo de processos da biblioteca YASMIN está ilustrado na figura 3.4, conforme mostra Rehling [REH 99].

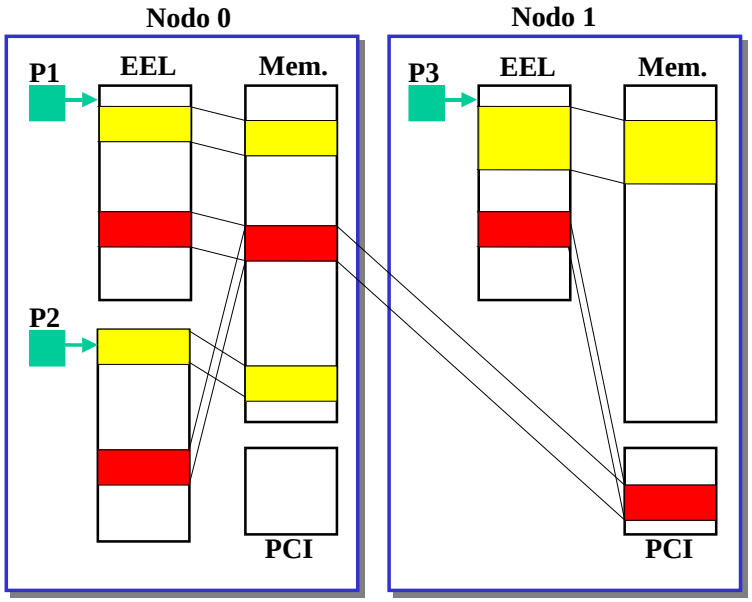

FIGURA 3.4 - O modelo de processos da biblioteca YASMIN.

Na figura 3.4, consideraram-se dois *nodos*, sendo o "*nodo* 0" dotado de duas CPUs, havendo três processos, cada qual atribuído a uma CPU, conforme o modelo SPMD. Supôs-se um grupo composto pelos três processos, que estabeleceram um segmento compartilhado. Observe-se que a área de memória compartilhada, fisicamente localizada no "*nodo* 0", é mapeada diretamente nos espaços lógicos de endereçamento dos processos P1 e P2. A mesma área é mapeada, através da placa de rede e do driver

SCI, no espaço lógico de endereçamento do processo P3, no "*nodo* 1". Convém salientar que os mapeamentos da área de memória compartilhada foram efetuados exatamente no mesmo endereço lógico dos três processos.

YASMIN, a exemplo de SMI, também fornece primitivas próprias para a alocação dinâmica de áreas de memória em segmentos compartilhados.

3.4.2 Sincronização

Todos os objetos de sincronização da biblioteca em discussão estão associados a grupos de processos. Para a sincronização de todos os processos de determinado grupo, YASMIN disponibiliza a abstração de barreira. As primitivas correlatas são `sci_create_barrier`, `sci_barrier` e `sci_destroy_barrier`. Diferentemente de SMI, uma barreira age no contexto de um grupo e não, necessariamente, sobre todos os processos da aplicação.

A sincronização de um grupo de processos, no acesso a um segmento compartilhado, pode ser feita através de *mutexes*. Particularmente, a interface de programação providencia dois tipos de *mutexes*: *mutex* tradicional, representado pelo objeto `sci_mutex`, e *mutex* de leitura e escrita, representado pelo objeto `sci_rw_mutex`. Em ambos os casos, as primitivas habituais estão disponíveis.

Outrossim, YASMIN proporciona mais uma facilidade de sincronização entre processos. O objeto `sci_sigobj`, associado a primitivas do tipo *wait* e *signal*, confere mais flexibilidade às aplicações.

A associação de objetos de sincronização a grupos pode trazer melhorias de desempenho aos programas paralelos. Além da flexibilidade decorrente da possibilidade de criação de grupos menores do que a totalidade de processos, os algoritmos de sincronização apresentam uma complexidade de tempo fortemente dependente do número de processos participantes.

3.4.3 Comunicação de grupo

Conquanto a biblioteca ora analisada prime pelos mecanismos de memória compartilhada, foram incorporadas à interface de programação primitivas de comunicação coletiva entre processos. Estão disponíveis funções para difusão (`sci_bcast`), espalhamento (`sci_scatter`), coleta (`sci_gather`), redução (`sci_reduce`) e generalizações das duas últimas — `sci_allgather` e `sci_allreduce`. A semântica das referidas funções é exatamente a mesma estabelecida pelo padrão MPI [MPI 94], que define um lauto conjunto de serviços atinentes à comunicação coletiva.

3.5 Sthreads

Uma proposta diferente das apresentadas até este ponto foi encetada pelo trabalho que redundou na biblioteca Sthreads [REH 99]. Desenvolvida pelo mesmo grupo que idealizou a biblioteca YASMIN, a qual se descreveu brevemente na seção anterior, Sthreads representa uma tentativa de facilitar a adaptação de programas originalmente baseados no padrão Pthreads [IEE 95] — que tem nas arquiteturas do tipo SMP seu principal alvo — para *clusters* SCI.

Esta proposta foi mormente motivada pela capacidade de compartilhamento de memória entre processadores, inerente à tecnologia SCI, o que permite que se pense na exploração de *multithreading* no contexto de um *cluster*, tal como em máquinas SMPs isoladas.

3.5.1 Arquitetura da biblioteca Sthreads

Sthreads segue uma abordagem bastante simples. Com base nos serviços oferecidos por YASMIN e, assumindo a disponibilidade de uma biblioteca para *multithreading* — conforme o padrão Pthreads — nativa no sistema operacional, Sthreads adapta as noções de *threads* e *mutexes* ao contexto de um *cluster* SCI, que passa a ser considerado como uma máquina SMP.

Durante a inicialização de uma aplicação Sthreads, o ambiente de execução YASMIN é disparado, sendo criado um processo YASMIN em cada *nodo* do *cluster*. Todas as chamadas a rotinas YASMIN efetuadas internamente por Sthreads foram protegidas por *mutexes*, evitando a ocorrência de *race conditions* possivelmente suscitadas pelo acesso concorrente de *threads* de um mesmo *nodo*.

3.5.2 Criação de *threads*

A biblioteca Sthreads permite a criação de *threads* locais ou remotas, cabendo ao programador a escolha do *nodo* em que determinada *thread* deve ser executada. A criação de *threads* locais exige o disparo de uma *thread* através da biblioteca nativa; a inicialização da estrutura de dados usada internamente por Sthreads, para fins de gerenciamento (`sthread_t`); e, finalmente, a disposição da referida estrutura em um segmento compartilhado, uma vez que todos os recursos associados à nova *thread* precisam estar disponíveis a todo *cluster*, de modo a permitir, por exemplo, que todas as *threads*, a despeito do *nodo* em que estejam, possam realizar uma operação de *join* futuramente.

Por outro lado, a criação de *threads* remotas é um pouco mais complexa. Neste caso, cada processo YASMIN atua como um *daemon* que atende a requisições, advindas de outros *nodos*, de criação de *threads*. Desta forma, para levar a efeito o disparo remoto de uma *thread*, as seguintes ações são executadas por Sthreads:

- a estrutura de dados `sthread_t` é devidamente inicializada e armazenada em um segmento compartilhado exportado pelo *nodo* em que a *thread* deve ser disparada;

- os argumentos da função a ser executada pela *thread* são também armazenados no segmento compartilhado exportado pelo *nodo-destino*;

- o *daemon* — processo YASMIN — do *nodo* em que a *thread* deve ser criada recebe uma mensagem de requisição, a qual lhe informa o local em que os recursos associados à *thread* a ser iniciada estão armazenados;

- o *daemon*, ante a solicitação, efetivamente dispara a nova *thread*, através da biblioteca nativa.

3.5.3 Sincronização

A biblioteca Sthreads estendeu a utilização de *mutexes* a fim de possibilitar a sincronização de todas as *threads* do *cluster*. *Threads* de um mesmo *nodo* compartilham o mesmo espaço de endereçamento e podem efetuar acesso concorrente à memória local ao *nodo* em que se encontram; de modo semelhante, *threads* de distintos *nodos* podem realizar acesso concorrente à memória compartilhada através da rede SCI, havendo a necessidade de sincronizá-las também.

Um *mutex* Sthreads possui dois componentes, a saber:

- um *mutex* da biblioteca Pthreads nativa, utilizado para sincronizar *threads* de um mesmo *nodo*;

- um *mutex* YASMIN, usado para garantir a sincronização de *threads* pertencentes a *nodos* diferentes.

A implementação de *mutexes* pela biblioteca Sthreads adota esta abordagem híbrida porque *mutexes* YASMIN não são *thread-safe*, impossibilitando seu emprego para a sincronização de *threads* locais.

Em sendo assim, uma operação de *lock* aplicada em um *mutex* Sthreads exige, primeiramente, um *lock* sobre o *mutex* Pthreads nativo, seguido de um *lock* sobre o *mutex* YASMIN. Este esquema assegura que somente uma *thread* local terá acesso ao *mutex* YASMIN, contido em um segmento compartilhado entre os *nodos* do *cluster* por meio da rede SCI.

Devido à necessidade de garantir que cada *mutex* Sthreads, criado por uma aplicação, seja globalmente identificado em de todo *cluster*, foi tomado um cuidado especial tocante a esta operação. No início da aplicação, aloca-se em cada *nodo* um vetor de *mutexes* Sthreads. A identificação global de cada *mutex* é feita por um valor inteiro que indexa o vetor. Dada a composição de um *mutex* Sthreads, cada posição do vetor refere-se a um *mutex* Pthreads local e a um *mutex* YASMIN global; obviamente,

posições correspondentes do vetor em *nodos* distintos devem conter referência ao mesmo *mutex* YASMIN, tornando possível a sincronização entre *threads* de todo *cluster*. A figura 3.5 ilustra um *mutex* Sthreads.

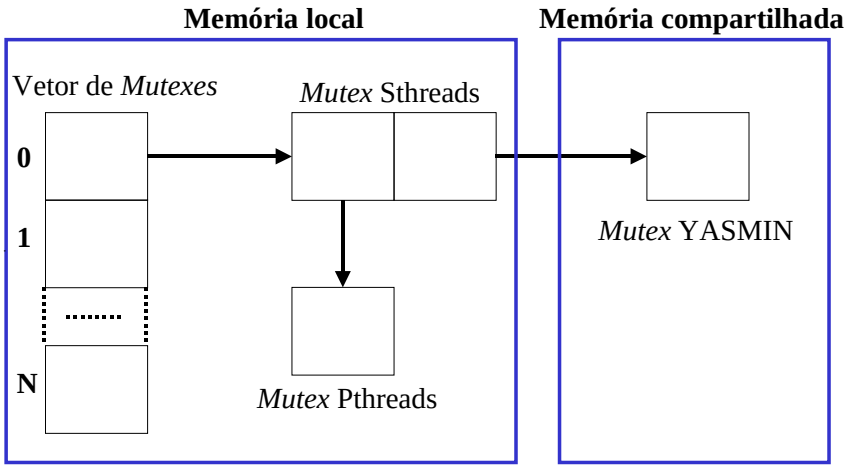

FIGURA 3.5 - A composição de um *mutex* Sthreads.

3.6 SCI-VM

A abstração de segmento compartilhado é a tônica de todos os trabalhos até aqui discriminados, embora esteja muito distante do modelo de compartilhamento real de memória suportado por hardware especializado, tal como arquiteturas da modalidade SMP. Isto se deve, sobretudo, à incapacidade do hardware SCI proporcionar, por si só, um espaço de endereçamento lógico global, que abranja todos os *nodos* do *cluster*.

Visando a estender a aplicabilidade de *clusters* SCI no contexto do paradigma de memória compartilhada, foi proposto e parcialmente implementado o ambicioso modelo SCI-VM (*SCI - Virtual Memory*) [SCH 99]. Concebido por pesquisadores da Universidade de Munique (Alemanha), SCI-VM tem como objetivo aproximar *clusters* SCI de arquiteturas multiprocessadas fortemente acopladas, ampliando sobremaneira o leque de modelos de programação passíveis de implementação.

3.6.1 Memória virtual global

A proposta de SCI-VM é instituir a abstração de memória virtual global transparente [SCH 98a], no âmbito de todo o *cluster* SCI, promovendo um único espaço lógico de endereçamento, independente do *nodo* a efetuar o acesso. Este é exatamente o cenário que se evidencia em arquiteturas SMPs.

Em associação à idéia de memória virtual global, SCI-VM constitui o que se chama de processo global distribuído, i.e., um processo único — e seu respectivo espaço lógico de endereçamento — que permeia todos os *nodos* do *cluster*. Estas

abstrações fornecem bases para a implementação irrestrita de modelos de programação por memória compartilhada em *clusters* SCI.

Para levar a termo estas idéias, às capacidades de acesso remoto à memória do hardware SCI uniram-se técnicas usualmente adotadas por ambientes de DSM puramente implementados em software, como por exemplo, distribuição de dados à granularidade de páginas, acesso sob demanda a páginas remotas e modelos de consistência relaxada.

A grande vantagem de SCI-VM, quando cotejado com mecanismos de DSM tradicionais — e.g. TreadMarks [AMZ 96] —, reside exatamente na exploração da capacidade de acesso remoto à memória do hardware SCI, evitando a necessidade de replicação de páginas e, por conseguinte, impedindo a manifestação de fenômenos como o *falso compartilhamento*. Ademais, todos os mecanismos de sincronização podem fazer uso das transações atômicas — *lock* — que o hardware SCI disponibiliza, conforme descrito na seção 2.4.2.

Um obstáculo ao desempenho de SCI-VM seria, devido às já mencionadas limitações de *clusters* SCI, a impossibilidade de utilização coerente de *cache* para dados armazenados em regiões remotas de memória, o que redundaria em desnecessários e custosos acessos através da rede. A fim de contornar este problema, SCI-VM habilita-se a utilizar as *caches* locais dos processadores dos *nodos*, implementando um modelo de consistência relaxada.

A implementação efetiva de SCI-VM exige estender o gerenciamento de memória virtual do sistema operacional e o próprio driver SCI, no intuito de interligar logicamente as diferentes cópias destas camadas de software que executam isoladamente em cada *nodo* do *cluster*. É desnecessário afirmar que este gerenciamento integrado de memória entre o driver SCI e o sistema operacional, bem como a união de cópias isoladas de um sistema operacional, envolvem tarefas que não são triviais. Mesmo assim, existe uma versão preliminar do SCI-VM, baseada em Windows-NT.

SCI-VM oferece uma API de baixo nível estritamente voltada para o desenvolvimento de modelos de programação por memória compartilhada. Atualmente, dois modelos de programação foram projetados, com o fito de validar as funcionalidades de SCI-VM: um ambiente de programação semelhante a SMI, e uma biblioteca para *multithreading* em *clusters* SCI, sendo esta última brevemente comentada a seguir.

3.6.2 SISCI-Pthreads

A noção de processo global único remete, prontamente, a um modelo de programação baseado em *multithreading*. Em havendo um único processo global distribuído, logicamente permeando todos os *nodos* de um *cluster* SCI, este processo pode manter vários fluxos de execução, exatamente como ocorre em uma máquina SMP, de forma totalmente transparente.

O primeiro uso de SCI-VM foi a implementação de uma biblioteca de *multithreading* — SISCI-Pthreads [SCH 98] — que possibilita executar, em *clusters*

SCI, programas escritos conforme o padrão Pthreads, sem qualquer alteração. Diferentemente da abordagem seguida pela biblioteca Sthreads, descrita na seção 3.5, todas as *threads* são mantidas logicamente por um único processo, compartilhando o mesmo espaço lógico de endereçamento. O programador simplesmente cria *threads* e utiliza, de forma irrestrita, todos os mecanismos de sincronização associados, tal qual o faria em um computador SMP.

Internamente, como mostra a figura 3.6, o processo global único instaurado por SCI-VM, em verdade, compõe-se de vários processos hospedeiros, cada qual localizado em um *nodo* do *cluster*, abrigando um conjunto de *threads*. Como SCI-VM provê um espaço lógico único, os contextos dos processos hospedeiros são consistentes e sincronizados, de modo que as *threads* não "percebem" que se encontram em processos distintos. Questões de implementação deste modelo envolvem o mapeamento de segmentos SCI sob demanda, em conjunto com mecanismos de gerenciamento de memória virtual — *swapping* —, e a adequação do grão de mapeamento proporcionado pelo hardware SCI àquele da plataforma subjacente — no caso, PCs rodando Windows-NT.

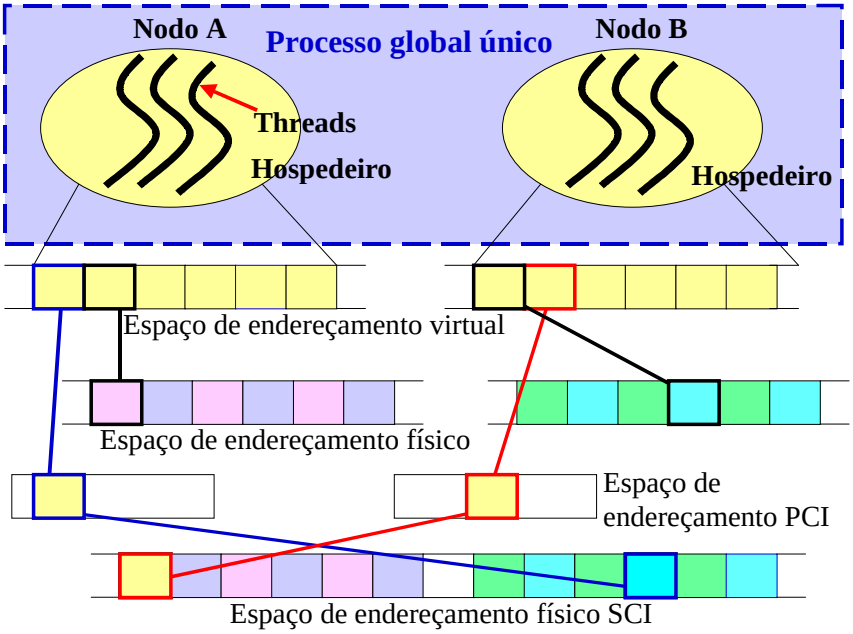

FIGURA 3.6 - Processo global distribuído e memória virtual global.

SCI-VM e SISCI-Pthreads fazem parte do projeto SISCI, ao qual se fez alusão na seção 3.2.1 e na figura 3.1 deste texto.

3.7 Considerações finais

Este capítulo delineou os principais trabalhos concernentes a modelos de programação por memória compartilhada em *clusters* SCI sem, no entanto, exaurir tudo o que tem sido feito neste campo de pesquisa.

Não obstante o tema desta Dissertação estar voltado a protocolos de comunicação por troca de mensagens em *clusters* SCI, dada a natureza desta tecnologia, não se pode simplesmente desprezar as pesquisas acerca de modelos de programação por memória compartilhada. Os trabalhos aqui descritos, assim o foram, não por uma mera apresentação do estado-da-arte, mas por facilitar o pleno entendimento da tecnologia SCI, suas funcionalidades e limitações. Ademais, a concepção de qualquer protocolo de troca de mensagens em *clusters* SCI exige o conhecimento de interfaces de programação de baixo nível, a exemplo do driver, da *SCI Physical Layer API* e da API SISCI, conhecimento este que permite nortear de forma profícua decisões de projeto, levando-se em consideração os objetivos traçados. Bibliotecas como SMI e YASMIN podem, outrossim, ser aproveitadas para a implementação de protocolos de troca de mensagens.

A leitura detida deste capítulo deve compelir a uma conclusão: a tecnologia SCI, por si só, não resolve os problemas inerentes à consecução de modelos de programação por memória compartilhada. Seu ponto favorável é possibilitar o acesso direto à memória remota, de forma semelhante a tecnologias como *Memory Channel* [GIL 96], e esta característica pode ser eficientemente explorada por protocolos de troca de mensagens.

A tabela 3.1 resume as principais propriedades das APIs que proporcionam a noção de porções compartilhadas de memória.

TABELA 3.1 - Resenha das principais propriedades das APIs baseadas em segmentos compartilhados.

APIs	Inicialização, exportação e mapeamento de segmentos DSM	Sincronização	Ambiente de execução	Serviços adicionais	Nível de abstração
Driver	programador: de forma explícita	inexistente	inexistente	DMA, interrupções remotas	muito baixo
SCI Physical Layer API	programador: de forma explícita	inexistente	inexistente	DMA, interrupções remotas	muito baixo
SISCI	programador: de forma explícita	inexistente	inexistente	DMA, interrupções remotas	baixo

APIs	Inicialização, exportação e mapeamento de segmentos DSM	Sincronização	Ambiente de execução	Serviços adicionais	Nível de abstração
SMI	ambiente: de forma transparente	barreira, *mutex* e *progress counter*	modelo SPMD	escalonamento e paralelização de laços	alto
YASMIN	ambiente: de forma transparente	barreira, *mutex* e *wait*/*signal*	modelo SPMD	comunicação de grupo	alto

4 Troca de mensagens em *clusters* SCI

Embora a aplicação originalmente idealizada para a tecnologia SCI tenha sido a interconexão de multiprocessadores em arquiteturas CC-NUMA, consoante mencionou-se no capítulo 2, seu emprego como rede de alto desempenho de *clusters* tem despertado o interesse de várias instituições e centros de pesquisa. No momento em que assume o papel de rede de alto desempenho de *clusters*, fazem-se prementes o desenvolvimento e adaptação de ambientes e bibliotecas de programação baseados no paradigma de comunicação de uso mais profuso em tais arquiteturas: troca de mensagens.

A tecnologia SCI, a um só tempo, fornece suporte em hardware para modelos de programação por memória compartilhada e à implementação de eficientes protocolos de comunicação por troca de mensagens. Especificamente, a capacidade de acesso direto a endereços remotos de memória, em nível de usuário, sem a intervenção do sistema operacional, garante uma baixíssima latência de comunicação — 2,3 a 2,5 μs —, permitindo que se vislumbrem protocolos *ad hoc* de troca de mensagens não apenas comparáveis, mas mais enxutos do que aqueles atualmente em voga em *clusters* baseados na rede Myrinet [BOD 95], a qual, indubitavelmente, logrou a maior aceitação na comunidade de *cluster computing*.

Muita pesquisa se tem feito acerca de protocolos de comunicação de alto desempenho, visando, sobretudo, a reduzir o caminho crítico entre processos executados por diferentes *nodos* de um *cluster*. A tendência atual é a concepção de mecanismos que tornem possível a comunicação em nível de usuário, sem a ingerência de protocolos adicionais ou do sistema operacional, a fim de reduzir a latência resultante. Citam-se, como exemplos de intensos esforços de pesquisa nesta área, os protocolos implementados com base no hardware Myrinet, tais como BIP [PRY 98], Fast Messages [PAK 96], GM [MYR 99] e Trapeze [YOC 97], além do padrão VIA [BUO 98].

A rede SCI insere-se imponente neste contexto, pelas razões acima expostas, dando ensejo ao projeto de protocolos de comunicação de baixa latência.

O presente capítulo trata de questões pertinentes à comunicação por troca de mensagens em *clusters* SCI, descrevendo as técnicas que têm sido adotadas para tanto.

4.1 Por que troca de mensagens em *clusters* SCI?

O paradigma de troca de mensagens exige do programador a distribuição explícita dos dados entre os *nodos* do *cluster*. Esta tarefa, ainda que seja extra, por vezes permite a consideração de questões de desempenho de forma mais simples ou natural, particularmente no que concerne à localidade dos dados.

Algumas peculiaridades do hardware SCI atualmente disponível para *clusters* impõem restrições ao compartilhamento de memória. Relembrando o que foi comentado no capítulo 2, as placas de rede SCI usadas em *clusters*, assim como qualquer outra, devem ser conectadas ao barramento de E/S de cada *nodo*. A decorrência imediata é a

impossibilidade de implementação do protocolo de coerência global de *cache* definido pelo padrão SCI, visto que a localização das placas as impede de monitorar as transações que se manifestam entre CPU e memória. Sem coerência global de *cache* em um *cluster*, acessos remotos devem obrigatoriamente redundar em atrasos através da rede de comunicação, o que restringe sobremaneira os modelos de memória compartilhada toleráveis eficientemente. Embora haja uma proposta, apresentada no capitulo 3, para contornar este entrave — SCI-VM/SISCI-Pthreads —, há questões que permanecem em aberto. Os mecanismos de coerência propostos são suficientemente eficientes? Aplicações baseadas em Pthreads, que assumem um ambiente no qual não ocorrem falhas no acesso à memória, podem lidar com possíveis erros de comunicação que se fazem presentes em certas situações?

Mesmo que as questões acima sejam respondidas em favor do compartilhamento de memória, remanesce uma mácula: um ambiente que propõe o uso de *clusters* SCI à guisa de máquinas SMPs, tal como SCI-VM/SISCI-Pthreads, inevitavelmente renuncia à portabilidade, tamanhas são as alterações necessárias no sistema operacional — gerenciamento de memória virtual e escalonamento de *threads* — e no próprio driver SCI (ver seção 3.6).

Vários são os projetos comprometidos com protocolos de comunicação por troca de mensagens especificamente para *clusters* SCI, incluindo adaptações de bibliotecas baseadas nos padrões MPI [MPI 94] e PVM [GEI 94]. Segue-se uma breve apresentação das principais propostas.

4.2 PVM-SCI

PVM-SCI [FIS 97][FIS 99] é uma adaptação do ambiente PVM, projetada na Universidade de Paderborn (Alemanha), no intuito de permitir a comunicação entre processos através da rede SCI.

Neste ponto, cabe uma rápida descrição da arquitetura subentendida pelo padrão PVM, para facilitar o entendimento da abordagem sugerida por PVM-SCI. O ambiente PVM baseia-se na construção de uma máquina virtual paralela, formada a partir de *nodos* de uma rede. Em cada um dos *nodos* da máquina virtual PVM, encontra-se um *daemon* — `pvmd` — em execução, o qual se responsabiliza pelo roteamento das mensagens. Por *default*, a comunicação entre um par de processos, pertencentes a *nodos* distintos, exige a intervenção de dois *daemons*: o processo emissor envia a mensagem ao *daemon* local, que então a encaminha ao *daemon* que se encontra no *nodo* hospedeiro do processo ao qual se destina a mensagem; por sua vez, este último *daemon* finalmente remete a mensagem ao processo que espera recebê-la.

Considerando-se um sistema PVM totalmente baseado na pilha de protocolos TCP/IP, a comunicação entre um processo e o *daemon* local dá-se através de uma conexão TCP, ao passo que a comunicação entre *daemons* é efetuada por intermédio do protocolo UDP. Adicionalmente, o programador pode determinar o estabelecimento de uma conexão direta entre um dado par de processos pertencentes a *nodos* distintos, a fim de evitar a intervenção dos *daemons*. Neste caso, uma conexão TCP interliga o par de processos em questão. A arquitetura típica de um ambiente PVM está ilustrada na figura 4.1.

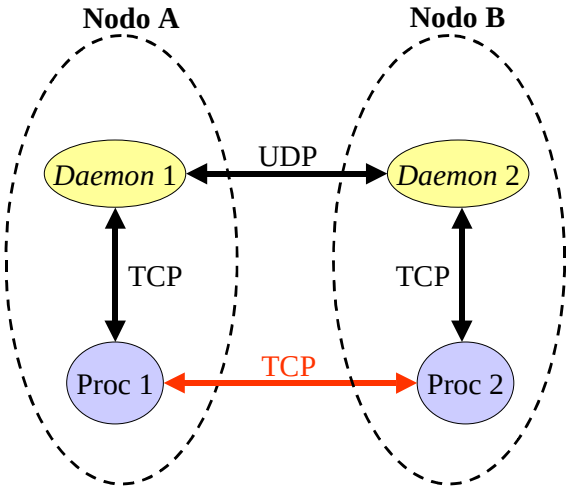

FIGURA 4.1 - Arquitetura típica de um ambiente PVM.

O ambiente PVM-SCI modifica apenas o esquema de comunicação direta entre processos, ou seja, permite que a rede SCI seja utilizada na comunicação entre qualquer par de processos, sem a intervenção dos *daemons*. Nos casos em que há o envolvimento dos *daemons*, a rede SCI não é usada.

No momento em que um processo, pela primeira vez, tenta estabelecer uma comunicação direta com outro, uma seqüência de ações são executadas pelo ambiente PVM-SCI, as quais substituem a instauração de uma conexão TCP. O processo emissor aloca, mapea e exporta um segmento de memória e, logo após, o *daemon* local requisita, ao *daemon* do *nodo* que hospeda o processo receptor, o estabelecimento de comunicação através da rede SCI, informando-lhe os dados referentes ao segmento compartilhado recém criado; em caso de aceitação, um segmento de memória também será, pelo processo receptor, criado, mapeado e exportado, sendo os dados a ele relativos submetidos pelo *daemon* local ao *daemon* do *nodo* do processo emissor.

Após esta seqüência de mapeamentos, o par de processos em questão pode começar a comunicar-se através da rede SCI, pela simples invocação das funções de troca de mensagens da biblioteca PVM.

Os segmentos compartilhados criados pelos processos comunicantes abrigam uma estrutura de dados comumente chamada de *Receive Ring Buffer* — RRB. Trata-se de uma área para o armazenamento de mensagens que está associada a dois ponteiros: um ponteiro indicando a próxima posição de escrita no buffer e, outro, a próxima posição de leitura. O RRB é gerenciado tal como uma fila circular, estando representado na figura 4.2.

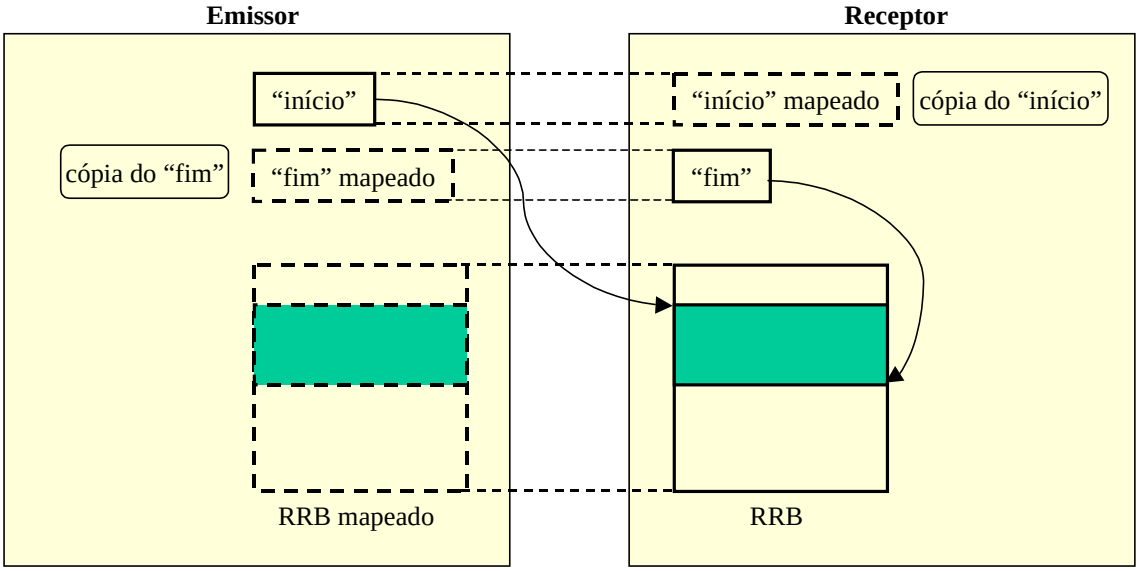

FIGURA 4.2 - Utilização do conceito de RRB para troca de mensagens.

A figura 4.2 mostra o RRB do processo referido como receptor. Este RRB, obviamente, precisa ser mapeado no espaço lógico de endereçamento do processo que deseja enviar uma mensagem, haja visto o funcionamento da rede SCI. Observe-se, também, que o ponteiro “fim” associado ao RRB está fisicamente localizado no processo mantenedor do RRB — o receptor —, embora o ponteiro “início” localize-se fisicamente na memória do processo emissor. Este esquema permite a implementação de um protocolo de troca de mensagens do tipo *write-only*, de modo que apenas escritas remotas pela rede SCI são necessárias, evitando-se leituras remotas. Esta escolha deve-se a uma particularidade da rede SCI: uma leitura remota (*load*) é aproximadamente dez vezes mais lenta do que uma escrita remota (*store*).

Os ponteiros “início” e “fim” são usados pelo emissor, para que possa analisar se há espaço no RRB para a mensagem, bem como pelo receptor, que deve ter a capacidade de perceber se existe ou não uma nova mensagem a ser retirada do RRB. O ponteiro “fim” somente é atualizado pelo emissor e, o “início”, apenas pelo receptor.

Quando uma primitiva do tipo `pvm_send` é invocada, o processo emissor, primeiramente, compara o ponteiro “início”, localmente mantido por ele, com o valor da sua cópia do ponteiro “fim”. A cópia do ponteiro “fim” evita que seja feita uma leitura remota, pois o mesmo é mantido pelo processo receptor. Posto que apenas o processo emissor atualiza o ponteiro “fim”, o valor de sua cópia estará sempre consistente. Após a comparação, se houver espaço no RRB, a mensagem é copiada a partir da posição indicada pelo ponteiro “fim”, sendo este — e sua cópia —, em seguida, atualizado. Por último, o emissor sinaliza uma interrupção no *nodo* do receptor, indicando o envio da mensagem. Isto é possível graças ao suporte a interrupções remotas fornecido pelas placas SCI.

As operações complementares são executadas quando da invocação de uma primitiva do tipo `pvm_recv`. Primeiramente, o processo receptor verifica se foi

sinalizada uma interrupção indicando a chegada de uma mensagem, para somente depois comparar sua cópia do ponteiro “início” com o ponteiro “fim”. Isto feito, o processo receptor pode copiar para o buffer do usuário a mensagem armazenada no RRB, a partir da posição indicada pelo ponteiro “início”, sendo que o tamanho da mensagem é especificado no cabeçalho da mesma. Finalmente, o ponteiro “início” e sua cópia são atualizados.

Note-se que a cópia da mensagem para o RRB, as atualizações dos ponteiros, os testes e a cópia da mensagem para o buffer do usuário envolvem escritas remotas, escritas locais e leituras locais, mas jamais leituras remotas; isto é fundamental para não degradar sobremaneira o desempenho.

A criação, a exportação e o mapeamento dos segmentos de memória que abrigam o RRB e os ponteiros, bem como a sinalização de interrupções remotas, são realizados através de funções do driver SCI. Os acessos remotos à memória são feitos diretamente através da tradicional rotina `memcpy`.

A latência do protocolo de comunicação implementado por PVM-SCI é muito alta, em comparação com o potencial do hardware SCI, devido à utilização das interrupções remotas como mecanismo de sinalização de mensagens. Ademais, a tradicional rotina `memcpy` não é a forma mais otimizada para cópias de memória em arquiteturas contemporâneas, tais como Pentium II e Pentium III, limitando a largura de banda máxima alcançável. Não surpreendentemente, o desempenho de PVM-SCI é péssimo: em PCs Pentium II de 400 MHz, com o *chipset* BX, Fischer e Reinefeld [FIS 99] mensuraram uma latência mínima de 85 μs e uma largura de banda máxima inferior a 15 Mbytes/s.

4.3 SCIPVM

SCIPVM [ZOR 99] representa outro esforço no sentido de promover a utilização do sistema PVM em *clusters* SCI. À semelhança do projeto PVM-SCI, sobre o qual se discorreu na seção anterior, SCIPVM somente utiliza efetivamente a rede SCI para troca de mensagens quando do estabelecimento de uma conexão direta entre dois processos; dito de outra forma, conforme a nomenclatura introduzida por PVM, quando a opção “*Direct Routing*” tiver sido selecionada.

Uma solução não-intrusiva no ambiente PVM nativo foi a proposta inicial de SCIPVM. Neste sentido, foram implementadas novas funções — `pvm_scisend` e `pvm_scirecv` —, que esperam exatamente os mesmos argumentos e apresentam a mesma semântica das funções originais `pvm_psend` e `pvm_precv`, incumbidas da transmissão e recepção de dados de mesmo tipo, em conjunto com a opção “*Direct Routing*”.

Conforme o tamanho da mensagem a ser transmitida, o sistema SCIPVM adota uma técnica diferente. Mensagens grandes são enviadas por DMA, através de chamadas de sistema do tipo `write` e `read` suportadas pelo driver SCI. Por sua vez, a fim de reduzir o impacto à latência devido às trocas de contexto necessárias à invocação de chamadas de sistema, mensagens pequenas são transmitidas diretamente através de escritas remotas em segmentos de memória compartilhados.

No caso de transmissões por DMA, soma-se às chamadas de sistema o *overhead* advindo da necessidade de alinhamento dos *buffers*. Foram também utilizadas técnicas de fragmentação de mensagens, no intuito de facilitar o gerenciamento do espaço limitado dos *buffers* do *kernel*.

Em se tratando do envio de mensagens por meio de memória compartilhada, SCIPVM manipula *buffers* que servem ao mesmo propósito dos RRBs implementados no ambiente PVM-SCI, descrito na seção anterior. Diferentemente, no entanto, o acesso aos *buffers* do sistema SCIPVM, cada qual associado a um dado par emissor/receptor, é controlado por um protocolo que garante a exclusão mútua, de modo que ou o emissor ou o receptor poderá, em um dado momento, efetuar qualquer operação sobre o *buffer* correlato. A garantia de exclusão mútua, neste caso, está relacionada ao gerenciamento do espaço dos *buffers*, e representa um *overhead* adicional. A técnica adotada para fins de exclusão mútua foi a clássica solução de Peterson, mostrada por Tanenbaum [TAN 92].

A implementação do sistema SCIPVM utiliza-se de primitivas do driver SCI tanto para transferências por DMA, quanto para as operações relativas a segmentos compartilhados — criação, exportação e mapeamento. SCIPVM foi originalmente projetado com fulcro em *clusters* baseados em estações de trabalho Sun Ultra-2, rodando o sistema operacional Solaris; portanto, as técnicas de comunicação desenvolvidas levaram em consideração as antigas placas SCI conectáveis ao barramento SBus (SBus-SCI), as quais são destituídas dos chamados *stream buffers* — buffers de leitura e buffers de escrita —, comentados na seção 2.6.2 e cujas principais funções são aumentar a largura de banda máxima e reduzir a latência de comunicação. Destarte, SCIPVM traz pouquíssima contribuição em termos de técnicas a serem empregadas na implementação de eficientes protocolos de troca de mensagens, voltados para as placas do tipo PCI-SCI, utilizadas em *clusters* de PCs, que constituem a arquitetura-alvo da presente Dissertação de Mestrado.

O desempenho de SCIPVM é limitado pelo mecanismo de interrupções, adotado com o fito de indicar o envio de uma mensagem, tal como no ambiente PVM-SCI. Conseqüentemente, observa-se uma utilização tacanha das potencialidades do hardware SCI.

4.4 SCI-MPICH

O mesmo grupo de pesquisadores de Aachen (Alemanha) que idealizou a biblioteca SMI, descrita na seção 3.3, está à frente do projeto SCI-MPICH [WOR 99], que se constitui em uma adaptação do MPICH [GRO 96] — uma implementação portável e livremente distribuída do padrão MPI — para *clusters* de PCs conectados pela tecnologia SCI.

A implementação de SCI-MPICH pressupõe a utilização das placas do tipo PCI-SCI e, atualmente, suporta os sistemas operacionais Solaris, Windows-NT e Linux.

4.4.1 Arquitetura

A adaptação do ambiente MPICH a uma nova tecnologia de comunicação exige somente o redesenho da camada ADI — *Abstract Device Interface* —, haja vista sua estrutura modular. Todas as outras camadas do MPICH permanecem inalteradas. A figura 4.3 mostra a estrutura do MPICH.

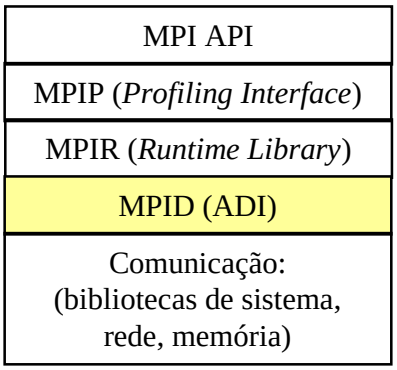

FIGURA 4.3 - Estrutura modular do MPICH.

A camada MPIR responsabiliza-se por transformar as complexas funções MPI em comunicações ponto-a-ponto, e estas são efetivamente realizadas pela camada ADI.

Os projetistas de SCI-MPICH adicionaram um novo dispositivo à camada ADI — `ch_smi`[*] —, o qual atua sobre a rede de comunicação SCI e foi implementado com base na biblioteca SMI (ver seção 3.3). A biblioteca SMI, por sua vez, utiliza funções da API SISCI e do driver SCI (ver seções 3.1 e 3.2).

Durante a inicialização dos processos que constituem uma aplicação MPI, a biblioteca SMI, a API SISCI e o driver SCI são requisitados a fim de possibilitar o estabelecimento de segmentos de memória globais compartilhados e seu mapeamento para os espaços lógicos de endereçamento de cada processo. Ao longo da execução da aplicação MPI, o dispositivo `ch_smi` utiliza-se de alguns serviços da biblioteca SMI, mormente a alocação dinâmica de memória em regiões compartilhadas e o mecanismo de sincronização por barreira.

Todavia, na maior parte do tempo, o dispositivo `ch_smi` está efetuando comunicação ponto-a-ponto. Para tanto, as camadas subjacentes de software — SMI, API SISCI e driver SCI — não precisam ser solicitadas, posto que a comunicação dá-se única e exclusivamente por acessos (escritas) ao espaço de endereçamento dos processos partícipes da aplicação, mais especificamente, aos buffers presentes nas regiões compartilhadas de memória devidamente mapeadas em cada um dos processos.

[*] O nome de cada dispositivo da ADI compõe-se de “ch_” (*channel*) seguido pela designação do mecanismo de comunicação empregado.

A figura 4.4 discrimina as camadas de software subjacentes ao dispositivo `ch_smi` e as funções por ele exercidas.

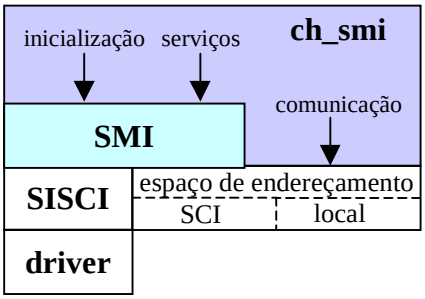

FIGURA 4.4 - O dispositivo `ch_smi` e suas funções.

4.4.2 Protocolos de comunicação

Os protocolos de comunicação do SCI-MPICH encerram técnicas especiais visando à obtenção de baixa latência para mensagens pequenas e elevada largura de banda de pico. Faz-se imprescindível a observância das principais características da rede SCI, a fim de tornar factível a especificação e implementação de protocolos de comunicação de alto desempenho. Dentre tais características, destacam-se:

- Conforme anteriormente mencionado, a escrita remota é aproximadamente dez vezes mais rápida do que a leitura remota. Portanto, é desejável que os protocolos de comunicação evitem leituras remotas.

- A utilização de interrupções remotas como mecanismo para sinalizar o envio de uma mensagem aumenta sobremaneira a latência, e este efeito é tanto mais visível quanto menor for a mensagem enviada.

- As placas PCI-SCI são dotadas de buffers de leitura e buffers de escrita. Particularmente, os buffers de escrita permitem que pacotes SCI sejam enviados pela rede ao mesmo tempo em que outros são submetidos aos buffers, para posterior envio. O resultado observado é a comunicação à maneira *pipeline*, desde que os endereços a que se destinam os pacotes sejam contíguos. O leitor é instado a recorrer à seção 2.6.2 e ao trabalho de Ryan et al. [RYA 96], para uma descrição mais detalhada sobre a técnica de *streaming*, presente nas placas PCI-SCI.

Dependendo do tamanho da mensagem, SCI-MPICH escolhe um dentre três protocolos de comunicação implementados. Esta diversidade de protocolos objetiva encontrar não apenas a forma mais otimizada de transmitir uma mensagem, mas também, um meio-termo entre desempenho e consumo de recursos. Os três protocolos do SCI-MPICH denominam-se *Short*, *Eager* e *Rendez-vous*.

4.4.3 Protocolo *Short*

O protocolo *Short* aplica-se ao envio de mensagens que cabem em um pacote SCI de 64 bytes. Optou-se por este tamanho porque cada buffer de escrita possui exatamente 64 bytes, o que permite o uso mais otimizado dos buffers, visto que, neste caso, prescinde-se do descarregamento (*flush*) explícito dos mesmos. Ademais, 64 bytes podem ser transmitidos em uma única transação SCI, garantindo-se a entrega de todos os bytes do pacote na exata ordem em que são enviados.

Cada pacote usado para mensagens pelo protocolo *Short* compõe-se de um cabeçalho de 12 bytes, um *payload* máximo de 47 bytes, um espaço de alinhamento de 4 bytes e um identificador, que ocupa 1 byte. Assim, o protocolo *Short* destina-se a mensagens de 0 a 47 bytes.

As mensagens enviadas pelo protocolo *Short* devem ser escritas, no espaço de endereçamento do processo receptor, em uma posição específica do *buffer em anel* correspondente ao processo emissor. Melhor dito, os processos mantêm um buffer distinto para cada um dos outros, evitando a ocorrência de acessos simultâneos e, por conseguinte, a necessidade de implementação de protocolos de exclusão mútua. Cada posição dos *buffers em anel* ocupa, obviamente, 64 bytes.

Quando do envio de uma mensagem, basicamente são desempenhadas as seguintes ações pelo emissor:

- Cálculo do identificador da mensagem;
- Cópia da mensagem para a posição corrente do *buffer em anel* apropriado;
- Sinalização do envio da mensagem, escrevendo o valor do identificador da mesma no último byte da posição corrente do *buffer em anel* adequado;
- Atualização da posição de escrita no *buffer em anel*.

Para receber uma mensagem, o processo receptor:

- Lê constantemente (*polling*) o último byte da posição corrente do *buffer em anel* relativo ao processo do qual receberá a mensagem, até que o valor lido seja igual ao identificador esperado;
- Copia a mensagem para o buffer do usuário;
- Atualiza a posição de leitura tocante ao *buffer em anel* referente ao emissor;
- Calcula o valor do identificador da próxima mensagem a ser recebida.

O protocolo acima descrito garante uma baixíssima latência de comunicação porque os pacotes carregam o conteúdo da mensagem em si e, também, o identificador pelo qual o receptor espera, identificador este que indica a chegada de uma mensagem. Isto pode ser feito devido ao fato de que somente uma transação SCI transmitirá os 64 bytes do pacote e, conseqüentemente, o byte com o identificador será, indubitavelmente, o último a ser recebido, de modo que se evita a sinalização da chegada de uma mensagem parcialmente recebida, o que poderia redundar na cópia de dados inválidos para o buffer do usuário.

A figura 4.5 ilustra um cenário em que o “processo 0” envia uma mensagem ao “processo 1”, por meio do protocolo *Short*. São mostradas as principais estruturas de dados usadas em conjunto com o protocolo. Tipicamente, cada *buffer em anel* possui 64 posições (*slots*).

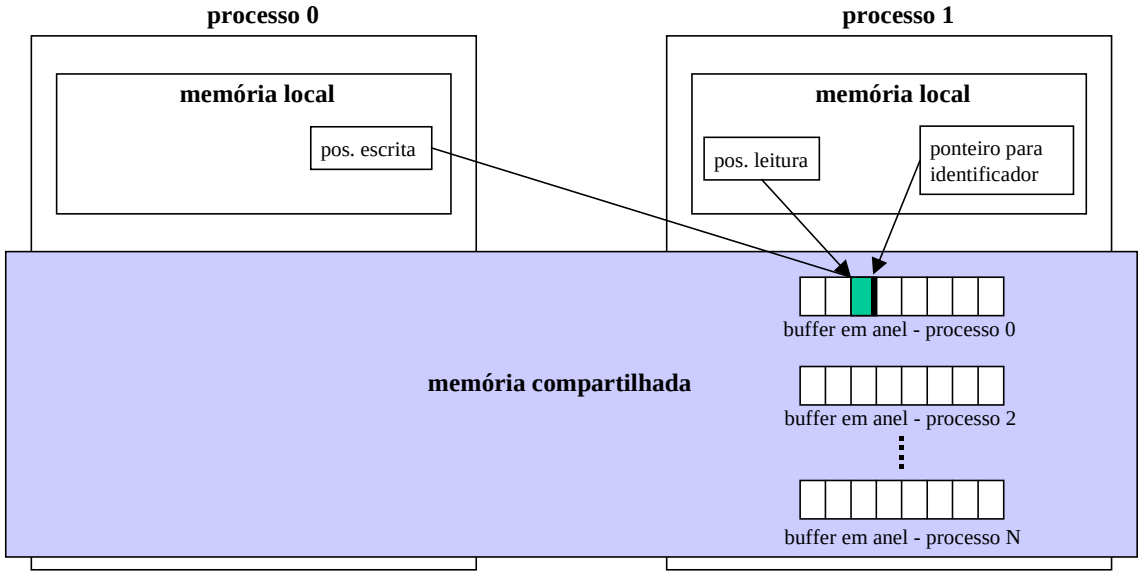

FIGURA 4.5 - Protocolo *Short* implementado no ambiente SCI-MPICH.

Diferentemente de PVM-SCI e de SCIPVM, os protocolos de comunicação do ambiente SCI-MPICH não se utilizam de interrupções para sinalizar o envio de mensagens. Mensagens maiores do que 47 bytes não podem ser enviadas pelo protocolo *Short*. Nestes casos, os outros dois protocolos entram em cena.

4.4.4 Protocolo *Eager*

Similarmente ao protocolo *Short*, cada processo mantém um *buffer eager* para cada um dos demais, a fim de evitar a necessidade de implementação de esquemas de exclusão mútua. A diferença principal está no tamanho dos buffers: por *default*, um *buffer eager* possui oito posições (*slots*), cada qual com 16 kbytes. Estes valores podem ser configurados pelo usuário, mas deve ser observado que o protocolo *Eager* será utilizado pelo SCI-MPICH para mensagens maiores do que 47 bytes e menores ou iguais ao tamanho do *slot* do *buffer eager*.

Outra diferença marcante entre os protocolos *Short* e *Eager* diz respeito à forma de sinalizar ao receptor o envio de uma mensagem. No protocolo *Short*, basta enviar um único pacote de 64 bytes, que conterá a mensagem em si e o seu identificador, ou seja, o ato de enviar a mensagem também serve para sinalizar a sua chegada. O receptor perceberá a chegada da mensagem quando houver, na última posição do *slot* corrente, o identificador esperado. No caso do protocolo *Eager*, tal esquema não é possível por dois motivos: a quantidade de dados a enviar é variável, conforme o tamanho da mensagem,

e serão necessárias, na grande maioria dos casos, mais de uma transação SCI para levar a efeito a transmissão.

Em sendo assim, é exigida uma etapa adicional de comunicação. Após o envio dos dados referentes à mensagem, transmite-se um pacote de controle ao receptor, que o espera, por meio de *polling*, em um endereço de memória próprio para a sua recepção. Após o recebimento do pacote de controle, o receptor toma ciência da chegada de uma mensagem no *slot* corrente do *buffer eager* associado ao processo emissor e, então, copia-a ao buffer do usuário, residente na memória local.

A observância de um detalhe sutil é vital para o correto funcionamento do protocolo *Eager*: o pacote de controle deve ser enviado após ter-se certeza de que todos os pacotes SCI constituintes da mensagem já chegaram ao devido *slot* do *buffer eager*. Embora isto seja óbvio, garantir esta condição requer conhecimento acerca do funcionamento da rede SCI. Posto que uma porção contígua de dados presentes na memória é submetida em blocos de 64 bytes a cada um dos oito buffers de escrita da placa SCI, é possível que, após a submissão dos últimos blocos de 64 bytes da mensagem, nem todos já tenham chegado ao seu destino ou, pior ainda, alguns ainda estejam armazenados na placa. Desta sorte, antes de encaminhar o pacote de controle, é imprescindível esvaziar os buffers de escrita da placa SCI.

4.4.5 Protocolo *Rendez-vous*

O protocolo *Eager* manipula buffers estaticamente alocados, ou seja, impossibilita a troca de mensagens maiores do que o tamanho comportado pelos *slots*. Isto implica na necessidade de se dispor de um protocolo capaz de lidar com mensagens de tamanho arbitrário. Para atender a esta requisição, o protocolo *Rendez-vous* adota recursos dinamicamente gerenciados, cuja preparação exige um mecanismo de *handshaking*.

O nome *Rendez-vous* advém do caráter síncrono do protocolo. Primeiramente, o processo que deseja estabelecer comunicação submete uma mensagem do tipo `REQUEST_SEND`, contendo o tamanho da mensagem de dados a ser transmitida. Ante a recepção da solicitação, o receptor aloca um determinado espaço de memória compartilhada para receber a mensagem. Primeiramente, é feita a tentativa de alocar uma porção de memória do tamanho informado pelo emissor; se não for possível, uma quantidade menor será alocada. Em qualquer dos casos, o receptor retorna ao emissor uma mensagem `OK_TO_SEND`, contendo o endereço e o tamanho da área de memória alocada.

De posse do endereço e do tamanho da área de memória disponibilizada pelo receptor, o emissor divide a mensagem e começa a copiar blocos de dados para o devido local. O número de bytes de cada bloco é divisível pela quantidade de memória alocada pelo receptor. Ao término da cópia de um bloco, o emissor envia uma mensagem de controle do tipo `BLOCK_READY` e inicia a cópia de outro bloco. Ao receber a mensagem `BLOCK_READY`, o receptor pode copiar para o buffer do usuário o bloco recém escrito pelo emissor, ou seja, o receptor pode ler a mensagem do buffer de transmissão antes do emissor tê-lo preenchido completamente. Este esquema permite aumentar a largura de banda máxima alcançável.

Quando a quantidade de memória alocada pelo receptor não for suficiente para armazenar toda a mensagem, uma nova cópia de blocos será necessária. Neste caso, após enviar o último bloco da primeira parte da mensagem, o emissor submete uma mensagem do tipo `CONT` ao receptor que, então, copia o último bloco da primeira parte para o buffer do usuário e retorna uma confirmação `OK_TO_SEND`. Em seguida, o emissor começa a transmissão da nova seqüência de blocos da mensagem. O protocolo assim prossegue, sucessivamente, até a transmissão do último bloco pertencente à última parte da mensagem.

O funcionamento do *Rendez-vous* encontra-se ilustrado na figura 4.6 [WOR 99].

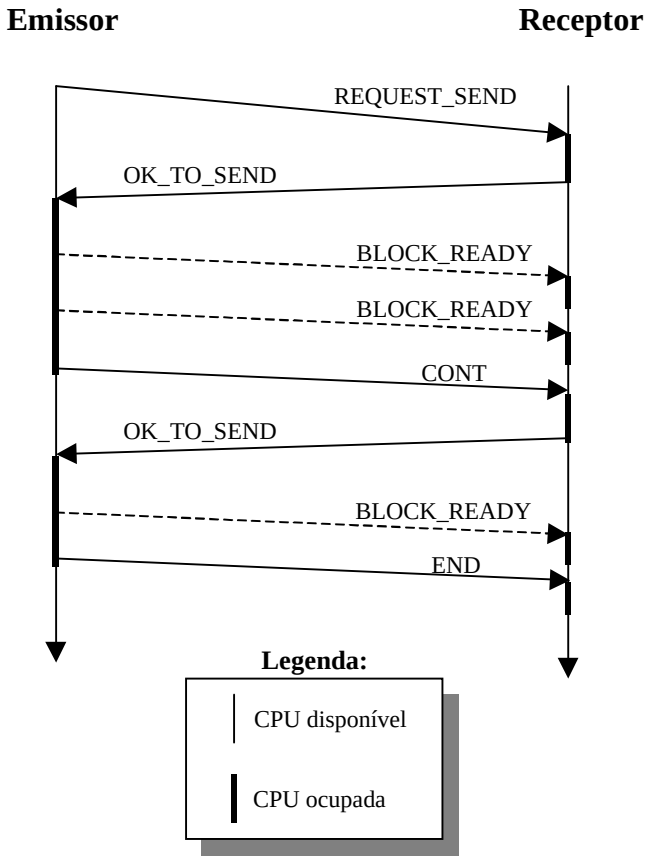

FIGURA 4.6 - Protocolo *Rendez-vous* do ambiente SCI-MPICH.

O protocolo *Rendez-vous* serve a dois propósitos. Além de permitir a troca de mensagens de tamanho arbitrário, é o responsável por aumentar a máxima largura de banda alcançável por SCI-MPICH. O protocolo *Eager*, uma vez que exige que o receptor repasse a mensagem ao buffer do usuário somente após a total transmissão da mesma, limita a largura banda a patamares situados muito aquém das possibilidades da rede SCI. O mecanismo de leituras e escritas sobrepostas do *Rendez-vous* eleva a largura de banda a valores próximos ao limite da rede SCI. Análises de desempenho mais detalhadas sobre SCI-MPICH serão feitas em um capítulo vindouro,

comparativamente com a biblioteca de programação paralela resultante desta Dissertação de Mestrado.

4.4.6 Considerações finais

As escritas remotas, necessárias durante as transferências de mensagens nos protocolos do SCI-MPICH, não são realizadas por meio da tradicional rotina `memcpy`, diferentemente de PVM-SCI e SCIPVM. SCI-MPICH emprega transferências de 64 bits do processador ao barramento PCI, utilizando a FPU — *Floating Point Unit*. Este artifício influi decisivamente para a elevação da máxima largura de banda que se pode obter.

O grupo envolvido com o projeto SCI-MPICH continua bastante ativo. A atual versão de SCI-MPICH [WOR 2000] sofreu alguns ajustes e recebeu novas funcionalidades. O protocolo *Short* foi adaptado para suportar eficientemente as mais recentes placas PCI-SCI de 64 bits, dotadas de 16 *stream buffers* de 128 bytes, em oposição aos antigos 8 *stream buffers* de 64 bytes. Ademais, os protocolos *Eager* e *Rendez-vous* foram estendidos de modo a permitir a troca assíncrona de mensagens por DMA e interrupções remotas. Outro avanço digno de nota foi a inclusão da interface de programação para E/S paralela, conforme definição do padrão MPI-2, cuja implementação utiliza-se da rede SCI. Por fim, as escritas remotas agora são levadas a efeito por meio de instruções do tipo MMX — *Math Matrix Extensions* —, presentes no Pentium II e Pentium III, e não mais através da FPU, o que permite aumentar ainda mais a máxima largura de banda alcançável.

4.5 ScaMPI

ScaMPI [HUS 99][SCA 2000] é uma implementação do padrão MPI para *clusters* SCI, distribuída comercialmente pela empresa *Scali Affordable Supercomputing*. A empresa *Scali AS* uniu-se à *Dolphin Interconnect Solutions*, fabricante das placas SCI, e responsabiliza-se pela distribuição do SSP — *Scali Software Platform* —, que abarca, além do ScaMPI, uma implementação do driver SCI (ver seção 3.1), baseada naquele desenvolvido na Universidade de Oslo [RYA 97], uma implementação da API SISCI (ver seção 3.2) e um software para o gerenciamento de *clusters* SCI.

Em verdade, os projetistas do ambiente SCI-MPICH, apresentado na seção anterior, buscaram inspiração no ScaMPI para o desenvolvimento dos protocolos de comunicação *Short*, *Eager* e *Rendez-vous*. ScaMPI define três protocolos de comunicação distintos, escolhidos transparentemente conforme o tamanho das mensagens trocadas. Obviamente, os protocolos de comunicação do ScaMPI adotam a estratégia *write-only*, evitando leituras remotas, a fim de não comprometer o desempenho. Além disso, como era de se esperar, cada receptor mantém um buffer para cada emissor, de modo a eliminar a possibilidade de acessos concorrentes e a necessidade de se garantir a exclusão mútua entre os processos.

Mensagens pequenas — tipicamente, menores ou iguais a 32 bytes — são trocadas através do protocolo *Inlining*. Este protocolo lida com *buffers em anel*, tal

como o protocolo *Short* do SCI-MPICH, transmitindo cada mensagem em um pacote de 64 bytes, que contém, além do *payload*, informações de controle. Neste protocolo, as mensagens são auto-sincronizantes, isto é, o pacote que as abriga serve também para sinalizar o seu próprio envio.

Para mensagens maiores, à semelhança de SCI-MPICH, ScaMPI utiliza o conceito de *eager buffers*. Trata-se exatamente da mesma estratégia empregada por SCI-MPICH, sendo a transferência de uma mensagem composta de duas etapas: cópia da mensagem para um compartimento do *eager buffer* reservado ao processo emissor; e, por fim, envio de um pacote de controle, sinalizando a chegada da mensagem recém transmitida.

O terceiro protocolo do ScaMPI é o chamado *Transporter* que, a rigor, consiste em um mecanismo de *rendez-vous* similar àquele do SCI-MPICH, permitindo não somente a troca de mensagens de tamanho arbitrário, mas também, o aumento da largura de banda de pico.

Apesar da nítida correspondência entre os protocolos do SCI-MPICH e estes aqui mencionados, o projeto do ScaMPI distingue-se pelo trato especial de duas questões relevantes: tolerância a falhas e *thread-safety*.

Todos os protocolos do ScaMPI empregam a técnica de *checkpointing*, possibilitando o reenvio de mensagens ante situações de falhas. A verificação de erros de comunicação é feita com o auxílio das informações de *status* mantidas pelo driver SCI. Em termos gerais, as seguintes etapas são seguidas durante a transmissão de uma mensagem:

- Estabelecimento de um *checkpoint*;
- Envio da mensagem;
- Esvaziamento dos buffers da placa SCI;
- Verificação de *status*;
- Reenvio da mensagem, se necessário.

A mesma seqüência de ações acima é também executada sempre que um pacote de controle precisa ser enviado para sinalizar a transmissão de uma mensagem, como no caso do protocolo *Eager*, por exemplo. Deste modo, as trocas de mensagens são realmente confiáveis.

Nos casos em que uma falha manifesta-se durante a transmissão do pacote *response* (ver seção 2.4.2 e figuras 2.3 e 2.4) de uma transação SCI atinente à comunicação em curso, um erro pode ser detectado ainda que a mensagem (ou um pacote de controle) tenha sido entregue com sucesso. Por esta razão, as estruturas de dados do ScaMPI são idempotentes, ou seja, a múltipla atualização de um dado não acarreta efeitos colaterais desastrosos.

Os protocolos de comunicação do ScaMPI foram desenvolvidos pelo uso de uma camada de software, também de autoria da *Scali AS*, que provê primitivas especializadas para a comunicação através de memória compartilhada. O denominado ScaFun [HUS 99a] fornece funções para a escrita e leitura de dados, tendo como alvo tanto segmentos SCI compartilhados entre diferentes *nodos* do *cluster*, quanto a memória local. As

primitivas do ScaFun são otimizadas conforme a arquitetura-alvo: em Pentiums II e III, as comunicações são levadas a efeito por meio de instruções MMX, ao passo que em máquinas Ultra-SPARC adotam-se as instruções *load*/*store* de 64 bytes. Todas as rotinas do ScaFun são *thread-safe*; outrossim, esta mesma propriedade é notada no ScaMPI.

As operações de inicialização realizadas internamente pelo ScaMPI, tais como criação, exportação e mapeamento de segmentos compartilhados, são efetivadas por intermédio das funções do driver SCI (ver seção 3.1).

Avaliações de desempenho acerca dos protocolos de comunicação do ScaMPI serão mostradas oportunamente no texto, através de uma análise que os cotejará com os protocolos de outros dois ambientes: o SCI-MPICH e a biblioteca de programação paralela que é o ponto culminante desta Dissertação de Mestrado.

4.6 CML: *Common Messaging Layer*

Outro trabalho relacionado ao desenvolvimento de bibliotecas voltadas à troca de mensagens em *clusters* SCI é o CML [HER 98]. Trata-se de um conjunto de primitivas, de baixo nível, especialmente projetadas para fornecer bases a implementações de MPI e PVM. CML encontra-se no contexto do projeto SISCI [EBE 97], brevemente descrito na seção 3.2.1 (ver figura 3.1) deste texto.

Apesar das existentes diferenças semânticas entre MPI e PVM, CML traz um conjunto único de funções a serem utilizadas por implementações de ambos ambientes de programação. Cabe salientar, contudo, que os trabalhos apresentados nas seções anteriores — PVM-SCI, SCIPVM, SCI-MPICH e ScaMPI — são completamente independentes do CML, tendo seus próprios protocolos de comunicação.

O ambiente ora delineado não introduz técnicas inovadoras no que tange à comunicação por troca de mensagens em *clusters* SCI. Suas estruturas de dados foram projetadas de modo a evitar leituras remotas e desnecessárias cópias de dados em memória, duas das principais causas de ineficiência em sistemas de troca de mensagens.

Mensagens cujo tamanho não ultrapasse um limite determinado são enviadas através de escritas remotas em segmentos compartilhados, segmentos estes que abrigam *buffers em anel* compostos de vários *slots*, gerenciados como uma fila circular. Estas estruturas de dados seguem a mesma abordagem dos RRBs — *Receive Ring Buffers* — do ambiente PVM-SCI (ver seção 4.2). Ao contrário de PVM-SCI, no entanto, a sinalização do envio de mensagens é feita através de *flags*, cada qual associada a um *slot* do RRB, e não por interrupções remotas, sendo imprescindível a realização de *polling* por parte do receptor de uma mensagem.

Por outro lado, a troca de mensagens consideradas grandes dá-se pelo mecanismo de DMA, liberando a CPU para outras tarefas. Os autores argumentam ser este procedimento próprio para elevar a largura de banda.

Eberl et al. [EBE 98] analisaram o desempenho do CML, em um *cluster* SCI baseado em máquinas do tipo Pentium II de 233 MHz, rodando o sistema operacional

Linux. Constataram-se uma latência mínima de comunicação em torno de 14 μs, e uma largura de banda máxima de 35 Mbytes/s.

4.7 Demais trabalhos correlatos

O mesmo grupo de pesquisadores que idealizou o CML tem investigado diversas questões concernentes à comunicação por troca de mensagens em *clusters* SCI. São também de sua autoria duas outras propostas: SCI AM [HEL 97] e SSLib [WEI 97]. SCI AM, como sugere o nome, é uma implementação da noção de *Active Messages* [EIC 92], em consonância com a versão 2.0 da especificação da API; SSLib, por sua vez, é uma adaptação da API BSD Sockets ao contexto de *clusters* SCI.

Apesar das diferenças inerentes a cada um dos modelos subentendidos por CML, SCI AM e SSLib, as três propostas compartilham diversas técnicas para a comunicação por troca de mensagens em SCI, mormente, a noção de RRBs e os mecanismos de sinalização da entrega de mensagens. Observam-se sutis variações tocantes aos procedimentos de inicialização e ao número e tamanho dos *slots* dos RRBs, adequados às necessidades de cada abordagem. No caso do SSLib, por exemplo, os *slots* não podem ter tamanho fixo, diferentemente dos outros dois ambientes, visto que a semântica imposta pelos Sockets prevê o envio de quantidades variáveis de bytes, sem a abstração de mensagem. De qualquer modo, as técnicas empregadas pelos três ambientes foram norteadas pelas mesmas idéias.

Eberl et al. [EBE 98] comparam as três propostas, levando em consideração questões de projeto e aspectos de desempenho. A largura de banda máxima praticamente permanece a mesma — por volta de 35 Mbytes/s —, mas observam-se nítidas diferenças na latência mínima de comunicação. SCI AM mostra uma latência de 7 μs, em contraposição aos valores 14 μs e 13 μs, obtidos por CML e SSLib, respectivamente. Esta diferença deve-se à natureza do modelo encerrado pela noção de *Active Messages*.

Destaca-se, igualmente, outro núcleo de comunicação para SCI baseado em *Active Messages*, concebido por um grupo da Universidade da Califórnia, em Santa Bárbara [IBE 96], cuja implementação difere um pouco da abordagem seguida por SCI AM. Cumpre ressaltar que a incorporação do conceito de *Active Messages* em núcleos de comunicação de baixo nível tem sido evidenciada no contexto das redes de alto desempenho, não sendo uma prerrogativa exclusiva de SCI.

O supramencionado SSLib não é a única proposta de adaptação da pilha de protocolos TCP/IP à rede SCI; outra iniciativa neste sentido é o SCIP [TAS 98a]. Ao invés de substituir os protocolos de transporte originais, como o faz SSLib, SCIP é uma interface IP implementada como um driver de rede. O restante da pilha de protocolos permanece inalterado, sendo apenas necessário modificar os mecanismos de transmissão e manipulação de pacotes IP. Novamente, o conceito de *buffer em anel* foi utilizado, sendo que a recepção de pacotes IP é sinalizada por interrupção.

Diferentemente dos modelos anteriormente estudados, entretanto, no âmbito do SCIP cada *nodo* possui apenas um *buffer em anel* que pode ser acessado concorrentemente pelos demais *nodos*. A fim de garantir que acessos concorrentes

sejam feitos em posições distintas, cada *nodo*, antes de escrever no buffer, atomicamente lê e altera o valor de uma variável que indica a próxima posição disponível. Esta variável-índice é acessada através de transações SCI atômicas do tipo *fetch & increment*, as quais exibem a desvantagem clara de degradar o desempenho, como toda leitura remota. Testes com o SCIP, rodando em um *cluster* SCI composto de Pentiums II de 400 MHz, apontaram uma largura de banda máxima de 31 Mbytes/s e uma latência de 77 μs, segundo Taskin et al. [TAS 98a].

4.8 Últimas considerações

Este capítulo versou sobre os principais trabalhos comprometidos com o desenvolvimento de bibliotecas de comunicação por troca de mensagens especificamente projetadas para *clusters* SCI, concentrando as discussões nas técnicas adotadas pelos protocolos de comunicação.

As observações aqui constantes, somadas aos detalhes técnicos apresentados no capítulo 2, e à apreciação dos ambientes de programação delineados no capítulo 3, ditaram as diretrizes para o projeto e implementação da biblioteca de programação paralela com a qual culmina a presente Dissertação de Mestrado.

5 DECK/SCI: a biblioteca de programação paralela proposta

O capítulo ora introduzido descreve as abstrações proporcionadas pela biblioteca de programação paralela desenvolvida neste Trabalho de Mestrado, bem como a semântica das primitivas especificadas. Desta sorte, as seções a seguir dedicam-se a uma apresentação cuja conotação assemelha-se à dos manuais de programação, abstendo-se de tecer comentários sobre a implementação do ambiente sendo descrito. Inicialmente, porém, as origens e o contexto em que se insere este Trabalho são aludidos.

As seções seguintes fornecem bases ao entendimento do próximo capítulo, que trata dos detalhes da implementação propriamente dita da biblioteca aqui apresentada, e das decisões de projeto norteadoras deste Trabalho.

5.1 Contexto e motivações

O advento das redes de interconexão de alto desempenho consagrou os *clusters* de estações de trabalho, ou de computadores pessoais, como arquiteturas para a execução eficiente de aplicações paralelas. O grande atrativo de tais arquiteturas é o baixo custo, associado a um desempenho comparável ao das máquinas paralelas dedicadas, cujo preço restringe o seu uso a poucas companhias ou centros de pesquisa.

A excelente relação custo–desempenho evidenciada pelos *clusters* impulsionou a sua disseminação, tanto no meio acadêmico e em instituições de pesquisa, quanto em companhias. A decorrência imediata desta proliferação foi o crescimento do interesse por pesquisas acerca do desenvolvimento de protocolos de comunicação *ad hoc*, especificamente projetados para extrair o máximo das redes de alto desempenho, haja visto que a simples presença do hardware de comunicação de alta performance não garante que o mesmo seja utilizado de forma profícua.

O GPPD — Grupo de Processamento Paralelo e Distribuído — do Instituto de Informática da Universidade Federal do Rio Grande do Sul tem conduzido pesquisas centradas na problemática advinda da utilização de *clusters* como arquiteturas para a execução de aplicações paralelas. Destaca-se, neste contexto, a biblioteca de programação paralela DECK — *Distributed Execution Communication Kernel* —, concebida pelo referido grupo de pesquisa, inicialmente, para explorar a rede de alto desempenho Myrinet, integrando comunicação e multiprogramação. A implementação do então chamado DECK/Myrinet [BAR 2000] utiliza diretamente as rotinas do núcleo de comunicação de baixo nível BIP [PRY 98], o qual provê protocolos que utilizam eficientemente a rede Myrinet.

Após a aquisição de um novo *cluster*, equipado com a rede SCI, anteviram-se novas frentes de pesquisa no GPPD. Com a disponibilidade de dois *clusters* nas dependências do GPPD, e buscando inspiração na emergente área de *Metacomputing* ou *Grid Computing* [FOS 99], idealizou-se o projeto MultiCluster [BAR 2000a], que tem como primordial objetivo a definição de um modelo de integração de *clusters*, de modo

que se possa constituir um "*cluster de clusters*" ou "*multicluster*". A idéia norteadora do MultiCluster é ensejar a construção de uma máquina paralela única, formada a partir de vários *clusters*, a fim de que esta arquitetura integrada possa vir a ser utilizada para o desenvolvimento e execução de aplicações paralelas, independentemente da tecnologia de comunicação em que se baseia cada um dos *clusters* componentes. Obviamente, o ambiente de programação para o desenvolvimento de aplicações paralelas no âmbito do MultiCluster será o DECK.

Antes desta Dissertação de Mestrado, apenas dispunha-se do DECK/Myrinet; no entanto, para possibilitar a consecução do modelo MultiCluster, havia a necessidade de se levar a termo o DECK/SCI, ou seja, uma biblioteca de programação paralela por troca de mensagens especificamente para *clusters* SCI, com exatamente a mesma API exportada pelo já existente DECK/Myrinet.

Embora as implementações sejam totalmente independentes e diferentes, devido ao MultiCluster ambas bibliotecas de programação paralela — DECK/Myrinet e DECK/SCI — devem fornecer o mesmo conjunto de primitivas, e as primitivas correspondentes devem apresentar a mesma semântica; do contrário, a integração dos *clusters* Myrinet e SCI torna-se impossível. Sob a ótica do programador do "*multicluster*", dispor-se-á de um único ambiente de programação, e os dois *clusters* serão vistos como apenas uma máquina paralela.

O modelo de integração do MultiCluster define a necessidade de um *nodo-gateway* em cada *cluster*, sendo sua incumbência entremear a comunicação entre máquinas pertencentes a *clusters* distintos. Destarte, considerando-se a infra-estrutura atualmente disponível no GPPD, a figura 5.1 ilustra o modelo de integração de *clusters* consoante a proposição do MultiCluster.

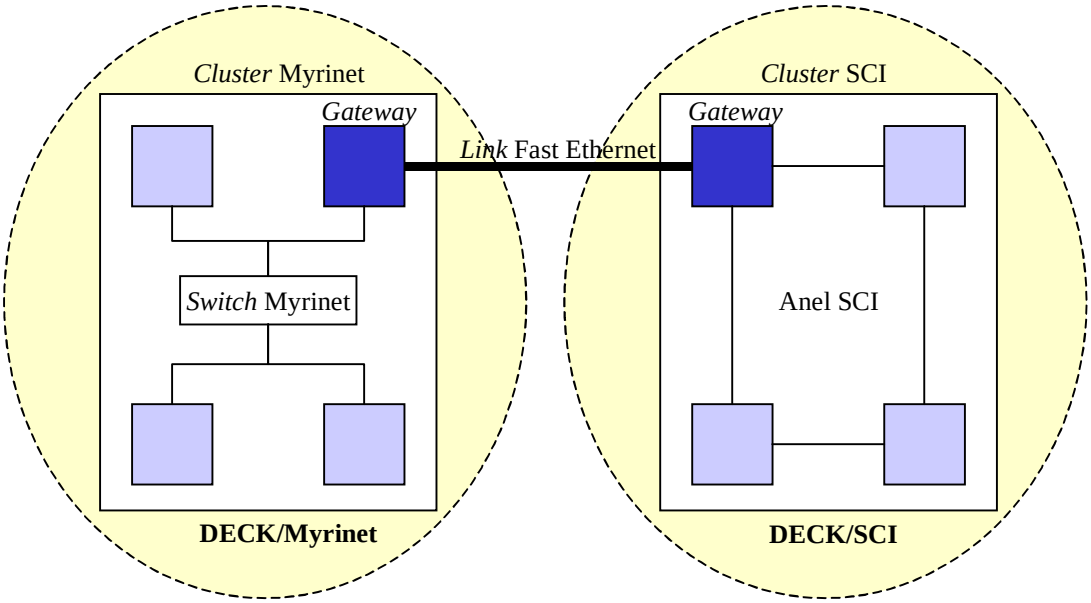

FIGURA 5.1 - Modelo de integração do MultiCluster.

Cumpre frisar, contudo, que o DECK/SCI, antes de ser um componente vital para a consecução do modelo MultiCluster, é uma biblioteca de programação paralela de *clusters* SCI por troca de mensagens, que garante aos programadores a utilização efetiva das potencialidades de alto desempenho da rede SCI. Dito de outra forma, a despeito do MultiCluster, o DECK/SCI permite o desenvolvimento e a execução eficiente de aplicações paralelas em *clusters* baseados na tecnologia SCI. Ademais, questões atinentes à integração de *clusters* Myrinet e SCI estão além do escopo desta Dissertação.

5.2 API e modelo de programação

À primeira vista, pode soar redundante e estranha a apresentação, nesta Dissertação, da API do DECK, pois, conforme supramencionado, a interface de programação do DECK/Myrinet, já descrita por Barreto [BAR 2000], deve ser exatamente a mesma do DECK/SCI, proposto no presente Trabalho. Todavia, faz-se necessária uma nova descrição formal da API dos DECKs, não somente para permitir ao leitor o melhor entendimento dos detalhes de implementação e das decisões de projeto do DECK/SCI, mas, principalmente, porque a API sofreu drásticas alterações, desde o surgimento do DECK/Myrinet.

Convém enfatizar que as alterações na interface de programação não foram feitas para facilitar a implementação do DECK/SCI; antes, o fito único desta reestruturação foi tornar mais confortável a utilização dos DECKs por parte dos programadores de aplicações paralelas.

A API a seguir apresentada é atualmente adotada pelo DECK/SCI e pelo DECK/Myrinet. Observe-se que os DECKs apenas dão suporte às linguagens de programação C e C++.

5.2.1 Funções de propósito geral

O DECK segue o modelo de programação SPMD, de modo que cada processo componente da aplicação paralela apresenta exatamente o mesmo código. É disparado um processo por *nodo*, e o conjunto de *nodos* do *cluster* que serão utilizados para a execução da aplicação deve ser informado ao utilitário `deckrun`, o qual efetivamente inicia os processos.

```
int deck_init(int *argc, char ***argv);
int deck_node(void);
int deck_numnodes(void);
int deck_barrier(void);
int deck_done(void);
```

FIGURA 5.2 - Funções de propósito geral do DECK.

Na figura 5.2, acham-se discriminadas as funções de propósito geral do DECK. São primitivas que tratam do ambiente de execução instaurado pelo DECK e,

adicionalmente, uma primitiva para a sincronização por barreira de todos os processos de uma aplicação.

Antes de qualquer outra primitiva da biblioteca DECK, todos os processos devem invocar a função `deck_init`, sendo de sua responsabilidade a inicialização do ambiente de execução, durante a qual diversas estruturas de dados internas são devidamente alocadas e preparadas.

A primitiva `deck_node` retorna o identificador único do *nodo* no qual o processo invocador está sendo executado. Note-se que o ambiente de execução do DECK atribui um identificador único — de *0* a *N – 1* — a cada um dos *N nodos* utilizados pela aplicação. De forma complementar, a função `deck_numnodes` retorna o número de *nodos* usados pela aplicação paralela em execução.

A função `deck_barrier` tem por incumbência estabelecer um ponto global de sincronização entre todos os processos da aplicação, através da noção de barreira. Segue-se a semântica habitual, qual seja, a função somente encerrará após todos os processos a terem invocado.

Para finalizar a aplicação paralela, cada um dos processos deve chamar a função `deck_done`, destinada a desalocar todas as estruturas de dados internas e a desfazer o ambiente de execução.

5.2.2 Multiprogramação

A API do DECK, além de oferecer funções para a comunicação por troca de mensagens, conforme será mostrado em seguida, especifica primitivas responsáveis pela criação e manipulação de *threads*. Também, para fins de sincronização entre *threads*, encontram-se na API funções que lidam com a abstração de semáforo.

Em verdade, as funções de multiprogramação do DECK visam a promover uma interface mais amigável do que a do padrão Pthreads [IEE 95], encapsulando suas primitivas. Entretanto, apenas um subconjunto bastante reduzido do referido padrão é atualmente providenciado.

Conquanto o modelo de programação do DECK preveja um processo por *nodo* e, possivelmente, mais de uma *thread* por processo, a implementação do DECK/SCI, resultante desta Dissertação e detalhada no próximo capítulo, não incorporou as funções que manipulam *threads* e semáforos. Posto que o objetivo maior deste Trabalho de Mestrado reside na investigação e projeto de protocolos de troca de mensagens para *clusters* SCI e, dada a simplicidade das funções de multiprogramação do DECK, optou-se, neste momento, pela concentração de todos os esforços na especificação e implementação de eficientes protocolos de comunicação para a rede SCI. Em sendo assim, as primitivas do DECK que dão conta dos recursos de multiprogramação não serão arroladas neste texto.

5.2.3 Mensagens

Qualquer operação de comunicação entre diferentes *nodos* executando uma aplicação paralela DECK exige, necessariamente, o envio e o recebimento de mensagens. A abstração de mensagem é explicitada pelo tipo `deck_msg_t`, estando arroladas na figura 5.3 as funções incumbidas da sua manipulação.

```
int deck_msg_create(deck_msg_t *m, unsigned long size);
int deck_msg_pack(deck_msg_t *m, int type, void *datum, int n);
int deck_msg_unpack(deck_msg_t *m, int type, void *datum, int n);
int deck_msg_clear(deck_msg_t *m);
int deck_msg_reset(deck_msg_t *m);
int deck_msg_getbuffer(deck_msg_t *m, void **buffer);
int deck_msg_destroy(deck_msg_t *m);
```

FIGURA 5.3 - Funções para a manipulação de mensagens.

A primeira ação a ser feita sobre uma variável do tipo `deck_msg_t` é efetivar a criação propriamente dita da mensagem. Para tanto, deve-se utilizar a primitiva `deck_msg_create`, que se destina a inicializar a variável do tipo mensagem e a reservar, aos dados a serem enviados ou recebidos, uma área de memória cujo tamanho especificou-se no argumento `size`.

Depois da criação da mensagem, a mesma passa a ficar apta ao envio ou recebimento de dados. Caso a mensagem tenha sido criada para o envio de dados, haverá que se preenchê-la antes da comunicação ser efetivada, existindo duas formas possíveis para isto: ou através da primitiva `deck_msg_pack`, ou pelo uso de `deck_msg_getbuffer`.

A primitiva `deck_msg_pack` responsabiliza-se por inserir na mensagem dados de tipos predefinidos. Quando da chamada à função, conforme pode ser visto na figura 5.3, especifica-se que se deseja inserir, na mensagem passada como argumento, "`n`" elementos do tipo "`type`", os quais estão localizados na área de memória apontada por "`datum`". Pode-se invocar quantas vezes forem necessárias a função `deck_msg_pack` e, a cada invocação, o cursor associado à mensagem é atualizado, indicando a próxima posição a partir da qual novos dados serão inseridos.

O argumento `type` pode assumir um dos valores discriminados na tabela 5.1, conforme o tipo do(s) dado(s) inserido(s) na mensagem. Note-se que, além dos tipos básicos, é possível inserir em uma mensagem ponteiros e, inclusive, outras mensagens.

Se, porventura, desejar-se inserir novos dados em uma mensagem, descartando aqueles que lhe tenham sido previamente inseridos, dever-se-á lançar mão da primitiva `deck_msg_clear`, antes de efetuar o preenchimento. A referida primitiva reinicializa o cursor associado à mensagem, de modo que novos dados possam ser inseridos a partir do início da mesma, sobrepondo tudo que nela se encontrava. Isto é bastante útil para a

utilização de uma única variável do tipo `deck_msg_t` em diferentes operações de comunicação.

TABELA 5.1 - Tipos de dados usados em mensagens no DECK.

Tipo do dado a ser inserido	Valor do argumento `type`
`char`	`DECK_CHAR`
`unsigned char`	`DECK_UCHAR`
`short`	`DECK_SHORT`
`unsigned short`	`DECK_USHORT`
`long`	`DECK_LONG`
`unsigned long`	`DECK_ULONG`
`float`	`DECK_FLOAT`
`double`	`DECK_DOUBLE`
`void*`	`DECK_POINTER`
`char*`	`DECK_STRING`
`deck_msg_t`	`DECK_MSG`

Além de `deck_msg_pack`, a função `deck_msg_getbuffer` proporciona outra forma de preenchimento de uma mensagem para posterior envio. Conforme pode-se observar na figura 5.3, `deck_msg_getbuffer` retorna, no ponteiro `buffer`, o endereço da área de dados associada à mensagem passada como argumento. De posse de tal endereço, o programador pode, explicitamente, preencher a mensagem como bem entender, sem a necessidade de invocar `deck_msg_pack`. É importante perceber que as cópias diretas de dados ao buffer da mensagem não implicam em atualizações do cursor a ela associado.

Variáveis do tipo `deck_msg_t` devem também ser utilizadas para o recebimento de dados. Neste caso, depois do recebimento propriamente dito, é necessário extrair os dados da mensagem. Para tanto, existe a primitiva `deck_msg_unpack`, cujo funcionamento é análogo ao da função de preenchimento de mensagens. A fim de se obterem os resultados esperados, é de vital importância extrair os dados da mensagem na mesma ordem em que foram inseridos. A cada invocação de `deck_msg_unpack`, o cursor associado à mensagem é atualizado, passando a apontar para a próxima posição a partir da qual mais dados serão extraídos.

Caso se venha a utilizar uma mesma variável do tipo `deck_msg_t` para sucessivos recebimentos de dados, far-se-á necessário invocar a função `deck_msg_reset` precedendo a cada seqüência de extrações. Analogamente à primitiva `deck_msg_clear`, `deck_msg_reset` reinicializa o cursor associado à mensagem, de modo que as operações de extração subseqüentes atuem a partir do início do buffer da mesma.

À semelhança da operação de preenchimento, é também possível ao programador extrair os dados diretamente, utilizando o endereço do buffer da mensagem, o qual pode ser obtido, conforme supracomentado, por meio da função `deck_msg_getbuffer`.

A última função de manipulação de mensagens da API do DECK é `deck_msg_destroy`. Conforme sugere o nome, esta função libera todos os recursos consumidos por uma variável do tipo mensagem, que, logo após, torna-se inválida.

5.2.4 Comunicação

No âmago do DECK, estão as primitivas de comunicação ponto-a-ponto e a abstração de caixa postal (*mail box*). Cada *thread* de uma aplicação DECK pode criar caixas postais para receber mensagens de outras *threads* da aplicação. De fato, o envio de mensagens a caixas postais é a única forma de comunicação entre *threads* pertencentes a *nodos* distintos. Na figura 5.4, arrolam-se as primitivas de comunicação e de manipulação de caixas postais do DECK, as quais serão explicadas a seguir.

```
int deck_mbox_create(deck_mbox_t *mb, char *name);
int deck_mbox_clone(deck_mbox_t *mb, char *name);
int deck_mbox_post(deck_mbox_t *mb, deck_msg_t *msg);
int deck_mbox_retrv(deck_mbox_t *mb, deck_msg_t *msg);
int deck_mbox_destroy(deck_mbox_t *mb);
```

FIGURA 5.4 - Funções de comunicação e de manipulação de caixas postais.

A criação de uma caixa postal, representada por uma variável do tipo `deck_mbox_t`, requer que um nome único, no âmbito de toda a aplicação paralela, lhe seja atribuído. A função `deck_mbox_create` reserva os recursos necessários a uma nova caixa postal, e a ela associa o nome passado como argumento, o qual, convém lembrar, não deve ter sido atribuído a nenhuma outra caixa postal criada por qualquer *thread* da aplicação paralela. Naturalmente, cada *thread* pode criar quantas caixas postais forem necessárias, sendo o limite determinado pela disponibilidade de recursos, tais como espaço em memória.

A caixa postal é a "entidade" através da qual uma *thread* espera receber mensagens de outras *threads* que se encontram em *nodos* diferentes do seu. Antes que possa enviar mensagens a um caixa postal, a *thread* emissora deve invocar a primitiva `deck_mbox_clone`, passando como argumentos uma referência a `deck_mbox_t` e o nome da caixa postal a que se destinarão as mensagens. A instância de `deck_mbox_t`, retornada por `deck_mbox_clone` em `mb`, deve ser utilizada nas subseqüentes operações de envio de mensagens.

Fazendo uma analogia com o paradigma de programação orientada a objetos, `deck_mbox_t` seria uma classe, e as funções `deck_mbox_create` e `deck_mbox_clone` seriam os construtores de objetos desta classe.

A figura 5.5 retrata o efeito das funções `deck_mbox_create` e `deck_mbox_clone`. Na figura, ilustram-se três *threads* sendo executadas em distintos *nodos*. A "*thread* 2" criou uma caixa postal; as outras duas *threads*, por sua vez,

invocaram a função `deck_mbox_clone`, o que lhes permitirá, futuramente, enviar mensagens à caixa postal criada pela "*thread* 2". Em última análise, pode-se dizer que a primitiva `deck_mbox_clone` relaciona uma variável do tipo `deck_mbox_t` a uma caixa postal previamente criada.

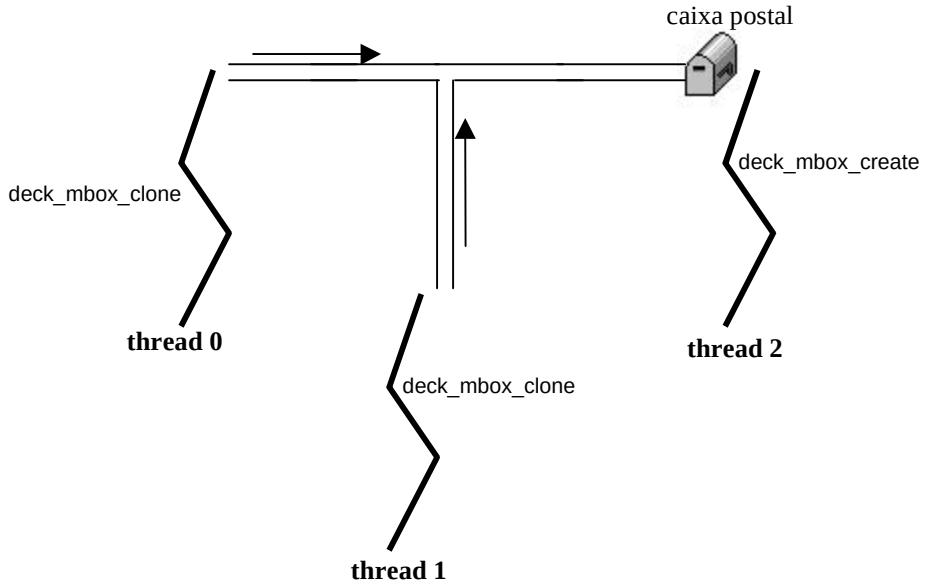

FIGURA 5.5 - Manipulação de caixas postais no DECK.

As funções de comunicação do DECK são `deck_mbox_post` e `deck_mbox_retrv`. A primeira efetiva o envio de uma mensagem — anteriormente inicializada e preenchida segundo os procedimentos descritos na seção 5.2.3 — a uma caixa postal, desde que se tenha estabelecido, de antemão, por intermédio da função `deck_mbox_clone`, uma relação entre o nome da caixa postal que receberá a mensagem e a variável do tipo `deck_mbox_t` passada como argumento à primitiva de comunicação.

Para o recebimento de mensagens a primitiva apropriada é `deck_mbox_retrv`. Esta primitiva retira uma mensagem de uma caixa postal que tenha sido criada pela *thread* invocadora, armazenando-a em uma variável do tipo `deck_msg_t`, também devidamente alocada. Após a recepção, os dados contidos na mensagem podem ser lidos, seja pelo uso da primitiva `deck_msg_unpack`, seja pelo acesso direto ao buffer da mensagem, cujo endereço obtém-se através de uma chamada a `deck_msg_getbuffer`, consoante descreveu-se na seção 5.2.3.

Caso não haja nenhuma mensagem disponível na caixa postal em questão, no momento da chamada à função `deck_mbox_retrv`, sua execução persistirá até que alguma transmissão seja feita. Tão logo uma mensagem esteja disponível na caixa postal, a mesma é copiada para a variável do tipo `deck_msg_t` passada como argumento, e somente depois a função `deck_mbox_retrv` encerra. Dito de outra forma, `deck_mbox_retrv` é uma primitiva bloqueante, assim como o é a função

`deck_mbox_post`, ou seja, ambas primitivas garantem, após seu retorno, o acesso seguro e correto aos buffers alocados pelos programadores em variáveis do tipo `deck_msg_t`. Assume-se, neste texto, a mesma conotação sugerida por Hwang e Xu [HWA 98] para primitivas bloqueantes, no cenário de troca de mensagens.

A abstração de caixa postal traz consigo algumas implicações. O DECK deve garantir que mensagens enviadas por uma *thread* a uma mesma caixa postal sejam retiradas na exata ordem em que foram transmitidas. Por outro lado, mensagens enviadas por diferentes *threads* à mesma caixa postal podem ser obtidas em qualquer ordem pela *thread* receptora.

Como era de se esperar, há também uma função responsável por liberar os recursos associados a uma variável do tipo `deck_mbox_t`. A referida função é `deck_mbox_destroy`, que pode atuar tanto sobre variáveis inicializadas através de `deck_mbox_create`, como sobre aquelas instanciadas por meio de `deck_mbox_clone`.

5.3 Exemplo de uso do DECK

A figura 5.6 exemplifica a utilização das principais funções da interface de programação do DECK. O singelo programa-exemplo, mostrado na figura, faz uma mensagem circular entre os *nodos* do *cluster*, tal como se fosse um *token* e, quando retorna ao seu originador — o "*nodo* 0", no caso —, o conteúdo da mensagem é mostrado na tela. Considere-se, a título de ilustração, que o programa seja executado em um *cluster* composto por quatro *nodos*. O "*nodo* 0" enviará uma mensagem ao "*nodo* 1", contendo os valores 56.89, 235 e 189. O "*nodo* 1", por sua vez, repassa esta mensagem ao "2", que a remete ao "3", o qual, finalmente, devolve-a ao "*nodo* 0". Ao receber a mensagem que circulou por todos os *nodos*, o "*nodo* 0" mostra seu conteúdo na tela.

Posto que o DECK segue o modelo SPMD, será disparado, em cada *nodo* do *cluster*, um processo com o código mostrado na figura. A primeira rotina invocada é `deck_init`, que instaura o ambiente de execução do DECK. Em seguida, a *thread* principal de cada processo cria uma caixa postal, a ela associando o nome "Caixa Postal - *n*", onde "*n*" é o identificador do *nodo* no qual a *thread* invocadora encontra-se em execução, identificador este obtido através da função `deck_node`. A caixa postal criada é representada pela variável `own_mb`.

Logo após, cada *thread* estabelece um "canal de acesso" com a caixa postal criada pela *thread* de seu *nodo-vizinho*, através da função `deck_mbox_clone`, da seguinte forma: a *thread* do "*nodo* 0" prepara-se para poder enviar mensagens à caixa postal da *thread* do "*nodo* 1"; a *thread* do "*nodo* 1" estabelece uma relação com a caixa postal da *thread* do "*nodo* 2", e assim, sucessivamente, sendo que a *thread* do "*nodo N–1*" obtém acesso à caixa postal da *thread* do "*nodo* 0". O "canal de acesso" de cada *thread* à caixa postal da sua vizinha é representado pela variável `next_mb`.

Todas as *threads*, que não a do "*nodo* 0", reservam espaço para uma mensagem, na variável `recv_msg`, e ficam bloqueadas no `deck_mbox_retrv`, esperando receber

a mensagem de uma de suas *threads* vizinhas. Ante o recebimento da mensagem, simplesmente a repassam para sua outra vizinha.

Diferentemente, a *thread* do "*nodo* 0" reserva espaço para duas mensagens, nas variáveis `recv_msg` e `send_msg`. Em seguida, insere três dados na mensagem representada por `send_msg` e a envia para a *thread* do "*nodo* 1", iniciando sua circulação pelo *cluster*. Depois disso, fica bloqueada até receber a mensagem da *thread* do "*nodo N–1*", que será armazenada na variável `recv_msg`. Finalmente, extrai os dados da mensagem recebida e os mostra na tela.

Após as operações de comunicação, cada *thread* libera os recursos associados a suas respectivas mensagens e à variável `next_mb`. Subseqüentemente à sincronização de todos os processos em uma barreira — `deck_barrier` —, as *threads* desfazem-se dos recursos consumidos por suas caixas postais. Por fim, a primitiva `deck_done` encerra o ambiente de execução DECK.

A figura 5.6, com o programa-exemplo acima explicado, acha-se na página seguinte.

```
#include <deck.h>
#include <stdio.h>

int main(int argc, char **argv) {
  char name[30];
  float f;
  long i[2];
  deck_mbox_t own_mb, next_mb;
  deck_msg_t send_msg, recv_msg;

  deck_init(&argc,&argv);

  sprintf(name, "Caixa postal - %d", deck_node());
  deck_mbox_create(&own_mb, name);

  sprintf(name, "Caixa postal - %d", (deck_node()+1)%deck_numnodes());
  deck_mbox_clone(&next_mb, name);

  deck_msg_create(&recv_msg, 20);

  if (deck_node() == 0) {
    deck_msg_create(&send_msg, 20);
    f = 56.89; i[0] = 235; i[1] = 189;
    deck_msg_pack(&send_msg, DECK_FLOAT, &f, 1);
    deck_msg_pack(&send_msg, DECK_LONG, i, 2);
    deck_mbox_post(&next_mb, &send_msg);

    deck_mbox_retrv(&own_mb, &recv_msg);
    f = 0.0; i[0] = 0; i[1] = 0;
    deck_msg_unpack(&recv_msg, DECK_FLOAT, &f, 1);
    deck_msg_unpack(&recv_msg, DECK_LONG, i, 2);
    printf("Valores recebidos: %f  %ld  %ld\n", f, i[0], i[1]);
    deck_msg_destroy(&send_msg);
  }
  else {
    deck_mbox_retrv(&own_mb, &recv_msg);
    deck_mbox_post(&next_mb, &recv_msg);
  }
  deck_msg_destroy(&recv_msg);
  deck_mbox_destroy(&next_mb);
  deck_barrier();
  deck_mbox_destroy(&own_mb);

  deck_done();

  return(0);
}
```

FIGURA 5.6 - Exemplo de utilização das primitivas do DECK.

6 Projeto e implementação do DECK/SCI

O presente capítulo trata dos detalhes de implementação do DECK/SCI, a biblioteca de programação paralela por troca de mensagens proposta nesta Dissertação de Mestrado. Especificamente, desvelam-se, nas seções que se seguem, as principais decisões de projeto que nortearam este Trabalho, e os motivos que compeliram a tais escolhas. O cerne deste capítulo é a descrição dos protocolos e técnicas de comunicação idealizados e implementados.

Os mecanismos propostos são confrontados com aqueles incorporados em trabalhos relacionados, trabalhos estes delineados no capítulo 4.

6.1 Objetivos específicos

As motivações da proposição e do desenvolvimento de uma nova biblioteca para programação paralela, baseada em comunicação por troca de mensagens, especificamente para *clusters* SCI, foram devidamente expostas no capítulo 5. Além de se mostrar peça fundamental do projeto MultiCluster [BAR 2000a], conforme já se mencionou no capítulo que a este antecede, o DECK/SCI traz consigo objetivos gerais que incluem proporcionar uma interface de programação de fácil assimilação, com abstrações e serviços apropriados para a programação de aplicações paralelas; e a utilização eficiente das potencialidades da rede de alto desempenho SCI.

Especificamente no que concerne à eficiente utilização da rede SCI, o projeto do DECK/SCI teve fulcro na garantia de baixa latência de comunicação durante a troca de mensagens pequenas entre diferentes *nodos* do *cluster*, e na consecução de uma largura de banda de pico próxima aos limites do hardware subjacente. Os protocolos de comunicação concebidos para integrar o arcabouço do DECK/SCI deixam transparecer os esforços no sentido de atingir à meta de assegurar, a aplicações paralelas, mecanismos compatíveis com alto desempenho computacional, demanda pelo qual cresce ininterruptamente.

6.2 Plataforma de hardware e software

O desenvolvimento do DECK/SCI, bem como todos os testes realizados e relatados no próximo capítulo, deu-se em um *cluster* composto por quatro máquinas SMPs dual Pentium III, de 500 MHz, com o *chipset* Intel-BX e dotadas de 256 Mbytes de memória. Cada máquina do *cluster* é equipada com uma placa PCI-SCI — para o barramento PCI de 32 bits, e operando a 33 MHz —, modelo D312, de duas dimensões, ou seja, com dois *links* de entrada e dois de saída, fabricada pela *Dolphin Interconnect Solutions*, contendo o *chip* controlador de *link* LC-2 e o PSB — interface entre o barramento PCI e o barramento interno da placa — revisão D. A topologia utilizada para a rede SCI foi um anel simples (*ringlet*) conectando os quatro *nodos* do *cluster*, embora as placas suportem um *torus* bidimensional.

As placas SCI em uso toleram uma largura de banda de até 400 Mbytes/s mas, devido às limitações do barramento PCI, o *throughput* máximo alcançável nesta arquitetura é inferior a 90 Mbytes/s. A latência mínima de comunicação possível fica por volta de 2,5 μs. Enfatizando o que se comentou na seção 2.6.2, as placas PCI-SCI, do modelo utilizado, possuem oito buffers de escrita e oito buffers de leitura.

Adicionalmente, os *nodos* do *cluster* também estão conectados por uma rede Fast Ethernet, operando a 100 Mbits/s *full-duplex*.

O sistema operacional adotado é o Linux, com *kernel* versão 2.2.14. Os drivers das placas SCI usados são distribuídos pela empresa *Scali Affordable Supercomputing*, como parte do SSP (*Scali Software Platform*) versão 2.0.2, e estão configurados para explorar os oito buffers de escrita, através da técnica de *stream combining* (ver seção 2.6.2 e o trabalho de Ryan et al. [RYA 96]).

6.3 Manipulação de segmentos compartilhados

Posto que a comunicação em uma rede SCI, no nível do hardware, é efetuada através de segmentos de memória compartilhados, a primeira decisão envolvida no projeto do DECK/SCI diz respeito à forma de lidar com este compartilhamento. À luz da infra-estrutura de software atualmente disponível para *clusters* SCI, fez-se necessário escolher uma dentre as interfaces de programação desenvolvidas com vistas ao gerenciamento e estabelecimento de segmentos compartilhados entre os *nodos* da rede SCI.

Neste contexto, possíveis opções recairiam sobre o driver SCI, a *SCI Physical Layer API*, a API SISCI, SMI ou YASMIN, todas apresentadas no capítulo 3 e explicitamente cotejadas na tabela 3.1. As três primeiras opções são interfaces de programação de baixo nível, enquanto SMI e YASMIN compreendem uma biblioteca de programação com recursos sofisticados e um ambiente de execução, sendo mais apropriadas ao desenvolvimento de aplicações paralelas do que ao de software básico.

Face à natureza de SMI e YASMIN, descartaram-se prontamente estes ambientes de programação do projeto do DECK/SCI. Os serviços necessários à implementação do DECK/SCI restringem-se tão-somente a primitivas para o estabelecimento de segmentos e os mapeamentos afins, prescindindo-se de mecanismos como paralelização de laços, presentes em SMI, por exemplo.

Ademais, tanto SMI quanto YASMIN exigem que os segmentos compartilhados criados pelos *nodos* sejam mapeados no mesmo espaço lógico de endereçamento de todos os processos, o que implica em um ponto global de sincronização associado à criação de cada segmento. Isto não é desejável porque o DECK/SCI está centrado na idéia de caixas postais pertencentes a *threads*, e cada caixa postal está associada a um segmento compartilhado, conforme será mostrado mais adiante. Quando uma nova caixa postal é criada em um *nodo*, nele cria-se um novo segmento compartilhado, evitando-se assim o desperdício de recursos que adviria caso um segmento compartilhado enorme, originado no início da aplicação, fosse usado para abrigar todas as caixas postais. Ora, como uma caixa postal somente poderá ser acessada pelos *nodos* nos quais alguma *thread* tenha efetuado uma operação do tipo `deck_mbox_clone`

(ver seção 5.2.4), é totalmente despropositada a imposição de um ponto global de sincronização toda vez em que uma caixa postal for criada. Por isso, SMI e YASMIN mostram-se incompatíveis com as necessidades do DECK/SCI, além do fato de instaurarem seu próprio ambiente de execução, impedindo o total controle sobre a implementação do DECK/SCI e, possivelmente, acarretando um demasiado *overhead* à inicialização de uma aplicação DECK.

Em eliminando SMI e YASMIN do rol de interfaces de programação elegíveis pelo DECK/SCI, restaram o driver, a *SCI Physical Layer API* e a API SISCI. O driver e a *SCI Physical Layer API* apresentam uma interface de programação de extremo baixo nível, sendo seu emprego suscetível a erros. Além disso, a única implementação disponível da *SCI Physical Layer API* é bastante limitada, com bem menos recursos do que prevê a sua especificação formal, e o único sistema operacional por ela suportado é o Windows-NT.

Entre o driver SCI e a API SISCI — afinal, as alternativas realmente consideráveis —, a última pareceu ser a melhor escolha. A API SISCI possui uma interface de programação de baixo nível, possibilitando o total controle da implementação do DECK/SCI e, ao mesmo tempo, permite uma programação mais confortável do que pelo uso direto do driver. Em verdade, a API SISCI (ver seção 3.2.3) encapsula as principais funções do driver e ainda dá ensejo ao acesso a recursos do hardware SCI.

Portanto, na implementação do DECK/SCI, utilizam-se as primitivas da API SISCI, basicamente, para as seguintes tarefas:

- inicialização da rede SCI;
- criação e exportação de segmentos de memória compartilhados;
- mapeamento de segmentos compartilhados aos espaços lógicos de endereçamento dos processos;
- esvaziamento (*flush*) dos buffers de escrita das placas SCI.

Há disponíveis, atualmente, duas implementações da especificação da API SISCI: uma distribuída pela empresa *Dolphin Interconnect Solutions*, e outra, pela *Scali Affordable Supercomputing*. No desenvolvimento do DECK/SCI, empregou-se a implementação da API SISCI distribuída pela *Scali* em conjunto com o SSP versão 2.0.2.

6.4 Protocolos de comunicação propostos

O âmago do DECK/SCI está nos seus protocolos de comunicação, onde se acham as decisões de projeto cruciais à consecução dos objetivos maiores deste Trabalho: garantir, às aplicações paralelas, um desempenho de comunicação próximo aos limites do hardware, particularmente, baixíssima latência de comunicação para mensagens pequenas e elevada largura de banda de pico.

A principal propriedade da tecnologia SCI, enquanto rede de alto desempenho para *clusters*, é a possibilidade de acesso direto a endereços remotos de memória, sem a

intervenção do sistema operacional e de qualquer camada adicional de software. É exatamente esta característica que confere à rede SCI a capacidade de assegurar uma latência de comunicação entre processos em torno de 2,5 μs.

Contudo, a concepção de protocolos de comunicação exige o tratamento de questões fundamentais, a exemplo do controle de fluxo, a fim de evitar a perda de mensagens que pode decorrer da diferença de velocidade entre o processo emissor e o receptor, e do seqüenciamento de mensagens, que visa a promover a entrega ordenada das mensagens trocadas entre um par emissor–receptor. O controle de fluxo e o seqüenciamento de mensagens representam um *overhead* inevitável, visto que, antes de eficiente, a comunicação deve, irrevogavelmente, estar comprometida com a correção. Em não havendo a interferência de chamadas de sistema durante operações de comunicação, como no caso da rede SCI, é de total responsabilidade dos protocolos evitar que a performance proporcionada pelo hardware seja demasiadamente onerada; melhor dito, o desempenho resultante é única e exclusivamente afetado pelas técnicas concebidas.

Além do controle de fluxo e da ordenação de mensagens, outro aspecto imprescindível ao projeto de protocolos de comunicação diz respeito à sinalização do envio de uma mensagem, isto é, os meios pelos quais o receptor é informado de que lhe foi remetida uma mensagem. Os mecanismos de sinalização são outra fonte inevitável de *overhead* aos protocolos, contribuindo para a redução do desempenho ótimo que o hardware pode proporcionar, de modo que tanto mais eficiente será um protocolo de comunicação, quanto mais otimizadas forem as técnicas de sinalização adotadas.

6.4.1 Desempenho máximo e particularidades da rede SCI

Na forma mais elementar de comunicação, desprovida de qualquer gerenciamento apurado, para que dois *nodos* possam trocar mensagens através da rede SCI, primeiramente, o receptor precisa criar um segmento de memória compartilhado, mapeá-lo ao seu espaço lógico de endereçamento e exportá-lo ao emissor que, por sua vez, também efetua o referido mapeamento. Após o estabelecimento dos mapeamentos, tudo que o emissor precisa fazer é copiar a mensagem para um endereço de memória pertencente ao segmento criado pelo receptor, através de instruções do tipo *store* da CPU, ou seja, por simples atribuições ou por meio de uma rotina como a tradicional `memcpy`. Isto feito, o emissor deve, de alguma forma, informar o receptor do envio da mensagem; então, ao perceber a chegada da mesma, o receptor pode, opcionalmente, copiá-la para algum buffer do usuário, localizado na memória local.

Conquanto conceitualmente simples a idéia supra-referida, é necessário atentar para algumas peculiaridades da rede SCI, visando a auferir o máximo desempenho possível. Obtiveram-se algumas medidas, na arquitetura descrita na seção 6.2, com o fito de absorver informações acerca das idiossincrasias do hardware SCI e, destarte, melhor conduzir a elaboração dos protocolos de comunicação do DECK/SCI.

As mensurações constantes das figuras 6.1 e 6.2 referem-se à comunicação “pura” em SCI, sem sinalização, e destituída de qualquer *overhead* que não o da transmissão por si só de dados.

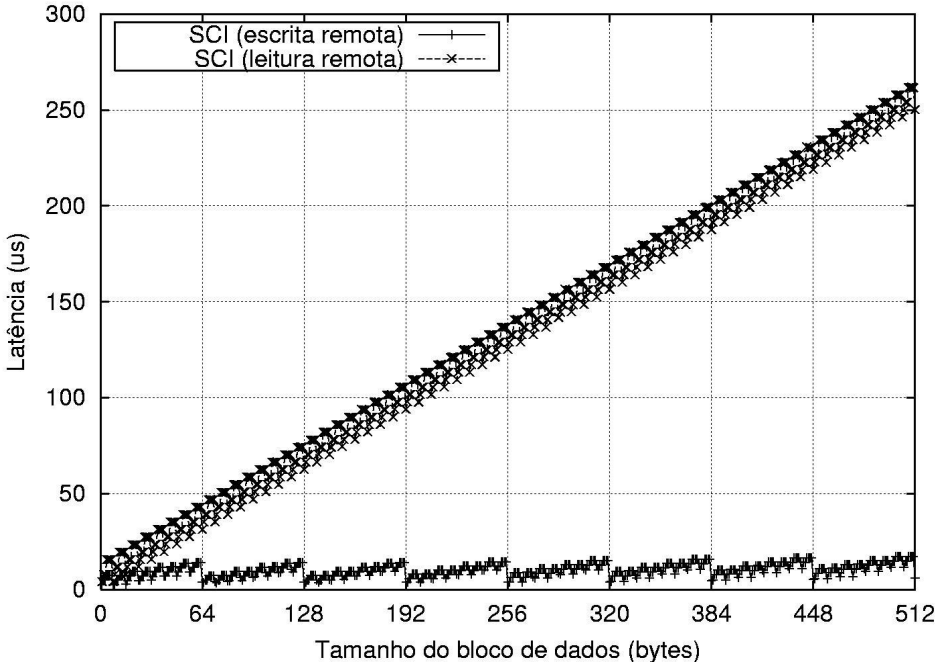

FIGURA 6.1 - Latência para o acesso à memória remota pela rede SCI.

A figura 6.1 revela algumas peculiaridades da rede SCI. Observe-se a cristalina diferença entre as taxas de crescimento da latência de comunicação resultante da leitura de dados de um segmento remoto, e da latência decorrente da escrita remota. Constata-se que as operações de escrita remota, comparadas à leitura remota, experimentam uma elevação extremamente suave na latência, à medida que a quantidade de dados transmitidos aumenta, sendo quase imperceptível na escala do gráfico mostrado. Isto corrobora o fato de que leituras remotas são, aproximadamente, dez vezes mais lentas do que escritas remotas, e sugere a elaboração de protocolos de comunicação do tipo *write-only*, isto é, baseados apenas e tão-somente em escritas remotas.

Ainda analisando a figura 6.1, pode-se notar outra propriedade da rede SCI. Nas faixas delimitadas por múltiplos consecutivos de 64 bytes, no eixo horizontal, a curva de latência concernente à escrita remota tem uma aparência relativamente "caótica", assemelhando-se a "dentes de serra". Outrossim, os menores valores de latência são obtidos para quantidades de dados múltiplas de 64 bytes; por isso, a curva mostra quedas abruptas nos valores 64, 128, 192, etc., como se pode aquilatar na figura. Perceba-se que é mais rápido transmitir 64 do que enviar quantidades entre 17 e 63 bytes; é mais rápido enviar 128 do que 66 a 127 bytes; e esse comportamento repete-se indefinidamente.

A explicação para o aspecto de "dentes de serra" da curva de latência, e para o ótimo desempenho da escrita remota de dados em quantidades múltiplas de 64 bytes, está na utilização dos buffers de escrita da placa SCI, aliada à técnica de *stream combining*, comentada na seção 2.6.2 e detalhada por Ryan et al. [RYA 96]. Cada um dos oito buffers de escrita da placa SCI tem exatamente 64 bytes. Quando um bloco

contíguo de dados é enviado, este bloco é internamente separado em grupos de 64 bytes, sendo cada grupo armazenado em um buffer diferente. O propósito desta técnica é conferir um certo grau de paralelismo à transmissão de dados, de forma que enquanto um buffer está sendo preenchido, o conteúdo de outro buffer pode ser submetido à rede de comunicação pela placa SCI. Com o uso combinado dois oito buffers, tem-se um *pipeline* de oito estágios para a transmissão de um bloco necessariamente contíguo de dados. Não coincidentemente, o tamanho máximo do *payload* de um pacote SCI é exatamente 64 bytes; assim, quando um buffer é completamente preenchido, uma única transação SCI é necessária para a transmissão dos seus 64 bytes. Se, porventura, um buffer for descarregado antes de completar os 64 bytes, serão necessários vários pacotes SCI, cada qual carregando, no máximo, 16 bytes. Por exemplo, a transmissão de 60 bytes requer o envio de quatro pacotes SCI, representando quatro transações, ao passo que o envio de 64 bytes exige apenas uma transação. Generalizando este cenário ao contexto dos oito buffers da placa, pode-se entender que a aparência serrilhada da curva de latência deve-se à variação da quantidade de pacotes necessários ao envio dos dados, sendo que esta variação não é diretamente proporcional ao número de bytes enviados.

Como era de se imaginar, ao ser preenchido um buffer, automaticamente é gerada uma transação SCI, o que contribui ainda mais para a redução do tempo de transmissão de dados em quantidades múltiplas de 64 bytes. Se o gráfico da figura 6.1 traçasse somente a latência dos valores múltiplos de 64 do eixo das abscissas, sua variação para a escrita remota seria sempre crescente e suave, como o leitor pode perceber analisando isoladamente estes valores. Isto é corolário do fato de que o número de pacotes SCI requeridos para a comunicação aumenta em uma unidade a cada acréscimo de 64 bytes na quantidade de dados.

Todas essas constatações propõem que eficientes protocolos de comunicação por troca de mensagens devam evitar transmitir pela rede um número de bytes que não seja múltiplo de 64, no intuito de otimizar a utilização dos buffers de escrita, fazendo pleno uso da comunicação à maneira *pipeline* proporcionada pelo hardware SCI.

A figura 6.2, diferentemente, possibilita uma avaliação da rede SCI sob outra perspectiva, mostrando a variação da largura de banda. Apresentam-se três curvas, sendo uma referente à largura de banda resultante de leitura remota e, as outras duas, tocantes à comunicação por escrita remota. Ratificando as conclusões anteriores, a largura de banda máxima alcançável por leitura remota é nitidamente inferior àquela que se pode auferir por escrita remota.

A informação mais importante denotada pela figura 6.2 é a diferença de desempenho entre duas formas distintas de efetuar a escrita remota. Através da tradicional rotina `memcpy`, presente na biblioteca padrão da linguagem C, a máxima largura de banda evidenciada foi 49,94 Mbytes/s. Em contrapartida, a utilização de instruções do tipo MMX na escrita remota garante uma largura de banda máxima de 87,72 Mbytes/s. As instruções MMX, com seu paralelismo SIMD — *Single Instruction Multiple Data* — inerente, produzem “rajadas” de dados no barramento PCI, reduzindo o número de transações que lhe são submetidas e, por conseguinte, otimizando o acesso à placa SCI. Destarte, a incorporação de instruções MMX em protocolos de comunicação por troca de mensagens surge como uma recomendável prática na busca por alto desempenho.

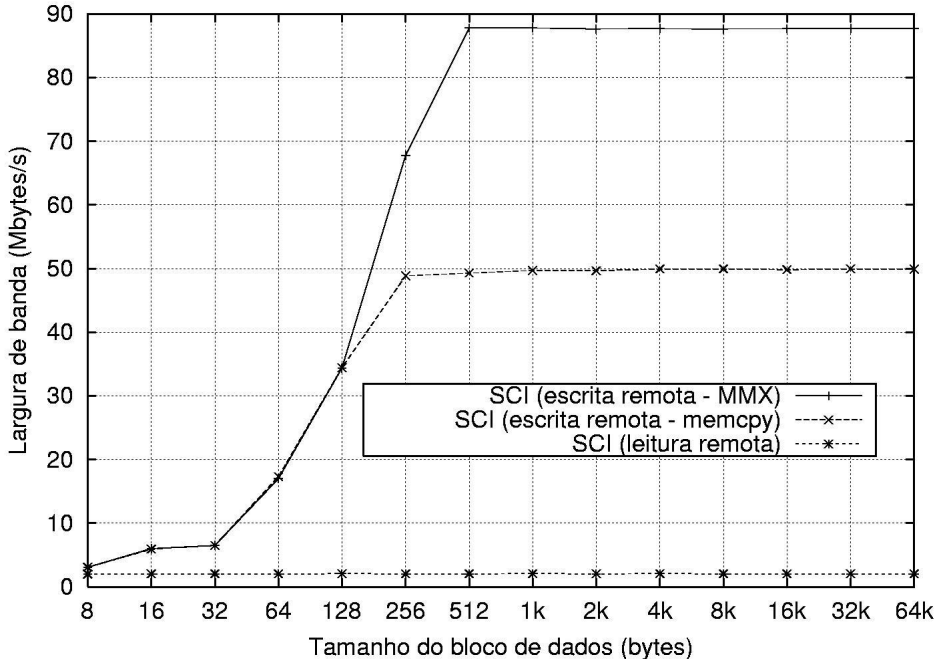

FIGURA 6.2 - Largura de banda para o acesso à memória remota pela rede SCI.

Seja através de instruções MMX, seja por meio da tradicional rotina `memcpy`, observa-se que a largura de banda de pico é atingida com a transmissão de 512 bytes, exatamente a quantidade máxima de dados comportada pelos oito buffers de escrita em conjunto ($8 \times 64 = 512$), representando o ponto, a partir do qual, o *pipeline* constituído pelos buffers permanece com todos os estágios ocupados. A obtenção de 87,72 Mbytes/s na transmissão de 512 bytes é um resultado esplêndido, que atesta a capacidade da rede SCI de propiciar baixas latências de comunicação, e comprova a potencialmente exitosa aplicabilidade de mecanismos de acesso direto à memória remota no projeto de eficientes protocolos de comunicação por troca de mensagens.

As medidas externadas pelos gráficos das figuras 6.1 e 6.2 retratam o máximo desempenho que se pode obter na rede SCI em que os testes foram realizados (ver seção 6.2), indicando os limites tecnológicos intransponíveis impostos aos protocolos de comunicação do DECK/SCI, a saber: latência mínima de 2,5 μs, e largura de banda máxima de 87,72 Mbytes/s.

6.4.2 Sinalização do envio de mensagens

O processo (ou *thread*) que espera receber mensagens precisa dispor de meios que o permitam perceber o momento da chegada de uma mensagem, ou, ao menos, se existe alguma disponível. Em termos gerais, há duas formas para isto: *polling* e interrupções.

No caso de *polling*, basicamente o que o processo receptor faz é ler constantemente o valor contido em um endereço de memória predeterminado, até que o

mesmo passe a indicar a condição de chegada de uma mensagem. Este endereço pode conter simplesmente uma *flag*, ou uma estrutura de dados mais elaborada, que traga informações como o tamanho da mensagem e o local onde foi armazenada.

Por outro lado, se a opção for lançar mão de interrupções, o processo receptor não precisa ler constantemente o conteúdo de endereços de memória; quando da chegada de uma mensagem, será disparada uma interrupção, ante a qual o processo suspende temporariamente suas atividades e encaminha as ações necessárias ao seu tratamento.

Cada abordagem tem suas vantagens e desvantagens. O mecanismo de interrupções evita o desperdício de ciclos gastos na leitura constante de um endereço de memória; em contrapartida, exige a intervenção do sistema operacional — troca do *modo-usuário* para o *modo-kernel* — para o tratamento de interrupções, redundando em uma elevação demasiada da latência. A estratégia de *polling*, por sua vez, não necessita da interferência do sistema operacional, sendo totalmente implementável em nível de usuário. Desta maneira, comunicação e sinalização, ambas, são efetivadas em nível de usuário, assegurando as baixas latências que o hardware SCI pode proporcionar.

Com vistas à obtenção da menor latência possível, todos os protocolos de comunicação do DECK/SCI baseiam-se em *polling*. Ambientes como SCI-MPICH, ScaMPI e CML seguem também esta mesma técnica.

6.4.3 Gerenciamento de buffers para as mensagens

Conforme já mencionado, a comunicação através do acesso direto à memória remota exige, em primeiro lugar, que o receptor de mensagens crie e exporte aos demais *nodos* um segmento de memória. Então, os emissores copiam suas mensagens ao segmento exportado pelo receptor em buffers que as armazenam temporariamente, até que sejam definitivamente copiadas ao buffer do usuário, em memória local.

Neste contexto, uma decisão de projeto relevante diz respeito ao modo de gerenciar os buffers presentes nos segmentos de memória que receberão mensagens. Duas são as possibilidades: um buffer pode ser compartilhado por todos os emissores, ou então, cada receptor mantém um buffer diferente para cada emissor. A primeira hipótese requer a coordenação do acesso, potencialmente concorrente, de todos os emissores a um recurso compartilhado, o que implica na necessidade de se garantir a exclusão mútua. A implementação de um mecanismo de exclusão mútua sobre uma área remota de memória é sobremaneira custosa, em termos de latência, visto que se torna inevitável o emprego de leitura remota para a sincronização dos emissores. A tecnologia SCI define uma transação atômica do tipo *fetch & increment*, justamente para auxiliar o desenvolvimento de protocolos de exclusão mútua. A utilização desta transação — que, a rigor, é uma variação da leitura remota convencional —, obviamente, mostra-se proibitiva, por ser conflitante com a meta de favorecer a baixa latência de comunicação. Além disso, o aumento do número de processos comunicantes acarreta uma maior contenção para a obtenção dos *locks*, degradando ainda mais o desempenho.

A opção que se parece, pois, mais adequada, é reservar, em cada receptor, um buffer para cada possível emissor. Neste caso, os emissores sabem exatamente aonde

devem copiar suas mensagens, sendo este controle totalmente local, uma vez que o acesso a cada buffer é, ao natural, exclusivo. O DECK/SCI, com fulcro na consecução de baixa latência, adota esta abordagem.

6.4.4 Protocolos de comunicação do DECK/SCI

À semelhança de SCI-MPICH e ScaMPI, projetaram-se três protocolos para compor o núcleo de comunicação do DECK/SCI. De forma transparente, em função do tamanho da mensagem a ser remetida, o DECK/SCI opta por um dentre os três protocolos de comunicação implementados.

Um dos protocolos é especialmente otimizado com a incumbência de garantir latências extremamente baixas para mensagens pequenas. Por outro lado, mensagens cujo tamanho não é comportado por esta técnica de transmissão podem ser enviadas através de um protocolo de propósito geral, o qual limita a largura de banda a um patamar consideravelmente abaixo das potencialidades do hardware SCI. Então, para elevar a largura de banda de pico, propôs-se um terceiro protocolo, do tipo *zero-copy*, que, por assim o ser, permite alcançar um *throughput* muito próximo dos limites tecnológicos inerentes à arquitetura subjacente.

A despeito de suas diferenças, decorrentes da especialização de cada um, os três protocolos de comunicação do DECK/SCI compartilham algumas técnicas, a saber:

- recebimento de mensagens por *polling*;
- manutenção de distintos buffers em cada receptor, para cada emissor, excetuando o protocolo *zero-copy*, em que a mensagem é enviada diretamente ao buffer do usuário;
- comunicação baseada somente em escrita remota;
- emprego de instruções MMX para a comunicação.

A figura 6.3 ilustra a estrutura básica do DECK/SCI, que se utiliza da API SISCI para a manipulação de segmentos compartilhados, conforme mencionou-se na seção 6.3, e de instruções do tipo MMX para efetivar operações de comunicação. Frisando o que já foi dito, a intenção do DECK/SCI é seu uso para a programação de aplicações paralelas.

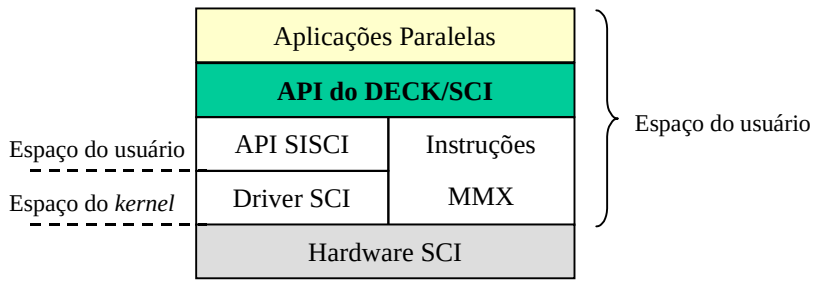

FIGURA 6.3 - Estrutura básica do DECK/SCI.

A seguir, detalham-se os protocolos de comunicação projetados e implementados. Uma minuciosa análise de desempenho acerca dos protocolos consta do próximo capítulo.

6.4.5 “Protocolo 1”: mensagens pequenas

O protocolo do DECK/SCI que dá conta da troca de mensagens pequenas será, doravante neste texto, chamado de “protocolo 1”, a fim de facilitar futuras alusões e comparações. Especial atenção foi prestada à transmissão de mensagens pequenas, por ser particularmente sensível a *overheads* adicionais, tais como a sinalização, que poderia promover um aumento demasiado da latência.

A característica que distingue o “protocolo 1” dos demais é a utilização de uma única transação SCI tanto para o envio da mensagem, como para sinalizar a chegada da mesma, de modo similar ao protocolo *Short* do SCI-MPICH. Em verdade, a técnica adotada pelo “protocolo 1” inspira-se no algoritmo apresentado por Omang [OMA 97], que introduz o conceito de *valid flag* associado ao último byte de um pacote SCI de 64 bytes.

O protocolo ora descrito aplica-se ao envio de mensagens com até 62 bytes. Os dados referentes à mensagem são necessariamente enviados em um pacote SCI de 64 bytes, cuja estrutura encontra-se ilustrada na figura 6.4.

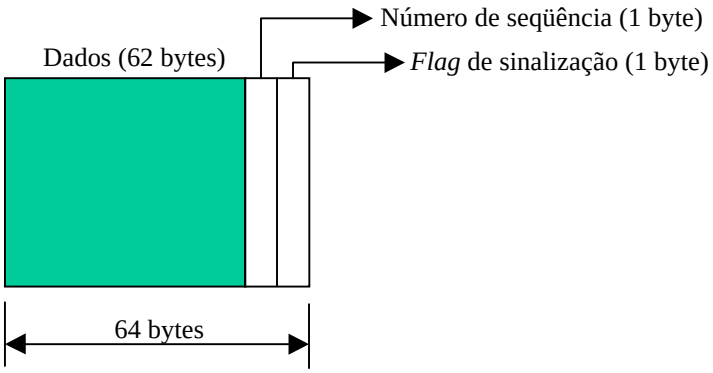

FIGURA 6.4 - Estrutura do pacote utilizado pelo “protocolo 1” do DECK/SCI.

Independentemente da quantidade de dados da mensagem, serão sempre enviados os 64 bytes do pacote, fazendo-se otimizado uso dos buffers de escrita da placa SCI — conforme mostrado anteriormente, a comunicação é mais rápida quando dados em quantidades múltiplas de 64 bytes são remetidos pela rede SCI. O último byte do pacote é usado para sinalizar o envio da mensagem nele contida, sem a necessidade de uma transação SCI adicional; destarte, é possível auferir baixas latências através do “protocolo 1”, sendo favorecida a troca de mensagens pequenas no DECK/SCI.

As principais estruturas de dados e o modelo encerrado pelo “protocolo 1” acham-se na figura 6.5, que ilustra a comunicação entre uma *thread*, presente no “*nodo* 1”, e uma *thread* sendo executada pelo “*nodo* 0”.

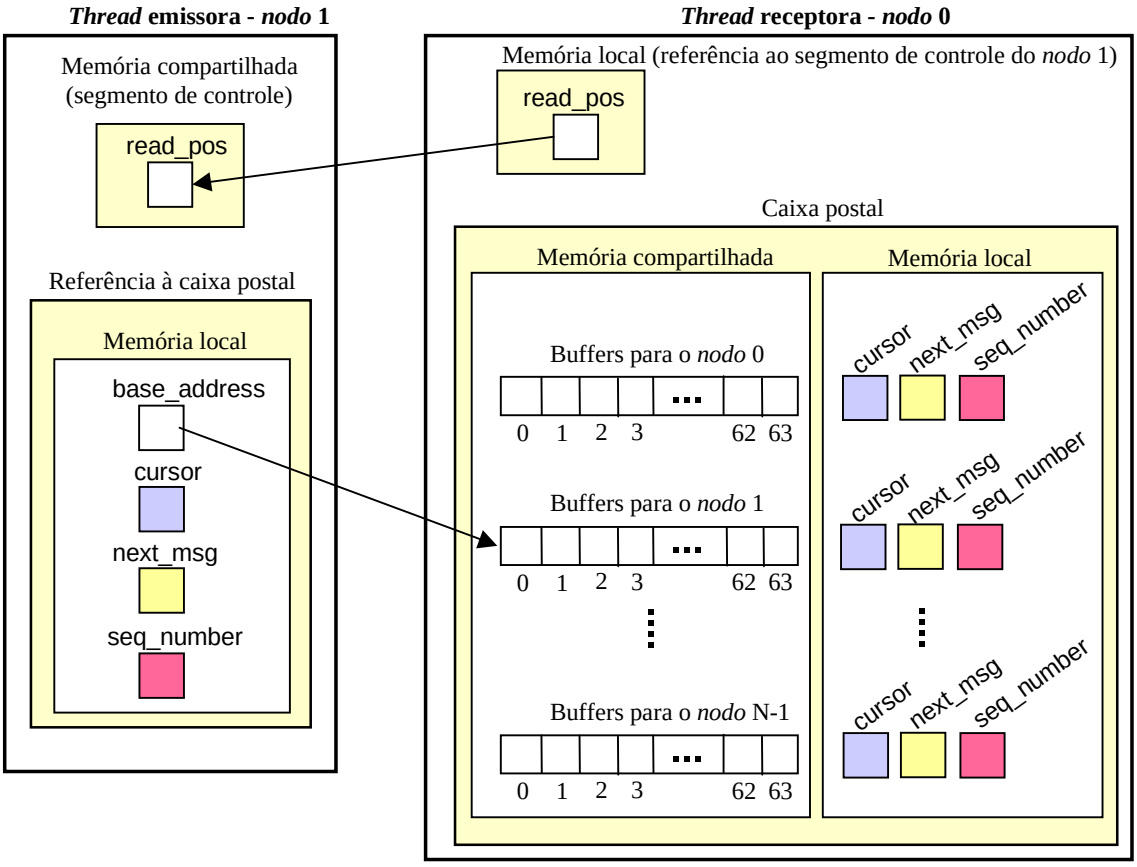

FIGURA 6.5 - Estruturas de dados utilizadas pelo “protocolo 1”.

Quando uma *thread* cria uma caixa postal (`deck_mbox_create`) para receber mensagens, um novo segmento compartilhado é alocado no *nodo* em que a referida *thread* encontra-se executando e, após, exportado aos demais *nodos* da aplicação. No segmento compartilhado associado à caixa postal, reserva-se, para cada *nodo*, um conjunto de 64 buffers de 64 bytes — que totalizam 4096 bytes — a serem usados para o armazenamento de mensagens transmitidas através do “protocolo 1”. Aliadas a cada conjunto de buffers estão três variáveis, mantidas na memória local, usadas pela *thread* mantenedora da caixa postal com o fito de controlar o recebimento de mensagens.

Ao invocar a primitiva `deck_mbox_clone` sobre uma caixa postal previamente criada, uma *thread* obtém acesso à caixa postal em questão, o que lhe permitirá enviar mensagens. Durante a operação, são instanciadas e inicializadas, na memória local, quatro variáveis, uma das quais é um ponteiro a que se atribui o endereço-base do conjunto de buffers reservado ao *nodo* da *thread* invocadora; as outras três são utilizadas no controle do envio de mensagens.

Na *thread* emissora, a variável `cursor` indica a posição (de 0 a 63), no conjunto de buffers reservado ao seu *nodo*, para a qual a próxima mensagem deve ser enviada. A variável `next_msg`, por sua vez, guarda o valor da *flag* de sinalização do pacote que carregará a próxima mensagem, e `seq_number` armazena o número de seqüência da próxima mensagem a ser enviada do *nodo* que contém a *thread* emissora à caixa postal destinatária, para fins de ordenação de mensagens. As variáveis correspondentes, na caixa postal da *thread* receptora, têm significado análogo: `cursor` representa a posição, no conjunto de buffers ao qual está relacionado, de onde deve ser extraída a próxima mensagem; `next_msg` armazena o próximo valor esperado no 64º byte da posição indicada por `cursor`; e, finalmente, `seq_number` guarda o número de seqüência associado à próxima mensagem a ser, de fato, copiada da caixa postal ao buffer do usuário em uma variável do tipo `deck_msg_t`.

A variável `read_pos`, presente no *nodo* da *thread* emissora, é atualizada pela *thread* receptora para informar a posição da qual a última mensagem foi extraída, representada pelo antigo valor de `cursor`, de modo a evitar que mensagens ainda não extraídas sejam sobrepostas pela emissora. Dito de outra forma, `read_pos` serve ao propósito de controle de fluxo, e está presente na *thread* emissora para evitar leituras remotas.

Quando da inicialização do ambiente de execução do DECK/SCI, cada *nodo* cria e exporta um segmento de controle, usado, dentre outras coisas, para o controle de fluxo. O endereço de `read_pos`, no segmento de controle do *nodo* da *thread* emissora, é calculado pela *thread* receptora em função da identificação interna — usada pelo DECK/SCI — da caixa postal, do identificador do *nodo* da *thread* receptora, e do número máximo de caixas postais que podem ser criadas por *nodo*, número este configurável pelo programador. Detalhes sobre a estrutura do segmento de controle serão mostrados oportunamente no texto.

Um resumo das ações básicas requeridas pelo “protocolo 1” encontra-se na figura 6.6. Obviamente, a implementação efetiva é muito mais complexa. A figura 6.6 traz uma notação compatível com os nomes de variáveis usados na figura 6.5, e visa a facilitar o entendimento do protocolo.

Analisando a figura 6.6, percebe-se que o protocolo apresenta apenas duas operações de comunicação: o envio propriamente dito da mensagem, em um pacote de 64 bytes conforme a estrutura mostrada na figura 6.4, e a informação passada da *thread* receptora para a emissora, sobre a última posição de leitura de mensagem, para fins de controle de fluxo. Cada conjunto de buffers é tratado como uma fila circular, conforme pode ser constatado, com os cursores variando entre 0 e 63. Note-se que os valores esperados pela *thread* receptora — *flag* e número de seqüência — são calculados da mesma forma por ambas *threads*, variando entre 0 e 255. Garante-se que uma mesma posição de um conjunto de buffers jamais receberá, consecutivamente, o mesmo valor para a *flag* — e para o número de seqüência —, o que elimina a necessidade de uma reinicialização dos últimos dois bytes de cada posição. Por exemplo, os valores tocantes à posição 0 alternarão entre 0, 64, 128 e 192, nesta ordem. Outrossim, não há qualquer risco da mensagem ser sinalizada antes do seu total recebimento, pois somente um pacote é transferido pela rede, impossibilitando a chegada dos dados fora de ordem; ao ser detectado o valor esperado no 64º byte, certamente todo o pacote já se encontrará armazenado no buffer da caixa postal.

***thread* emissora**

```
// Retém o fluxo de transmissão de
//mensagens, se necessário.
while (read_pos == cursor);

Monta o pacote com a mensagem;

//Envia a mensagem para a posição
//   apontada por cursor.
*(base_address + cursor*64)= mensagem;

//Atualiza variáveis de controle.
cursor = (cursor+1)%64;
next_msg = (next_msg+1)%256;
seq_number = (seq_number+1)%256;
```

***thread* receptora**

```
//Lê os dois últimos bytes da posição
//corrente de todos os conjuntos de buffers,
//até chegar uma mensagem.
while(1) {
  for(i=0;i<deck_numnodes();i++) {
    // Faz polling na flag e no
    //    número de seqüência.
    if ((*(buffers[i]+cursor[i]*64+63) ==
           next_msg[i] &&
        (*(buffers[i]+cursor[i]*64+62) ==
           seq_number[i]) {

      Copia a mensagem para o buffer
       do usuário;

      //Informa ao emissor o valor do cursor.
      *(read_pos[i]) = cursor[i];

      //Atualiza variáveis de controle.
      cursor[i] = (cursor[i]+1)%64;
      next_msg[i] = (next_msg[i]+1)%256;
      seq_number[i] = (seq_number[i]+1)%256;

      //Força o término da função.
      return;
    }
  }
}
```

FIGURA 6.6 - Pseudocódigo atinente ao “protocolo 1”.

Cumpre ressaltar uma situação que se pode evidenciar, durante a troca de mensagens no DECK/SCI. É possível que a *thread* receptora perceba a chegada de uma mensagem nova, em um dos conjuntos de buffers do “protocolo 1”, cujo número de seqüência não seja o esperado. Isto significa que o *nodo* originador da referida mensagem enviou, anteriormente, outra(s) mensagem(ns), através do “protocolo 2”; neste caso, o consumo da mensagem pequena deve ser protelado até que o número de seqüência nela contido seja igual ao valor esperado. Este esquema é necessário para garantir a entrega ordenada das mensagens enviadas por um *nodo* a uma caixa postal, já que o *polling* é feito, primeiramente, sobre os buffers do “protocolo 1” e, somente depois, sobre os buffers do “protocolo 2”, descrito a seguir. Caso não houvesse números de seqüência, mensagens enviadas pelo “protocolo 1” poderiam ser consumidas antes de mensagens enviadas pelo “protocolo 2”, a despeito da real ordem em que foram remetidas.

6.4.6 “Protocolo 2”: buffers espaçosos

O protocolo apresentado na seção anterior é especializado na troca de mensagens pequenas — até 62 bytes. Portanto, houve que se definir um protocolo de comunicação mais genérico, capaz de lidar com a transmissão e recepção de mensagens maiores.

O doravante denominado “protocolo 2” gerencia buffers de maior porte, necessários para conter mensagens mais vultosas, sendo suficientemente genérico para a troca de mensagens de tamanho arbitrário, limitado apenas pela capacidade de armazenamento dos buffers, a qual, diga-se de passagem, pode ser configurada pelo usuário. As principais estruturas de dados imprescindíveis ao funcionamento do “protocolo 2” estão ilustradas na figura 6.7.

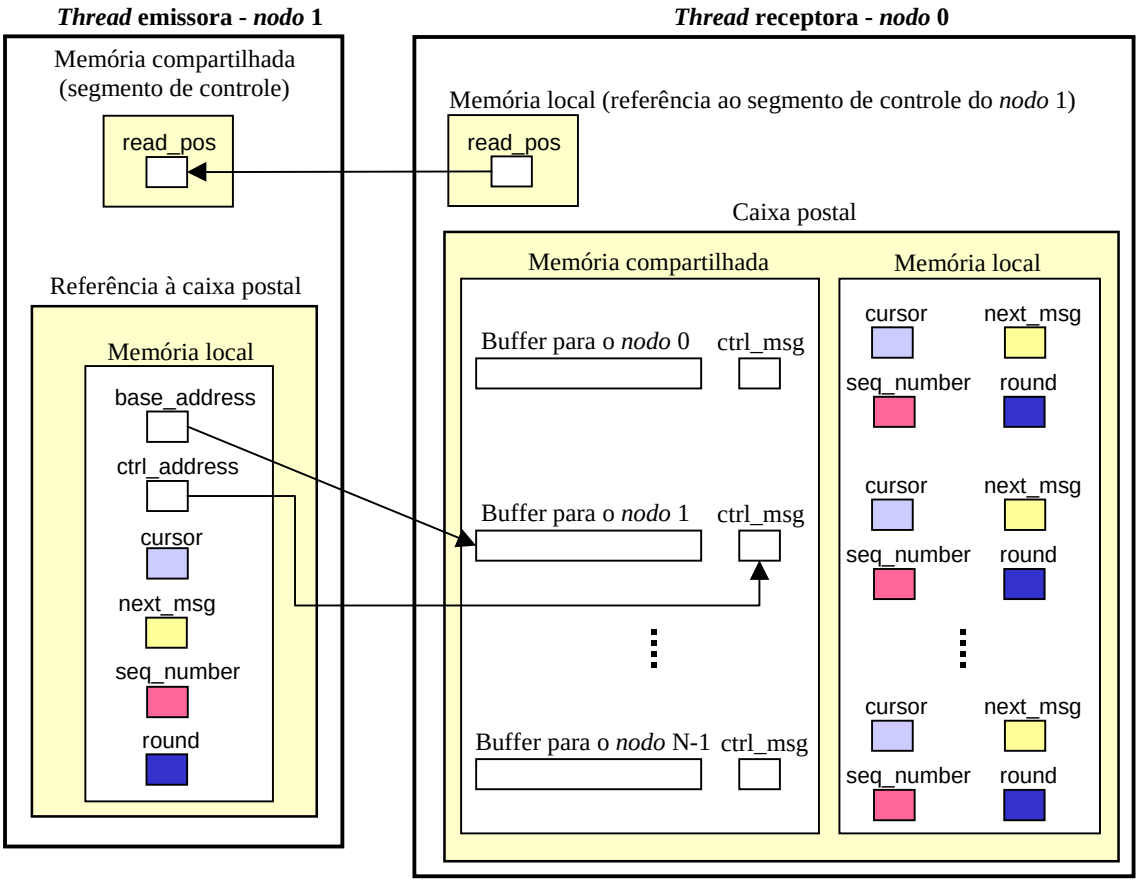

FIGURA 6.7 - Estruturas de dados subentendidas pelo “protocolo 2”.

As variáveis de nome `seq_number`, associadas a cada um dos buffers e presentes na referência à caixa postal, são exatamente as mesmas utilizadas no âmbito do “protocolo 1”, visto que têm a função de assegurar a ordenação das mensagens encaminhadas por um dado *nodo* a uma caixa postal, independentemente do protocolo utilizado. Por outro lado, as demais variáveis são específicas ao “protocolo 2”, e nada têm a ver com aquelas adotadas pelo “protocolo 1”, embora alguns nomes coincidam nas figuras 6.5 e 6.7. Escolheram-se nomes idênticos, neste texto, para facilitar o estabelecimento de analogias entre os protocolos, favorecendo a compreensão por parte do leitor.

Observe-se, na figura 6.7, que os buffers reservados a cada um dos *nodos* não são divididos em compartimentos, diferentemente dos buffers do “protocolo 1”; o “protocolo 2” preenche-os contiguamente. Perceba-se, também, que cada buffer vem

acompanhado, na região de memória compartilhada, por uma área de 64 bytes denominada `ctrl_msg`. É justamente nestes endereços que a *thread* receptora realiza *polling* para verificar a chegada de novas mensagens.

As mensagens manipuladas pelo "protocolo 2" contêm um cabeçalho composto por dois campos: `datalen`, que indica o espaço, em bytes, ocupado pelos dados da mensagem no corpo da mesma, calculado em função do tipo e da quantidade de elementos que foram inseridos na variável `deck_msg_t` que a representa; e `real_datalen`, que é o múltiplo de 64 bytes mais próximo — maior ou igual — de `datalen`, indicando o número de bytes transmitidos de fato pela rede SCI para enviar o corpo da mensagem, haja visto que o uso da rede é otimizado para quantidades de bytes múltiplas de 64. Embora tenha somente os dois campos mencionados, o cabeçalho da mensagem ocupa 64 bytes, também em consonância com a prática de otimizar as operações de comunicação. A estrutura das mensagens do "protocolo 2" é mostrada na figura 6.8.

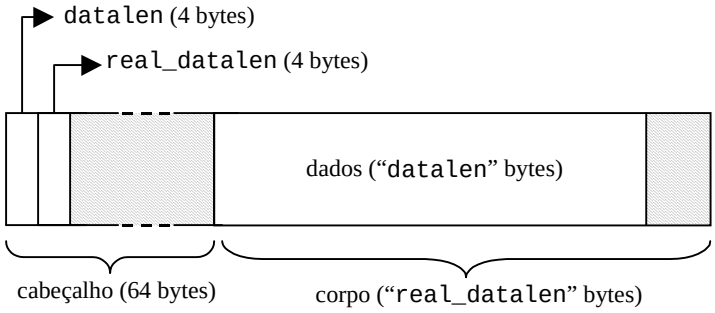

FIGURA 6.8 - Composição das mensagens usadas pelo "protocolo 2".

Para o envio de uma mensagem, primeiramente, a *thread* emissora constrói o cabeçalho e cria uma estrutura como a mostrada na figura 6.8. Após, toda a mensagem é enviada ao buffer relativo ao *nodo* da *thread* emissora — cujo endereço-base está armazenado em `base_address` — na posição dada por `base_address+cursor` (ver figura 6.7). Antes de sinalizar o envio da mensagem, é necessário esvaziar os buffers da placa SCI, pois pode haver transações pendentes. Após o esvaziamento dos buffers da placa, garante-se que todas as transações pendentes foram concluídas, ou seja, todos os pacotes SCI constituintes da mensagem estão devidamente armazenados no buffer da caixa postal. Em seguida, a *thread* emissora envia uma mensagem de controle, carregando os valores das variáveis `next_msg` e `round`, para o endereço `ctrl_msg`, apontado por `ctrl_address`, sinalizando que foi remetida uma mensagem identificada por este par de valores. Por fim, a *thread* emissora atualiza os valores de `cursor`, `next_msg` e `seq_number`. O próximo valor de `cursor` será o valor corrente acrescido de `real_datalen` + 64, que é exatamente o espaço total ocupado pelo cabeçalho e corpo da mensagem no buffer da caixa postal. A variável `next_msg` é atualizada conforme a expressão `next_msg` = (`next_msg`+1)%1.000.000, isto é, pode assumir valores situados entre 0 e 999.999, e `seq_number` é atualizado como no "protocolo 1".

A mensagem de controle é enviada em um único pacote SCI de 64 bytes, garantindo a transmissão atômica de `next_msg`, que ocupará os quatro primeiros bytes da área `ctrl_msg`, e `round`, que ficará armazenado no quinto byte de `ctrl_msg`. Isto se faz necessário porque SCI não garante a entrega ordenada de pacotes. Caso os valores fossem enviados em transações distintas, chegariam em tempos diferentes; pior ainda, os quatro bytes do valor `next_msg` poderiam ser divididos em duas porções de dois bytes, as quais seriam recebidas separadamente e em qualquer ordem.

Para o recebimento de uma mensagem, a *thread* receptora lê constantemente as áreas reservadas às mensagens de controle (`ctrl_msg`) de cada buffer, até que, em alguma delas, os valores `next_msg` (os quatro primeiros bytes) ou `round` (o quinto byte) sejam diferentes daqueles contidos nas variáveis locais `next_msg` e `round`, respectivamente. Esta diferença indica que há, no buffer da caixa postal, uma ou mais mensagens já enviadas e ainda não consumidas. A próxima mensagem a ser consumida pela *thread* receptora está armazenada a partir da posição indicada pela sua variável `cursor`; então, a *thread* lê o cabeçalho da mensagem, a fim de determinar seu tamanho, e a copia para o buffer do usuário na memória local. Após, a *thread* receptora envia uma mensagem (em um pacote SCI de 64 bytes) à emissora, contendo o valor de `cursor` e o valor de `round`, para efeito de controle de fluxo — isto está representado por `read_pos` na figura 6.7 —; em seguida, atualiza suas variáveis `cursor`, `next_msg` e `seq_number`.

Um aspecto interessante no “protocolo 2” é o tratamento dos buffers como regiões contiguamente endereçáveis, diferentemente dos buffers do “protocolo 1”, que são divididos em 64 posições. Em virtude desta característica, as mensagens precisam de um cabeçalho que informe seu tamanho, para que a *thread* receptora saiba exatamente quantos bytes precisa copiar para o buffer do usuário e, também, de quanto deve aumentar o valor da variável `cursor`.

A idéia de buffers contíguos exige do “protocolo 2” esquemas completamente diferentes do “protocolo 1”, para o recebimento de mensagens. Imagine-se uma situação em que a *thread* emissora envie, por exemplo, três mensagens, antes da *thread* receptora invocar a função de recebimento (`deck_mbox_retrv`). Neste caso, foram remetidas três mensagens de controle, uma para cada mensagem de dados; porém, como as mensagens de controle são submetidas sempre para o mesmo endereço, a *thread* receptora terá acesso somente à última e, mesmo assim, consumirá a primeira das três mensagens de dados enviadas. Primeiramente, a receptora percebe a chegada de uma nova mensagem porque o valor `next_msg`, referente à última mensagem de controle, difere do valor da sua variável local, e esta diferença é exatamente três. Dado que `cursor` aponta para a posição da próxima mensagem a ser consumida, será copiada ao buffer do usuário a primeira das três mensagens. Com o auxílio do cabeçalho da mensagem recém consumida, a *thread* receptora, antes de terminar a função `deck_mbox_retrv`, atualiza `cursor` de modo a apontar para a segunda mensagem que lhe foi enviada e, em seguida, atualiza sua variável local `next_msg`. Caso seja invocada mais uma vez a função `deck_mbox_retrv`, a *thread* receptora prontamente perceberá que há mensagens disponíveis, pois a mensagem de controle ainda tem um valor `next_msg` diferente do de sua variável local; desta feita, entretanto, a diferença é dois. Quando a diferença for zero, todas as mensagens terão sido consumidas.

Outro aspecto interessante do “protocolo 2” diz respeito à forma como o controle de fluxo é feito. Utiliza-se o conceito de *round*, associado ao buffer. Inicialmente, emissora e receptora estão no *round* 0. Quando a emissora não puder enviar uma mensagem ao buffer, por falta de espaço, isto é, quando `cursor+real_datalen+64` for maior do que o tamanho do buffer, é necessário reter o fluxo de transmissão de mensagens. A emissora, então, espera até que o `cursor` e o `round` da receptora sejam iguais ao seu — a receptora, a cada mensagem consumida, informa à emissora o último valor das variáveis `cursor` e `round`, conforme já mencionado. Quando os valores das variáveis coincidirem, a emissora atualiza sua variável `round`, empregando a expressão `round` = (`round`+1)%256. Marcando o início de um novo *round*, a emissora atribui zero à variável `cursor`, no intuito de enviar as mensagens subseqüentes para posições a partir do endereço-base do buffer; a variável `next_msg` é também colocada em zero. Pode-se recomeçar a escrever mensagens no início do buffer porque a receptora, certamente, consumiu as mensagens antigas, posto que seu `cursor` emparelhara com o `cursor` da emissora. Ao perceber a mudança de *round* na próxima mensagem de controle enviada pela emissora, a receptora passará a ler mensagens a partir do início do buffer, atualizando sua variável `round` e reinicializando `cursor` e `next_msg` com zero.

O “protocolo 2” do DECK/SCI é análogo ao protocolo *Eager* dos ambientes SCI-MPICH e ScaMPI. Diferentemente destes, no entanto, o DECK/SCI faz uso otimizado dos buffers, manipulando-os como regiões de memória compartilhada preenchidas contiguamente pelas mensagens. SCI-MPICH e ScaMPI dividem cada *eager buffer* em grandes compartimentos — tipicamente, há oito posições de 16 kbytes cada —, o que redunda em um desperdício de espaço. Mesmo que seja enviada uma mensagem de, por exemplo, 70 bytes, esta mensagem impedirá a utilização do espaço restante no compartimento para o qual foi copiada. Além do desperdício de espaço, outra conseqüência imediata deste esquema é a tendência em refrear o fluxo de transmissão de mensagens mais rapidamente do que o “protocolo 2” do DECK/SCI, que aproveita cada byte dos buffers.

6.4.7 “Protocolo 3”: *zero-copy*

Embora o “protocolo 2” possa ser adotado para a troca de mensagens de, virtualmente, qualquer tamanho, uma séria restrição ao desempenho é por ele imposta, conforme será mostrado no próximo capítulo. A grande desvantagem do “protocolo 2”, e também do protocolo *Eager* de SCI-MPICH e ScaMPI, é iniciar a cópia da mensagem para o buffer do usuário somente após seu recebimento completo. O emissor remete toda a mensagem de dados e, depois, envia uma mensagem de controle sinalizando a comunicação. Por fim, o receptor procede à copia dos dados para o buffer do usuário. Este esquema limita seriamente a máxima largura de banda alcançável.

Para elevar a largura de banda de pico, SCI-MPICH e ScaMPI propuseram um protocolo do tipo *Rendez-vous*, através do qual a cópia da mensagem para o buffer do usuário é feita durante a sua transmissão para os buffers de sistema, de forma intercalada (ver seção 4.4.5 e figura 4.6). O “protocolo 3” do DECK/SCI, no intuito de elevar ainda mais a largura de banda de pico, em vez de adotar a estratégia de SCI-MPICH e ScaMPI, oferece um mecanismo que evita a cópia extra da mensagem para o buffer do usuário; a única etapa requerida é a transmissão propriamente dita da

mensagem pela rede SCI, seguida pela sinalização. O desempenho resultante, como se constata no próximo capítulo, é nitidamente superior ao das outras bibliotecas de comunicação.

No caso dos protocolos 1 e 2, anteriormente descritos, cada mensagem é mantida em um buffer da caixa postal da *thread* receptora, até ser copiada para o buffer da variável de tipo `deck_msg_t` passada como argumento para a primitiva `deck_mbox_retrv`. Através do "protocolo 3", o emissor envia a mensagem diretamente para o buffer da variável do usuário, sem o armazenamento temporário nos buffers da caixa postal. Obviamente, este mecanismo é totalmente transparente ao programador DECK, que sempre lida com mensagens e caixas postais, e sequer precisa ter conhecimento acerca da diversidade de protocolos de comunicação oferecidos pelo DECK/SCI.

Internamente, estabelece-se o tamanho máximo para as mensagens manipuláveis pelo "protocolo 2", de modo que mensagens, cujo espaço ocupado pelos dados ultrapasse este limite, sejam enviadas e recebidas por meio do "protocolo 3". A análise de desempenho, constante do próximo capítulo, mostra qual é o limite ideal para a transição do "protocolo 2" ao *zero-copy*.

No momento em que uma mensagem DECK é criada (`deck_msg_create`) pelo programador, testa-se o argumento `size` passado como argumento, o qual indica o tamanho máximo da mensagem. Caso este valor seja menor ou igual ao limite estabelecido para a troca de protocolos — `DECK_MSG_BUF_LIMIT` —, o buffer da variável de tipo `deck_msg_t` retornada pela função `deck_msg_create` será alocado através da tradicional rotina `malloc`, e o "protocolo 2" será, certamente, usado pelas primitivas `deck_mbox_post` e `deck_mbox_retrv`. Contudo, se o valor do argumento `size` for superior ao máximo tamanho de mensagem manipulável pelo "protocolo 2", o buffer da variável `deck_msg_t`, retornada por `deck_msg_create`, não será alocado através da rotina `malloc`, mas sim, será a ele reservado um espaço no segmento de mensagens, que é um segmento compartilhado criado quando da inicialização do ambiente de execução. Neste caso, as primitivas de comunicação `deck_mbox_post` e `deck_mbox_retrv` utilizar-se-ão do "protocolo 3" se, e somente se, o espaço ocupado pelos dados inseridos na mensagem for maior do que o limite para a troca de protocolos; senão, o "protocolo 2" será usado.

A implementação de um mecanismo de *zero-copy* é possível devido à natureza da API do DECK, que exige a criação de mensagens (`deck_msg_t`), através de primitivas específicas, tanto para o envio quanto para o recebimento de dados. Assim, em se reservando uma área de um segmento compartilhado — o segmento de mensagens — para o buffer de uma mensagem, e graças ao mecanismo de acesso direto à memória remota, proporcionado pela tecnologia SCI, pode-se vislumbrar um protocolo de troca de mensagens por *zero-copy*.

Assim como os outros dois protocolos do DECK/SCI, o "protocolo 3" também é baseado em *polling*. Em cada caixa postal, além das estruturas de dados do "protocolo 1" e do "protocolo 2", exibidas, respectivamente, nas figuras 6.5 e 6.7, há áreas de *polling* específicas ao "protocolo 3", onde uma *thread* receptora pode perceber a chegada de mensagens por *zero-copy*. As estruturas de dados envolvidas no protocolo *zero-copy* encontram-se ilustradas na figura 6.9.

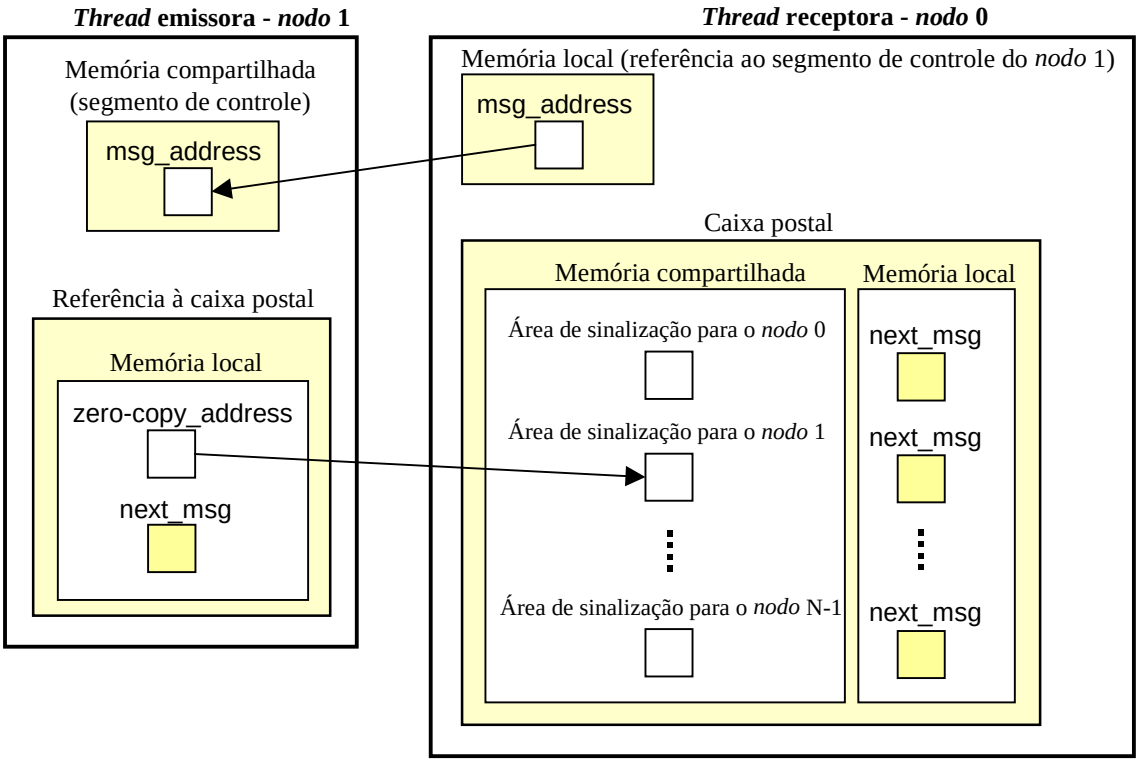

FIGURA 6.9 - Estruturas de dados envolvidas no funcionamento do “protocolo 3”.

Perceba-se que, na caixa postal, não há buffers para as mensagens de dados, mas somente áreas que recebem mensagens de controle enviadas pelas *threads* emissoras. Conquanto as mensagens de dados sejam, a rigor, enviadas diretamente para o buffer do usuário, em uma variável do tipo `deck_msg_t`, sob a ótica do programador DECK mensagens são sempre enviadas para caixas postais, e não para *nodos*; por esta razão, as estruturas de dados do protocolo *zero-copy* devem, necessariamente, estar abrigadas em caixas postais. Cumpre enfatizar, mais uma vez, que os programadores DECK não precisam estar cientes dos três protocolos do DECK/SCI; especificaram-se distintos protocolos com vistas à obtenção do melhor desempenho possível, sem prejuízo às abstrações da API.

O “protocolo 3”, obviamente, é síncrono, diferentemente dos protocolos 1 e 2, havendo o emprego de um esquema de *handshaking* entre emissora e receptora, visto que a *thread* emissora necessita ser informada do endereço do buffer da variável de tipo `deck_msg_t` a que se destina a mensagem.

Primeiramente, a *thread* emissora envia, para a área de sinalização reservada ao seu *nodo*, apontada por `zero-copy_address`, uma mensagem de controle de um byte, a qual carrega o valor armazenado na sua variável local `next_msg`. Então, a emissora espera até que sua parceira informe-lhe o endereço do buffer da variável de tipo `deck_msg_t` para o qual a mensagem deve ser remetida. A *thread* receptora, através de *polling*, percebe que, na referida área de sinalização, está um valor que coincide com o de sua variável local `next_msg` correlata, e o interpreta como uma requisição,

originada pela emissora, para o envio de uma mensagem de dados por *zero-copy*. Na seqüência, a receptora, atendendo à solicitação, envia, a uma posição específica do segmento de controle[*] do *nodo* da *thread* emissora, o endereço do buffer a que se destina a mensagem — isto está representado por `msg_address`, na figura 6.9 —, e põe-se a esperar pelo término da comunicação. De posse do endereço-destino, a emissora efetivamente remete a mensagem de dados, esvazia os buffers da placa SCI e, finalmente, sinaliza o término da comunicação, enviando, para a devida área de sinalização, desta feita, o valor -1. Ao perceber a chegada do valor -1, na área de sinalização tocante ao *nodo* de sua parceira, a *thread* receptora atualiza o valor de sua variável `next_msg`, empregando a fórmula `next_msg` = (`next_msg`+1)%127, para que possa atender a futuras requisições, e encerra a função `deck_mbox_retrv`. A emissora, desfechando a seqüência de ações do protocolo, atualiza o valor de sua variável `next_msg`, através da mesma expressão, a fim de que suas próximas solicitações tenham efeito.

Em virtude do sincronismo que lhe é inauferível, o "protocolo 3" prescinde da incorporação de mecanismos de controle de fluxo, ao contrário dos demais protocolos do DECK/SCI.

As etapas de comunicação do "protocolo 3" estão esquematizadas na figura 6.10.

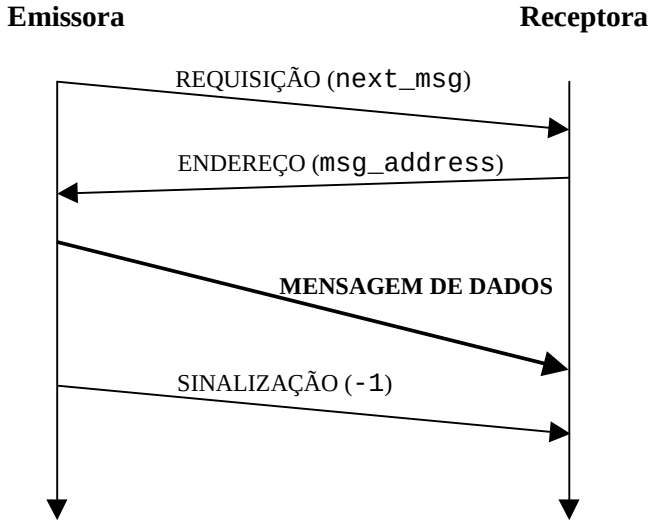

FIGURA 6.10 - Operações de comunicação necessárias ao "protocolo 3".

No caso dos outros dois protocolos do DECK/SCI, a *thread* emissora sabe, de antemão, para onde transmitir as mensagens, posto que a referência à caixa postal destinatária, obtida através da primitiva `deck_mbox_clone`, já contém o endereço-base

[*] Na inicialização do ambiente de execução do DECK/SCI, cada *nodo* cria um segmento compartilhado, que se denominou "segmento de controle", usado para várias funções, a saber: implementação da primitiva `deck_barrier`, armazenamento da tabela de caixas postais, controle de fluxo para os protocolos 1 e 2, e recebimento do endereço-destino de mensagens a serem enviadas por *zero-copy*. A estrutura do segmento de controle será exibida na seção 6.5.

do conjunto de buffers do “protocolo 1”, e do buffer utilizado pelo “protocolo 2”. Cabe à receptora copiar a mensagem da caixa postal para o buffer da variável de tipo `deck_msg_t` em questão.

6.5 Inicialização do ambiente de execução

A cooperação inicial entre todos os *nodos* para a instauração do ambiente de execução é condição *sine qua non* ao funcionamento de uma aplicação paralela no DECK/SCI. Tal é denotado, no programa do usuário, pela obrigatória chamada à primitiva `deck_init`, a ser efetuada por cada um dos *nodos* da aplicação. Dentre as diversas tarefas desempenhadas por `deck_init`, destaca-se o estabelecimento de dois segmentos compartilhados especiais: o segmento de controle e o segmento de mensagens.

Primeiramente, cada processo DECK, fazendo uso das primitivas da API SISCI, cria, no *nodo* em que se encontra, seu próprio segmento de controle, e o exporta aos demais; a seguir, cada processo conecta-se a todos os segmentos de controle remotos, e mapea-os ao seu espaço lógico de endereçamento. A mesma seqüência de ações é, logo após, realizada com relação aos segmentos de mensagens. Como resultado desta interação, cada processo DECK passa a ter acesso, a partir de seu espaço lógico de endereçamento, a todos os segmentos — de controle e de mensagens.

Os segmentos de controle são estruturados logicamente conforme o esquema gráfico da figura 6.11.

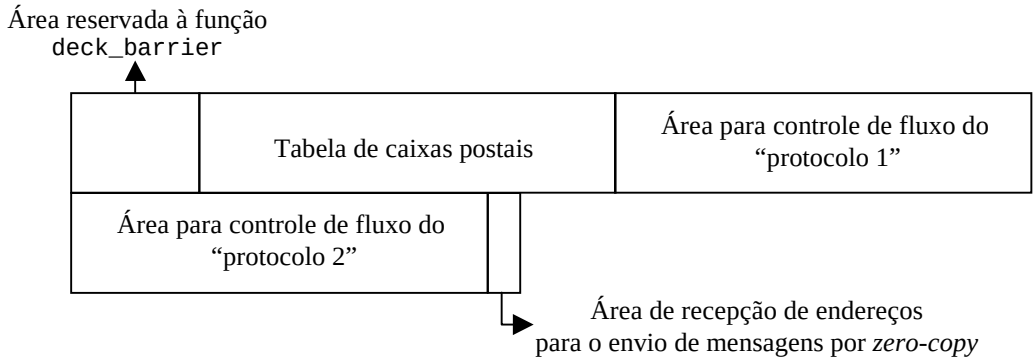

FIGURA 6.11 - Estrutura lógica do segmento de controle.

A primeira parte do segmento de controle é uma área, de 64 bytes, utilizada pela implementação da função `deck_barrier`, especificamente para as trocas de informações entre os processos DECK necessárias à geração de um ponto global de sincronização, o qual produz a abstração de barreira.

Adjacente à área reservada à função `deck_barrier`, acha-se a tabela de caixas postais. Todas as caixas postais criadas durante a execução da aplicação paralela, por todas as *threads*, de todos os *nodos*, são mantidas no segmento de controle de cada um

dos *nodos*, ou seja, existe uma tabela de caixas postais replicada em cada *nodo* da aplicação. O objetivo desta replicação não é tolerância a falhas, mas sim, garantir mais eficiência na execução da função `deck_mbox_clone`, responsável pela obtenção de uma referência a uma caixa postal previamente criada.

Ainda no segmento de controle, há duas áreas reservadas ao mecanismo de controle de fluxo, sendo uma para o "protocolo 1" e, outra, para o "protocolo 2". Um endereço específico, localizado no interior de cada uma destas áreas, está ilustrado nas figuras 6.5 e 6.7, denotado por `read_pos`. Neste endereço, a *thread* receptora informa, à emissora, sua atual posição de leitura no buffer da caixa postal reservado ao *nodo* da sua parceira. O cálculo do endereço `read_pos` leva em conta o identificador do *nodo* da *thread* receptora, e a identificação interna da caixa postal usada, identificação esta que nada tem a ver com o nome que o programador lhe atribuiu, sendo conhecida apenas pelo DECK/SCI. Garante-se que o referido endereço, calculado pela receptora para a caixa postal em uso, seja único, o que é imprescindível ao correto funcionamento do controle de fluxo de mensagens enviadas a cada caixa postal da aplicação. Visto que, no DECK, mensagens são remetidas, necessariamente, a caixas postais, e que as *threads* podem criar quantas caixas postais desejarem, o controle de fluxo tem de ser feito no âmbito da caixa postal, e não do *nodo* como um todo.

A última porção do segmento de controle é uma área, de quatro bytes, utilizada durante o funcionamento do "protocolo 3". É nesta região, representada na figura 6.9 por `msg_address`, que a *thread* emissora recebe o endereço para o qual deve enviar determinada mensagem por *zero-copy*.

Em suma, os segmentos de controle exercem um papel imprescindível no ambiente de execução do DECK/SCI, qual seja, promover áreas de memória compartilhada para a comunicação entre todos os processos DECK, de modo que as funções mais elementares, inerentes às primitivas da API, possam ser levadas a efeito.

As cinco regiões discriminadas na figura 6.11 estão alinhadas, na memória compartilhada, em 64 bytes. Destarte, assegura-se o uso otimizado dos buffers da placa SCI, através do pleno aproveitamento da comunicação à maneira *pipeline* proporcionada pelo hardware, sempre que escritas remotas atuarem em qualquer porção dos segmentos de controle.

Além do segmento de controle, cada *nodo* mantém outro segmento compartilhado especial: o segmento de mensagens. Este segmento é utilizado, pela função `deck_msg_create`, na criação de uma mensagem — variável do tipo `deck_msg_t` — cujo espaço a ser-lhe reservado aos dados ultrapassa o limite estabelecido — `DECK_MSG_BUF_LIMIT` — para a transição do "protocolo 2" ao *zero-copy*. Neste caso, será designado, ao buffer da mensagem em criação, uma porção do segmento de mensagens, de modo a possibilitar o uso do "protocolo 3" para a recepção de dados. Posto que o buffer da mensagem esteja em uma porção de um segmento compartilhado, e não na memória local, o mecanismo de *zero-copy* pode ser empregado, se necessário.

Outrossim, a função `deck_init` tem por incumbência inicializar as demais estruturas de dados usadas internamente pelo DECK/SCI, e alinhar, em 64 bytes, todos

os ponteiros utilizados nas operações de escrita remota, em busca do melhor desempenho possível.

6.6 Manipulação de mensagens

Insta-se o leitor a consultar a seção 5.2.3, a fim de que possa melhor acompanhar os parágrafos seguintes, que versam sobre os mais relevantes detalhes de implementação acerca das primitivas presentes na API do DECK com o fito de manipular mensagens.

No DECK, mensagem é uma abstração vital a toda e qualquer operação de comunicação, tanto quanto o é a caixa postal. Faz-se imprescindível, por parte do programador, a criação de variáveis do tipo mensagem — `deck_msg_t` — para o envio e recebimento de dados, durante a execução de uma aplicação paralela.

Conforme já comentado, a função `deck_msg_create` atua de duas formas distintas, dependendo do espaço a ser reservado aos dados da variável-mensagem em criação. Se o espaço reservado aos dados — o qual é informado pelo programador — for menor ou igual à constante `DECK_MSG_BUF_LIMIT`, o buffer da mensagem será alocado em memória local, pois tal mensagem jamais será usada pelo protocolo *zero-copy*. Entretanto, se o espaço reservado aos dados superar o limite estabelecido, o buffer da mensagem será associado a uma porção do segmento de mensagens, posto que a mesma poderá vir ser usada pelo protocolo *zero-copy*, contanto que o espaço efetivamente ocupado pelos dados nela inseridos ultrapasse o limite de transição de protocolos.

A cada porção utilizada no segmento de mensagens, incrementa-se uma variável indicadora da próxima posição disponível. Caso o espaço a ser reservado aos dados da mensagem, somado ao valor da variável indicadora, seja maior do que o tamanho do segmento de mensagens, a função `deck_msg_create` retornará um erro de falta de recursos.

Qualquer que seja a localização física do buffer da mensagem — memória local ou segmento de mensagens —, sempre é respeitado seu alinhamento em 64 bytes, pelas razões anteriormente expostas neste texto. Ademais, o espaço alocado, de fato, para a mensagem, não é exatamente aquele passado como argumento à função `deck_msg_create`. Toma-se o próximo múltiplo de 64, em relação ao valor informado, no intuito de permitir a transmissão de quantidades múltiplas de 64 bytes. Depois, somam-se 64, de modo a reservar espaço ao cabeçalho utilizado em conjunto com o “protocolo 2”, quando for o caso (ver figura 6.8). Por fim, mais 64 são somados, incluindo espaço para que se possa proceder ao alinhamento do buffer.

Uma vez que os primeiros 64 bytes do buffer da mensagem tenham sido designados, exclusivamente, ao cabeçalho atinente ao “protocolo 2”, os dados serão sempre inseridos, através da primitiva `deck_msg_pack` — ou copiados, quando do seu recebimento, através de `deck_mbox_retrv` —, a partir do 65º byte do buffer.

Três atributos importantes de uma variável-mensagem são `cursor`, `datalen` e `real_datalen`. Toda vez em que a função `deck_msg_pack` é invocada, insere-se, no

buffer da mensagem passada como argumento, uma determinada quantidade de elementos, de um tipo; por conseguinte, o ponteiro `cursor` é avançado, de modo a indicar a próxima posição a partir da qual novos dados serão incluídos. Além do avanço do `cursor`, incrementa-se o atributo `datalen`, que contém o espaço efetivamente ocupado pelos dados inseridos na mensagem. Como somente quantidades múltiplas de 64 bytes trafegam pela rede, através dos protocolos do DECK/SCI, o atributo `real_datalen` mantém armazenado o múltiplo de 64 seguinte a `datalen`. Em se tratando do "protocolo 1", `real_datalen` será, obrigatoriamente, 64.

A execução da função `deck_msg_clear`, concebida para possibilitar a sobreposição dos dados já presentes em uma mensagem, redunda na reinicialização dos seus atributos com os valores originais: `cursor` passa a apontar para o 65º byte do buffer da mensagem; `datalen` é colocado em zero; e, `real_datalen`, em 64. Por outro lado, a primitiva `deck_msg_reset`, simplesmente, reinicializa o `cursor`, visto que sua incumbência é, apenas, permitir que os dados de uma variável-mensagem possam ser extraídos, mais de uma vez, com a função `deck_msg_unpack`, a qual não faz uso de `datalen` e `real_datalen`.

A primitiva `deck_msg_getbuffer`, que retorna o endereço do buffer de uma variável-mensagem, confere mais liberdade ao programador, dando ensejo à manipulação direta dos dados da mensagem, sem a necessidade de inseri-los por meio de `deck_msg_pack`, ou retirá-los através de `deck_msg_unpack`. Porém, a utilização explícita do buffer da mensagem impede a atualização dos atributos `cursor`, `datalen` e `real_datalen` à medida que o buffer é preenchido. A função `deck_msg_getbuffer` atribui a `datalen` o valor passado como argumento para `deck_msg_create`, ou seja, o espaço total reservado ao buffer da mensagem, e a `real_datalen`, o próximo múltiplo de 64, e estes valores não mais serão alterados, a menos que se lance mão da função `deck_msg_pack` para o preenchimento do buffer.

Destarte, quando da manipulação explícita do buffer da mensagem, recomenda-se ao programador reservar, na criação da mesma, exatamente o espaço que seus dados irão ocupar, no intuito de otimizar as operações de comunicação, haja visto que o fator determinante na escolha do protocolo a ser usado é o atributo `datalen`. Por exemplo, se o programador reservar 200 bytes a uma mensagem, e preencher diretamente seu buffer com somente 60 bytes úteis, o DECK/SCI, a despeito do conteúdo da mensagem, transmitirá, pelo "protocolo 2", 320 bytes, sendo 256 (que é, em relação a 200, o próximo múltiplo de 64) para o corpo, mais 64 de cabeçalho, embora o ideal seja transmitir somente 64 bytes, através do protocolo de baixa latência — o "protocolo 1".

A última primitiva de gerenciamento de mensagens, `deck_msg_destroy`, dá conta da liberação da área de memória designada ao buffer, invalidando a variável-mensagem que lhe é passada como argumento.

6.7 Gerenciamento de caixas postais

Neste momento, visando a favorecer a plena assimilação das minúcias de que tratam os próximos parágrafos, convida-se o leitor a reportar-se à seção 5.2.4, que expôs

a noção de caixa postal, do ponto de vista do programador DECK, e a semântica das primitivas afins. Pode-se aquilatar que a abstração de caixa postal, idealizada com vistas à formulação de mecanismos de comunicação entre *threads* pertencentes a distintos *nodos*, indubitavelmente é o elemento central para a modelagem de aplicações paralelas no DECK.

6.7.1 Arquitetura das caixas postais

Conforme comentários anteriores deixaram antever, o ato de criar uma caixa postal tem dois efeitos: criação de um segmento compartilhado, o qual abriga os buffers utilizados pelos protocolos de comunicação; e inicialização de estruturas de dados, presentes na memória local, responsáveis por controlar o tráfego de mensagens para os buffers. As figuras 6.5, 6.7 e 6.9 ilustram, em separado, os componentes de uma caixa postal utilizados exclusivamente por cada protocolo de comunicação; a figura 6.12, por sua vez, exibe a estrutura lógica completa.

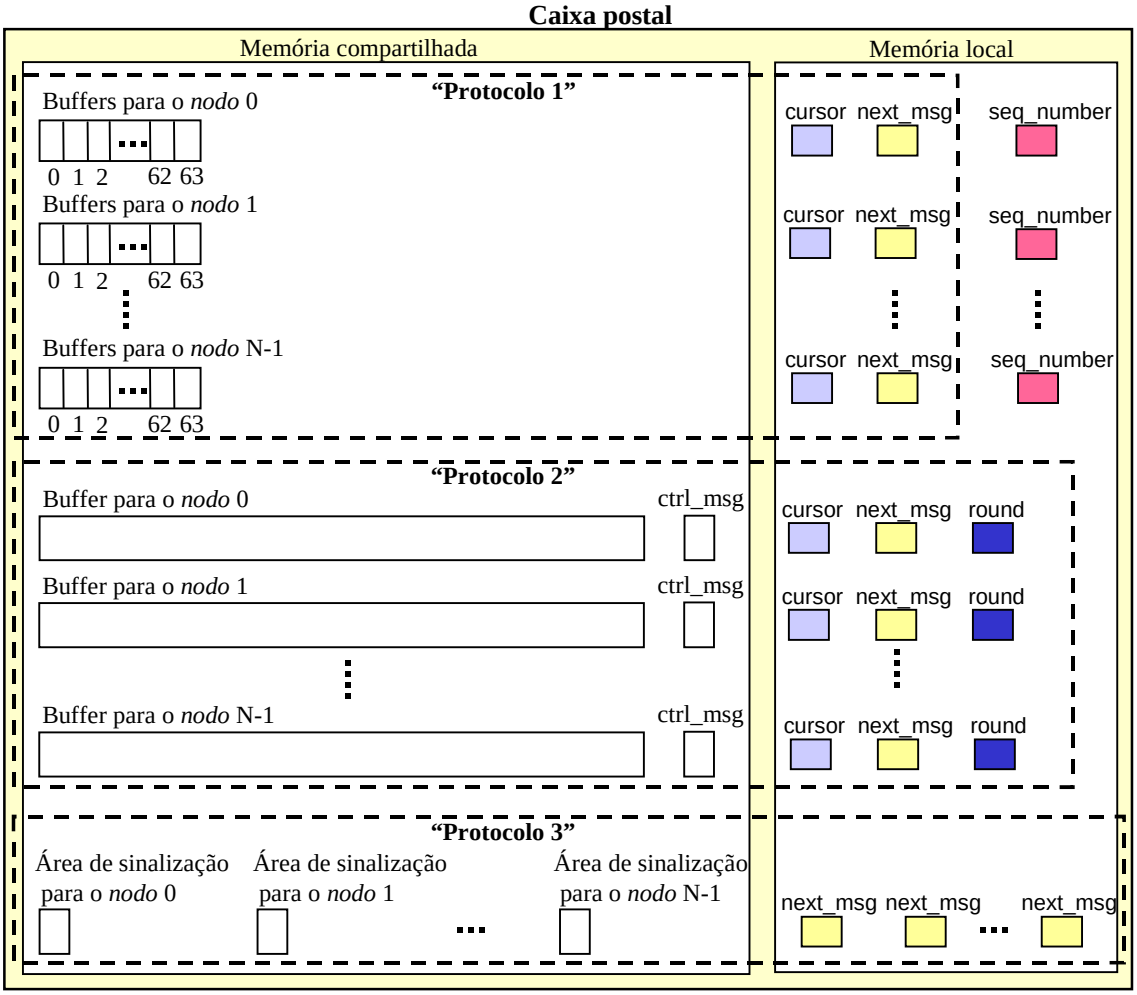

FIGURA 6.12 - Componentes de uma caixa postal.

Os conjuntos de buffers pertinentes ao “protocolo 1”, como já mencionado, apresentam 64 posições, de 64 bytes cada, totalizando 4096 bytes, ao passo que o tamanho dos buffers do “protocolo 2” é dado pelo valor `DECK_MEDBUF_SIZE`, configurável pelo usuário. É mister a observância de que `DECK_MEDBUF_SIZE` deve ser, no mínimo, suficiente para que os buffers do “protocolo 2” possam comportar a mensagem de maior tamanho por ele manipulável. Posto que `DECK_MSG_BUF_LIMIT` representa o limite após o qual o protocolo *zero-copy* passa a substituir o “protocolo 2”, e que o “protocolo 2” transmite um cabeçalho de 64 bytes, antecedendo ao corpo de cada mensagem, pode-se concluir que `DECK_MEDBUF_SIZE` tem de ser maior ou igual a `DECK_MSG_BUF_LIMIT`+64.

A área `ctrl_msg`, associada a cada buffer do “protocolo 2”, possui 64 bytes, necessários para receber atomicamente, em uma única transação SCI, os valores de `next_msg` (4 bytes) e `round` (1 byte), oriundos de alguma *thread* emissora, conforme descrição do referido protocolo, constante da seção 6.4.6. No que tange ao “protocolo 3”, cada área de sinalização ocupa somente 1 byte; em verdade, estas áreas estão agrupadas em um bloco contíguo de 64 bytes.

Como era de se esperar, garante-se que os endereços aos quais as *threads* emissoras referem-se, no segmento compartilhado de uma caixa postal, estejam alinhados em 64 bytes, exceto aqueles das áreas de sinalização do “protocolo 3”. Estas últimas não precisam estar alinhadas em 64 bytes porque a escrita remota de um único byte é extremamente rápida, apresentando uma latência de 2,5 μs, de modo que o fato de não estarem alinhadas não compromete o desempenho do protocolo.

Nitidamente, pode-se constatar que o espaço de memória ocupado por uma caixa postal depende do número de *nodos* que estão executando a aplicação paralela. Considerando-se a memória compartilhada, calcula-se o espaço demandado por uma caixa postal, em bytes, através da expressão N×(4096 + `DECK_MEDBUF_SIZE` + 64) + 64, onde “N” é o número de *nodos*. Todavia, este valor é automaticamente ajustado para o próximo múltiplo do tamanho de página usado pela arquitetura subjacente; no caso de Pentium rodando Linux, o tamanho de página é 4096 bytes.

Em uma plausível configuração dos protocolos de comunicação, poder-se-ia atribuir 8192 a `DECK_MSG_BUF_LIMIT`, e 24768 a `DECK_MEDBUF_SIZE`. Em tal situação, supondo-se quatro *nodos* executando a aplicação paralela, uma caixa postal teria para si reservados 118784 bytes de memória compartilhada, já devidamente ajustados aos 4096 bytes das páginas de memória. A configuração em questão postula que mensagens, cuja totalidade dos dados perfaçam mais de 8192 bytes, sejam transmitidas através do protocolo *zero-copy*, o qual utiliza, em vez dos buffers da caixa postal, os segmentos de mensagens estabelecidos por cada *nodo* durante a inicialização do ambiente de execução.

Convém aqui frisar que o máximo tamanho de mensagem manipulável pelo DECK/SCI é limitado pelo espaço dos segmentos de mensagens, configurável pelo usuário e denotado por `DECK_MSEG_SIZE`. Se, por exemplo, cada segmento ocupar 10 Mbytes, este será o maior tamanho possível de uma mensagem. Observe-se, entretanto, que o espaço total de um segmento é dividido entre as mensagens, criadas no *nodo* que o abriga, contanto que se lhes tenha reservado ao buffer uma quantidade superior a `DECK_MSG_BUF_LIMIT`.

6.7.2 Ordenação de mensagens

O DECK/SCI resolve a questão de ordenação de mensagens em dois níveis: primordialmente, trata-se do problema da chegada fora de ordem dos pacotes SCI componentes de uma mesma mensagem; em outro momento, leva-se em consideração a ordenação de diferentes mensagens, no nível das caixas postais.

Conquanto a rede SCI disponha de meios para assegurar a entrega confiável de pacotes, o padrão nada define acerca da ordem de entrega dos mesmos. O mecanismo de confirmação, baseado em pacotes do tipo *echo* (ver seção 2.4.2 e figura 2.3), e os buffers de escrita da placa SCI contribuem para que os pacotes constituintes de uma mensagem possam, não raro, chegar ao destino em ordem desconhecida. Os três protocolos de comunicação do DECK/SCI foram projetados de forma a contornar este inconveniente. O "protocolo 1" transmite cada mensagem em um único pacote SCI, de 64 bytes; os protocolos 2 e 3, por sua vez, aguardam pelo término de todas as transações SCI pendentes, antes de procederem à sinalização do envio de cada mensagem.

Estas técnicas, porém, não são suficientes para garantir que mensagens remetidas por um mesmo *nodo*, a uma mesma caixa postal, sejam recebidas pela *thread* receptora na exata ordem em que foram enviadas. É exatamente neste contexto que se inserem as variáveis de nome `seq_number`, mantidas nas caixas postais e associadas a cada *nodo* da aplicação (ver figura 6.12).

O âmago da questão de ordenação de mensagens está na estratégia de *polling* implementada pela primitiva `deck_mbox_retrv`. Quando invoca a função `deck_mbox_retrv`, a fim de obter a próxima mensagem disponível na caixa postal passada como argumento, a *thread* receptora põe-se a ler constantemente os buffers reservados a cada um dos *nodos*, começando pelos buffers reservados ao "*nodo* 0". Caso, após verificar se foram enviadas mensagens pelo "*nodo* 0" através dos protocolos 1, 2 e 3, nesta ordem, constate-se que nenhuma comunicação fora efetuada, a *thread* passa a averiguar os buffers do "*nodo* 1", e assim, sucessivamente. Se, depois de ler todos os buffers, de todos os *nodos*, nenhuma mensagem tiver sido sinalizada, recomeça-se o procedimento novamente, a partir do "*nodo* 0". Em suma, a tendência natural deste esquema de *polling* é que a *thread* perceba, para cada *nodo*, primeiramente as mensagens enviadas pelo "protocolo 1", depois, aquelas remetidas pelo "protocolo 2" e, finalmente, as requisições de envio de mensagem por *zero-copy*.

Se nenhum cuidado especial fosse tomado, o seguinte cenário poderia ser suscitado: determinada *thread* envia, à mesma caixa postal, uma mensagem pelo "protocolo 2" e, em seguida, outra, pelo "protocolo 1", mas a receptora recebe-as na ordem inversa, segundo a estratégia de *polling* acima descrita. A solução para este problema foi a utilização de números de seqüência nas caixas postais, representados pelas variáveis `seq_number` na figura 6.12.

O *nodo* da *thread* emissora armazena, na referência à caixa postal destinatária, um número de seqüência, que é incrementado a cada mensagem enviada (ver figura 6.14). Analogamente, a receptora mantém na sua caixa postal um número de seqüência para cada *nodo*, indicando a próxima mensagem que deve ser, de fato,

retirada. O número de seqüência da emissora é enviado juntamente com as mensagens do “protocolo 1”, conforme mostrou-se na figura 6.4, por ser o protocolo cujos buffers são primeiramente examinados. Caso o valor trazido por uma mensagem enviada pelo “protocolo 1” não seja aquele esperado pela receptora, ignora-se a mensagem e verifica-se o buffer do “protocolo 2”. Se houver uma nova mensagem no buffer do “protocolo 2”, certamente esta deve ser retirada da caixa postal; senão, lê-se a área de sinalização referente ao “protocolo 3” e, em havendo requisição de envio de mensagem por *zero-copy*, deve-se atender-lhe. Dito de outra forma, somente será retirada uma mensagem do buffer relativo ao “protocolo 1” quando seu número de seqüência coincidir com o valor esperado pela receptora.

A cada mensagem extraída da caixa postal, independentemente do protocolo usado para seu envio, a receptora incrementa a variável `seq_number`, possibilitando o recebimento de novas mensagens. Consoante se mencionou anteriormente, nas seções que descreveram os três protocolos de comunicação, emissora e receptora atualizam seus números de seqüência com a expressão `seq_number=(seq_number+1)%256`.

A ordenação de mensagens enviadas através do mesmo protocolo é decorrência imediata, visto que os buffers são manipulados como filas circulares e, em se tratando do “protocolo 3”, não há qualquer problema, dada a sua natureza síncrona.

Nas ocasiões em que mensagens são originadas por distintos *nodos*, com destino a uma mesma caixa postal, nada se pode afirmar a respeito da ordem de recebimento. O que o DECK/SCI faz, para evitar o favorecimento de algum *nodo*, é armazenar o identificador do *nodo* originador da última mensagem retirada da caixa postal. Assim, quando a rotina `deck_mbox_retrv` for novamente invocada, o *polling* começará a partir dos buffers do *nodo* com identificador subseqüente ao daquele contemplado, e não a partir dos buffers do “*nodo* 0”.

6.7.3 Identificação de caixas postais

No âmbito de uma aplicação paralela, uma caixa postal é identificada, em ocasião de sua criação, pelo nome que lhe atribuiu o programador. No entanto, o DECK/SCI, internamente, identifica cada caixa postal com um par de valores inteiros, único em todo o *cluster*, composto pelo identificador do *nodo* da *thread* criadora, e por um identificador de caixa postal, local ao *nodo* em que foi criada. Este identificador local de caixa postal é seqüencialmente atribuído, à medida que se criam novas caixas postais no *nodo* em questão. A título de exemplo, considere-se uma aplicação paralela sendo executada por três *nodos*, numerados pelo DECK com 0, 1 e 2. Suponha-se que tenham sido criadas duas caixas postais no “*nodo* 0”, uma no “*nodo* 1”, e duas no “*nodo* 2”. Os identificadores das caixas postais, utilizados pelo DECK/SCI, serão, respectivamente, (0,0), (0,1), (1,0), (2,0) e (2,1).

Para a criação de um segmento compartilhado com a API SISCI, deve-se atribuir-lhe um identificador, cuja abrangência é local ao *nodo* em que tal segmento está sendo criado. Quando da criação de uma caixa postal, a primeira ação executada pela função `deck_mbox_create` é justamente estabelecer um novo segmento compartilhado, necessário para conter os buffers pressupostos pelos protocolos de

comunicação. O identificador atribuído ao segmento compartilhado pertencente à caixa postal será o segundo elemento do par que a identifica globalmente.

Após a criação do segmento compartilhado e da inicialização de todas as estruturas de dados da caixa postal, a função `deck_mbox_create` inclui uma entrada na tabela de caixas postais mantida por cada um dos *nodos* da aplicação, através de um *broadcast* . Lembre-se de que existe uma tabela de caixa postais replicada no segmento de controle de cada *nodo* (ver seção 6.5 e figura 6.11).

O tamanho da tabela de caixas postais depende do número de *nodos* da aplicação e da quantidade máxima de caixas postais que podem ser criadas por *nodo*, denotada por `DECK_MAX_MBOX` e configurável pelo usuário. Reservam-se 64 bytes a cada entrada da tabela, a fim de garantir o alinhamento que tem sido preconizado ao longo do texto. Destarte, o tamanho da tabela de caixas postais, em bytes, é dado pela expressão $N \times$ `DECK_MAX_MBOX` $\times 64$, onde "N" é o número de *nodos* que estão executando a aplicação paralela. Cada entrada da tabela possui três campos, quais sejam, o nome da caixa postal, atribuído pelo programador, e os dois valores do par utilizado pelo DECK/SCI para identificá-la.

A estrutura interna da tabela de caixas postais permite a cada *thread* calcular de forma inequívoca o endereço, relativo ao endereço-base da tabela, no qual deve armazenar a entrada referente à caixa postal recém criada. A figura 6.13 ilustra a divisão lógica da tabela de caixas postais.

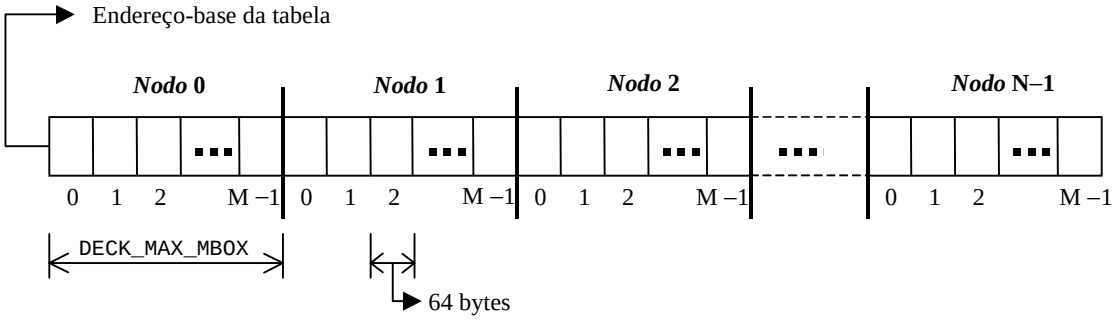

FIGURA 6.13 - Estrutura lógica da tabela de caixas postais.

Desde a inicialização do ambiente de execução, a porção do segmento de controle concernente à tabela de caixas postais tem espaço reservado suficiente para armazenar informações sobre todas as caixas postais que podem ser criadas em uma aplicação paralela, considerando o número total de *nodos*, denotado na figura 6.13 por "N", e o número máximo de caixas postais por *nodo*, representado por "M" ou `DECK_MAX_MBOX`, na figura. Cada processo DECK obtém, na inicialização dos segmentos de controle, o endereço-base da tabela local de caixas postais, e os endereços-base das tabelas dos demais *nodos*, a fim de que possa incluir novas entradas em todas as tabelas, por *broadcast*. Para tanto, a *thread* criadora de uma caixa postal calcula, em relação ao endereço-base, o endereço em que deve ser armazenada a nova entrada na tabela de caixas postais, empregando a fórmula (*local_id* + *nodo* $\times$ M) $\times$ 64,

onde “local_id” é o identificador local de caixa postal, no âmbito de um *nodo*, i.e., é o segundo elemento do par que identifica a caixa postal; “nodo” é o identificador do *nodo* da *thread* criadora; e “M” é o número máximo de caixas postais que podem ser criadas por *nodo* da aplicação. Note-se que “local_id” varia entre 0 e M−1, e “nodo” varia entre 0 e N−1, onde “N” é o número total de *nodos*.

A estrutura lógica da tabela deve ser interpretada da seguinte forma: as primeiras “M” posições, a partir do endereço-base, são reservadas a caixas postais criadas por *threads* do “*nodo* 0”; as “M” posições subseqüentes são destinadas a caixas postais pertencentes a *threads* do “*nodo* 1”; e assim, sucessivamente. A função `deck_mbox_create` encerra após efetivar a inserção de uma entrada, na devida posição, em todas as tabelas de caixas postais.

De acordo com o modelo de programação sugerido pelo DECK, se uma *thread* desejar enviar mensagens a uma caixa postal, precisará obter uma “referência” à caixa postal destinatária, o que se consegue por intermédio da primitiva `deck_mbox_clone`. Os argumentos de `deck_mbox_clone` são dois: o nome da caixa postal destinatária, e o endereço de uma variável do tipo `deck_mbox_t`, para a qual a referência à caixa postal é retornada.

A primeira ação de `deck_mbox_clone` é consultar a tabela de caixas postais local ao *nodo* da *thread* invocadora, no intuito de encontrar a entrada cujo campo nome coincide com aquele passado como argumento. Ao encontrar, `deck_mbox_clone` descobre, a partir dos demais campos da entrada, o *nodo* em que a caixa postal fora criada e o identificador local da caixa postal no referido *nodo*, identificador este que é idêntico ao do segmento compartilhado referente à caixa postal, conforme comentou-se anteriormente.

De posse de tais informações, `deck_mbox_clone` obtém uma conexão ao segmento compartilhado da caixa postal, e o mapea ao espaço lógico de endereçamento do processo em que se encontra a *thread* invocadora. Por fim, inicializam-se as estruturas de dados que comporão a referência à caixa postal, retornada pela função, de modo que a *thread* invocadora possa enviar mensagens à caixa postal destinatária, por qualquer dos protocolos de comunicação do DECK/SCI.

A estrutura de uma referência para caixa postal foi mostrada, em separado, pelas figuras 6.5, 6.7 e 6.9, com respeito aos protocolos 1, 2 e 3, respectivamente. A constituição de uma referência para caixa postal é exibida integralmente na figura 6.14. Observe-se que estão presentes todas as estruturas de dados, atinentes aos três protocolos de comunicação, as quais controlam o tráfego de mensagens com destino aos buffers da caixa postal reservados ao *nodo* responsável pela *thread* que obteve a referência.

Outra primitiva fornecida pelo DECK para a manipulação de caixas postais é `deck_mbox_destroy`, que atua sobre variáveis do tipo `deck_mbox_t`, representem elas caixas postais propriamente ditas ou referências para caixas postais. Em qualquer dos casos, o efeito de `deck_mbox_destroy` é liberar os recursos associados à variável passada como argumento.

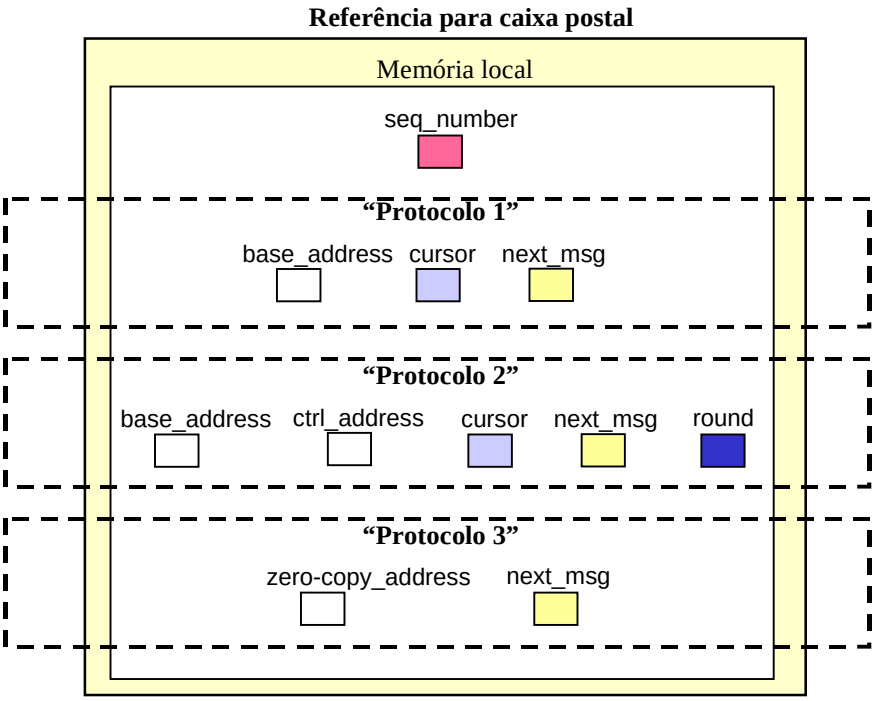

FIGURA 6.14 - Componentes de uma referência para caixa postal.

6.7.4 Controle de fluxo

A formulação de protocolos de comunicação há que passar, obrigatoriamente, pela concepção de mecanismos de controle de fluxo, porquanto é intolerável a perda de mensagens advinda de um pífio gerenciamento dos buffers de sistema. Em virtude do modelo de programação inerente ao DECK, o controle de fluxo assume uma perspectiva mais ampla, estando estritamente relacionado à noção de caixa postal: não basta controlar o fluxo de mensagens encaminhadas a um determinado *nodo*; necessário é proceder-se a tal controle no contexto de uma caixa postal, isoladamente.

O “protocolo 3” do DECK/SCI, por ser síncrono e ter por fulcro uma técnica de *zero-copy*, não requer controle de fluxo. Diferentemente, o projeto dos protocolos 1 e 2 exigiu a implementação de esquemas de controle de fluxo, por caixa postal.

A despeito das diferenças entre os dois protocolos, a idéia subjacente à estratégia adotada para evitar a sobreposição de mensagens nos buffers de uma caixa postal é a mesma, qual seja: a *thread* proprietária da caixa postal, ao consumir uma mensagem, atualiza um cursor que indica a posição donde deve extrair a próxima mensagem, e informa o valor de seu cursor à *thread* emissora. Uma questão impõe-se para a concretização desta idéia: como comunicar à emissora o progresso da receptora?

A solução encontrada foi a exploração do então denominado segmento de controle, que reserva espaço para informações de controle de fluxo originadas pelo “protocolo 1”, e pelo “protocolo 2”, em separado, como se pode ver na figura 6.11. A

estrutura lógica interna das áreas do segmento de controle reservadas para controle de fluxo é idêntica àquela da tabela de caixas postais, mostrada na figura 6.13.

À luz de tal estrutura lógica, a *thread* receptora pode calcular exatamente o endereço, dentro do segmento de controle pertencente ao *nodo* da *thread* emissora, para o qual deve remeter as informações necessárias. O endereço é calculado pela receptora levando em consideração a identificação do *nodo* em que está sendo executada e, também, o identificador local de sua caixa postal. Deste modo, garante-se que cada caixa postal, de cada *thread*, tenha uma posição única em cada área de controle de fluxo. A expressão usada para o cálculo do endereço, em relação ao endereço-base da área de controle de fluxo, não surpreendentemente, é $(local\_id + nodo \times M) \times 64$, onde "local_id" é a identificação local da caixa postal; "nodo" é o identificador do *nodo* em que se acha a *thread* proprietária da caixa postal; e "M" é o número máximo de caixas postais que podem ser criadas em cada *nodo*.

6.8 Finalização do ambiente de execução

A primitiva `deck_done`, a ser invocada por cada *nodo* para marcar o encerramento da aplicação paralela, realiza as tarefas de liberação dos recursos associados às estruturas de dados utilizadas internamente pelo DECK/SCI. Ademais, desfaz os mapeamentos dos segmentos remotos — de controle e de mensagens — e remove os segmentos locais.

Os componentes do ambiente de execução, em cada um dos *nodos*, são extintos, de modo que o *cluster* SCI possa vir a ser utilizado posteriormente por outras aplicações paralelas.

6.9 Comentários finais

O capítulo que aqui finda apresentou os detalhes de maior relevância do projeto e implementação do DECK/SCI, a biblioteca de programação paralela por troca de mensagens, específica para *clusters* SCI, proposta e concluída a contento neste Trabalho.

O êxito da implementação deveu-se, em grande parte, à etapa de estudo e pesquisa, durante a qual investigou-se a tecnologia SCI, enquanto rede de alto desempenho para *clusters*, identificando-se-lhe as limitações e observando-se as benesses que poderiam, potencialmente, advir-lhe do uso.

Concomitantemente às investigações acerca da tecnologia SCI, foram encetados estudos aprofundados sobre os ambientes de programação desenvolvidos para *clusters* baseados em tal tecnologia, dando-se ênfase às interfaces de programação e aos serviços oferecidos. Abrangeram-se, nestes estudos, diversos níveis de abstração, desde os drivers, até sofisticadas bibliotecas que visam ao desenvolvimento de aplicações paralelas por compartilhamento de memória. Esta perscrutação foi de suma importância para que se pudesse identificar, inequivocamente, com fundamentação proporcionada por um sólido substrato de conhecimento técnico, a interface de programação que melhor se coaduna aos objetivos traçados pelo projeto do DECK/SCI.

Examinaram-se, outrossim, diversos mecanismos empregados pelas existentes bibliotecas de comunicação por troca de mensagens, voltadas para *clusters* SCI, no intuito de não apenas julgá-los, crítica e criteriosamente, arrolando-lhes os méritos e deméritos, mas também, de considerar, desde que plausível, a possibilidade de adaptação de algumas das estratégias propostas às necessidades do DECK/SCI.

As decisões de projeto, à luz das metas estabelecidas, foram pautadas pela intensa pesquisa realizada, a qual se relatou nos capítulos 2, 3 e 4 do presente texto.

Com o fito de contextualizar os mecanismos implementados no DECK/SCI, a tabela 6.1 coteja a biblioteca resultante desta Dissertação de Mestrado com aquelas que se lhe assemelham.

TABELA 6.1 - Resumo das principais características de implementação do DECK/SCI e das demais bibliotecas de comunicação por troca de mensagens.

Biblioteca	API usada	Manipulação de buffers	Recebimento de mensagens	Acesso à memória remota	Número de protocolos	Envio de mensagens grandes
PVM-SCI	driver	buffers exclusivos	interrupção	`memcpy` tradicional	2	DMA
SCIPVM	driver	buffers exclusivos	interrupção	`memcpy` tradicional	2	DMA
SCI-MPICH	SMI	buffers exclusivos	*polling*	instruções MMX	3	transmissão e cópia local intercaladas
ScaMPI	driver e ScaFun	buffers exclusivos	*polling*	instruções MMX	3	transmissão e cópia local intercaladas
CML	SISCI	buffers exclusivos	*polling*	`memcpy` tradicional	2	DMA
DECK/SCI	SISCI	buffers exclusivos	*polling*	instruções MMX	3	*zero-copy*

7 Análise comparativa de desempenho e validação

Após os comentários sobre as motivações principais deste Trabalho e a apresentação da proposta, externados pelo capítulo 5, e o detalhamento do projeto e da implementação da biblioteca de programação paralela, objeto desta Dissertação, que se seguiu no capítulo 6, remanesce mostrar os resultados concretos obtidos com o uso do DECK/SCI.

O capítulo em curso analisa os protocolos de comunicação do DECK/SCI, em relação ao desempenho ótimo da rede SCI. Em seguida, ainda sob a perspectiva de desempenho, o DECK/SCI é confrontado com dois ambientes de programação que servem ao mesmo propósito — SCI-MPICH e ScaMPI.

Ainda no presente capítulo, apresentam-se os resultados obtidos com a execução de uma aplicação paralela clássica, modelada e desenvolvida através do DECK.

7.1 Introdução

Dentre todas as bibliotecas de programação por troca de mensagens pesquisadas, SCI-MPICH e ScaMPI auferem o melhor desempenho de comunicação, em conseqüência de três fatores: utilização de instruções otimizadas para o acesso à memória remota; preterição de esquemas baseados em interrupções; e especificação de três protocolos de comunicação, com vistas a um tratamento realmente especializado de mensagens muito pequenas, e de mensagens tão grandes quanto se queira. Como se não bastasse, ambas bibliotecas fornecem uma interface de programação em consonância com o padrão MPI, que é, certamente, o mais difundido e utilizado para o desenvolvimento de aplicações paralelas.

Pelas razões acima expostas, somadas à questão de disponibilidade, SCI-MPICH e ScaMPI mostram-se excelentes parâmetros de comparação para o DECK/SCI.

Todos os experimentos relatados neste capítulo foram realizados no *cluster* SCI disponível no Instituto de Informática da UFRGS, cuja especificação técnica descreveu-se na seção 6.2 deste texto. Como de praxe, as métricas utilizadas nas análises de desempenho foram latência e largura de banda, mensuradas através do típico algoritmo *ping-pong*: um *nodo* envia uma mensagem a outro, que, ao recebê-la, devolve-a ao emissor. Para cada tamanho de mensagem, o *nodo* emissor mede o tempo de 1000 repetições do *ping-pong* — 1000 idas e voltas da mensagem —, e calcula a latência como

$$lat = \frac{tempo}{2 \times 1000}$$

onde *tempo* é o tempo gasto para a mensagem ir e voltar 1000 vezes. Em função da latência, calcula-se a largura de banda, dada por

$$lb = \frac{tamanho}{lat}$$

onde *tamanho* é o tamanho da mensagem, em bytes, e *lat* é a latência, em segundos.

7.2 *Overhead* dos protocolos de comunicação

Quando se está analisando o desempenho de uma biblioteca de comunicação de alto nível, tal como o DECK/SCI, é mister que se meça e avalie o *overhead* incutido pelos protocolos implementados, em relação ao desempenho máximo que a rede pode proporcionar. Medidas deste tipo permitem verificar o impacto causado por mecanismos como controle de fluxo, ordenação de mensagens, sinalização, codificação de cabeçalhos e *handshaking*, além de cópias de mensagens para a memória local, enfim, por tudo aquilo que se adiciona à simples comunicação "pura".

7.2.1 Avaliação do "protocolo 1"

A figura 7.1 permite avaliar o "protocolo 1" do DECK/SCI em relação à comunicação "pura" na rede SCI.

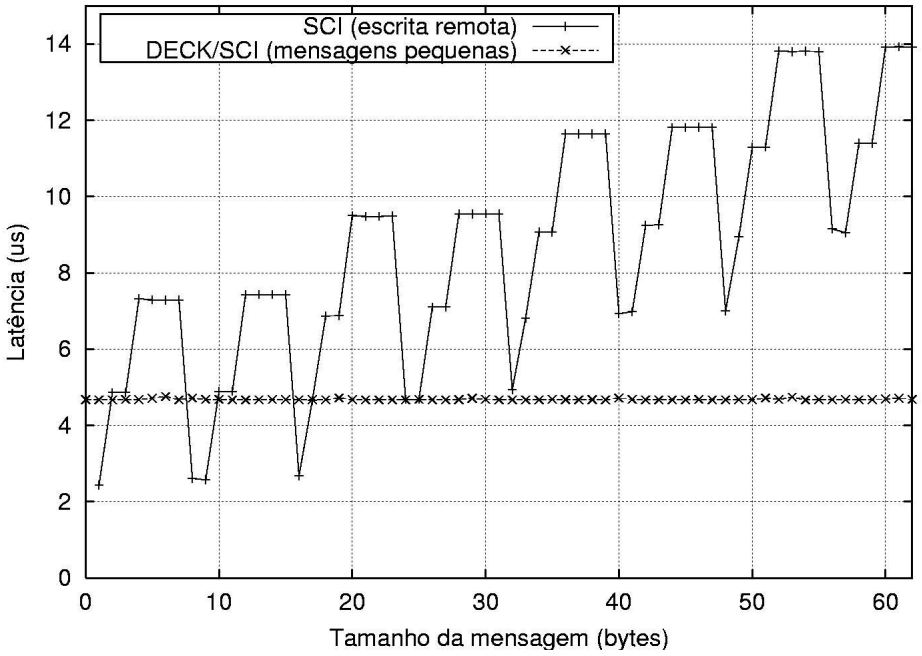

FIGURA 7.1 - Latência do "protocolo 1" do DECK/SCI.

Apresenta-se, na figura, a latência que se mediu para mensagens cujo tamanho varia entre 0 e 62 bytes, justamente o escopo manipulável pelo "protocolo 1". Pode-se observar que a latência medida para o "protocolo 1" do DECK/SCI manteve-se

constante, abaixo de 5 μs. Explica-se esta constância pelo fato de que, independentemente do tamanho da mensagem a ser enviada, o "protocolo 1" sempre transmite um pacote SCI de 64 bytes, conforme descrito na seção 6.4.5.

O resultado aparentemente surpreendente, revelado pela figura 7.1, é o desempenho nitidamente superior do DECK/SCI em comparação com a escrita remota "pura", salvo para mensagens de 1, 8 e 16 bytes. A aparência irregular da curva de latência para a escrita remota deve-se à variação do número de pacotes necessários para enviar cada mensagem, sendo que esta variação não é diretamente proporcional ao tamanho da mensagem, conforme esclareceu-se na seção 6.4.1 (ver figura 6.1). Diferentemente, o "protocolo 1" do DECK/SCI apresenta uma latência constante porque requer, sempre, apenas um pacote SCI para o envio de mensagens; além disso, por transmitir necessariamente 64 bytes, otimiza a utilização dos buffers de escrita da placa SCI e obtém latências mais baixas do que a própria escrita remota "pura". Lembre-se de que a rede SCI comporta-se melhor quando quantidades múltiplas de 64 bytes são transmitidas.

Estes resultados atestam os cuidados especiais que exige o trato de mensagens pequenas, e comprovam que o "protocolo 1", de fato, cumpre seu papel de garantir baixíssima latência de comunicação, tendo obtido 4,66 μs. A figura 7.1 enaltece os méritos técnicos dos mecanismos cuidadosamente planejados para o "protocolo 1", que compensa seus *overheads* intrínsecos, acarretados por controle de fluxo, ordenação de mensagens e sinalização, utilizando eficientemente as potencialidades de baixa latência da rede SCI, de modo que se mostra extremamente enxuto.

7.2.2 Avaliação dos protocolos 2 e 3

Ao contrário do "protocolo 1", os demais protocolos de comunicação do DECK/SCI, inevitavelmente, deixam transparecer o *overhead* que lhes é inerente, apresentando um desempenho inferior ao da escrita remota "pura". Em verdade, este resultado era esperado, pois seria impossível implementar protocolos de comunicação mais eficientes do que o mecanismo elementar por eles próprios utilizado.

As figuras 7.2 e 7.3 comparam, respectivamente, em termos de latência e largura de banda, os protocolos 2 e 3 com a escrita remota "pura".

Analisando a figura 7.2, pode-se perceber que a curva tocante à escrita remota não apresenta o comportamento irregular anteriormente exibido, visto que foi medida a latência apenas para mensagens com os tamanhos mostrados no eixo das abscissas, que são múltiplos de 64.

O propósito dos gráficos das figuras 7.2 e 7.3 é comprovar que o *overhead* dos protocolos impede-os de obter o desempenho ótimo da escrita remota "pura", e avaliar o impacto deste *overhead* no desempenho resultante. Observe-se que a curva de latência da escrita remota permanece abaixo das curvas do DECK/SCI. Perceba-se, também, que a latência do "protocolo 2" é inferior à do "protocolo 3" para mensagens até 1024 bytes; após, o mecanismo de *zero-copy* passa a ser mais vantajoso. De fato, a curva de latência do "protocolo 3" tende a acompanhar a curva da escrita remota, mantendo a diferença constante, ao passo que a curva do "protocolo 2" começa a se distanciar quando o

tamanho de mensagem ultrapassa 1024 bytes, devido ao *overhead* extra necessário para copiar a mensagem do buffer da caixa postal para o buffer do usuário.

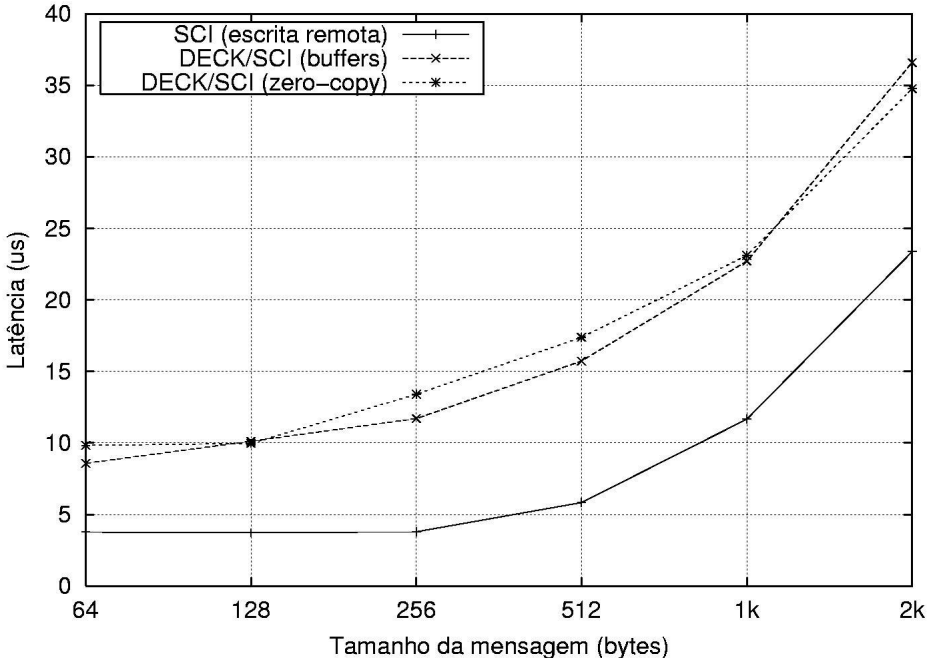

FIGURA 7.2 - Latência dos protocolos 2 e 3 do DECK/SCI.

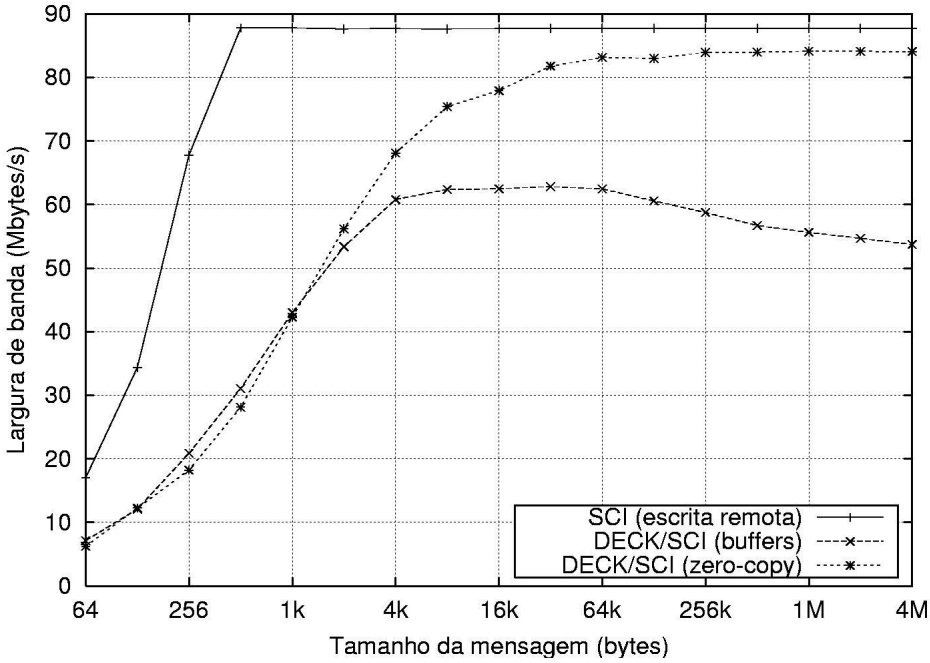

FIGURA 7.3 - Largura de banda dos protocolos 2 e 3 do DECK/SCI.

As curvas de largura de banda, exibidas na figura 7.3, enfatizam a discrepância existente entre os protocolos 2 e 3 do DECK/SCI. Corroborando a tendência mostrada pelos gráficos de latência, o “protocolo 3”, graças ao mecanismo de *zero-copy*, é capaz

de obter um desempenho comparável ao da escrita remota; por sua vez, o "protocolo 2", ante o aumento do tamanho das mensagens, a partir de 1024 bytes, experimenta uma diminuição na taxa de crescimento da largura de banda, que atinge o ápice em 62,81 Mbytes/s, para mensagens de 32 kbytes. Caso mensagens maiores sejam enviadas, a largura de banda obtida pelo "protocolo 2" começa a decair. O fraco desempenho de tal protocolo advém da cópia extra da mensagem para o buffer do usuário, efetuada somente depois da conclusão de sua transmissão pela rede. Este *overhead* da cópia de mensagem da memória compartilhada para a memória local é tanto mais visível quanto maior for a quantidade de bytes.

Os resultados aqui comentados mostram a necessidade de um tratamento especial para mensagens grandes, razão pela qual o protocolo *zero-copy* foi idealizado. A largura de banda máxima alcançada pelo "protocolo 3" foi 84,12 Mbytes/s, que representa um desempenho fantástico, quando comparado ao limite suportado pela rede SCI — 87,72 Mbytes/s. Em sendo assim, o "protocolo 3" atinge 95,9% da máxima largura de banda alcançável, exercendo efetivamente seu papel de proporcionar, às aplicações paralelas, desempenho próximo aos limites impostos pelo hardware subjacente.

7.3 Confrontação dos três protocolos do DECK/SCI

A figura 7.4 confronta, explicitamente, os três protocolos de comunicação do DECK/SCI, com respeito à latência, trazendo à tona os efeitos da especialização de cada um.

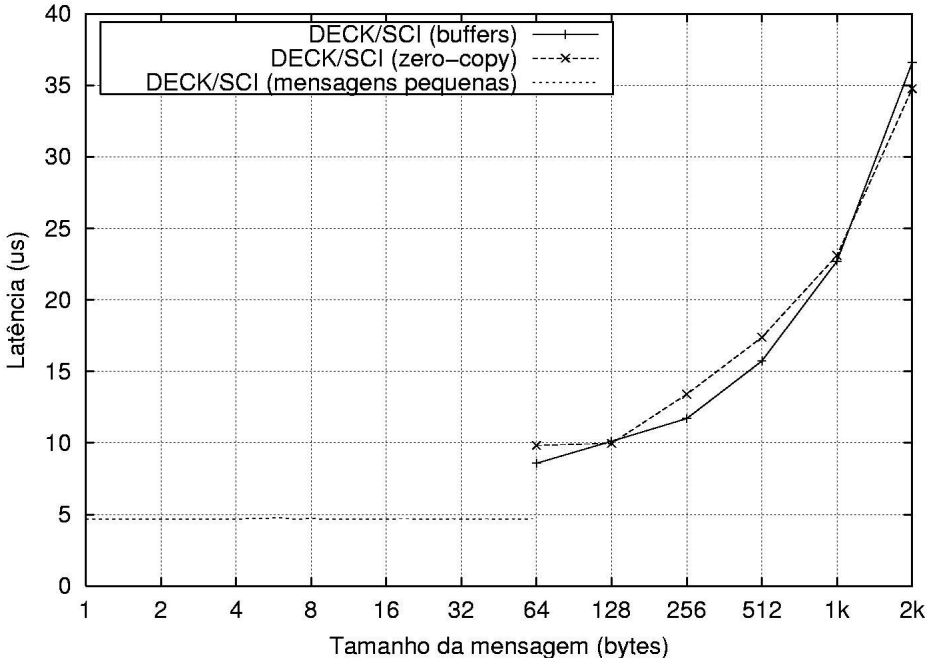

FIGURA 7.4 - Latência dos três protocolos de comunicação do DECK/SCI.

Torna-se evidente, a partir da figura, o acentuado aumento de latência quando da transição do "protocolo 1" aos demais protocolos. A latência passa de 4,66 para 8,56 μs,

no momento em que o “protocolo 2” entra em cena. Esta duplicação da latência não é surpreendente, visto que o “protocolo 2”, para enviar uma mensagem de 64 bytes, transmite 128 bytes, sendo 64 para o cabeçalho e 64 para o corpo da mesma e, adicionalmente, remete outro pacote SCI de 64 bytes, a fim de sinalizar a operação de comunicação. Em contraposição, o “protocolo 1”, utilizado para mensagens com tamanho entre 0 e 62 bytes, exige o tráfego de apenas e tão-somente um pacote SCI de 64 bytes. A figura 7.4 mostra que, na ausência de um protocolo especializado em mensagens pequenas, jamais seria possível obter latências inferiores a 5 μs, reforçando a necessidade de implantação de múltiplos protocolos, se o objetivo é desenvolver uma biblioteca de comunicação comprometida com alto desempenho.

O “protocolo 3”, por outro lado, exibe uma latência de 9,82 μs para o envio de uma mensagem de 64 bytes. O esquema de *handshaking*, envolvendo requisição, envio de endereço, envio da mensagem e sinalização, imprescindível à comunicação por *zero-copy*, produz um *overhead* ao qual são particularmente sensíveis as mensagens com tamanho menor ou igual a 1024 bytes. A partir deste ponto, comparativamente ao “protocolo 2”, as etapas de sincronização do “protocolo 3” são compensadas pelo fato de se evitar a cópia extra da mensagem para o buffer presente na memória local do receptor. Ademais, cada mensagem de controle, usada durante o *handshaking*, acarreta a menor latência possível na rede SCI — 2,5 μs —, visto que possui apenas um byte. Isto explica por que o “protocolo 3” não tarda muito a ser mais vantajoso do que o “protocolo 2”.

A figura 7.5 exibe a mesma confrontação entre os protocolos do DECK/SCI, porém, sob a ótica da largura de banda.

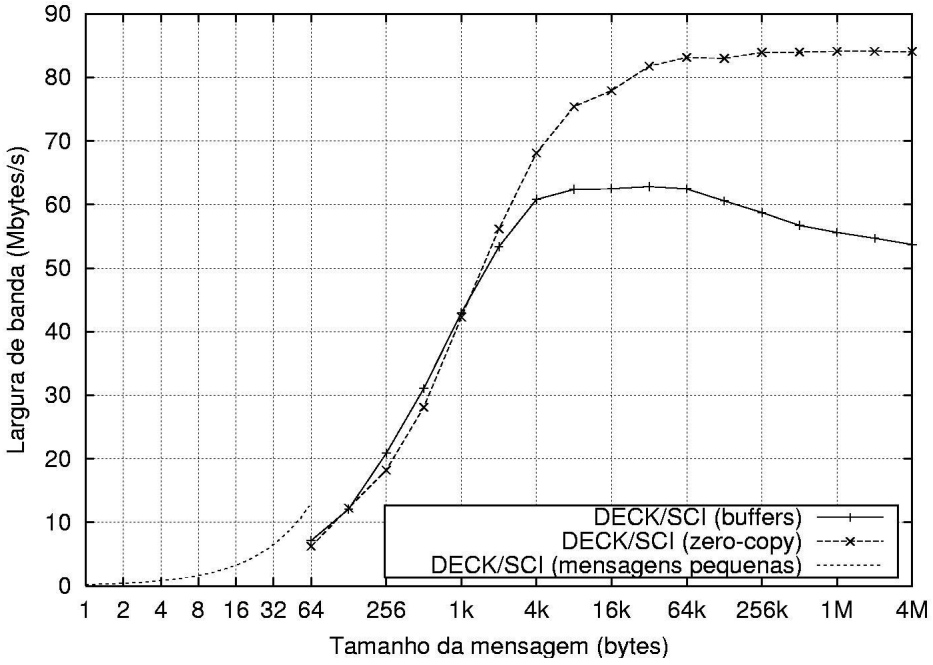

FIGURA 7.5 - Largura de banda dos três protocolos de comunicação do DECK/SCI.

Refletindo a elevação de latência ocasionada pela transição do “protocolo 1” aos demais, a figura 7.5 evidencia a queda da largura de banda. A largura de banda obtida

pelo “protocolo 1”, para uma mensagem de 62 bytes, foi 12,63 Mbytes/s, ao passo que os protocolos 2 e 3 iniciam com uma largura de banda de 7,12 e 6,21 Mbytes/s, respectivamente.

A conclusão a ser extraída da figura 7.5 é que seria impraticável obter uma largura de banda de pico próxima aos limites do hardware SCI — acima de 80 Mbytes/s — sem a elaboração de um protocolo especializado em mensagens grandes. As figuras 7.4 e 7.5 justificam todos os esforços envidados na especificação e implementação de três protocolos de comunicação distintos.

Cabe um último comentário a respeito da diversidade de protocolos do DECK/SCI. Embora o tamanho de mensagem sugerido pelos gráficos para a transição do “protocolo 2” ao “protocolo 3” seja 1024 bytes, talvez seja interessante adotar outro valor. Não deve ser esquecido que o “protocolo 3” é síncrono, impedindo a liberação do emissor até que o receptor esteja pronto para receber a mensagem. Em algumas aplicações, este comportamento pode não ser desejável. Um meio-termo poderia ser protelar ao máximo a transição para o “protocolo 3”, a fim de impor o caráter síncrono da troca de mensagens somente quando o “protocolo 2” tender a degradar o desempenho. Um bom limite poderia ser 8 kbytes, visto que, para mensagens com este tamanho, o “protocolo 2” chega praticamente ao ápice de sua largura banda, conforme pode-se constatar na figura 7.5. Lembre-se de que o limite para a transição entre os protocolos 2 e 3 é configurável pelo usuário, sendo representado pelo valor `DECK_MSG_BUF_LIMIT`.

7.4 DECK/SCI, SCI-MPICH e ScaMPI

A seguir, o DECK/SCI é comparado, em termos de desempenho, às bibliotecas SCI-MPICH e ScaMPI. Nos experimentos realizados, manteve-se a configuração *default* dos *eager buffers* de ambas bibliotecas.

A figura 7.6 mostra a latência medida para os três ambientes de programação, considerando mensagens de 0 a 62 bytes. Nitidamente, o desempenho do DECK/SCI é melhor: além de apresentar latências mais baixas em todo o intervalo, o DECK/SCI é o único ambiente a conseguir manter a latência constante e abaixo de 5 μs, demonstrando o efeito dos apurados mecanismos do “protocolo 1”.

O ambiente SCI-MPICH experimenta latências em torno de 7 μs para mensagens até 44 bytes, que é justamente o intervalo de atuação do seu protocolo *Short*, o qual se mostra ligeiramente mais otimizado do que o equivalente do ScaMPI. Porém, entre 44 e 62 bytes, o ScaMPI exibe latência menor que a do SCI-MPICH, mantendo-a por volta de 10 μs até 56 bytes, sofrendo em seguida alguns acréscimos, até atingir 15 μs. Neste mesmo intervalo, SCI-MPICH tem sua latência elevada consideravelmente, chegando a passar de 35 μs.

O “protocolo 1” do DECK/SCI, além de ser mais eficiente do que os protocolos *Short* das outras bibliotecas, possui um intervalo de atuação maior, mantendo uma baixíssima latência para mensagens de 0 até 62 bytes.

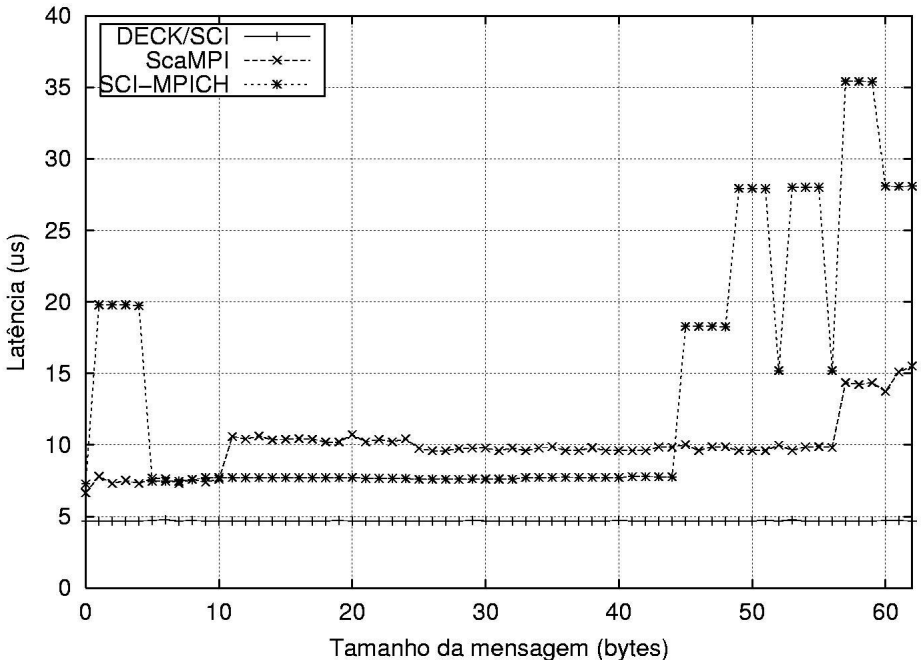

FIGURA 7.6 - Latência para mensagens pequenas: DECK/SCI, SCI-MPICH e ScaMPI.

A tabela 7.1 discrimina os valores de latência obtidos por cada uma das bibliotecas, para mensagens vazias (0 byte). Cumpre salientar, mais uma vez, que o DECK/SCI consegue manter sua latência mínima para mensagens de 0 a 62 bytes.

TABELA 7.1 - Latência mínima obtida por DECK/SCI, SCI-MPICH e ScaMPI.

Biblioteca	Latência mínima
DECK/SCI	4,66 µs
ScaMPI	6,63 µs
SCI-MPICH	7,26 µs

A comparação dos outros dois protocolos do DECK/SCI — protocolos 2 e 3 — com SCI-MPICH e ScaMPI acha-se nas figuras 7.7 e 7.8. Estas figuras, no intuito de permitir uma avaliação do desempenho global das bibliotecas de comunicação, mostram a latência para mensagens variando de 8 bytes a 2 kbytes, e a largura de banda para mensagens entre 8 bytes e 8 Mbytes, embora a figura 7.6 já tenha detalhado a latência resultante do envio de mensagens pequenas — de 0 a 62 bytes.

A figura 7.7 não detalha a latência medida para as mensagens com tamanho compreendido pelo intervalo entre 44 e 64 bytes, em que o ScaMPI obtém um desempenho superior ao do SCI-MPICH. Diferentemente, são mostradas as medidas referentes somente aos tamanhos de mensagem explicitados no eixo horizontal, com o fito de facilitar a apreensão da tendência geral do comportamento das bibliotecas.

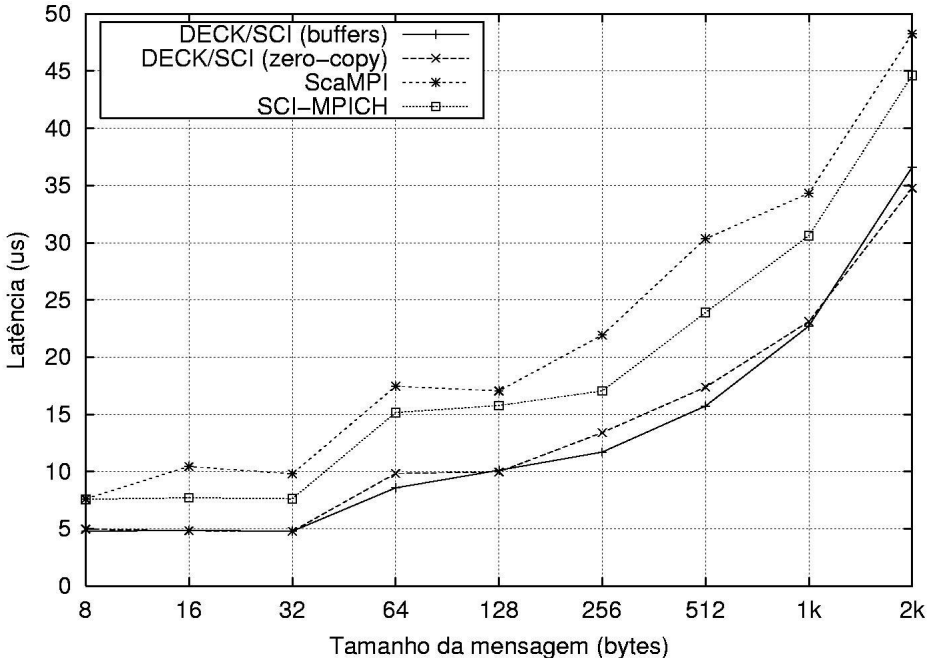

FIGURA 7.7 - Latência obtida pelas bibliotecas de comunicação.

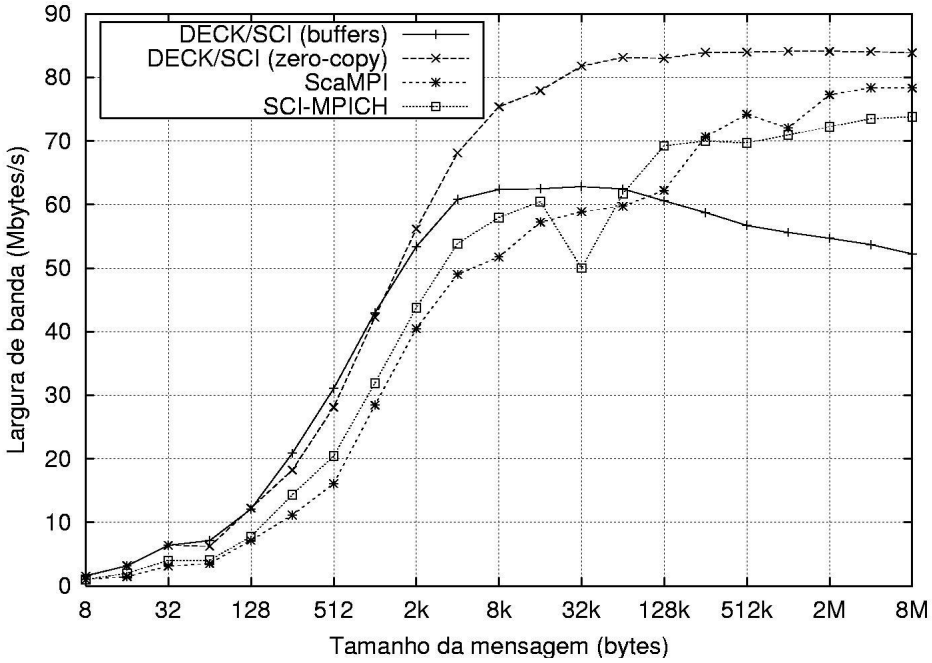

FIGURA 7.8 - Largura de banda obtida pelas bibliotecas de comunicação.

Mais uma vez, constata-se o desempenho visivelmente superior do DECK/SCI. Conforme a figura 7.7, os três protocolos de comunicação do DECK/SCI apresentam latência menor do que a dos protocolos de SCI-MPICH e ScaMPI.

Através da análise dos gráficos atinentes à largura de banda, explicitados pela figura 7.8, pode-se chegar a algumas conclusões. O “protocolo 2” do DECK/SCI obtém maiores larguras de banda do que os protocolos *Eager* de SCI-MPICH e ScaMPI, os quais atuam de 64 bytes a 32 kbytes. Por sua vez, o protocolo *Eager* do SCI-MPICH é mais otimizado do que o protocolo equivalente do ScaMPI.

Para tamanhos de mensagem acima de 32 kbytes, as implementações de MPI, com vistas à elevação da largura de banda, empregam o protocolo *Rendez-vous*, por meio do qual o ScaMPI supera o SCI-MPICH, em termos de máxima largura de banda alcançável. Todavia, o “protocolo 3” do DECK/SCI, com seu eficiente mecanismo de *zero-copy*, sobrepuja facilmente os protocolos *Rendez-vous* de ambas implementações de MPI, mantendo sempre uma largura de banda consideravelmente mais elevada.

O melhor desempenho do “protocolo 2” do DECK/SCI, em relação aos equivalentes protocolos *Eager* de SCI-MPICH e ScaMPI, deve-se a uma profícua exploração dos buffers de escrita da placa SCI, otimizando o número de transações necessárias para enviar cada mensagem. O protocolo *Eager* do ScaMPI apresenta um desempenho ainda pior do que o correspondente protocolo do SCI-MPICH, provavelmente, em virtude do seu mecanismo de *checkpointing* utilizado para fins de tolerância a falhas.

A técnica de *zero-copy* do DECK/SCI mostrou-se efetiva na consecução da meta de obter largura de banda próxima dos limites inerentes à arquitetura subjacente, e claramente demonstrou, como era de se esperar, um desempenho superior ao esquema de intercalação entre transmissão e cópia local, adotado pelos protocolos *Rendez-vous* das implementações de MPI.

Encontram-se arrolados, na tabela 7.2, os valores obtidos para a máxima largura de banda de cada biblioteca de comunicação, bem como a relação entre tais valores e o limite máximo suportado pela rede SCI.

TABELA 7.2 - Máxima largura de banda alcançável pelas bibliotecas de comunicação.

Biblioteca	Máxima largura de banda	Relação entre a máxima largura de banda obtida e o limite suportado pela rede SCI
DECK/SCI	84,12 Mbytes/s	95,9 %
ScaMPI	78,35 Mbytes/s	89,3 %
SCI-MPICH	73,80 Mbytes/s	84,1 %

Em suma, conclui-se que o DECK/SCI apresenta desempenho superior às duas implementações de MPI, para qualquer tamanho de mensagem, haja visto que os protocolos 1, 2 e 3 do DECK/SCI são mais eficientes do que os equivalentes *Short*, *Eager* e *Rendez-vous* de SCI-MPICH e ScaMPI.

7.5 Validação do DECK/SCI

Um grupo de alunos da Pontifícia Universidade Católica do Rio Grande do Sul começou, há pouco tempo, a desenvolver aplicações paralelas com o DECK. Uma aplicação já por eles concluída foi a geração paralela dos clássicos fractais de Mandelbrot [MAN 82].

O grupo da PUC-RS utilizou-se do DECK/Myrinet para o desenvolvimento da citada aplicação; porém, como a API exportada pelo DECK/Myrinet é idêntica à interface de programação fornecida pelo DECK/SCI, o programa teria de funcionar, sem quaisquer alterações, em ambos ambientes. Em sendo assim, para validar a implementação do DECK/SCI, aproveitou-se a disponibilidade de uma aplicação que sabidamente produzia resultados corretos com o DECK/Myrinet.

De fato, o programa paralelo também funcionou perfeitamente com o DECK/SCI, sem modificações, comprovando que as primitivas e objetos de ambos os DECKs seguem a mesma semântica, de forma que podem ser usados indistintamente. Um fator deveras interessante neste processo de validação é que a aplicação paralela foi construída por pessoas alheias aos detalhes de implementação do DECK/SCI, corroborando mais ainda a manutenção estrita da semântica definida para as primitivas da API, a despeito das particularidades de cada DECK.

7.5.1 Fractais de Mandelbrot

Os fractais de Mandelbrot são imagens abstratas produzidas pela aplicação sucessiva de operações matemáticas sobre cada ponto da figura. Por definição, cada ponto da imagem é representado por um número *c* do plano dos complexos, da forma $c = a + bi$. Define-se formalmente uma órbita como a seqüência de pontos gerada, a partir do número $z_0 = 0 + 0i$, empregando-se a fórmula $z_{n+1} = z_n^2 + c$, onde *c* é um ponto qualquer no plano dos complexos Diz-se que uma órbita “escapou” quando a distância de algum de seus pontos até a origem for maior do que 2. Assim, o conjunto de Mandelbrot é representado pelos pontos cujas órbitas não “escapam”. A figura 7.9 mostra a imagem referente ao conjunto de Mandelbrot.

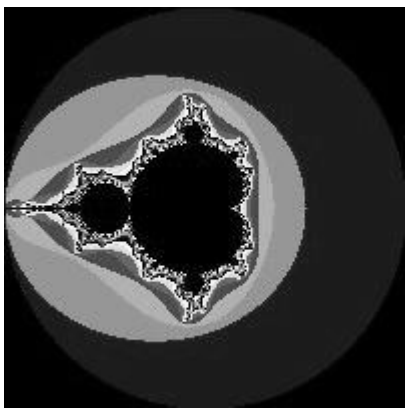

FIGURA 7.9 - O conjunto de Mandelbrot.

A implementação do algoritmo de geração de fractais de Mandelbrot exige que se estabeleça o número máximo de iterações para o cálculo de cada ponto. Se a órbita não “escapar” até o número máximo de iterações ser atingido, assume-se que o ponto faz parte do conjunto. A aparência da imagem exibida na figura 7.9 é obtida definindo-se a cor de cada ponto em função do número de iterações necessárias para calculá-lo. No intuito de se obter o conjunto de Mandelbrot, basta a aplicação da fórmula acima mencionada sobre os pontos pertencentes à região compreendida pelas coordenadas (-2, 2) e (2, 2), visto que a distância máxima de cada ponto à origem é 2.

O algoritmo para a geração do conjunto de Mandelbrot é facilmente paralelizável. A estratégia adotada pelos alunos da PUC-RS, usando o DECK, foi dividir a imagem em sub-regiões, atribuindo-as, sob demanda, a diferentes processadores. Em um *cluster* com *N nodos*, haverá um mestre e *N–1* escravos. O mestre divide a imagem em sub-regiões com um determinado número de pontos. Cada escravo calcula uma sub-região distinta e, após o término do cálculo, envia os resultados ao mestre e requisita outra sub-região. Quando não houver mais sub-regiões a calcular, o mestre terá consigo o conjunto completo de Mandelbrot. A atribuição de tarefas é feita sob demanda porque a simples divisão da imagem em um número de sub-regiões igual à quantidade de escravos redundaria em problemas de balanceamento de carga, haja visto que a convergência do cálculo depende da localização da sub-região.

A facilidade de paralelização deste algoritmo advém da independência entre os cálculos de sub-regiões diferentes. Em se tratando do modelo de programação do DECK, há a necessidade de criação de uma caixa postal em cada *nodo*, para que a comunicação entre mestre e escravos seja possível.

Nos experimentos realizados com o DECK/SCI, utilizou-se uma imagem com 600x600 pixels, dividida em 400 sub-regiões, tendo-se atribuído 17500 ao número máximo de iterações. Quanto à configuração do DECK/SCI, adotaram-se os valores definidos por *default*: `DECK_MSG_BUF_LIMIT` = 8192 bytes — limite para a transição do “protocolo 2” ao “protocolo 3” —, e `DECK_MEDBUF_SIZE` = 24768 bytes — tamanho dos buffers do “protocolo 2”.

A tabela 7.3 mostra os tempos de processamento medidos para a geração do conjunto de Mandelbrot com diferentes números de escravos, bem como o *speed-up* calculado.

TABELA 7.3 - Tempos de processamento para a geração do conjunto de Mandelbrot.

Número de escravos	Tempo de processamento	*Speed-up*
1	59,77 s	—
2	29,91 s	1,998
3	19,94 s	2,997

Observa-se que os ganhos obtidos pelo aumento do número de escravos estão muito próximos do ideal. Atribui-se isto não somente à natureza da aplicação, mas também, ao alto desempenho proporcionado pelos protocolos de comunicação do

DECK/SCI. Entretanto, cumpre enfatizar que, mesmo na situação em que apenas um escravo é utilizado, há comunicação entre dois *nodos* — o mestre e o escravo —, ou seja, o cálculo do ganho não foi feito em relação a uma implementação genuinamente seqüencial, mascarando alguns custos intrínsecos das aplicações paralelas, tais como, o *overhead* devido ao disparo da aplicação em múltiplos *nodos* e, no caso do DECK, o tempo de criação de caixas postais.

8 Conclusão

A tecnologia SCI, enquanto rede de alto desempenho para *clusters*, a um só tempo, facilita o desenvolvimento de ambientes de programação baseados em memória compartilhada, graças ao suporte em hardware a DSM, e permite a implementação de bibliotecas de comunicação por troca de mensagens. A grande vantagem evidenciada por SCI é a baixa latência que proporciona no acesso direto à memória remota. Esta característica pode ser explorada para a elaboração de eficientes protocolos de comunicação, contanto que sejam levadas em consideração algumas particularidades que afetam diretamente o desempenho resultante, tais como, o funcionamento dos buffers de escrita das interfaces de rede, a necessidade do uso de instruções otimizadas para o acesso à memória remota e a lentidão inerente a leituras remotas.

A performance de protocolos de comunicação por troca de mensagens é fortemente influenciada pelo mecanismo de sinalização de mensagens adotado, pela localização das estruturas de dados utilizadas para o gerenciamento de buffers de sistema, e pelos mecanismos de controle de fluxo implementados. Outrossim, em busca de baixa latência e elevada largura de banda de pico, há que se incorporar múltiplos protocolos a uma biblioteca de comunicação, de modo que se possa tomar cuidados especiais na troca de mensagens pequenas, cujo desempenho é particularmente sensível a qualquer *overhead* adicional, e que se dê tratamento especial também ao envio de mensagens grandes, desempenho do qual é incisivamente prejudicado por cópias adicionais em memória.

A técnica proposta nesta Dissertação para o tratamento de mensagens pequenas é muito semelhante às técnicas empregadas por SCI-MPICH e ScaMPI. Entretanto, no tratamento de mensagens grandes, o DECK/SCI inovou, em relação a todos ambientes estudados, promovendo um protocolo de comunicação baseado em um método que reduz as cópias de mensagem ao mínimo necessário. Isto somente foi possível em função das características da API fornecida pelo DECK, que exige a criação explícita de "objetos" do tipo mensagem, através de primitivas da própria API, antes de qualquer operação de comunicação ser efetivada. Diferentemente, a API definida no padrão MPI impede a concepção de esquemas de transmissão de mensagens por *zero-copy*, pois a semântica das primitivas de comunicação exige que os dados que trafegam pela rede sejam copiados a um buffer localizado na memória local do *nodo* receptor. Por esta razão, SCI-MPICH e ScaMPI utilizam-se de um método de intercalação entre comunicação e cópia, em vez de *zero-copy*.

Os experimentos realizados comprovaram o melhor desempenho do DECK/SCI, em comparação a SCI-MPICH e ScaMPI. Visto que SCI-MPICH e ScaMPI, sabidamente, são mais otimizados do que os demais ambientes estudados, pode-se estender a comparação a todos os ambientes apresentados. O "protocolo 1" do DECK/SCI, para mensagens pequenas — de 0 a 62 bytes —, obtém uma latência de comunicação constante, de 4,66 μs. Trata-se de um protocolo com mínimo *overhead*. Por sua vez, o "protocolo 3" — *zero-copy* — permite alcançar, para mensagens grandes, uma largura de banda realmente próxima ao limite do hardware SCI — 84,12 Mbytes/s. Mesmo o protocolo de propósito geral do DECK/SCI — "protocolo 2" — mostra-se mais otimizado do que o equivalente *Eager* das implementações de MPI. Estas constatações, portanto, atestam que os objetivos do DECK/SCI, no que tange ao

desempenho, foram atingidos a contento. O DECK/SCI é, de fato, uma biblioteca de comunicação de baixa latência e elevada largura de banda.

A correta execução, no DECK/SCI, do programa de geração paralela de fractais de Mandelbrot, desenvolvido por alunos da PUC-RS originalmente com o DECK/Myrinet, serviu para validar a biblioteca de comunicação resultante desta Dissertação de Mestrado, comprovando que o DECK/SCI cumpriu a meta de manter a mesma semântica das primitivas da API do DECK/Myrinet. Em sendo assim, foi dado o primeiro passo em direção à consecução do modelo MultiCluster, que visa à integração de um *cluster* Myrinet e um *cluster* SCI. Pretende-se, futuramente, que o DECK — DECK/Myrinet + DECK/SCI — seja o ambiente de programação de tal arquitetura integrada, razão pela qual fez-se imprescindível o respeito incondicional à semântica das primitivas da API original.

As técnicas e métodos de comunicação implementados no DECK/SCI não têm seu emprego limitado apenas à tecnologia SCI. Qualquer rede de interconexão que suporte o acesso direto à memória remota, a exemplo de *Memory Channel* [GIL 96], pode vir a se beneficiar do DECK/SCI. Logo, conclui-se que os resultados obtidos nesta pesquisa transcendem as fronteiras da tecnologia especificamente utilizada.

Bibliografia

[AHM 2000] AHMAD, Ishfaq. Cluster computing: a glance at recent events. **IEEE Concurrency**, Los Alamitos, CA, v.8, n.1, Jan.–Mar. 2000.

[AMZ 96] AMZA, C. et al. TreadMarks: Shared Memory Computing on Networks of Workstations. **Computer**, Los Alamitos, CA, v.29, n.2, p.18-28, Feb. 1996.

[BAR 2000] BARRETO, Marcos Ennes. **DECK:** um ambiente para programação paralela de agregados de multiprocessadores. Porto Alegre: PPGC/UFRGS, 2000. Dissertação de Mestrado.

[BAR 2000a] BARRETO, M.; ÁVILA, R.; NAVAUX, P. The MultiCluster model to the integrated use of multiple workstation clusters. In: WORKSHOP ON PERSONAL COMPUTER BASED NETWORKS OF WORKSTATIONS, 3., 2000, Cancun. **Proceedings...** Berlin: Springer-Verlag, 2000. p.71-80. (Lecture Notes in Computer Science, n.1800).

[BOD 95] BODEN, N. et al. Myrinet: a gigabit-per-second local-area network. **IEEE Micro**, Los Alamitos, CA, v.15, n.1, p.29-36, Feb. 1995.

[BUG 99] BUGGE, Håkon; OMANG, Knut. Affordable Scalability Using Multi-Cubes. In: HELLWAGNER, Hermann; REINEFELD, Alexander (Eds.). **SCI:** Scalable Coherent Interface - Architecture and Software for High-Performance Compute Clusters. Berlin: Springer-Verlag, 1999. 490p. p.167-175. (Lecture Notes in Computer Science, n.1734).

[BUO 98] BUONADONNA, Philip; GEWEKE, Andrew; CULLER, David. An Implementation and Analysis of the Virtual Interface Architecture. In: SUPERCOMPUTING, 1998, Orlando, Florida. **Proceedings...** [S.l.:s.n.], 1998.

[BUT 98] BUTENUTH, Roger; HEISS, Hans-Ulrich. Shared Memory Programming on PC-based SCI clusters. In: SCI-EUROPE, 1998, Bordeaux. **Proceedings...** [S.l.:s.n.], 1998.

[BUT 99] BUTENUTH, Roger; HEISS, Hans-Ulrich. Interfacing SCI Device Drivers to Linux. In: HELLWAGNER, Hermann; REINEFELD, Alexander (Eds.). **SCI:** Scalable Coherent Interface - Architecture and Software for High-Performance Compute Clusters. Berlin: Springer-Verlag, 1999. 490p. p.179-190. (Lecture Notes in Computer Science, n.1734).

[BUY 99] BUYYA, Rajkumar (Ed.). **High performance cluster computing: architectures and systems**. Upper Saddle River: Prentice Hall PTR, 1999. 849p.

[CUL 99] CULLER, David E.; SINGH, Jaswinder Pal; GUPTA, Anoop. **Parallel Computer Architecture:** A Hardware/Software Approach. San Francisco: Morgan Kaufmann, 1999. 1025p.

[DAT 99] DATA GENERAL CORPORATION. **SCI Interconnect Chipset and Adapter:** Building Large Scale Enterprise Servers with Pentium II Xeon SHV Nodes. [S.l.:s.n.], 1999.

[DOR 97] DORMANNS, M.; SPRANGERS, W.; ERTL, H.; BEMMERL, T. A Programming Interface for NUMA Shared-Memory Clusters. In: HIGH PERFORMANCE COMPUTING AND NETWORKING, 1997, Vienna. **Proceedings...** Berlin: Springer-Verlag, 1997. p.698-707. (Lecture Notes in Computer Science, n.1225).

[DOR 98] DORMANNS, M. **SMI - Shared Memory Interface: Reference Manual**. Aachen: RWTH - Lehrstuhl für Betriebssysteme, 1998. 39p.

[EBE 97] EBERL, M.; HELLWAGNER, H.; HERLAND, B.; SCHULZ, M. SISCI — Implementing a Standard Software Infrastructure on an SCI Cluster. In: WORKSHOP CLUSTER COMPUTING, 1., 1997, Chemnitz. **Proceedings...** [S.l.:s.n.], 1997.

[EBE 98] EBERL, Michael et al. Fast Communication Libraries on an SCI Cluster. In: SCI-EUROPE, 1998, Bordeaux. **Proceedings...** [S.l.:s.n.], 1998.

[EIC 92] EICKEN, T. et al. Active Messages: A mechanism for integrated communication and computation. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 19., 1992, Gold Coast. **Proceedings...** [S.l.:s.n.], 1992. p.256-266.

[FIS 97] FISCHER, M.; SIMON, J. Embedding SCI into PVM. In: EUROPEAN PVM/MPI USERS GROUP MEETING, 4., 1997, Cracow. **Proceedings...** Berlin: Springer-Verlag, 1997. p.177-184. (Lecture Notes in Computer Science, n.1332).

[FIS 99] FISCHER, M.; REINEFELD, A. PVM for SCI Clusters. In: HELLWAGNER, Hermann; REINEFELD, Alexander (Eds.). **SCI:** Scalable Coherent Interface - Architecture and Software for High-Performance Compute Clusters. Berlin: Springer-Verlag, 1999. 490p. p.239-248. (Lecture Notes in Computer Science, n.1734).

[FOS 99] FOSTER, Ian T.; KESSELMAN, Carl (Eds.). **The grid:** blueprint for a new computing infrastructure. San Francisco: Morgan Kaufmann, 1999. 677p.

[GEI 94] GEIST, A. et al. **PVM:** Parallel Virtual Machine — A Users Guide and Tutorial for Network Parallel Computing. Boston: MIT Press, 1994.

[GIA 98] GIACOMINI, F. et al. **Low-level SCI software:** Requirements, Analysis and Functional Specification. [S.l.:s.n.], 1998. (ESPRIT Project 23174).

[GIL 96] GILLETT, Richard B. Memory Channel Network for PCI. **IEEE Micro**, Los Alamitos, CA, v.16, n.1, p.12-18, Feb. 1996.

[GON 99] GONZÁLEZ, V.; SANCHIS, E.; TORRALBA, G. Multinode Performance Evaluation: experience using the Dolphin 4-port switch. In: SCI-EUROPE, 1999, Toulouse. **Proceedings...** [S.l.:s.n.], 1999.

[GRO 96] GROPP, W. et al. A high-performance, portable implementation of the MPI message passing interface standard. **Parallel Computing**, Netherlands, v.22, p.789-828, Sept. 1996.

[GUS 94] GUSTAVSON, David B. The Many Dimensions of Scalability. In: COMPCON, 1994. **Proceedings...** [S.l.:s.n.], 1994.

[GUS 96] GUSTAVSON, David B.; LI, Qiang. The Scalable Coherent Interface (SCI). **IEEE Communications Magazine**, New York, v.34, n.8, p.52-63, Aug. 1996.

[HEL 97] HELLWAGNER, H; KARL, W.; LEBERECHT, M. Fast Communication Mechanisms - Coupling Hardware Distributed Shared Memory and User-Level Messaging. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS, 1997, Las Vegas. **Proceedings...** [S.l.:s.n.], 1997.

[HEL 99] HELLWAGNER, Hermann. The SCI Standard and Applications of SCI. In: HELLWAGNER, Hermann; REINEFELD, Alexander (Eds.). **SCI:** Scalable Coherent Interface - Architecture and Software for High-Performance Compute Clusters. Berlin: Springer-Verlag, 1999. 490p. p.3-37. (Lecture Notes in Computer Science, n.1734).

[HER 98] HERLAND, B. G.; EBERL, M.; HELLWAGNER, H. A Common Messaging Layer for MPI and PVM over SCI. In: HIGH PERFORMANCE COMPUTING AND NETWORKING, 1998, Amsterdam. **Proceedings...** Berlin: Springer-Verlag, 1998. p.576-587. (Lecture Notes in Computer Science, n.1401).

[HIP 97] HIPPER, G.; TAVANGARIAN, D. Advanced workstation and cluster architectures for parallel computing. **Journal of Systems Architecture**, [S.l.], v.44, n.3, p.207-226, Dec. 1997.

[HUS 99] HUSE, L. P. et al. ScaMPI - Design and Implementation. In: HELLWAGNER, Hermann; REINEFELD, Alexander (Eds.). **SCI:** Scalable Coherent Interface - Architecture and Software for High-Performance Compute Clusters. Berlin: Springer-Verlag, 1999. 490p. p.249-261. (Lecture Notes in Computer Science, n.1734).

[HUS 99a] HUSE, L. P.; OMANG, K.; BUGGE, H. ScaFun - a fundament for process communication. In: SCI-EUROPE, 1999, Toulouse. **Proceedings...** [S.l.:s.n.], 1999.

[HWA 98] HWANG, Kai; XU, Zhiwei. **Scalable parallel computing:** technology, architecture, programming. Boston: McGraw-Hill, 1998. 802p.

[IBE 96] IBEL, M. et al. Implementing Active Messages and Split-C for SCI Clusters and Some Architectural Implications. In: INTERNATIONAL WORKSHOP ON SCI-BASED HIGH-PERFORMANCE LOW-COST COMPUTING, 6., 1996, Santa Clara. **Proceedings...** [S.l.:s.n.], 1996. p.39-48.

[IEE 93] INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. **IEEE Standard for Scalable Coherent Interface (SCI)**, IEEE 1596-1992. New York, 1993.

[IEE 95] INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. **Information Technology — Portable Operating System Interface (POSIX), Threads Extension [C Language]**, IEEE 1003.1c-1995. New York, 1995.

[IEE 97] INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. **Physical Layer Application Programming Interface for the Scalable Coherent Interface (SCI PHY-API)**, Proposed IEEE 1596.9-1997, Draft Version 0.51. New York, 1997.

[IEE 97a] INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. G**igabit Ethernet**, IEEE P802.3z. New York, 1997.

[LIA 99] LIAAEN, Marius C.; KOHMANN, Hugo. Dolphin SCI Adapter Cards. In: HELLWAGNER, Hermann; REINEFELD, Alexander (Eds.). **SCI:** Scalable Coherent Interface - Architecture and Software for High-Performance Compute Clusters. Berlin: Springer-Verlag, 1999. 490p. p.71-87. (Lecture Notes in Computer Science, n.1734).

[MAN 82] MANDELBROT, B. **The fractal geometry of nature**. New York: W. E. Freeman and Company, 1982.

[MPI 94] MPI FORUM. **The MPI message passing interface standard**. Knoxville: University of Tennessee, 1994.

[MYR 99] MYRICOM INC. **The GM Message Passing System**. 1999. Disponível em: <http://www.myri.com/GM>. Acesso em: out. 1999.

[OMA 97] OMANG, Knut. Synchronization Support in I/O Adapter Based SCI Clusters. In: INTERNATIONAL WORKSHOP ON COMMUNICATION AND ARCHITECTURAL SUPPORT FOR NETWORK-BASED PARALLEL COMPUTING, 1., 1997, San Antonio, TX. **Proceedings...** Berlin: Springer-Verlag, 1997. p.158-172. (Lecture Notes in Computer Science, n.1199).

[PAK 96] PAKIN, S.; LAURIA, M.; CHIEN, A. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In: SUPERCOMPUTING, 1995, San Diego, CA. **Proceedings...** [S.l.]:IEEE Computer Society Press, 1996.

[PFI 98] PFISTER, Gregory F. **In search of clusters**. 2nd ed. Upper Saddle River: Prentice Hall PTR, 1998. 575p.

[PRY 98] PRYLLI, L.; TOURANCHEAU, B. BIP: A New Protocol Designed for High Performance Networking on Myrinet. In: INTERNATIONAL PARALLEL PROCESSING SYMPOSIUM, 12.; SYMPOSIUM ON PARALLEL AND DISTRIBUTED PROCESSING, 9., 1998, Orlando. **Proceedings...** Berlin: Springer-Verlag, 1998. p.472-485. (Lecture Notes in Computer Science, n.1388).

[REH 99] REHLING, E. Sthreads: Multithreading for SCI clusters. In: SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 11., 1999, Natal. **Proceedings...** Natal: SBC, 1999. p.230-234.

[RIC 99] RICHTER, Harald; KLEBER, Richard; OHLENROTH, Matthias. SCI Rings, Switches and Networks for Data Acquisition Systems. In: HELLWAGNER, Hermann; REINEFELD, Alexander (Eds.). **SCI:** Scalable Coherent Interface - Architecture and Software for High-Performance Compute Clusters. Berlin: Springer-Verlag, 1999. 490p. p.125-149. (Lecture Notes in Computer Science, n.1734).

[RYA 96] RYAN, Stein Jørgen; GJESSING, Stein; LIAAEN, Marius. Cluster Communication using a PCI to SCI interface. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED COMPUTING AND SYSTEMS, 8., 1996, Chicago. **Proceedings...** [S.l.:s.n.], 1996.

[RYA 97] RYAN, Stein Jørgen. **The design and implementation of a portable driver for shared memory cluster adapters**. Oslo: Department of Informatics, University of Oslo, 1997. 69p. (Research Report n.255).

[SCA 2000] SCALI AS. **ScaMPI User's Guide**, version 1.9.1. Oslo, 2000. 69p.

[SCH 98] SCHULZ, M. SISCI-Pthreads: SMP-like Programming on an SCI-cluster. In: HIGH PERFORMANCE COMPUTING AND NETWORKING, 1998, Amsterdam. **Proceedings...** Berlin: Springer-Verlag, 1998. p.566-575. (Lecture Notes in Computer Science, n.1401).

[SCH 98a] SCHULZ, M.; HELLWAGNER, H. Global Virtual Memory based on SCI-DSM. In: SCI-EUROPE, 1998, Bordeaux. **Proceedings...** [S.l.:s.n.], 1998. p.59-67.

[SCH 99] SCHULZ, M. SCI-VM: A flexible base for transparent shared memory programming models on clusters of PCs. In: HIGH-LEVEL PARALLEL PROGRAMMING MODELS AND SUPPORTIVE ENVIRONMENTS WORKSHOP, 1999, San Juan. **Proceedings...** Berlin: Springer-Verlag, 1999. (Lecture Notes in Computer Science, n.1586).

[STE 99] STERLING, Thomas L. et al. **How to build a beowulf:** a guide to the implementation and application of PC clusters. Cambridge: MIT, 1999. 239p.

[TAN 92] TANENBAUM, Andrew S. **Modern Operating Systems**. [S.l.]: Prentice-Hall, 1992.

[TAS 98] TASKIN, H. **Synchronisationsoperationen für gemeinsamen speicher in SCI-clustern**. Paderborn: Universität GH Paderborn, 1998. Dissertação de Mestrado.

[TAS 98a] TASKIN, H.; BUTENUTH, R.; HEISS, H. SCI for TCP/IP with Linux. In: SCI-EUROPE, 1998, Bordeaux. **Proceedings...** [S.l.:s.n.], 1998.

[WEI 97] WEIDENDORFER, J. **Entwurf und Implementierung einer Socket-Bibliothek für ein SCI-Netzwerk**. München: Technische Universität München, 1997. Dissertação de Mestrado.

[WOR 99] WORRINGEN, Joachim; BEMMERL, Thomas. MPICH for SCI-connected Clusters. In: SCI-EUROPE, 1999, Toulouse. **Proceedings...** [S.l.:s.n.], 1999. p.3-11.

[WOR 2000] WORRINGEN, Joachim. SCI-MPICH: The Second Generation. In: SCI-EUROPE, 2000, Munich. **Proceedings...** [S.l.:s.n.], 2000.

[YOC 97] YOCUM, Kenneth G. et al. Cut-Through Delivery in Trapeze: An Exercise in Low Latency Messaging. In: IEEE SYMPOSIUM ON HIGH-PERFORMANCE DISTRIBUTED COMPUTING, Aug. 1997, Portland, OR. **Proceedings...** [S.l.:s.n.], 1997.

[ZOR 99] ZORAJA, I.; HELLWAGNER, H.; SUNDERAM, V. SCIPVM: Parallel Distributed Computing on SCI Workstation Clusters. **Concurrency: Practice and Experience**, [S:l.], v.11, n.13, Mar. 1999.