UNIVERSIDADE FEDERAL DE PELOTAS

Bacharelado em Ciência da Computação
Instituto de Física e Matemática
Departamento de Informática

Trabalho Acadêmico

Desenvolvimento de uma biblioteca de serviços de comunicação para um aglomerado de computadores

César Henrique Vortmann

Pelotas, 2008

CÉSAR HENRIQUE VORTMANN

Desenvolvimento de uma biblioteca de serviços de comunicação para um aglomerado de computadores

Trabalho acadêmico apresentado ao Bacharelado em Ciência da Computação da Universidade Federal de Pelotas, como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Prof. Dr. Gerson Geraldo Homrich Cavalheiro

Pelotas, 2008

Dados de catalogação na fonte:
Maria Beatriz Vaghetti Vieira – CRB-10/1032
Biblioteca de Ciência & Tecnologia - UFPel

V957d Vortmann, César Henrique

Desenvolvimento de uma biblioteca de serviços de comunicação para um aglomerado de computadores / César Henrique Vortmann ; orientador Gerson G. H. Cavalheiro. – Pelotas, 2008. – 51f. - Monografia (Conclusão de curso). Curso de Bacharelado em Ciência da Computação. Departamento de Informática. Instituto de Física e Matemática. Universidade Federal de Pelotas. Pelotas, 2008.

1.Informática. 2.Processamento de alto desempenho. 3. Aglomerado de computadores. I.Cavalheiro, Gerson G. H. II.Título.

CDD: 005.275

“The only way of finding the limits of the possible
is by going beyond them into the impossible.”
— ARTHUR C. CLARKE

AGRADECIMENTOS

Primeiramente, agradeço aos meus pais pelo grande apoio e dedicação prestada durante todo o período em que estive a 547Km de distância. Esta distância sempre me pareceu mais curta por causa de vocês.

Agradeço ao meu irmão pela companhia durante esses anos em que estivemos morando sozinhos e longe de casa, não consigo imaginar como seria se tivesse que me virar sem ti.

Agradeço à minha namorada Simone, pelo suporte e estímulo, principalmente nos momentos difíceis encontrados durante a realização do trabalho de conclusão. Sem ti não seria possível!

Agradeço ao professor Lucas pela amizade e pelos conhecimentos compartilhados enquanto estive sob sua “tutela”. Espero que essa amizade continue além do final do curso.

Agradeço ao professor Gerson por sua orientação e pela sua paciência, principalmente por ter que responder minhas dúvidas mesmo tendo que cuidar de seu recém nascido filho.

Agradeço ao professor José Luís Almada Güntzel pelo exemplo de dedicação, seriedade e comprometimento com seu trabalho.

Além destes, gostaria de agradecer todos os professores que tive contato durante a realização do curso, seus ensinamentos foram importantíssimos para que eu atingisse o objetivo de me formar.

Agradeço também a todos os meus colegas, pelas madrugadas em que passamos juntos estudando e fazendo trabalhos, pelas jantas realizadas e pelo companheirismo demonstrado.

Muito obrigado a todos!

RESUMO

Vortmann, César Henrique. **Desenvolvimento de uma biblioteca de serviços de comunicação para um aglomerado de computadores**. 2008. 51 f. Monografia (Bacharelado em Ciência da Computação). Universidade Federal de Pelotas, Pelotas.

O processamento de alto desempenho tem nos aglomerados de computadores um grande aliado, devido ao poder computacional e ao baixo custo destas arquiteturas quando comparadas com supercomputadores. Buscando atingir alto desempenho em aglomerados de computadores pela sobreposição de comunicação com cálculo efetivo desenvolveu-se o modelo de Mensagens Ativas, que faz uso de troca de mensagens assíncronas para troca de informações entre os nós da arquitetura. Entretanto, a proposta original depende de um processador, em hardware, dedicado ao processamento das comunicações. Seu elevado custo, em função do hardware específico, dificulta a aquisição por parte das intituições interessadas. Já a facilidade de aquisição e a capacidade de processamento concorrente dos multiprocessadores permite, juntamente ao uso de multiprogramação leve, a implementação do modelo de Mensagens Ativas em software, criando, assim, uma alternativa de alto desempenho que é independente do hardware utilizado. Neste trabalho desenvolveu-se uma biblioteca de comunicação que implementa o modelo de Mensagens Ativas e que faz uso da multiprogramação leve para a simulação do processador de comunicação. Com uma avaliação de desempenho, mediu-se o desempenho das diversas operações realizadas pela biblioteca para a realização da troca de informações entre os nós da arquitetura e que seu desempenho é favorável a futura inserção no ambiente de programação Anahy.

Palavras-chave: Processamento de Alto Desempenho, Aglomerado de Computadores, Mensagens Ativas, Anahy.

ABSTRACT

Vortmann, César Henrique. **Development of a communication services' library to clusters of computers**. 2008. 51 f. Monography. Universidade Federal de Pelotas, Pelotas.

Cluster of Computers is one of the most popular architecture for high performance computing since its offers high computational power with low cost when compared to traditional supercomputers. In order to reduce the overhead at communication and to achieve effective high performance at execution time, it was proposed the Active Message asynchronous communication model to provide communication support for applications in such kind of architecture. The Active Message model was originally proposed to be used associated with special communication hardware, reducing the adoption of this solution. Nevertheless, it is possible to explore multiprocessor architectures with multithreading to implement the Active Message model in software. This work presents a software library implementing the Active Message model. This library was developed to be threadaware, exploiting multithreading execution to simulate the special communication hardware. A performance assessment of this library shows that this model is suitable to be introduced in the Anahy runtime environment.

Keywords: High Performance Computing, Cluster of Computers, Active Messages, Anahy.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
AVC	Anahy Virtual Machine
FSB	Front Side Bus
MA	Mensagens Ativas
MPI	Message Passing Interface
PAD	Processamento de Alto Desempenho
PVM	Parallel Virtual Machine
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RSR	Remote Service Request
SPMD	Single Program Multiple Data
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

SUMÁRIO

1 INTRODUÇÃO

Muitas configurações de hardware para o Processamento de Alto Desempenho (PAD) são apoiadas em arquiteturas com memória distribuída. Este é o caso de agregados de computadores e constelações. O TOP 500 (TOP500.ORG, 2008) mostra, no seu último levantamento, que, dentre as 500 máquinas mais potentes, mais de 80% pertencem a estas classes de arquiteturas. Entre os fatores que contribuem para o grande número de aglomerados de computadores presentes nesta lista estão sua facilidade de aquisição e o ganho computacional proporcionado quanto se compara este tipo de arquitetura com supercomputadores (UNDERWOOD; SASS; LIGON III, 2001).

Ferramentas clássicas utilizadas para programação paralela e distribuída nestas arquiteturas empregam o modelo de troca de mensagens, síncronas ou assíncronas. Tais ferramentas são mais voltadas a implementação de sistemas do que ao desenvolvimento de software propriamente dito. Outros recursos mais elaborados, como *Remote Procedure Call* (RPC) ou *Remote Method Invocation* (RMI) oferecem níveis de abstração mais elevados, entretanto, não possuem foco em alto desempenho. O padrão *Message Passing Interface* (MPI), por sua vez, oferece uma abstração de alto nível para manipulação de mensagens, sendo muito popular no contexto do PAD.

Quando o objetivo é obter PAD em aglomerados de computadores, os custos de comunicação devem ser minimizados. O modelo de Mensagens Ativas (MA) propõe a realização da comunicação com um custo pequeno. Este modelo foi proposto (EICKEN et al., 1992) para ser utilizado com hardware específico, empregando um processador dedicado à comunicação. A popularização de multiprocessadores, com o advento tecnológico dos processadores multi-core permite implementar este modelo em software, com o uso de multiprogramação leve para simular o processador de comunicação, utilizando a capacidade de processamento concorrente disponível além de permitir uma implementação independente do hardware específico antes utilizado.

Este trabalho propõe o desenvolvimento e a implementação de uma biblioteca de serviços de comunicação em software para ambientes com memória distribuída, baseado no modelo de Mensagens Ativas, que seja eficiente e que permita sua utilização independente do hardware utilizado.

1.1 Motivação

A popularização dos aglomerados de computadores como alternativa para suporte ao processamento de alto desempenho gerou uma necessidade de exploração do potencial destas arquiteturas, devido ao grande poder computacional gerado e pela sua facilidade de aquisição. Para isto, faz-se necessário o desenvolvimento de ferramentas de programação eficientes e que atendam às expectativas dos programadores em termos de facilidade de uso. Como constatação deste fato pode-se citar a não identificação de um modelo que seja universalmente aceito para programação paralela (ZOMAYA, 1996).

Outra motivação é a possibilidade de inserção da biblioteca desenvolvida no ambiente de programação paralela Anahy, permitindo assim a extensão deste ambiente para arquiteturas com memória distribuída, e permitindo que estas desfrutem do mecanismo de escalonamento presente em Anahy, capaz de controlar a execução de um programa paralelo segundo suas dependências de dados expressas por suas atividades.

1.2 Objetivos

O objetivo principal deste trabalho foi o desenvolvimento e a implementação de uma biblioteca de MA em software, explorando arquiteturas multiprocessadas para oferecer uma alternativa de alto desempenho para comunicação em aglomerados de computadores. Este objetivo foi atingido, e a biblioteca está disponível para sua futura inserção no ambiente Anahy. Deve-se destacar os objetivos específicos atingidos com a realização deste trabalho:

- Disponibilização de uma biblioteca para realização de comunicação entre os nós de um aglomerado de computadores;
- Avaliação de desempenho da biblioteca implementada, permitindo verificar sua eficiência em relação aos resultados esperados.

1.3 Resultados Alcançados

Os objetivos perseguidos por este trabalho foram atingidos com a implementação da biblioteca de comunicação. A biblioteca encontra-se disponível para inserção

no ambiente Anahy. Uma avaliação de desempenho permitiu verificar que a biblioteca atingiu os níveis de performance esperado, além de permitir o avanço do projeto Anahy com sua futura inserção no mesmo, extendendo este ambiente para arquiteturas com memória distribuída.

1.4 Estrutura do texto

O restante desta monografia está organizado como segue:

O Capítulo 2 descreve conceitos a respeito da programação paralela e distribuída, dissertando sobre a arquitetura utilizada em aglomerados de computadores, sobre como é realizada a comunicação entre seus nós, além de explanar conceitos sobre a multiprogramação leve.

O Capítulo 3 apresenta o conceito de Mensagens Ativas, conceito este muito importante para o trabalho, já que o mesmo propõe seu desenvolvimento em software a partir da proposta em hardware. Juntamente com os conceitos de interrupção e *polling* são apresentadas implementações alternativas à original, que buscam corrigir ou amenizar os problemas encontrados nesta. Além disso, ainda são apresentados alguns trabalhos que fazem uso do conceito de Mensagens Ativas na realização da comunicação entre os nós de um aglomerado.

O Capítulo 4 descreve a implementação da biblioteca de comunicação, como utilizá-la, como foi simulado o processador de comunicação proposto em hardware na proposta original de Mensagens Ativas e as primitivas implementadas, além de apresentar um pequeno exemplo de uso.

No Capítulo 5 são apresentados os resultados da avaliação de desempenho das operações elementares realizadas pela biblioteca de comunicação implementada.

Finalmente, no Capítulo 6 são apresentadas as conclusões retiradas do trabalho, e expectativas para trabalhos futuros.

2 PROGRAMAÇÃO PARALELA E DISTRIBUÍDA

Matemática, Física, Biologia e Química têm sido as principais áreas quando se fala em demanda por poder computacional, devido principalmente aos complexos problemas apresentados.

Na busca de soluções, o primeiro pensamento é o de utilização de supercomputadores com hardware e software especializados para resolução de cada um dos problemas apresentados. No entanto, adquirir, a baixo custo, computadores de grande porte não é uma realidade com as tecnologias atuais. Devido às condições financeiras da grande maioria das instituições onde são desenvolvidas pesquisas nas áreas citadas, e os grandes investimentos necessários a aquisição e manutenção de supercomputadores, esta solução torna-se, portanto, impraticável.

Surgem então como alternativa, os aglomerados de computadores (Clusters of Computers). Aglomerados caracterizam-se por serem sistemas de computação paralela, constituídos por um conjunto de computadores independentes que trabalham de forma integrada, como um recurso computacional único (BUYYA, 1999). O crescimento no uso de aglomerados de computadores pode ser observado através das estatísticas apresentadas no *site* www.top500.org (TOP500.ORG, 2008), que classifica semestralmente as 500 máquinas mais poderosas do mundo e onde se pode notar que a presença de aglomerados chega a cerca de 80%.

Após ter-se disponível tal arquitetura, dotada de múltiplos recursos de processamento, observa-se a necessidade de desenvolvimento de uma forma de explorar eficientemente esta arquitetura, utilizando-se do paradigma programação concorrente. Este paradigma, também chamado de programação paralela ou distribuída, caracteriza-se por descrever a aplicação sob forma de vários fluxos de execução independentes.

Este capítulo tem por objetivo apresentar tópicos relacionados a programação paralela e distribuída. A Seção 2.1 diz respeito aos componentes da arquitetura de uma aglomerado de computadores e suas formas de exploração. A Seção 2.2

destaca algumas ferramentas de comunicação empregadas na troca de dados e tarefas entre os nós de um aglomerado de computadores. E por fim, na seção 2.3 é apresentado o conceito de multiprogramação leve, utilizada para exploração da concorrência intra-nó.

2.1 Arquitetura de um Aglomerado

Pode-se definir um aglomerado de computadores como sendo um grupo de computadores, chamados de nós, os quais podem ser multiprocessados ou não. Estes nós são conectados entre si, trabalhando cooperativamente de modo a formar uma única unidade computacional. Seu objetivo principal é o compartilhamento de recursos, sendo que o ganho de desempenho com o uso da programação concorrente é um dos seus usos mais freqüentes. Durante a execução de uma aplicação, dependendo da arquitetura dos nós, podem ser explorados dois níveis concorrência, a concorrência intra-nó e a concorrência entre-nós, que diferem somente na forma como é realizada a interação entre os diferentes fluxos de execução.

A concorrência intra-nó é aquela que ocorre internamente aos nós do aglomerado e tem como principal característica fluxos de execução que compartilham um mesmo espaço de endereçamento, sendo a troca de dados realizada com apoio desta área em comum. Deve-se ter atenção especial para garantir a sincronização dos dados tendo em vista que a comunicação é feita através de leituras e escritas concorrentes na memória.

Existem para isso diversos mecanismos, entre eles *mutexes*, mecanismos de controle no avanço da execução por meio de primitivas *create* e *join*. As ferramentas mais populares para a exploração da concorrência intra-nó baseiam se no padrão POSIX (CORPORATE IEEE; ELECTRONICS ENGINEERS, 1994) para processos leves (ou threads).

Já a concorrência entre-nós é aquela que ocorre entre os nós do aglomerado e tem como principal característica fluxos de execução que necessitam algum mecanismo de comunicação, como por exemplo, troca de mensagens, para a troca de informações. Neste nível de concorrência pode-se dizer que o paralelismo real da arquitetura esté sendo explorado, pois não há competição pelos recursos de processamento de um nó específico. Já que os fluxos de execução são tarefas independentes, capazes de executarem paralelamente, serão executados em nós diferentes da arquitetura, não competindo assim, pelos mesmos recursos. Mesmo envolvendo tarefas independentes, pode ser necessário algum tipo de sincronização ou troca de dados entre esses diferentes fluxos de execução, mas diferentemente da concorrência intra-nó,

essa sincronização não pode ser feita em um espaço de endereçamento comum, já que cada nó possui seu próprio espaço de endereçamento. Utiliza-se então a rede que conecta os nós para a execução da aplicação, caracterizando uma *aplicação distribuída*, já que a memória encontra-se distribuída entre nós da arquitetura.

Como a memória encontra-se distribuída entre os nós, a utilização de primitivas de escrita e leitura não faz mais sentido, passando-se a utilizar primitivas do tipo *envia* e *recebe*. As tarefas agora interagem pela da troca de mensagens, forma de comunicação muito mais complexa que a memória comum às tarefas, pois envolve o uso de mecanismos de endereçamento, e que, além disso, é muito mais onerosa. Quando uma tarefa faz uma chamada a primitiva *envia*, a tarefa pode ser considerada terminada e os dados produzidos como resultado serão enviados a tarefa receptora. Já quando é realizada uma chamada à primitiva *recebe*, a tarefa chamadora é considerada terminada e as instruções após o *recebe* definem a próxima tarefa, que estará apta a executar quando receber os dados necessários ao início de sua execução.

Entre as ferramentas encontradas na literatura que possibilitam o compartilhamento de dados por meio da troca de mensagens, podem ser citadas *Parallel Virtual Machine* (PVM) (GEIST et al., 1994) e *Message Passing Interface* (MPI) (FORUM, 1994), além do padrão sockets (STEVENS, 1997).

Para uma melhor exploração dos recursos oferecidos por um aglomerado de computadores, deve-se explorar tanto a concorrência intra-nó como a concorrência entre-nós. Entretanto, a utilização simultânea destes dois mecanismos não é uma tarefa simples (CARISSIMI, 1999). Pode-se justificar o uso dos dois níveis de concorrência pela possibilidade da sobreposição de parte do tempo gasto com comunicação entre-nós por cálculo efetivo (VALIANT, 1990), extendendo-se a idéia de que um processo deve perder o processador para outro processo ao realizar uma operação de E/S. Podem ser citadas várias ferramentas que permitem a exploração dos dois níveis de concorrência, entra elas as bibliotecas Athapascan (CARISSIMI, 1999), Nexus (FOSTER; KESSELMAN; TUECKE, 1996) e *PM*^{2} (DENNEULIN; NAMYST; MéHAUT, 1997), além da linguagem de programação Java (DEITEL; DEITEL, 2004).

2.2 Comunicação em Aglomerados

Dados produzidos por determinada tarefa em um nó poderão ser consumidos por outras tarefas em outros nós, durante a execução de uma aplicação em um aglomerado de computadores. Esta operação, comunicação entre tarefas executando em nós distintos, é uma das mais custosas, quando se fala em

processamento concorrente em arquiteturas com memória distribuída. Devido a necessidade de processamento de alto desempenho, deve-se reduzir o tempo desde a produção de determinado dado até seu consumo.

Portanto, o desempenho da aplicação está diretamente ligado ao custo introduzido pela comunicação entre os nós do aglomerado. Devido a isso, têm sido desenvolvidos tanto protocolos mais eficientes de comunicação quanto hardware com melhor desempenho, com o objetivo, sempre, de explorar mais eficientemente os recursos oferecidos pelas redes de comunicação além de reduzir o custo das comunicações.

O restante desta seção dedica-se a apresentar algumas ferramentas de comunicação que podem ser utilizadas para a troca de dados entre os nós de um aglomerado.

2.2.1 Troca de Mensagens

Troca de mensagens é a estratégia empregada, tradicionalmente, para a realização do compartilhamento de dados entre os nós de um aglomerado de computadores. São utilizadas operações do tipo *send* para envio de dados e *receive* para recebimento de dados. Existem variantes desta estratégia, as quais permitem comunicações tanto síncronas quanto assíncronas.

No caso síncrono, *send* e *receive* são operações bloqueantes, ou seja, uma chamada *send* permanece bloqueada até que o *receive* correspondente seja executado. Neste caso, o transmissor somente enviará os dados após receber do receptor uma confirmação de que este encontra-se pronto para a recepção. Então, utilizando-se da troca de mensagens síncronas, a realização, em paralelo, de processamento e comunicação, é impossível (EICKEN et al., 1992).

Por outro lado, a variante assíncrona permite que parte do tempo gasto com comunicações seja sobreposto por cálculo efetivo pois o transmissor de uma mensagem não necessita esperar pelo requerimento da mensagem por parte do receptor ou pela confirmação de recebimento desta mensagem. Após o envio de uma mensagem, o transmissor estará livre tanto para realizar novas transmissões quanto para a realização de cálculo efetivo. Entretanto, o uso desta técnica implica em uma maior complexidade no controle da evolução do programa e, consequentemente, na troca de dados.

Duas das mais populares bibliotecas de comunicação baseadas em troca de mensagens para processamento de alto desempenho são PVM (GEIST et al., 1994) e MPI (FORUM, 1994). Pode-se citar, também, o padrão sockets (STEVENS, 1997), bastante popular em ambientes UNIX.

2.2.1.1 PVM

A biblioteca PVM (GEIST et al., 1994) permite transformar uma rede de computadores heterogêneos em única máquina virtual para a execução de aplicações paralelas e distribuídas. A máquina virtual pode ser composta por um número praticamente ilimitado de hosts heterogêneos (COSTA; STRINGHINI; CAVALHEIRO, 2002).

O sistema PVM é composto de duas partes: uma biblioteca de funções paralelas e um processo que roda em segundo plano (daemon), chamado pvmd3, e que reside em todos os computadores que fazem parte da máquina virtual. A biblioteca é responsável por fornecer primitivas de comunicação, inicialização e término de processos PVM além de adição e remoção de hosts da máquina virtual, sincronização e envio de sinais entre processos PVM. Já o daemon é responsável pelo gerenciamento das tarefas paralelas oferecidas pelo PVM. O conjunto de daemons executando em diferentes máquinas, forma a máquina virtual do PVM (COSTA; STRINGHINI; CAVALHEIRO, 2002).

Qualquer processo PVM pode enviar uma mensagem para qualquer outro processo PVM, não havendo limite para o tamanho nem para o número destas mensagens. Esta comunicação entre processos é baseada em operações *send* e *receive* assíncronas além de recepção síncrona.

2.2.1.2 MPI

A biblioteca MPI é uma biblioteca de comunicação para programação paralela baseada em troca de mensagens, que surgiu durante o MPI Fórum, organizado por laboratórios, indústrias e universidades, com o objetivo de definir um conjunto de primitivas relacionadas a sincronização e comunicação entre tarefas.

Assim como PVM, MPI também cria uma máquina virtual sobre uma arquitetura real. Cada nó virtual consiste em um processo executando sobre um nó real da arquitetura, sendo que mais de um nó virtual poderá residir no mesmo nó real e nós virtuais poderão estabelecer comunicação para envio e recebimento de dados quando necessário.

Além das primitivas para troca de dados entre processos, que vão desde as tradicionais *send* e *receive* até mecanismos de comunicação em grupo, MPI ainda permite que um processo envie dados para todos os outros processos assim como receba dados de todos os outros processos.

2.2.1.3 Sockets

Bibliotecas sockets são populares em ambientes UNIX, tal como o ambiente GNU/Linux, e sua implementação segue o padrão sockets definido em (STEVENS, 1997). É o mecanismo mais básico utilizado para comunicação

interprocesso em um computador ou entre aplicações em diferentes computadores conectados por redes locais ou redes de longa distância. Possui um conjunto pequeno de primitivas e pode ser empregado tanto com o protocolo Transmission Control Protocol (TCP) (orientado a conexão) quanto com User Datagram Protocol (UDP) (orientado a pacotes), sendo muito eficiente em redes padrão Ethernet.

2.2.2 Chamada Remota de Procedimento

Outra estratégia a ser empregada para comunicação em aglomerados de computadores consiste em chamadas remotas de procedimento (*Remote Procedure Call* - RPC) (BIRRELL; NELSON, 1984), as quais oferecem um maior nível de abstração ao programador. RPC oferece ao programador uma estrutura de programação bastante próxima a chamada de um procedimento ordinário para invocar um serviço a ser executado em outro nó (CAVALHEIRO, 2004), passando, como parâmetro, um conjunto de dados.

Um ambiente que implementa RPC é composto por um servidor, responsável pelo cálculo, um cliente, que invoca os serviços do servidor, e um *stub*, que é executado junto ao cliente e consiste em um procedimento responsável por capturar as invocações ao serviço remoto.

A popularidade de RPC, originalmente síncrona, deve-se ao emprego das conhecidas chamadas de procedimento, onde serviços remotos são vistos como procedimentos locais. Além disso, este modelo, onde o cliente espera sincronamente pelo servidor, é bem próximo ao esquema sequencial conhecido pelos programadores (BARCELLOS; GASPARY, 2003).

2.2.3 Mensagens Ativas

Mensagens Ativas (EICKEN et al., 1992) surgiram como soluções clássicas ao problema de comunicação em ambientes paralelos (LUMETTA; MAINWARING; CULLER, 1997), tendo como objetivo a realização das comunicações sem introdução de grande quantidade de sobrecusto de execução (WALLACH et al., 1995). O mecanismo de comunicação é assíncrono, sendo que cada mensagem enviada carrega, em seu cabeçalho, o endereço de um serviço que será executado no receptor e, em seu corpo, os dados correspondentes aos parâmetros desde serviço (EICKEN et al., 1992). No momento da recepção de uma mensagem ativa, um procedimento é acionado para recuperar a mensagem da rede e inserí-la no processamento em andamento no nó, de forma que o serviço requisitado execute rapidamente e por completo.

Este mecanismo emprega a estratégia de bufferizar somente o necessário pelo sistema operacional das Mensagens Ativas, tentando minimizar a sobrecarga de comunicação.

Apesar das semelhanças entre Mensagens Ativas e RPC, cabe destacar que as primeiras foram desenvolvidas com o objetivo de obterem alto desempenho em arquiteturas distribuídas e usam para isso um processador dedicado à comunicação, enquanto as outras preocupam-se apenas com a comunicação em arquiteturas deste tipo.

2.3 Multiprogramação Leve

A multiprogramação leve (multithreading) é a ferramenta comumente utilizada para explorar a concorrência intra-nó. Esta estratégia permite que vários fluxos de execução sejam criados no interior de um processo. Cada um destes fluxos recebe o nome de processo leve ou thread. O termo leve refere-se ao compartilhamento de recursos, alocados a um processo, por todas suas threads ativas (VAHALIA, 1996), não sendo necessário que cada thread possua sua própria descrição de recursos. Isto torna a manipulação de threads menos onerosa ao sistema operacional.

Um dos recursos compartilhados pelos threads é a memória do processo, que serve como base de comunicação e troca de dados, sendo acessada com primitivas de leitura e escrita. Neste caso, o problema está em garantir um acesso correto a esta memória compartilhada, sendo necessário o emprego de algum mecanismo de sincronização para este fim.

3 MENSAGENS ATIVAS

O desenvolvimento de software que tire o máximo proveito de arquiteturas como os aglomerados de computadores, apresenta três desafios: (1) minimizar o sobrecusto gasto em comunicações, (2) permitir que o tempo gasto em comunicações seja sobreposto por computação (cálculo efetivo) e (3) coordenar os dois anteriores sem sacrificar o desempenho do processador (EICKEN et al., 1992).

Uma aplicação implementada para executar em arquiteturas como aglomerados e que vise alto desempenho, deve buscar minimizar o tempo entre a produção de um dado e seu consumo. Este tempo, que pode ser visto como a reatividade da aplicação, refere-se à capacidade da aplicação em disparar novos serviços a medida que novos dados são produzidos. Considerando-se a execução em um aglomerado de computadores, um dado produzido em um nó do aglomerado por uma tarefa poderá ser consumido por outra tarefa em um nó diferente. Para tanto, será necessária uma troca de mensagens entre os dois nós, a qual deverá ocorrer de maneira rápida, já que o dado produzido deve ser consumido no menor tempo possível.

Uma solução eficiente para implementação de troca de mensagens entre os nós de um aglomerado, consiste na utilização de Mensagens Ativas (EICKEN et al., 1992), que, normalmente, está entre os mecanismos de comunicação mais rápidos disponíveis (LUMETTA; MAINWARING; CULLER, 1997). Este mecanismo é assíncrono e tenta explorar toda a flexibilidade e desempenho das modernas redes de computadores (EICKEN et al., 1992), permitindo realizar comunicações sem introduzir grande quantidade de sobrecustos de execução (WALLACH et al., 1995).

Neste mecanismo, cada mensagem enviada contém, em seu cabeçalho, o endereço de uma função a ser executada no receptor e, em seu corpo, os dados que compõem os argumentos (LUMETTA; MAINWARING; CULLER, 1997). A idéia é que o transmissor envie uma mensagem para a rede e continue seu processamento. A recepção de uma mensagem ativa provoca o acionamento de

um procedimento para recuperar a mensagem da rede e a invocação do serviço desejado. Para minimizar o sobrecusto de comunicação, as Mensagens Ativas são bufferizadas apenas o necessário para o transporte na rede.

A diferença entre Mensagens Ativas e RPC, está no fato de que o mecanismo de Mensagens Ativas busca extrair os dados da rede o mais rápido possível e executar o processamento requisitado pela mensagem, de forma a minimizar o tempo entre a produção e o consumo dos dados. A grande vantagem deste mecanismo está na forma como a mensagem recebida é tratada: ela é inserida dentro do fluxo de execução corrente, o que evita o sobrecusto de criação de um novo fluxo (CARISSIMI, 1999).

Este capítulo apresenta um estudo sobre o mecanismo de Mensagens Ativas. A Seção 3.1 apresentada a implementação original proposta (EICKEN et al., 1992). Já a seção 3.2 ilustra algumas implementações alternativas que buscam minimizar as limitações do modelo original. Na Seção 3.3 são apresentados alguns ambientes de programação que empregam o mecanismo de Mensagens Ativas.

3.1 Implementação Original

O modelo original de Mensagens Ativas foi desenvolvido em hardware e apresentado em (EICKEN et al., 1992), e podia funcionar de duas maneiras: por interrupção ou por *polling*, como pode ser visto nas Fig. 1 e Fig. 2, onde a Fig. 1 representa a estratégia de interrupção e a Fig. 2, a estratégia de *polling*, respectivamente.

Na política de interrupção, a interface de rede interrompe o thread em execução, avisando da chegada da mensagem (1). Esta interrupção é tratada (2) e o descritor da mensagem é retirado da interface de rede (3). É então iniciado o tratamento da mensagem (4), retirando-se os dados da rede (5) e armazenando-se em uma área de dados.

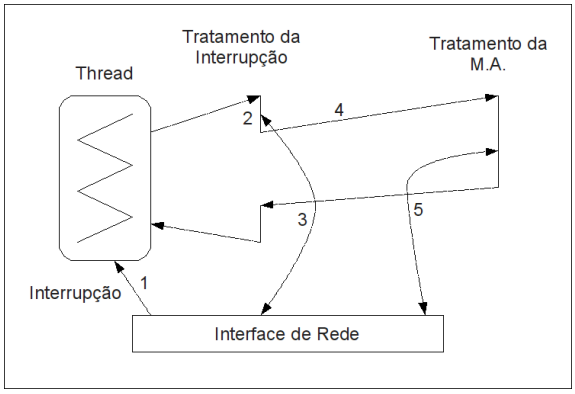

Figura 1: *Política de interrupção na implementação original de Mensagens Ativas*

Já na política de *polling*, o thread em execução consulta, de forma periódica, a interface de rede, com o objetivo de verificar a chegada de novas mensagens. Caso ocorra a chegada de uma mensagem, seu descritor será retirado da rede (1) e o tratamento da mensagem ativa será disparado (2). Após isso, os dados necessários a execução da tarefa são retirados da rede (3).

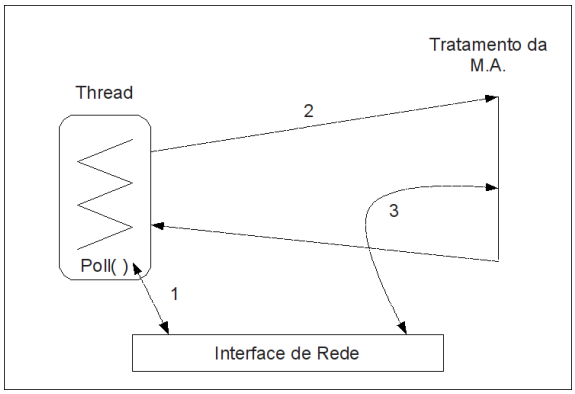

Figura 2: *Política de polling na implementação original de Mensagens Ativas*

Entretanto, estas políticas apresentam um sério problema, por possuírem um único thread no receptor. Esse thread é responsável pelo recebimento e tratamento das Mensagens Ativas, não podendo realizar operações de sincronização, pois isto poderia resultar em uma situação de *deadlock*.

3.2 Implementações Alternativas

As restrições impostas ao uso de sincronizações pelo modelo original de Mensagens Ativas não é interessante ao desenvolvimento de aplicações. Como solução ao problema relatado, foram desenvolvidas algumas variações do modelo original.

O restante da seção apresentará 3 variações do modelo original de Mensagens Ativas. São eles: UpCall, onde existe um thread específico para a comunicação; PopUp, que trabalha com vários threads; e por fim Fila de Execução (PERANCONI, 2005).

3.2.1 Upcall

Nesta variação todas as mensagens recebidas são tratadas por um único thread, o qual é chamado de *daemon* de comunicação. Esta forma de encarar o problema da recepção das mensagens permite que o thread executando no momento do recebimento de uma mensagem continue seu processamento, sem parar para o tratamento da mensagem (CARISSIMI, 1999). Na Fig. 3 pode-se conferir o funcionamento do mecanismo de UpCall para a política de interrupção

e na Fig. 4 a política de *polling*.

Na política de interrupção, a chegada de uma mensagem gera uma interrupção (1), que acionará o escalonador. Ocorrerá então a perda do processador por parte do processo em execução (2) e o processador será atribuído ao *daemon* de comunicação (3). É então feita a retirada da mensagem da interface de rede (4) e seu tratamento é iniciado (5).

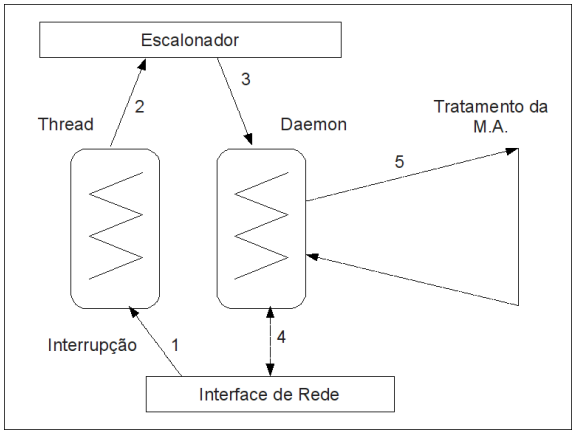

Figura 3: *Política de interrupção na implementação UpCall de Mensagens Ativas*

Já quando se utiliza da política de *polling*, o *daemon* verificará a interface de rede constantemente, com o objetivo de detectar a chegada de novas mensagens. Para que isso seja possível, o escalonador deve ceder o direito de execução ao *daemon* (1). Assim que é detectada a chegada de uma nova mensagem, esta é retirada da interface de rede (2) e tratada (3).

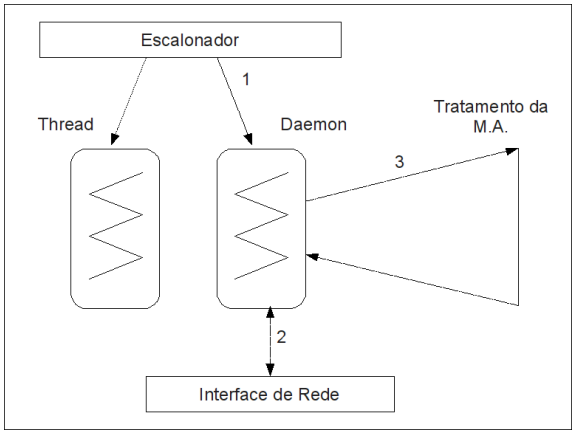

Figura 4: *Política de polling na implementação UpCall de Mensagens Ativas*

Nesta variação, diferentemente do que acontece com a implementação original, os threads são independentes, permitindo assim a realização de sincronizações. Entretanto, a existência de um thread único para tratamento das mensagens, em cada nó, faz com que a reatividade ao atendimento de mensagens recebidas seja comprometida.

3.2.2 PopUp

Nesta variação, cada mensagem recebida gera a criação de um novo thread para o seu tratamento. A vantagem mais clara nessa maneira de tratar a mensagem é o aumento da reatividade da aplicação, já que os dados são consumidos assim que recebidos, além de permitir que o thread em execução não interrompa sua execução para retirar dados da mensagem da interface de rede (CARISSIMI, 1999). Como no caso anterior, esse mecanismo pode funcionar tanto por interrupção quanto por *polling* (Fig. 5 e Fig. 4).

Na política de interrupção, a chegada de uma mensagem gera uma interrupção (1), que será tratada pelo thread atualmente em execução (2). É então retirado da rede o cabeçalho da mensagem (3), e dá-se a criação de uma nova thread para o tratamento da mensagem (4). Esse novo thread retirará, conforme necessário, os dados que foram repassados como parâmetros na mensagem (5).

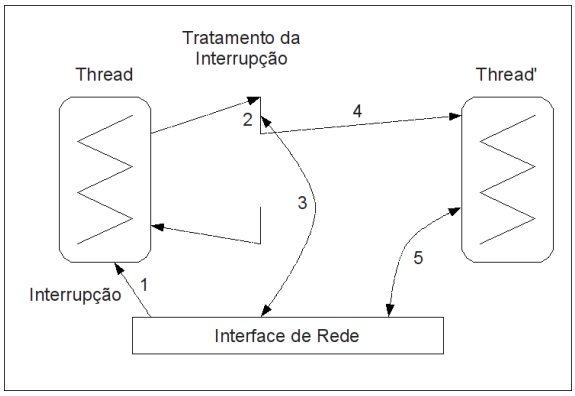

Figura 5: *Política de interrupção na implementação PopUp de Mensagens Ativas*

Já quando utiliza-se da política de *polling*, o thread ocupando o processador atualmente verificará a interface de rede constantemente, com o objetivo de detectar a chegada de novas mensagens (1). Ocorrendo esse recebimento, o thread atual cria um novo thread para tratamento da mensagem (2). A retirada da mensagem da interface de rede e seu devido tratamento é realizado pelo novo thread (3).

Nesta variação, apesar de resolver tanto os problemas encontrados na implementação original, dizendo respeito à sincronização, já que os threads continuam independentes, e em *UpCall*, diminuindo o tempo entre a produção e o consumo dos dados através da criação de um novo thread para tratamento das mensagens e, portanto, aumentando a reatividade da aplicação, acaba por introduzir custos adicionais de criação de novos threads a cada recebimento de novas mensagens.

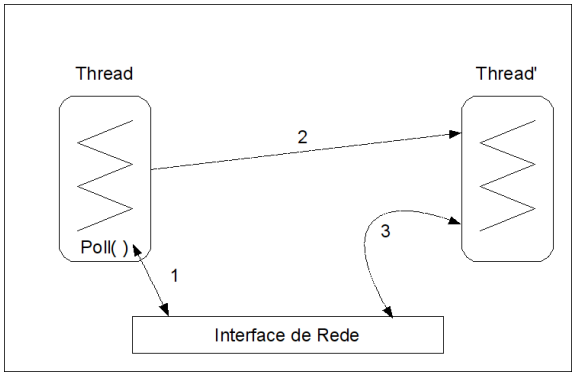

Figura 6: *Política de polling na implementação PopUp de Mensagens Ativas*

3.2.3 Fila de Execução

No modelo original de Mensagens Ativas havia a possibilidade de ocorrência de *deadlock* se a mensagem sendo processada realizasse sincronização. O modelo *UpCall* resolve este problema disponibilizando um thread específico para o tratamento de mensagens. O modelo *PopUp* resolve tanto os problemas de *deadlock* quanto de reatividade, através da criação de um novo thread para tratamento de cada mensagem recebida, ao preço de adicionar os custos de criação de threads na aplicação. Concluí-se daí que nenhum dos modelos anteriores é significativamente interessante quando se tem aplicações altamente paralelas.

Um modelo híbrido, que unisse características dos modelos *UpCall* e *PopUp*, onde existiría um *daemon* responsável por retirar as mensagens da interface de rede e onde não fosse necessário a criação de novos threads para execução de vários serviços paralelos, foi proposto (ROLOFF; CARISSIMI; CAVALHEIRO, 2004), inicialmente, fazendo uso de processadores virtuais e listas de tarefas, esse modelo foi concebido para inserção no modelo de execução do ambiente Anahy (CAVALHEIRO; DALL'AGNOL; REAL, 2003).

Este modelo caracteriza-se por possuir um número *n* de threads executando as tarefas definidas pela aplicação, em cada nó, mas, principalmente, porque ao criar-se uma tarefa, esta é inserida em uma lista de tarefas (fila de execução). Os threads verificam a lista de tarefas após concluírem suas tarefas atuais, e no caso de haver alguma tarefa pronta para ser executada, esta é tomada pelo thread. Caso não haja tarefa na lista, o thread é bloqueado à espera de uma nova tarefa. Quando uma mensagem é recebida, ela é automaticamente inserida na lista de tarefas, e aguarda até que um thread encarregue-se de sua execução.

Na Fig. 7 pode-se visualizar esse modelo. O *daemon* de comunicação é responsável por verificar a rede constantemente com o objetivo de verificar a chegada de mensagens (1). Ao detectar alguma mensagem, o *daemon* retira-a

da interface de rede, armazena-a na lista de tarefas, e volta a verificar a interface de rede (2). Toda vez que o thread em execução acaba seu processamento, uma verificação à lista de tarefas é feita, com o objetivo de obter alguma tarefa pronta para processamento (3). Caso haja tarefa, esta é executada, e caso contrário, o thread é bloqueado, até que nova tarefa seja inserida na lista de tarefas.

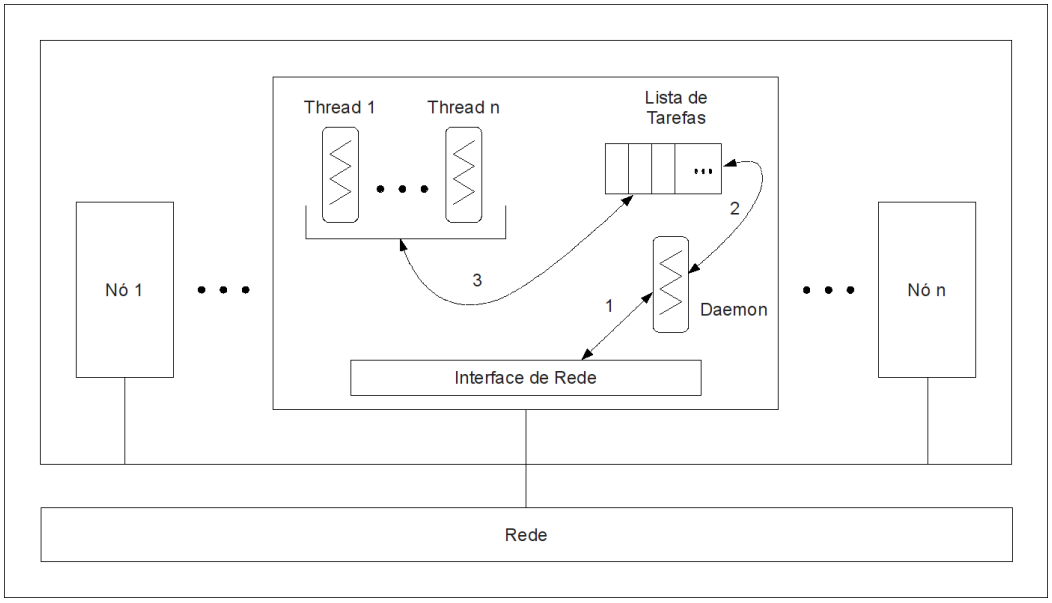

Figura 7: *Modelo Fila de Execução de Mensagens Ativas*

3.3 Trabalhos Selecionados

Esta seção apresenta alguns ambientes de programação paralela e distribuída que fazem uso de Mensagens Ativas para a comunicação entre os nós da arquitetura.

3.3.1 Split-C

Split-C (KRISHNAMURTHY et al., 1993) é uma extensão, paralela, da linguagem de programação C, possuindo suporte a um espaço de endereçamento global em uma arquitetura com memória distribuída. Segue um modelo de programação, conhecido como *Single Program Multiple Data* (SPMD), onde todos os processadores iniciam a execução em um ponto comum do código, mas podem seguir fluxos de execução distintos e onde os processadores realizam sincronizações através do uso de barreiras.

Em Split-C os processadores podem acessar qualquer endereço em um espaço de endereçamento global e possuem uma área específica, que pode ser acessada por meio de ponteiros locais padrão da linguagem C, neste espaço de endereçamento global. O acesso à área de endereçamento global é feito, assim

como o acesso à área local de cada processador, através do uso de ponteiros padrão da linguagem C, que, neste caso, serão ponteiros globais.

Mensagens Ativas, em Split-C, é um recurso de implementação do próprio ambiente, ao qual o programador não tem acesso. Os serviços da camada de comunicação são disponibilizados pela interface *SP AM* (CHANG et al., 1996), a qual manipula o hardware de comunicação diretamente.

3.3.2 Chant

Chant (HAINES; CRONK; MEHROTRA, 1994) é um ambiente de programação paralela desenvolvido para sistemas com memória distribuída e tem como principal característica o conceito de *talking threads*. *Talking threads* pode ser entendido como um conceito onde dois threads realizam comunicação direta entre si através do uso de primitivas do tipo *send*( ) e *receive*( ), independentemente do espaço de endereçamento onde se encontram.

Através de uma extensão da interface e das funcionalidades providas pelo padrão POSIX para threads (Pthreads) (CORPORATE IEEE; ELECTRONICS ENGINEERS, 1994), Chant defina uma categoria de objetos computacionais definidos como *chanters*, que nada mais são do que threads com identificadores únicos dentro da máquina paralela e que podem comunicar-se com outros threads. Ao invés da utilização da primitiva *pthread_create*( ) para criação de um *thread chanter* deve-se usar a primitiva *pthread_chanter_create*( ).

São utilizadas primitivas tipo *send* e *receive*, no padrão MPI, para a passagem de parâmetros e retorno de dados entre dois thread comunicantes, podendo haver também o envio síncrono e o recebimento tanto síncrono como assíncrono de mensagens. Chant também provê um mecanismo de *Remote Service Request* (RSR), o qual permite a execução remota de funções, para quando a troca de mensagens entre *chanters* não pode ser realizada.

O suporte à comunicação disponibilizado por Chant permite de que sejam feitas tanto troca de dados ponto-a-ponto entre threads quanto a invocação remota de threads. A invocação remota de threads é possível através da utilização, por parte do ambiente de execução, de um *server thread*. A responsabilidade do *server thread* é aguardar o recebimento de uma mensagem solicitando um serviço, e executá-lo assim que o recebimento for realizado, fazendo, para isso, uso de uma prioridade de execução superior aos outros threads.

Os serviços de Mensagens Ativas estão disponíveis ao programador no ambiente Chant, entretanto é responsabilidade deste fazer o correto uso dos serviços, afim de evitar uma situação de *deadlock*.

3.3.3 Nexus

Nexus (FOSTER; KESSELMAN; TUECKE, 1994, 1996) consiste em uma camada de suporte a execução de aplicações paralelas irregulares, com o objetivo de utilizar um aglomerado de computadores heterogêneos e geograficamente distribuídos como uma Grade Computacional (FOSTER; KESSELMAN, 1999).

A Fig. 8 ilustra as cinco abstrações que formam a interface de Nexus: nós, contextos, threads, ponteiros globais e requisições de serviços remotos.

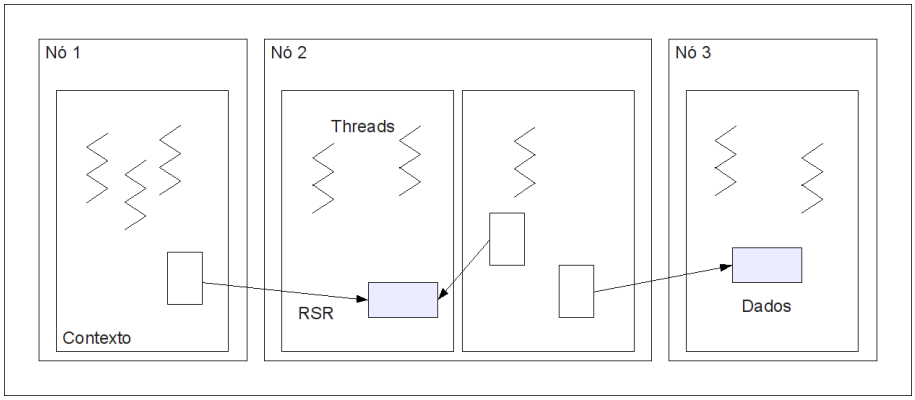

Figura 8: *Abstrações que formam a interface Nexus*

Uma aplicação consiste, então, de vários threads executando dentro de espaços de endereçamento chamados de *contextos*, mapeados em um conjunto de *nós*, os quais representam os recursos físicos da arquitetura. A memória compartilhada é utilizada na comunicação de threads internamente à um mesmo contexto, enquanto que, para a comunicação entre threads em contextos distintos, os threads fazem-se uso de ponteiros globais para o disparo de requisições de serviços remotos, os quais executam funções dentro de outros contextos.

Os ponteiros globais também provêm um espaço de endereçamento global em Nexus. Caracteriza-se um ponteiro global por uma estrutura de dados contendo uma referencia direta a um objeto dentro de um contexto, através do uso de um par (*startpoint, endpoint*) para a comunicação ponto-a-ponto. Para realização de comunicação multicast associa-se um *startpoint* a vários *endpoints*.

A comunicação entre threads de diferentes contextos é realizada através de requisições de serviços remotos (RSR). Ao realizar uma requisição de serviço remoto, uma função especial, chamada de handler, é executada, internamente ao contexto referenciado pelo ponteiro global. A invocação desta função é realizada de forma assíncrona dentro do contexto. Já sua execução pode ser realizada tanto em um novo thread com em um thread já criado, desde que a função não seja bloqueante. Salienta-se ainda que em uma RSR não existe confirmação ou

retorno de resultados e o thread que realizou a RSR não permanece bloqueado aguardando resposta do thread remoto.

3.3.4 Athapascan-0

O núcleo executivo de Athapascan-0 (GINZBURG, 1997) fornece um conjunto de primitivas para troca de mensagens, combinando multiprogramação e comunicação. Em Athapascan-0 pode-se criar threads tanto localmente quanto remotamente. O núcleo de Athapascan-0 oferece mecanismos de sincronização como mutexes e semáforos a threads criados localmente, já que sua comunicação é realizada através da memória compartilhada, garantindo assim o acesso correto aos dados compartilhados. Já os threads remotos são criados com a invocação de um chamado à um procedimento remoto assíncrono, o qual corresponde à execução de um serviço.

Além de não restringir o uso de primitivas de sincronização por parte dos threads remotos, Athapascan-0 ainda permite que estes usem ”mensagens urgentes”. O tratamento de uma mensagem urgente é realizado por um daemon específico, o qual, ao recebimento de uma mensagem, executa imediatamente o serviço descrito nesta mensagem. Entretanto, os serviços executados por este daemon não devem possuir primitivas de sincronização.

4 IMPLEMENTAÇÃO

Este capítulo apresenta a biblioteca desenvolvida para suporte em software de Mensagens Ativas e as ferramentas utilizadas na sua implementação. O capítulo inicia (Seção 4.1) apresentando a utilização da biblioteca, a forma como se definiu-se as primitivas da interface de programação aplicativa e como a mensagem ativa foi abstraída em software. A Seção 4.2 apresenta a implementação em software do processador de comunicação e seu objetivo.

Já na Seção 4.3 é explicado o processo de inicialização dos nós virtuais e do processador de comunicação assim como suas finalizações. As primitivas da interface de programação aplicativa são apresentadas, assim como descritas, na Seção 4.4. A Seção 4.5 apresenta um pequeno exemplo do uso da biblioteca.

4.1 Utilização da biblioteca

O código foi desenvolvido na forma de biblioteca na linguagem de programação C, devendo esta ser ligada à aplicação que a utilizará. Foi utilizada a biblioteca Open MPI (OPENMPI, 2008) para a realização da comunicação entre os nós da arquitetura. Esta biblioteca de comunicação implementa o padrão de comunicação MPI (GROPP; LUSK; SKJELLUM, 1994; PACHECO, 1996), uma alternativa de mais alto nível se comparada com outras opções de ferramentas para comunicação entre processos, como a utilização de sockets. A biblioteca Open MPI deve, também, ser ligada ao programa aplicativo. Por se tratar de um padrão, outras implementações de MPI podem substituir a biblioteca Open MPI sem modificação no código desenvolvido para a biblioteca de Mensagens Ativas. Não é necessário usar cláusulas *#include* para inclusão da biblioteca Open MPI, sendo necessário somente a inclusão da biblioteca desenvolvida no presente trabalho.

Para compilação da biblioteca deve-se entrar com o seguinte comando na linha de comando, sendo *app.c* o código-fonte da aplicação desenvolvida utilizando-se da biblioteca de mensagens ativas aqui desenvolvida:

```
mpicc app.c -o app -Wall
```

O formato das primitivas da interface de programação aplicativa da biblioteca seguiu o seguinte padrão: tem-se primeiramente a união dos acrônimos act e msg, separados pela caractere sublinhado, seguido do correspondente, em inglês, do serviço executado por aquela primitiva. Nas primitivas utilizadas para a inicialização e término dos processadores virtuais e do processador comunicação fez-se uso do prefixo av, seguido das palavras init e finalize, para a inicialização e término, respectivamente. Estas primitivas não são de uso exclusivo da biblioteca desenvolvida, mas de todas as funcionalidades da arquitetura virtual Anahy, a AVC – Anahy Virtual Computer.

O corpo de uma mensagem ativa entre dois nós é abstraída pelo uso de uma estrutura de dados opaca contendo uma área de memória onde serão armazenados os dados, chamado de buffer. A inserção e a remoção de dados deste buffer são realizadas por meio de primitivas *pack/unpack* que, além de inserir ou remover dados da área de memória, atualizam um ponteiro com a posição atual no buffer.

Esta estrutura também possui campos contendo o tamanho total do buffer e o tamanho ainda livre para inserção de dados, juntamente com um campo informando o serviço atrelado a esta mensagem.

Por fazer uso de uma biblioteca MPI, a aplicação deve ser lançada, além de compilada, como uma aplicação MPI, conforme especificado pela implementação de MPI utilizada. A execução se dá da forma tradicional de MPI, ou seja, vários processos executando em nós virtuais da máquina virtual MPI sobre os nós reais de uma arquitetura distribuída. O modelo de execução seguido por esses nós virtuais é o modelo *Single Program Multiple Data* (SPMD), onde vários processadores autônomos executam o mesmo código, mas em diferentes pontos, como pode ser visto na Fig. 9.

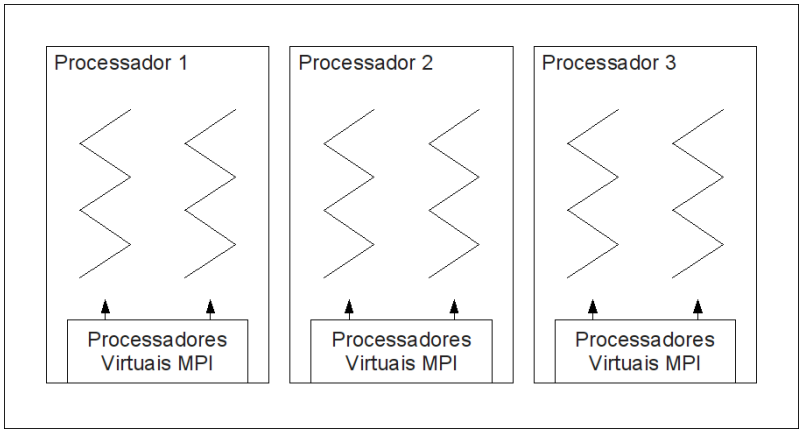

Figura 9: *Processadores seguindo o modelo de execução SPMD*

4.1.1 Serviços

Os serviços consistem em pontos de entrada das comunicações no modelo de Mensagens Ativas. Devem ser implementados pelo programador para realizar tarefas associadas à aplicação desenvolvida. A implementação de serviços se dá na forma de funções sem retorno (*void*) que recebem como parâmetro de entrada um ponteiro para uma área de memória (*void **). Um protótipo para um serviço pode ser definido como segue.

```
void *servico(void *in);
```

A função associada a um determinado serviço é ativada pelo daemon de comunicação conforme a mensagem ativa recebida, segundo o modelo UpCall (Seção 3.2.1). O programador deve atentar para que o código dos serviços não efetue operações bloqueantes ou sejam por demais extensos. O código dos serviços é executado no fluxo de execução associado ao daemon de comunicação e há risco de perda de reatividade do sistema, desviando a função principal do processador de comunicação para execução de computação.

É importante observar que todos os serviços devem ser reconhecidos por todos os nós virtuais da arquitetura virtual MPI da mesma forma. A biblioteca desenvolvida oferece uma primitiva para registro de serviços. Os serviços devem ser registrados em cada nó da arquitetura, na mesma ordem para cada nó, pois a identificação de um serviço se dá pelo índice de um vetor onde estão armazenados seus endereços na memória do nó atual.

4.2 O processador de comunicação

O processador de comunicação, também chamado de daemon de comunicação, é, no nó virtual 0 da máquina virtual MPI, executado por um thread criado pela primitiva *av_init*. Já nos outros nós virtuais, o fluxo de execução associado à função principal main é responsável pela execução do processador de comunicação, uma vez que o código da função main deve ser executado somente por um dos nós virtuais.

Este processador de comunicação implementa um dos modelos alternativos de Mensagens Ativas para execução dos serviços, o modelo *UpCall*. A política de *polling* é empregada, pois o daemon de comunicação fica bloqueado em uma operação de recebimento de mensagem síncrona de MPI, verificando, portanto, a interface de rede de forma constante.

O código do processador de comunicação desenvolvido em software é o seguinte:

```
void *daemon() {
    int source; //identificacao do transmissor
    act_msg_t *m = malloc(sizeof (act_msg_t)); //alocacao da mensagem
    void *(*func)(void*); //ponteiro para o servico
    for (;;) {
        act_msg_create(m, 2048); //cria mensagem
        act_msg_recv(m, &source); //bloqueia aguardando recebimento
        act_msg_realloc(m, m->sizebuff); //realoca o buffer
        func = (void *) act_msg_getserv(m->service); //end. servico
        func(m); //execucao do servico
        act_msg_reset(m); //reinicia a mensagem
    }
}
```

O código inicia criando uma variável do tipo *int*, a qual é responsável por armazenar a identificação do nó responsável pelo envio da mensagem para o daemon. Depois, é alocada uma área na memória para armazenagem da mensagem e criado o ponteiro que será responsável por armazenar o endereço do serviço identificado na mensagem. Finalmente, o código entra em um laço infinito onde, após criada a mensagem, o daemon bloqueia esperando por novas mensagens. Ao receber uma nova mensagem, sua área destinada a dados é realocada, com o objetivo de minimizar o uso da memória. Esta função pode ser omitida. Após, o serviço identificado pela mensagem é retirado do campo *service* e usado para obter o endereço na memória do nó atual da função correspondente e esta função é então executada recebendo como parâmetro a mensagem recebida. Após a execução do serviço, a mensagem é reiniciada, e o daemon reinicia o laço, bloqueando sua execução no aguardo de uma nova mensagem.

4.3 Processo de Inicialização e término

Todos os nós virtuais MPI iniciam a execução da função principal (*main*) conforme especificado no modelo de execução SPMD. A primitiva *av_init* faz a inicialização das estruturas de dados associadas a biblioteca e, no nó 0 (*zero*) cria um thread específico para suportar a execução do daemon de comunicação. Na seqüência o código da aplicação deve realizar o registro dos serviços e, então, nos demais nós, como a função main não deve ser executada, o *daemon* de comunicação ocupa o mesmo thread associado ao main.

Como já foi mencionado, um serviço é uma função implementada pelo programador da aplicação, que recebe parâmetros, mas que não retorna nenhum resultado, pois Mensagens Ativas são assíncronas.

Utiliza-se uma função para o registro dos serviços da aplicação, o qual é responsável por armazenar o endereço do serviço na memória do nó atual em um vetor e retornar para a aplicação o índice onde este serviço foi armazenado no vetor, permitindo assim, seu acesso pelo outros nós da arquitetura pela identificação do índice neste vetor de serviços.

Estes serviços não podem ser bloqueantes, pois no recebimento de uma mensagem, o serviço será executado diretamente pelo *daemon* de comunicação, podendo acarretar assim em uma situação onde o processador de comunicação ficará indisponível.

Segue um exemplo de código de como deve ser feita a inicialização até o registro dos serviços, sendo as variáveis *my_rank* e *p*, variáveis análogas as utilizadas em aplicações MPI para o controle de execução dos processadores virtuais MPI.

```
int main(int argc, char *argv[]) {
    int my_rank, p;
    pthread_t thid;

    av_init(argc, argv, &my_rank, &p, &thid);
    serviço[0] = act_msg_regserv(serviço1);
    serviço[1] = act_msg_regserv(serviço2);
    serviço[2] = act_msg_regserv(serviço3);
```

Inicialmente cria-se as variáveis responsáveis pela identificação do *rank* MPI do nó atual, assim como a identicação do número total de processadores virtuais da máquina virtual MPI. Cria-se também um identicador para o thread responsável pela execução do daemon de comunicação no nó virtual 0. Após é chamada a primitiva *av_init* responsável pela sincronização dos nós virtuais MPI e pela inicialização do processador de comunicação no nó 0. Finalmente, após o retorno da primitiva *av_init*, deve-se registrar os serviços que o daemon de comunicação poderá executar.

Para a realização do término dos processadores virtuais faz-se uso da primitiva *av_finalize*, a qual é acionada no nó virtual 0, enviando um mensagem ativa para todos os outros nós virtuais, requisitando que sejam terminados, e então finalizando sua própria execução.

4.4 Primitivas da Interface de Programação Aplicativa

Está seção traz uma breve descrição das primitivas implementadas que estão disponíveis ao programador, assim como da primitiva *act_msg_recv* utilizada no processador de comunicação para fazer o recebimento das mensagens.

```
int act_msg_create(act_msg_t *m, unsigned long size)
```

Esta função tem por objetivo criar uma nova mensagem. Para isso, essa função inicializa os vários campos da mensagem responsáveis por armazenar informações a respeito do tamanho da mensagem e do serviço a ser realizado por aquela mensagem, após a alocação de uma área, de tamanho *size*(segundo parâmetro da função), reservada para os dados a serem armazenados na mensagem. O primeiro parâmetro é um ponteiro para *act_msg_t*, uma estrutura de dados opaca responsável por abstrair a mensagem ativa, como já havia sido mencionado. Esta função deve retornar 0 caso sua execução seja bem sucedida.

```
int act_msg_realloc(act_msg_t *m, unsigned long size)
```

Esta função tem por objetivo realocar a área de memória alocada para os dados da mensagem. Esta função pode ser utilizada para diminuir o uso de memória no nó atual, entretanto, introduz um overhead que pode não ser desejado para a realização de processamento de alto desempenho. O primeiro parâmetro é um ponteiro para a mensagem a qual se deseja realocar a área de dados da mensagem e o segundo parâmetro é o novo tamanho desejado para a área de dados da mensagem. Esta função retorna 0 se sucedida.

```
int act_msg_reset(act_msg_t *m)
```

Esta função tem por objetivo liberar a área de memória apontada pela mensagem passada como parâmetro. Além disso, esta função atribiu um valor inválido para o campo que identifica o serviço na mensagem e também atribui 0 para os campos que armazem informações a respeito do tamanho da mensagem. Deve retornar 0 em caso de sucesso.

```
int act_msg_init(act_msg_t *m, int service)
```

Esta função tem por objetivo inicializar o serviço a ser realizado na mensagem. Para isso, o campo da mensagem apontada pelo primeiro parâmetro da função que identifica o serviço a ser realizado no processador de comunicação no recebimento de uma mensagem recebe o valor passado no segundo parâmetro. Este valor deve corresponder a um índice de um serviço armazenado no vetor de serviços pela primitiva *act_msg_regserv*. Esta função retorna 0 se bem sucedida.

```
unsigned int act_msg_regserv(void * (*function) (void *))
```

Esta função registra um serviço no nó atual. Seu principal objetivo é permitir que os diferentes nós da arquitetura acessem o mesmo serviço em qualquer nó, mesmo estes serviços estando em diferentes áreas de memória. Para isso, todos os serviços devem ser registrados na mesma ordem, em todos os nós da arquitetura. A função armazena em um vetor o endereço na memória do nó atual do serviço passado como parâmetro, e retorna para a função que a chamou o índice da posição no qual este serviço foi registrado.

```
unsigned int act_msg_getserv(int indice)
```

Esta função tem por objetivo retornar o endereço de um determinado serviço registrado no nó atual. Usa o valor passado como parâmetro como índice para realizar a busca no vetor que armazena o endereço dos serviços. É utilizada para o encapsulamento dos dados, não necessitando o programador fazer uso direto do vetor de serviços.

```
int act_msg_send(act_msg_t *m, int dest)
```

Esta função é utilizada para o envio de uma mensagem a um nó destino. O primeiro parâmetro identifica a mensagem a ser enviada e o segundo identifica o nó virtual da máquina virtual MPI a qual deverá ser feito o envio.

Todas as estruturas da mensagem ativa, como seu buffer, e os campos contendo informações de tamanho e sobre o serviço a ser realizado no nó de destino são encapsuladas em uma mensagem MPI através da primitiva *MPI_Pack* (PACHECO, 1996), fazendo-se uso de tipos de dados MPI. Logo após, a mensagem é enviada para o nó destino através da primitiva *MPI_Send*. A função retorna zero em caso de sucesso.

```
int act_msg_recv(act_msg_t *m, int *source)
```

Esta função é utilizada pelo processador de comunicação para realizar o recebimento de uma mensagem ativa. O primeiro parâmetro identifica a mensagem onde deverá ser armazenado os dados retirados da interface da rede e a função retorna no segundo parâmetro a identificação do nó que realizou o envio.

O recebimento é realizado através da primitiva *MPI_Recv*, que recebe um mensagem do tipo *MPI_PACKED* (PACHECO, 1996), onde estão encapsulados todos os dados que descrevem a mensagem recebida. Através do uso da primitiva *MPI_Unpack*, a mensagem é reconstruída no processador de

comunicação, retirando da mensagem MPI os dados que descrevem os vários campos da Mensagem Ativa recebida. Esta função deverá retornar 0 em caso de sucesso.

```
int act_msg_pack(act_msg_t *m, const void *data, size_t size)
```

O objetivo desta função é armazenar determinado dado na área destinada a dados de uma mensagem. Copia-se os dados de uma área de memória de tamanho *size* que é apontada por *data* para a área de memória apontada pelo ponteiro da área de dados da mensagem. Logo após, atualiza-se ponteiro da área de dados da mensagem com a nova posição, além de atualizar-se o campo da mensagem que armazena informações a respeito do tamanho livre na mensagem. Retorna o valor 0 caso seja bem sucedida.

```
int act_msg_unpack(act_msg_t *m, void *data, size_t size)
```

O objetivo desta função é análogo à função act_msg_pack, ou seja, retirar determinado dado da área de dados de uma mensagem. Para isso, a função copia os dados de uma área de memória de tamanho *size* apontada pelo ponteiro da área de dados da mensagem para uma área de memória apontada por *data*. Então, a função atualiza o ponteiro da área de dados da mensagem para apontar para sua nova posição e atualiza as informações no campo com informações sobre o tamanho livre na mensagem. Caso sua execução seja bem sucedida retorna o valor 0.

```
int av_init(int argc, char *argv[], int *my_rank, int *p, pthread_t *thid)
```

O objetivo dessa função é lançar os nós virtuais MPI, além de, no nó virtual 0, lançar o processador de comunicação. São passados os parâmetros recebidos da linha de comando para que sejam processados pela primitiva *MPI_Init* (PACHECO, 1996), responsável por inicializar os nós virtuais MPI. O código após esta função será executado por todos os nós virtuais, exceto onde explícitamente indicado, portanto, deve-se, após a chamada desta função, registrar-se todos os serviços a serem utilizados entre os nós da arquitetura. Ainda dentro desta função, caso o nó virtual seja o nó 0, será lançado um thread independente responsável por rodar o código do processador de comunicação. Esta função deverá retornar 0.

```
int av_finalize(pthread_t thid, int p)
```

Esta função objetiva finalizar os nós virtuais MPI e o processador de comunicação lançado pelo nó virtual 0 na chamada da função *av_init*. Esta função é chamada no escopo do nó virtual 0, enviando uma mensagem ativa a todos os outros nós, requisitando que sejam terminados. Então o nó 0 finaliza a execução do thread lançado para simular o processador de comunicação e chama a função *MPI_Finalize* para finalizar os nós virtuais da máquina virtual MPI.

4.5 Exemplo de uso

Para exemplificar o uso da interface de programação aplicativa da biblioteca desenvolvida foi implementado um pequeno exemplo, com três serviços básicos. O primeiro serviço é o serviço utilizado para finalizar os processos ao final da execução, chamado *quit*. Sua execução se dará após o nó virtual 0 enviar um Mensagem Ativa para os outros nós virtuais requisitando suas finalizações. O segundo serviço chamado de *soma*, recebe uma mensagem, desempacota um número armazenado na área de dados da mensagem, soma o valor 1 à este número e armazena-o em uma Mensagem Ativa com o terceiro serviço, *imprime*, enviando-o para o nó 0. Já o terceiro serviço, *imprime*, simplesmente desempacota o número armazenado na área de dados da mensagem e imprime-o em um arquivo.

Abaixo, o código-fonte do exemplo:

```
#include <stdio.h>   //
#include <stdlib.h>  // BIBLIOTECAS BASICAS DA LINGUAGUEM C
#include <string.h>  //
#include "act_msg.h" // BIBLIOTECA DESENVOLVIDA

void *soma(void *msg) {
    int *x = (int *) malloc(sizeof (int));
    act_msg_t *msg2 = malloc(sizeof (act_msg_t));
    act_msg_unpack(msg, x, sizeof (int)); //desempacotamento
    *x = *x + 1; //incremento do valor
    act_msg_create(msg2, 10); //nova mensagem eh criada
    act_msg_pack(msg2, x, sizeof (int)); //dado empacotado
    act_msg_init(msg2, 2);
    act_msg_send(msg2, 0); //envio da mensagem para o no 0
}
```

```
void *imprime(void *msg) {
    FILE* file = fopen("teste.txt", "w"); //abertura do arquivo
    int *x = (int *) malloc(sizeof (int));
    act_msg_unpack(msg, x, sizeof (int)); //desempacotamento
    fprintf(file, "Resultado: \%d\n", *x); //impressao
    fclose(file); //fechamento do arquivo
}

void *quit(void *msg) {
    exit(0);
}

int main(int argc, char *argv[]) {
    int my_rank, p;
    pthread_t thid;
char c;

    av_init(argc, argv, &my_rank, &p, &thid); //inicializao
    int indices[10];
    indices[0] = act_msg_regserv(quit);    //
    indices[1] = act_msg_regserv(soma);    // REGISTRO DOS SERVICOS
    indices[2] = act_msg_regserv(imprime); //

    if (my_rank == 0) {
        act_msg_t *m1 = malloc(sizeof (act_msg_t));  //aloca mensagem
        int *numero1 = (int *) malloc(sizeof (int));
        *numero1 = 6; //valor inicial
        act_msg_create(m1, sizeof (int)); //cria nova mensagem
        act_msg_init(m1, 1); //inicializa com servico soma
        act_msg_pack(m1, numero1, sizeof (int)); //empacota dado
        act_msg_send(m1, 1); //envia dado para o no 1
scanf(c);
        av_finalize(thid, p); //finalizacao
    }
    if (my_rank != 0) {
        daemon();
    }
}
```

A aplicação inicia introduzindo algumas bibliotecas básicas da linguagem C, para tratamento de entrada e saída e tratamento de *strings* por exemplo, atráves de cláusulas *#include*. Além destas é também introduzida a biblioteca desenvolvida neste trabalho através do comando *#include "act_msg.h"*. Logo abaixo são definidos os 3 serviços utilizados na aplicação. No serviço *soma*, o dado armazenado na área de dados da mensagem é desempacotado para uma área de memória que é apontada por um ponteiro para tipo de dado *int*. Incrementa-se o valor apontado por este ponteiro e então cria-se uma nova mensagem, empacota-se o novo valor e envia-se esta mensagem de volta ao nó 0 da máquina virtual MPI.

O serviço *imprime* é responsável por imprimir o resultado do serviço *soma* em um arquivo. Após abrir o arquivo, o dado é desempacotado da área de dados da mensagem, armazenado em uma área de memória apontada por um ponteiro para *int* e então o valor apontado por este ponteiro é impresso no arquivo. Após, o arquivo é fechado e o serviço é finalizado. Já o serviço *quit* simplesmente chama a função *exit*, responsável por finalizar a execução do trecho de código atual.

Já na função principal, segue-se o esquema de inicialização descrito na seção 4.3, registrando-se os serviços previamente definidos. Após, faz-se com que o nó virtual 0 da máquina virtual MPI crie uma nova mensagem e também crie uma ponteiro para *int*. Este ponteiro então é inicializado com um valor qualquer, no caso do exemplo este valor é igual a 6. Então, inicializa-se a mensagem com o serviço correspondente, serviço *soma*, e faz-se o envio da mensagem para o nó virtual 1.

Já que o envio de mensagens ocorre de forma assíncrona, utilizou-se a primitiva *scanf* para que a execução do serviço correspondente pudesse ser concluída antes que os daemons de comunicação fossem finalizados através do uso da primitiva *av_finalize*. Já o virtual 1 executa o código do daemon de comunicação apresentado na seção 4.2. Uma representação deste exemplo pode ser conferido na Fig. 10.

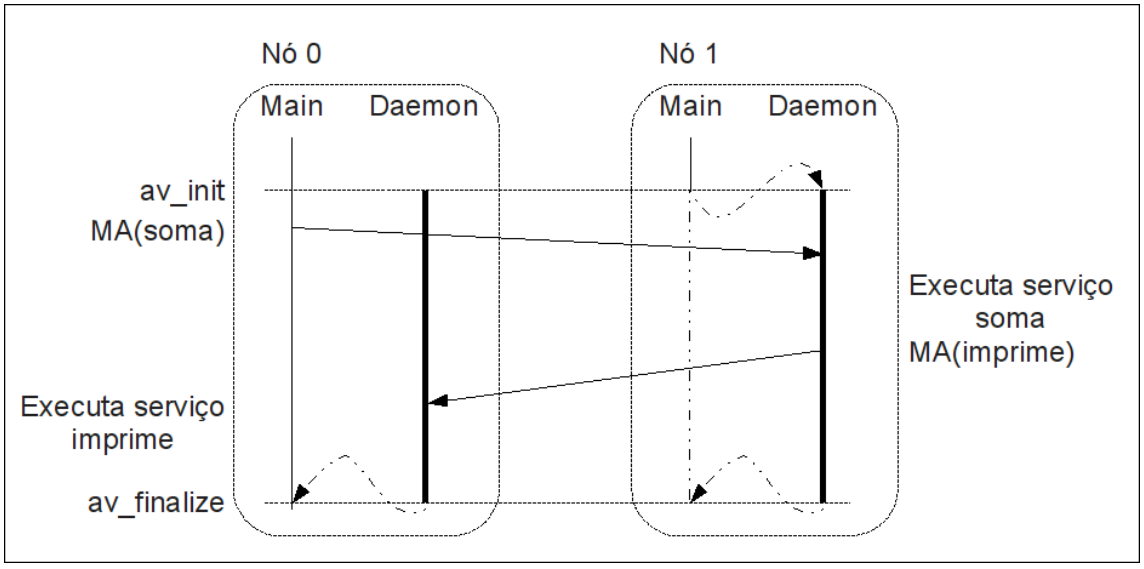

Figura 10: *Exemplo de uso da interface de programação aplicativa disponível ao programador*

5 AVALIAÇÃO DE DESEMPENHO

Os experimentos foram realizados localmente em uma máquina com processador Intel Core Duo T2300, 1.66Ghz, 667MHz de Front Side Bus (FSB) e 2MB de cache. A máquina também possuía 2,5GB de memória RAM, e o sistema utilizado como base foi o GNU/Linux distribuição Ubuntu 8.04 com kernel 2.6.24-21. A versão utilizada da biblioteca Open MPI foi a de número 1.2.6.

Foram realizados testes com o objetivo de medir o tempo de execução necessário para a realização das operações elementares desenvolvidas na biblioteca. Entre as operações medidas estão as operações de empacotar e desempacotar dados em uma mensagem através das primitivas *pack/unpack* (Seção 5.1) e operação de criação de mensagens com o uso da primitiva *act_msg_create* (Seção 5.2), com diferentes tamanhos de mensagens. Também avaliou-se o tempo de execução de uma operação de envio de mensagem (Seção 5.3), operação básica para o funcionamento correto da biblioteca desenvolvida. Além disso desenvolveu-se um serviço com o objetivo de medir o uso de todas essas operações (Seção 5.4) em conjunto.

5.1 Empacotamento e Desempacotamento de Dados

As operações de *pack* e *unpack* são utilizadas para empacotar e desempacotar determinado dado de uma mensagem conforme explicado na Seção 4.4. O desenvolvimento da biblioteca foi motivado pelo seu potencial de desempenho em aglomerados de computadores, logo, tais operações não devem ocupar muito tempo do processador em suas execuções, permitindo assim que o processador possa ocupar maior parte do seu tempo realizando cálculo efetivo.

O tempo médio de execução de cada operação foi encontrado através da medição da média de 100 operações de empacotamento de um dado do tipo *int* e de 100 operações de desempacotamento de um dado do tipo *int*. O tempo médio de cada operação pode ser conferido (em segundos) na Tab. 1. Deve-se ainda destacar que as operações foram realizadas utilizando-se somente um

processador virtual.

Operação	Tempo de Execução Médio (s)
Pack	0,0005
Unpack	0,0004

Tabela 1: *Tempo de execução médio das operações de Pack e Unpack*

A análise quantitativa destes resultados permitiu verificar que são satisfatoriamente relevantes aos objetivos procurados.

5.2 Criação de Mensagens

Para criação de uma nova mensagem deve-se utilizar a primitiva *act_msg_create* que recebe em seu segundo parâmetro (como visto na Seção 4.4) o tamanho desejado da mensagem. Este tamanho é dado em bytes e é alocado dinâmicamente na invocação da primitiva. Optou-se por medir o tempo de execução da operação de criação de uma nova mensagem gerando-se mensagens com tamanho igual a 1024 bytes e 2048 bytes. Assim como na avaliação das primitivas *pack/unpack*, mediu-se a média do tempo de execução de 100 testes, novamente em segundos, rodando sobre somente um processador virtual. Os tempos médios de execução podem ser conferidos na Tab. 2.

Tam. da Mensagem	Tempo de Execução Médio (s)
1024 bytes	0,00012
2048 bytes	0,00025

Tabela 2: *Tempo de execução médio de criação de mensagens com tamanho 1024 e 2048 bytes*

Assim como os resultados obtidos na avaliação das operações de empacotamento e desempacotamento de dados, o tempo médio de criação de novas mensagens, tanto com tamanho igual a 1024 bytes quanto com tamanho igual a 2048 bytes, mostraram-se satisfatórios ao resultado esperado.

5.3 Envio de Mensagens

A biblioteca desenvolvida baseia-se no envio de mensagens assíncronas entre os nós de um aglomerado de computadores, sendo, portanto, a operação de envio de mensagens para outros nós a operação mais relevante na avaliação de desempenho da biblioteca. Esta operação deve ser executada da forma mais rápida possível, permitindo que o desempenho da aplicação desejada não seja afetado pela sua realização. Para a realização desta avaliação foram

utilizados dois processadores virtuais, sendo que o serviço contido na msg enviada não realizava nenhum tipo de computação, um serviço vazio. Além disso, a mensagem criada possuía um dado do tipo *int* empacotada em sua área de dados, a qual possuía tamanho igual ao tamanho do tipo de dado *int*, ou seja, 4 bytes. Novamente foram realizadas 100 operações, daonde retirado-se o tempo médio de execução da operação de envio de mensagens, tempo este que pode ser conferido na Tab. 3.

Operação	Tempo de Execução Médio (s)
Send	0,004

Tabela 3: *Tempo de execução médio de envio de mensagens*

Mais uma vez, a avalição quantitativa dos resultados obtidos demonstrou que, assim como as operações previamente avaliadas, a operação de envio de mensagens para diferentes nós da arquitetura teve desempenho adequado ao objetivado.

5.4 Serviço de Avaliação das Operações em conjunto

Com o objetivo de avaliar o desempenho das diversas operações em conjunto, desenvolveu-se um serviço onde os processadores de comunicação desempacotam um dado, executam alguma operação básica, como a adição, sobre o dado obtido e enviam este a outro nó, após prévio empacotamento. Utilizando dois processadores virtuais obteve-se o tempo médio de execução desde o envio de uma mensagem por um nó até o recebimento da resposta, como em uma operação ”pingpong”. A média do tempo de execução foi então obtida dividindo-se o tempo total pelo número de testes realizados, ou seja, 20 pares ”pingpong”. O tempo médio de execução deste serviço pode ser conferido na Tab. 4.

	Tempo de Execução Médio (s)
Serviço	0,01

Tabela 4: *Tempo de execução médio do serviço de avaliação das diversas operações*

Mostrou-se com este serviço que o desempenho de todas as operações previamente avaliadas, assim como da biblioteca em si, esteve dentro das expectativas.

6 CONCLUSÃO

Neste trabalho foi proposto o desenvolvimento e a implementação de uma biblioteca de comunicação para aglomerados de computadores que fizesse uso do modelo de Mensagens Ativas para obtenção de alto desempenho, sobrepondo o tempo gasto na comunicação com cálculo efetivo empregando troca de mensagens assíncronas entre os nós da arquitetura. Esta biblioteca fez uso de multiprogramação leve para simulação do processador dedicado à comunicação presente na proposta original de Mensagens Ativas.

Com o uso do modelo de Mensagens Ativas na implementação da biblioteca foi possível atingir os níveis de desempenho que são esperados de uma biblioteca de comunicação que se propõe ser utilizada em aglomerados de computadores, arquiteturas estas de suporte ao processamento de alto desempenho. Entretanto, ainda deverão ser realizados experimentos com o objetivo de validar seu desempenho em uma aglomerado de computadores, já que os experimentos foram realizados localmente.

Além disto, a biblioteca foi implementada de forma que pudesse ser usada tanto independentemente quanto integrada a outro ambiente, no desenvolvimento de aplicações que necessitem de alto poder computacional.

Como trabalhos futuros tem-se a integração da biblioteca de comunicação desenvolvida com o ambiente de programação paralela Anahy, já que a biblioteca mostrou ter desempenho satisfatório, além de ter sido desenvolvida para ser independente do ambiente à qual estará sendo usada.

Isto permitirá que o ambiente Anahy seja extendido a uma arquitetura com memória distribuída, com a possibilidade de migração de tarefas e dados de forma a distribuir a carga computacional entre os nós da arquitetura.

REFERÊNCIAS

BARCELLOS, A. M. P.; GASPARY, L. P. Tecnologias de Rede para Processamento de Alto Desempenho. In: III ESCOLA REGIONAL DE ALTO DESEMPENHO, 2003, Santa Maria - Rio Grande do Sul - Brasil. **Anais...** SBC, 2003. p.67–102.

BIRRELL, A. D.; NELSON, B. J. Implementing remote procedure calls. **ACM Transactions on Computer Systems**, New York, NY, USA, v.2, n.1, p.39–59, 1984.

BUYYA, R. **High Performance Cluster Computing**: architectures and systems. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1999.

CARISSIMI, A. **Athapascan-0**: exploitation de la multiprogrammation legere sur grappes de multiprocesseurs. 1999. Tese (Doutorado em Ciência da Computação) — Institut National Polytechnique de Grenoble, Grenoble, France.

CAVALHEIRO, G. G. H. Princípios da programação concorrente. In: IV ESCOLA REGIONAL DE ALTO DESEMPENHO, 2004, Pelotas - Rio Grande do Sul - Brasil. **Anais...** SBC, 2004. p.3–39.

CAVALHEIRO, G. G. H.; DALL'AGNOL, E. C.; REAL, L. C. V. Uma Biblioteca de Processos Leves para a Implementação de Aplicações Altamente Paralelas. In: IV WORKSHOP EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO, 2003, São Paulo - Brasil. **Anais...** SBC, 2003. p.117–124.

CHANG, C.-C.; CZAJKOWSKI, G.; HAWBLITZEL, C.; EICKEN, T. von. Low-latency communication on the IBM RISC system/6000 SP. In: SUPERCOMPUTING '96: PROCEEDINGS OF THE 1996 ACM/IEEE CONFERENCE ON SUPERCOMPUTING (CDROM), 1996, Washington, DC, USA. **Anais...** IEEE Computer Society, 1996. p.24.

CORPORATE IEEE, I. o. E.; ELECTRONICS ENGINEERS, I. S. **IEEE Standard for Information Technology - Portable Operating System Interface (POSIX)**:

system application program interface (api), amendment 1: realtime extension (c language), ieee std 1003.1b-1993. New York, NY, USA: IEEE Standards Office, 1994.

COSTA, C. M. da; STRINGHINI, D.; CAVALHEIRO, G. G. Programação concorrente: threads, mpi e pvm. In: II ESCOLA REGIONAL DE ALTO DESEMPENHO, 2002, São Leopoldo - Rio Grande do Sul - Brasil. **Anais...** SBC, 2002. p.31–65.

DEITEL, H. M.; DEITEL, P. J. **Java**: como programar (6th edition). Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2004.

DENNEULIN, Y.; NAMYST, R.; MéHAUT, J.-F. Architecture Virtualization with Mobile Threads. In: PARCO'97 (PARALLEL COMPUTING), 1997. **Anais...** Elsevier Science Publishers, 1997. p.477–484.

EICKEN, T. von; CULLER, D. E.; GOLDSTEIN, S. C.; SCHAUSER, K. E. Active messages: a mechanism for integrated communication and computation. In: ISCA '92: PROCEEDINGS OF THE 19TH ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 1992, New York, NY, USA. **Anais...** ACM, 1992. p.256–266.

FORUM, M. P. **MPI**: a message-passing interface standard. Knoxville, TN, USA: [s.n.], 1994.

FOSTER, I.; KESSELMAN, C. **The grid**: blueprint for a new computing infrastructure. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999.

FOSTER, I.; KESSELMAN, C.; TUECKE, S. The Nexus Task-Parallel Runtime System. In: I INTL WORKSHOP ON PARALLEL PROCESSING, 1994. **Anais...** Tata McGraw Hill, 1994. p.457–462.

FOSTER, I.; KESSELMAN, C.; TUECKE, S. The Nexus approach to integrating multithreading and communication. **Journal of Parallel and Distributed Computing**, Orlando, FL, USA, v.37, n.1, p.70–82, 1996.

GEIST, A.; BEGUELIN, A.; DONGARRA, J.; JIANG, W.; MANCHEK, R.; SUNDERAM, V. **PVM**: parallel virtual machine: a users' guide and tutorial for networked parallel computing. Cambridge, MA, USA: MIT Press, 1994.

GINZBURG, I. **Athapascan-0b**: intégration efficace et portable de multiprogrammation légère et de communication. 1997. Tese (Doutorado em Ciência da Computação) — Institut National Polytechnique de Grenoble, Grenoble, França.

GROPP, W.; LUSK, E.; SKJELLUM, A. **Using MPI**: portable parallel programming with the message-passing interface. Cambridge, MA, USA: MIT Press, 1994.

HAINES, M.; CRONK, D.; MEHROTRA, P. On the design of Chant: a talking threads package. In: SUPERCOMPUTING '94: PROCEEDINGS OF THE 1994 ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 1994, New York, NY, USA. **Anais...** ACM, 1994. p.350–359.

KRISHNAMURTHY, A.; CULLER, D. E.; DUSSEAU, A.; GOLDSTEIN, S. C.; LUMETTA, S.; EICKEN, T. von; YELICK, K. Parallel programming in Split-C. In: SUPERCOMPUTING '93: PROCEEDINGS OF THE 1993 ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 1993, New York, NY, USA. **Anais...** ACM, 1993. p.262–273.

LUMETTA, S. S.; MAINWARING, A. M.; CULLER, D. E. Multi-protocol active messages on a cluster of SMP's. In: SUPERCOMPUTING '97: PROCEEDINGS OF THE 1997 ACM/IEEE CONFERENCE ON SUPERCOMPUTING (CDROM), 1997, New York, NY, USA. **Anais...** ACM, 1997. p.1–22.

OPENMPI. **Open MPI**. http://www.open-mpi.org. Acesso: ago. 2008.

PACHECO, P. S. **Parallel programming with MPI**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.

PERANCONI, D. S. **Alinhamento de Sequência Biológicas em Arquiteturas com Memória Distribuída**. 2005. Dissertação (Mestrado em Ciência da Computação) — Universidade do Vale do Rio dos Sinos, São Leopoldo.

ROLOFF, E.; CARISSIMI, A.; CAVALHEIRO, G. G. H. Variações de mensagens ativas para aglomerados de computadores. In: IV ESCOLA REGIONAL DE ALTO DESEMPENHO, 2004, Pelotas - Rio Grande do Sul - Brasil. **Anais...** SBC, 2004. p.289–292.

STEVENS, W. R. **UNIX Network Programming**: networking apis: sockets and xti. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997.

TOP500.ORG. **TOP 500 Supercomputer Sites**. http://www.top500.org. Acesso: ago. 2008.

UNDERWOOD, K. D.; SASS, R. R.; LIGON III, W. B. Cost effectiveness of an adaptable computing cluster. In: SUPERCOMPUTING '01: PROCEEDINGS OF THE 2001 ACM/IEEE CONFERENCE ON SUPERCOMPUTING (CDROM), 2001, New York, NY, USA. **Anais...** ACM, 2001. p.54–54.

VAHALIA, U. **UNIX internals**: the new frontiers. Upper Saddle River, NJ, USA: Prentice Hall Press, 1996.

VALIANT, L. G. A bridging model for parallel computation. **Commun. ACM**, New York, NY, USA, v.33, n.8, p.103–111, 1990.

WALLACH, D. A.; HSIEH, W. C.; JOHNSON, K. L.; KAASHOEK, M. F.; WEIHL, W. E. Optimistic active messages: a mechanism for scheduling communication with computation. In: PPOPP '95: PROCEEDINGS OF THE FIFTH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 1995, New York, NY, USA. **Anais...** ACM, 1995. p.217–226.

ZOMAYA, A. Y. H. **Parallel and distributed computing handbook**. New York, NY, USA: McGraw-Hill, Inc., 1996.