Explorando o Uso de Mensagens Ativas em Ambientes para o Processamento de Alto Desempenho*

Daniela Saccol Peranconi, Gerson Geraldo Homrich Cavalheiro
Universidade do Vale do Rio dos Sinos
Programa Interdisciplinar de Pós-Graduação em Computação Aplicada
Av. Unisinos, 950 - São Leopoldo - RS - Brasil
dsaccol@turing.unisinos.br, gersonc@unisinos.br

Resumo

Este artigo apresenta um mecanismo de suporte à comunicação em aglomerados de computadores, desenvolvido para exploração de processamento de alto desempenho. O mecanismo é disponibilizado sob a forma de uma biblioteca de funções em C e está basedo no modelo de Mensagens Ativas. Sua implementação é realizada na camada aplicativa, empregando técnicas de multiprogramação leve. Uma aplicação para alinhamento de seqüências biológicas utilizando o método de programação dinâmica foi implementada para avaliar a biblioteca de comunicação desenvolvida. Por fim, são apresentados resultados de desempenho obtidos e o uso do suporte desenvolvido na construção de Anahy, um ambiente para processamento de alto desempenho em aglomerados.

1. Introdução

A popularização dos aglomerados de computadores vem aumentando consideravelmente nos últimos anos (verificável através de dados apresentados em http://www.top500.org.). Isto se deve, entre outros fatores, a estas arquiteturas oferecerem alta capacidade de processamento e disponibilidade de memória a preços relativamente baixos. No entanto, não é tarefa trivial desenvolver aplicações para este tipo de arquitetura que tirem o máximo proveito dos recursos de processamento disponíveis. As ferramentas de programação paralela e distribuída tradicionais (como *threads*, MPI, OpenMP, RPC) exigem intervenção direta do programador no momento de preparar sua aplicação para utilizar melhor os recursos de hardware disponíveis, não tendo sido desenvolvidas com finalidade de obtenção de alto desempenho.

Anahy [6] é uma ferramenta proposta para ocupar este espaço, tal como outros ambientes encontrados na literatura (Athapascan-0, Nexus, Chant, Split-C). O ambiente Anahy provê uma interface de programação concorrente de alto nível e um núcleo executivo, permitindo que o programador descreva a concorrência de sua aplicação independentemente dos recursos computacionais disponíveis na arquitetura. Além disso, Anahy preserva a ordem de execução entre as tarefas, conforme especificado pelo programa. Atualmente, o ambiente Anahy encontra-se disponível para arquiteturas SMP, tendo como principais serviços, a atribuição de tarefas aos nós reais de processamento, a criação de tarefas e a sincronização entre elas e o acesso aos dados compartilhados. Para que Anahy torne-se operacional sobre aglomerados de computadores, estes serviços necessitam ser aplicáveis ao contexto distribuído e *threads* Anahy precisam ser migradas entre os nós do aglomerado, fazendo-se necessária a introdução de serviços de comunicação entre os nós.

Dentre as diferentes ferramentas que podem ser empregadas para comunicação, tais como [10, 17, 2], identificou-se nas Mensagens Ativas [19] as que introduzem menos sobrecustos na execução de uma aplicação. Existem diversos ambientes que implementam este mecanismo, no entanto nenhum deles adequou-se às necessidades de implementação de Anahy. Portanto, um novo mecanismo de comunicação necessitou ser implementado [15]. Este artigo apresenta tal mecanismo, o qual foi disponibilizado sob a forma de uma biblioteca de funções em C e validado através de seu uso em uma aplicação real, o alinhamento de seqüências biológicas, desenvolvida seguindo os modelos de programação e execução de Anahy.

O restante deste artigo está organizado como segue.

* Trabalho desenvolvido como dissertação de Mestrado, parcialmente financiado por PROSUP/Capes.

A Seção 2 destaca as ferramentas de Mensagens Ativas e multiprogramação leve utilizadas na implementação do mecanismo de comunicação desenvolvido. Nas seções 3 e 4 são descritos, respectivamente, alguns ambientes de programação que implementam o mecanismo de Mensagens Ativas e a biblioteca desenvolvida. Na Seção 5 faz-se uma análise dos resultados de desempenho obtidos com o uso de tal biblioteca como suporte à execução de uma aplicação para alinhamento de seqüências biológicas em aglomerados de computadores. A Seção 6 apresenta uma proposta de extensão da interface de programação do ambiente Anahy de forma a possibilitar a operacionalização de tal ambiente em arquiteturas com memória distribuída. Algumas considerações finais são apresentadas na Seção 7.

2. Programação em Aglomerados de Computadores

A utilização eficiente dos recursos de processamento disponíveis em aglomerados de computadores requer que sejam exploradas, simultaneamente, as concorrências intra e entre-nós. O emprego destes dois níveis se justifica pelo ganho que pode ser obtido pelo recobrimento de parte do tempo gasto em comunicações pela execução de cálculo efetivo [18].

Para exploração da concorrência intra-nó utiliza-se, comumente, *multithreading*. Com isso, é possível que vários fluxos de execução (*threads*) sejam criados no interior de um processo e compartilhem os recursos alocados a este processo. Em particular, existe o compartilhamento de um espaço de endereçamento para comunicação de dados, sendo necessário garantir um acesso correto a este espaço com o emprego de algum mecanismo de sincronização para este fim. Para exploração da concorrência entre-nós, existe a necessidade de troca de dados e tarefas entre os nós da arquitetura. Para tanto, diferentes ferramentas de comunicação podem ser empregadas [10, 17, 2, 19]. Dentre estas ferramentas, o mecanismo de Mensagens Ativas, normalmente, encontra-se entre os mais rápidos [14], permitindo realizar comunicações sem introduzir grande quantidade de sobrecustos de execução [20].

O mecanismo de Mensagens Ativas é assíncrono e tenta explorar toda a flexibilidade das modernas redes de computadores [19], buscando diminuir o tempo entre a produção e o consumo dos dados. Neste caso, 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 [14]. 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.

O funcionamento do mecanismo de Mensagens Ativas pode ocorrer por interrupção ou *polling*, segundo uma das três variantes: modelo original [19], *UpCall* [4] e *PopUp* [4]. Dadas as características de cada variante, a opção por uma delas deve considerar o(s) tipo(s) de serviço(s) a ser(em) executado(s) (tamanho, bloqueante ou não-bloqueante, freqüência de ativação, entre outros). Em particular, destaca-se que o modelo original permite que atividades de comunicação executem de forma concorrente com a aplicação por ser implementado em hardware. Nas demais variantes, esta concorrência é buscada pelo uso de *multithreading*. Buscando equacionar as características destas variações em função do modelo de execução de Anahy, é proposta uma nova variante: fila de execução. Esta introduz um *daemon* de comunicação para tratamento das mensagens e um número n de *threads* para execução das tarefas definidas pelo programa em execução. Com isso, é possível que várias tarefas executem simultaneamente, mantendo os processadores ocupados com cálculo efetivo o maior tempo possível. A concorrência das atividades de comunicação e da aplicação é obtida através do *pipeline* de fluxos de execução determinado pelo *multithreading* [18].

Embora se mostrem eficientes para comunicação em aglomerados, as Mensagens Ativas apresentam-se unicamente assíncronas. Isso torna necessário o emprego de mecanismos de controle externos que permitam uma correta evolução do programa em execução. Em geral, este controle fica a cargo do programador da aplicação e não do mecanismo de Mensagens Ativas.

3. Ambientes que Implementam Mensagens Ativas

Na literatura são encontrados diversos ambientes que empregam o mecanismo de Mensagens Ativas. Entre estes estão Split-C, Nexus, Chant, Athapascan-0 e Anahy.

3.1. Split-C

Split-C [7] é uma extensão paralela da linguagem C, que suporta o acesso a um espaço de endereçamento global em arquiteturas com memória distribuída. O modelo de programação seguido é o SPMD (*Single Program Multiple Data*), onde todos os processadores iniciam a execução em um ponto de um código comum, seguindo seu próprio fluxo de controle conforme seus dados. A sincronização entre processadores é possível através de barreiras. As Mensagens Ativas são um recurso de implementação do próprio ambiente, não sendo oferecido acesso a estes recursos ao

programador. Esta camada tem seus serviços disponibilizados pela interface SPMA [5], construída para manipular diretamente um hardware de comunicação.

3.2. Nexus

Nexus [8, 9] consiste em uma camada de suporte à execução de aplicações paralelas irregulares, oferecendo uma arquitetura do tipo *Grid Computing*. Neste ambiente, uma aplicação consiste em um conjunto de *threads* executando dentro de contextos (espaços de endereçamento), os quais representam os recursos físicos de processamento que são mapeados para um conjunto de nós. Dentro de um mesmo contexto, as *threads* comunicam-se por meio da memória compartilhada; estando em contextos diferentes, elas podem disparar requisições de serviços remotos (RSR) através de ponteiros globais. Um ponteiro global consiste em uma estrutura de dados que contém uma referência direta a um objeto dentro de um determinado contexto. Uma RSR resulta na execução de uma função especial, chamada *handler*, no contexto referenciado pelo ponteiro global. Esta função é invocada assincronamente dentro do contexto; nenhuma ação, tal como a execução de um *receive*, precisa ser tomada para que a função seja executada. Esta execução pode ocorrer de dois modos: em uma nova *thread* ou em uma *thread* pré-alocada, caso a função não seja bloqueante. Cabe salientar que para uma RSR não existe confirmação ou retorno de resultados e a *thread* chamadora não permanece bloqueada.

3.3. Chant

Chant [12] é um ambiente para programação paralela desenvolvido para sistemas com memória distribuída baseado em *threads* comunicantes (*talking threads*). A comunicação é permitida através de primitivas *send()* e *receive()*, independente de as *threads* estarem ou não no mesmo espaço de endereçamento. Chant estende a interface e as funcionalidades do padrão POSIX para *threads* (Pthreads) [13] para uma categoria de objetos computacionais chamados *chanters*. Para criar uma *thread chanter*, local ou remota, a primitiva *pthread_create()* de Pthreads foi estendida para *pthread_chanter_create()*, que retorna o identificador global único da *thread* criada. Para situações onde a comunicação entre *chanters* não pode ser realizada através da troca de mensagens ponto-a-ponto, Chant provê um mecanismo também denominado de RSR, que permite que funções sejam executadas em um processo remoto. O suporte à comunicação em Chant oferece a possibilidade de realizar troca de dados ponto-a-ponto entre *threads* e a invocação remota de *threads*. Este segundo serviço é viabilizado pela introdução no ambiente de execução de uma *thread* especializada (*server thread*), que aguarda o recebimento de mensagens solicitando a execução de um serviço. Ao receber uma mensagem, a *server thread* passa a executar o serviço solicitado, possuindo prioridade de execução mais alta em relação às demais *threads*. Em Chant, os serviços de Mensagens Ativas estão disponíveis ao programador, mas deve-se ter cuidado para que os serviços não sejam bloqueantes, de forma a não ocorrerem situações de *deadlock*.

3.4. Athapascan-0

O núcleo executivo de Athapascan-0 [11, 3] combina multiprogramação e comunicação, fornecendo um conjunto de primitivas para a realização de troca de mensagens. Athapascan-0 possibilita a criação de *threads* locais e remotas. No caso de serem criadas localmente, as *threads* comunicam-se por meio da memória compartilhada e o núcleo oferece mecanismos de sincronização (como *mutexes*, semáforos e variáveis de condição) para garantir exatidão no acesso concorrente aos dados compartilhados. No caso de *threads* remotas, sua criação ocorre através de uma chamada a um procedimento remoto assíncrono, correspondendo à execução de um serviço. Athapascan-0 introduz, também, a possibilidade de uso de "mensagens urgentes", as quais são tratadas por um *daemon* especializado, que reage ao recebimento de uma mensagem pela ativação imediata do serviço solicitado - estes serviços não devem realizar nenhuma operação de sincronização.

3.5. Anahy

O objetivo do ambiente Anahy [6] é oferecer recursos para a exploração do processamento de alto desempenho, provendo tanto uma interface de programação concorrente de alto nível, como um núcleo executivo. Com isso, o programador é capaz de descrever a concorrência de sua aplicação independentemente dos recursos computacionais disponíveis na arquitetura. Além disso, a atribuição das tarefas aos nós reais de processamento, a criação e sincronização de tarefas e o controle de acesso aos dados compartilhados, passam a ser atribuições do ambiente, não mais do programador. A interface de programação de Anahy oferece ao programador operações do tipo *fork/join* para criação e sincronização de fluxos de execução. Tais operações correspondem, respectivamente, às operações *pthread_create* e *pthread_join* do padrão POSIX para *threads*. Em Anahy, as correspondentes para estas duas primitivas são, respectivamente, *athread_create* e *athread_join* e, por questões de compatibilidade com o padrão POSIX, trabalham da mesma maneira que suas respectivas correspondentes neste padrão. O núcleo executivo de Anahy explora um grafo de de-

pendências entre tarefas, garantindo a correta evolução do programa. Além disso, é responsável pelo mapeamento das tarefas nos recursos físicos de processamento, pelo gerenciamento das listas de tarefas e pela distribuição da carga gerada entre os nós que compõem a arquitetura. Em Anahy, as Mensagens Ativas são utilizadas como recurso do ambiente para trocas de dados e tarefas entre os nós da arquitetura, não sendo de conhecimento do programador sua utilização.

4. Mecanismo Desenvolvido

A operacionalização de Anahy em ambientes distribuídos requer que os serviços de atribuição de tarefas aos nós reais de processamento, criação de tarefas e sincronização entre elas e controle de acesso aos dados compartilhados, atualmente disponíveis no contexto local a um nó [6], estejam igualmente disponíveis no contexto global. Além destes serviços, toda migração de *threads* Anahy entre os nós do aglomerado deve ocorrer de maneira transparente ao usuário.

Embora tenham sido destacadas algumas bibliotecas que implementam o mecanismo de Mensagens Ativas, nenhuma mostrou-se capaz de atender as necessidades de implementação do ambiente Anahy. A biblioteca Split-C, primeira a empregar Mensagens Ativas, possui seu suporte implementado em hardware. Chant e Athapascan-0 realizam as comunicações com base no padrão MPI, o qual não apresenta resultados satisfatórios quando empregados em redes padrão Ethernet, com a qual optamos por trabalhar. Por fim, no ambiente Nexus, uma RSR implica na ativação de um serviço dentro de uma *thread*, não permitindo a composição das tarefas Anahy. Por isso, tornou-se imprescindível o desenvolvimento do mecanismo aqui apresentado. Tal mecanismo [15] teve seu desenvolvimento baseado no princípio de atender a solicitação de execução de uma tarefa o mais rápido possível, procurando minimizar a adição de sobrecusto ao tempo total da aplicação. Para tanto, foram empregadas, simultaneamente, as ferramentas de *multithreading* e Mensagens Ativas apresentadas, de maneira a utilizar eficientemente os recursos de processamento disponíveis em um aglomerado de computadores.

Duas estruturas foram introduzidas para oferecer suporte ao mecanismo de escalonamento: os processadores virtuais e o *daemon* de comunicação. Os processadores são *threads* especializadas na execução de uma tarefa específica e não realizam comunicações nem tratamento de mensagens. Toda troca de mensagens entre os nós do aglomerado e tratamento das mensagens enviadas ou recebidas por um nó é de responsabilidade do *daemon* de comunicação. Com isso, os processadores virtuais dedicam-se exclusivamente à realização de cálculo efetivo, possibilitando que o tempo gasto em comunicações seja parcialmente sobreposto por computação [18].

O módulo implementado fez uso do serviço de *sockets*, do sistema operacional GNU/Linux, para possibilitar a comunicação entre os nós do aglomerado. Optou-se pela utilização de *sockets* por este mecanismo ser implementado sobre o protocolo TCP, diretamente suportado pelo hardware disponível (rede padrão Ethernet), apresentando bons resultados de desempenho [1]. A implementação das funcionalidades do módulo considerou, também, o padrão POSIX para *threads*, por buscar-se a portabilidade de código, a exemplo da construção dos demais módulos de Anahy.

Uma visão esquemática do mecanismo implementado pode ser observada na Figura 1, correspondendo a uma nova variante do modelo básico de Mensagens Ativas, denominado fila de execução. Toda mensagem entregue a um nó é retirada da interface de rede pelo *daemon* de comunicação (1). Se esta mensagem corresponde a uma tarefa que já pode ser executada, ela é colocada em uma lista de tarefas prontas (2); caso contrário (dados de entrada não disponíveis), ela é colocada em uma lista de tarefas que aguardam para serem liberadas para execução. Quando um dos processadores virtuais do nó encerrar o processamento que estava realizando, ele consulta a lista de tarefas prontas para verificar a disponibilidade de tarefas para execução (3). Caso a lista não esteja vazia, ele retira a primeira tarefa e passa a executá-la; caso contrário, ele permanece bloqueado até que a inclusão de uma tarefa na lista o desbloqueie e ele passe a executá-la. Como as tarefas somente são atribuídas aos processadores virtuais quando estão prontas para execução, existe garantia de que a evolução da aplicação será correta.

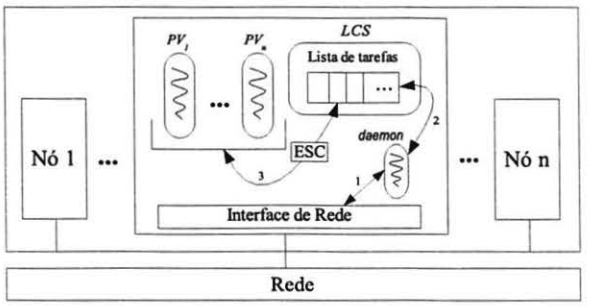

Figura 1. Visão esquemática do mecanismo de Mensagens Ativas.

No entanto, o mecanismo apenas garante a distribuição de tarefas e dados entre os nós do aglomerado. O controle de acesso aos dados compartilhados, como é o caso das lis-

tas de tarefas, deve ser realizado pelo programador (representado na Figura 1 pelo módulo LCS - Lógica de Controle Semântico). Também é de responsabilidade do programador controlar a atribuição de tarefas aos processadores virtuais (na Figura 1, representado por ESC - Escalonamento). Com a operacionalização de Anahy sobre aglomerados de computadores, estas atividades de escalonamento passarão a ser atribuições do ambiente, não mais do programador.

O mecanismo implementado foi materializado sob a forma de uma biblioteca de funções cujas primitivas são apresentadas e brevemente descritas na Tabela 1.

Nome	Descrição
act_msg_regserv	Recebe um nome de função e a registra na tabela de serviços do *daemon*.
act_msg_create	Aloca espaço em memória, fazendo o *buffer* da mensagem apontar para o novo espaço.
act_msg_init	Inicializa a mensagem com o serviço previamente retornado por **act_msg_regserv**.
act_msg_pack	Copia n bytes da área de dados para dentro do *buffer* da mensagem.
act_msg_unpack	Copia n bytes do *buffer* da mensagem para uma área de memória.
act_msg_send	Envia uma mensagem para o *host* destino.
act_msg_reset	Desaloca o espaço em memória utilizado pelo *buffer* da mensagem e atribui -1 ao identificador de serviço.
av_init	Abre as conexões entre todos os nós e inicializa os *daemons* em todos eles.
av_terminate	Finaliza os *daemons* em todos os nós do aglomerado e fecha todas as conexões abertas.

Tabela 1. Primitivas da biblioteca de Mensagens Ativas.

5. Análise de Desempenho

Dois conjuntos de testes foram realizados para avaliar o mecanismo implementado. O primeiro consiste na análise de desempenho através de uma aplicação sintética e o segundo explora uma aplicação real cujas características a tornam própria ao modelo de execução de Anahy. Os testes foram realizados em um aglomerado composto por 12 máquinas XEON 2.8GHz, biprocessadas, dotadas de 2GB de memória principal, executando Gentoo 2.6.8, rede Gigabit Ethernet.

A aplicação sintética consiste na realização de n envios de um vetor de t bytes entre um nó origem e um nó destino. O recebimento da mensagem implica na execução de um serviço com peso computacional c. O gráfico na Figura 2 apresenta os resultados para $n = 100$ e $t = 300$ com $c = \{grande\}$. Os resultados apresentados correspondem a uma média de 100 execuções, sendo que o desvio padrão medido não é apresentado por ter sido inferior a 1% em todos os casos. O objetivo do experimento é verificar a reatividade do mecanismo implementado em função do número de processadores virtuais presentes no nó.

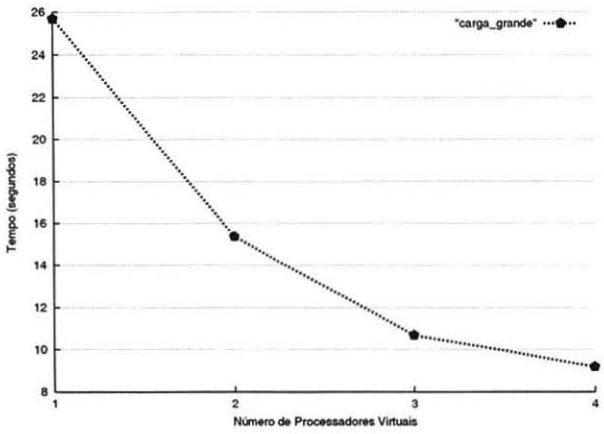

Figura 2. Tempos de transmissão de um vetor de 300 elementos entre dois nós.

O que pode ser observado no gráfico da Figura 2 é a redução do tempo de processamento à medida que o número de processadores virtuais aumenta. Isso demonstra que o objetivo de atender uma solicitação de execução de uma tarefa o mais rápido possível foi atendida pelo mecanismo. Além disso, a redução de tempo com o aumento do número de processadores virtuais indica que estes introduzem pouca sobrecarga ao tempo total de processamento.

O segundo conjunto de testes [15] explora a estrutura de execução de uma estratégia de programação dinâmica, base para algoritmos de alinhamento de seqüências biológicas [16]. Esta aplicação descreve um grafo de tarefas em termos de dependências de dados, descrevendo um caminho crítico de execução, caracterizando uma aplicação típica para Anahy. A Figura 3 apresenta um grafo para esta aplicação composto por 25 tarefas e as dependências entre estas tarefas são indicadas pelas setas entre elas. Por exemplo, as setas que partem de 1 para 2, 3 e 5 indicam que 1 produz resultados que serão utilizados como dados de entrada por 2, 3 e 5 e estes últimos somente iniciam suas execuções quando 1 tiver produzido seus resultados.

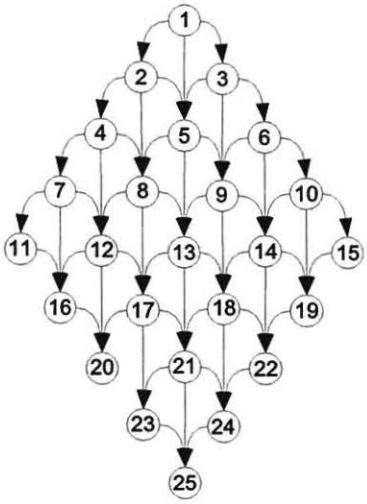

Figura 3. Grafo de dependências para uma aplicação composta de 25 tarefas.

O processo do alinhamento se caracteriza pela execução de um algoritmo em dois passos básicos. O primeiro passo consiste no preenchimento de uma matriz de tamanho $(m+1) \times (n+1)$, onde m e n correspondem ao tamanho das seqüências a serem alinhadas. Essa matriz é preenchida de acordo com a relação de recorrência (1), onde *g* corresponde à pontuação por um espaço (*gap*) inserido no alinhamento e *p(i,j)*, à pontuação pelo alinhamento dos elementos *i* e *j* da matriz.

$$A[i,j] = max \left\{ \begin{array}{c} A[i,j-1] + g \\ A[i-1,j-1] + p(i,j) \\ A[i-1,j] + g \\ 0 \end{array} \right\} \quad (1)$$

De acordo com a relação de recorrência (1), o cálculo de cada célula da matriz depende de três valores previamente calculados ([*i*, *j*-1], [*i*-1,*j*-1] e [*i*-1,*j*]).

O segundo passo consiste em encontrar o alinhamento propriamente dito entre as duas seqüências. Partindo-se da célula da matriz com o maior valor, faz-se uma operação de *traceback* na matriz, sempre se optando pelo vizinho com valor mais alto. No caso de mais de um vizinho com a mesma pontuação, dá-se preferência àquele que se encontra na diagonal da matriz, evitando a inclusão de um *gap* em uma das seqüências que estão sendo alinhadas.

Dada a característica de recorrência do primeiro passo e as necessidades computacionais do mesmo, o interesse encontra-se em realizar o cálculo concorrente da matriz de similaridades. Conforme as dependências de dados existentes entre os elementos da matriz, esta pode ser preenchida linha a linha, coluna a coluna ou pelas antidiagonais. Aos dois primeiros casos não é vantajoso introduzir concorrência devido às dependências existentes entre os elementos de uma mesma linha ou coluna e o conseqüente número de sincronizações necessárias. São calculadas, então, as antidiagonais em paralelo e para diminuir a quantidade de comunicações e sincronizações necessárias para o cálculo dos elementos da matriz, esta é dividida em blocos de *q* linhas e *r* colunas. Considerando-se o grafo da Figura 3, este refere-se à representação de uma matriz de similaridades composta de 25 blocos. As dependências existentes entre os blocos de células são as mesmas existentes entre as células.

A alocação de tarefas aos nós reais de processamento é realizada de maneira que blocos pertencentes à mesma antidiagonal (fluxo de execução) sejam atribuídos ao mesmo nó, de forma a explorar a localidade física dos dados, diminuindo o número de comunicações e sincronizações necessárias. No caso do grafo da Figura 3, são identificados 9 fluxos de execução, como, por exemplo, o fluxo formado pelos nós 2, 8, 17 e 23.

Os resultados apresentados no gráfico da Figura 4 correspondem ao alinhamento de duas seqüências de 100.000 elementos cada uma. Para este caso, a matriz de similaridades foi dividida em 400 blocos (20 x 20), totalizando blocos com 25.000.000 de elementos cada um. As mensagens trocadas entre os nós têm o tamanho correspondente a um bloco (25.000.000 de bytes) mais dois inteiros identificando as posições *i* e *j* do bloco, totalizando 25.000.008 bytes. Os tempos apresentados estão referenciados em segundos e são uma média de 100 execuções para cada valor apresentado (desvio padrão insignificante).

O gráfico da Figura 4 indica ganho de desempenho tanto com a introdução de concorrência intra-nó (aumento do número de processadores virtuais), quanto entre-nós (aumento do número de nós do aglomerado). Observa-se, ainda, que a adição de concorrência entre-nós possibilitou ganho considerável de desempenho, com uma redução de cerca de 50% do tempo de processamento quando se aumentou de 1 para 2 o número de nós. Destaca-se, também, que ocorre ganho de desempenho mesmo quando é utilizado um único processador virtual em cada nó. Este ganho foi possível, pois existe em cada nó uma segunda *thread* dedicada exclusivamente à comunicação entre os nós - o *daemon*. A existência deste permite que o processador virtual dedique-se exclusivamente ao cálculo do bloco destinado a ele. A presença do *daemon* de comunicação também possibilitou a obtenção de ganho quando um segundo processador virtual foi introduzido no nó. O ganho, neste caso, foi possível, pois cada processador virtual é atribuído para execução em um dos processadores reais do nó e o *daemon* permanece dedicado à comunicação. Por fim, observa-

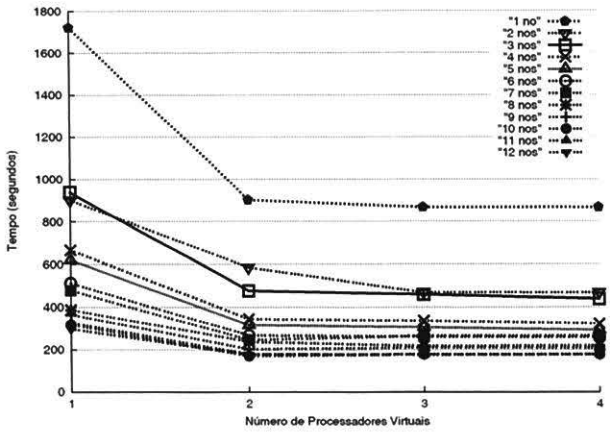

Figura 4. Tempos de execução para o alinhamento de uma matriz de 100.000.000 de elementos, dividida em 400 blocos.

se que a partir da inclusão do terceiro processador virtual, o ganho deixa de ser considerável, pois os dois processadores virtuais são suficientes para ocupar os processadores reais com as operações de cálculo intensivo do alinhamento, estando a saturação do sistema próxima ao paralelismo da máquina.

6. Extensão da Interface de Programação de Anahy

Embora a interface de programação de Anahy forneça um estilo de programação *multithreaded*, programas Anahy podem ser executados em arquiteturas com memória distribuída. O mecanismo de escalonamento de Anahy foi desenvolvido para migrar *threads* sem que o programador preocupe-se com o mapeamento das tarefas em processadores ou de dados nos módulos de memória. No entanto, o programador deve fornecer informação sobre os dados requeridos (parâmetros) e produzidos (resultados) pelas *threads*, possibilitando a transferência dos dados.

Desta forma, propõe-se um mecanismo baseado em operações *pack/unpack*, considerando a manipulação do dado tipo void* requerido pela *thread*. O uso deste mecanismo tem por objetivo viabilizar a transferência de *threads* Anahy entre diferentes nós. Os protótipos das funções *pack/unpack* para uma dada *thread* podem ser exemplificados por:

```
int packInFunc(void *in, char **buff);
int unpackInFunc(void *in, char **buff);
int packOutFunc(void *res, char **buff);
int unpackOutFunc(void *res, char **buff);
```

O primeiro parâmetro de cada função *pack/unpack* representa os dados a serem enviados (*in*) ou produzidos (*res*) para/por uma *thread*. O segundo parâmetro (*buff*) indica o *buffer* onde os dados de entrada da *thread* devem ser empacotados ou de onde devem ser lidos para serem desempacotados. Cada função retorna o tamanho (em bytes) dos dados empacotados ou desempacotados na operação.

Como as operações *pack/unpack* estão relacionadas a *threads*, o programador associa específicas funções *pack/unpack* às *threads* no *athread_attr_t*:

```
int athread_attr_setpackinput(
     athread_attr_t *attr,
     int (*func) (void *in, char **buff));
int athread_attr_setunpackinput(
     athread_attr_t *attr,
     int (*func) (void *in, char **buff));
int athread_attr_setpackoutput(
     athread_attr_t *attr,
     int (*func) (void *res, char **buff));
int athread_attr_setunpackoutput(
     athread_attr_t *attr,
     int (*func) (void *res, char **buff));
```

Uma vez que as operações *pack/unpack* são atributos de uma *thread*, elas podem ou não ser informadas no *athread_attr_t* (o valor *default* é NULL). No caso onde estes atributos não são especificados, a execução da *thread* é restrita ao nó onde foi criada.

7. Conclusão

Este artigo apresentou um mecanismo de suporte à execução de aplicações em aglomerados de computadores. Tal mecanismo foi desenvolvido para ser integrado ao ambiente de programação Anahy, de forma a viabilizar a operacionalização deste ambiente em arquiteturas com memória distribuída. A implementação realizada foi avaliada através de seu uso em uma aplicação típica de Anahy: o alinhamento de seqüências biológicas. Tal aplicação pode ser representada por um grafo que descreve as tarefas da aplicação e as relações de dependências entre elas. O mecanismo apresentou bons resultados de desempenho quando utilizado como suporte à execução desta aplicação em aglomerados de computadores.

Para melhor utilizar os recursos de processamento disponíveis em um aglomerado, o mecanismo desenvolvido combinou as ferramentas de Mensagens Ativas e *multithreading*. Por ser assíncrono, o mecanismo de Mensagens Ativas permite descrever a concorrência da aplicação, mapeando as atividades em fluxos de execução que executam simultaneamente à troca de dados e tarefas entre os nós do aglomerado.

Como suporte de execução do ambiente Anahy, este modelo, que é adaptado ao alto desempenho, é acessível

através de uma interface de alto nível, responsável por gerenciar o uso eficiente dos recursos de hardware. Esta interface disponibiliza ao usuário duas funções básicas: *athread_create* e *athread_join*, responsáveis, respectivamente, pela criação e sincronização de fluxos de execução. As demais operações necessárias à exploração eficiente dos recursos disponíveis em um aglomerado são realizadas pelo ambiente de maneira transparente ao usuário.

Referências

[1] E. D. Benitez and G. G. H. Cavalheiro. Análise Comparativa do Uso de MPI e Sockets Aplicados na Convolução de Imagens. In 4^a *Escola Regional de Alto Desempenho*, pages 321–324, Pelotas - Rio Grande do Sul - Brasil, Jan. 2004. SBC.

[2] A. Birrell and B. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computers Systems*, 2(1):39–59, feb 1984.

[3] J. Briat, I. Ginzburg, and M. Pasin. Athapascan-0b: Un Noyau Exécutif Parallèle. *Lettre du Calculateur Parallèle*, 10(3):273–293, 1998.

[4] A. S. Carissimi. *Athapascan-0: Exploitation de la Multiprogrammation Légère Sur Grappes de Multiprocesseurs*. PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, France, Sept. 1999.

[5] C.-C. Chang, G. Czajkowski, C. Hawblitzel, and T. von Eicken. Low-Latency Communication on the ibm risc System/6000 sp. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing (CDROM)*, page 24, Pittsburgh, Pennsylvania, United States, 1996. ACM Press.

[6] O. C. Cordeiro, D. S. Peranconi, L. C. Villa Real, E. C. Dall'Agnol, and G. G. H. Cavalheiro. Exploiting Multithreaded Programming on Cluster Architectures. In *Proceedings of the 19th Inernational Symposium on High Performance Computing Systems and Applications*, pages 90–96, May 2005.

[7] D. E. Culler, A. C. Arpaci-Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. A. Yelick. Parallel Programming in Split-C. In *Supercomputing*, pages 262–273, 1993.

[8] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Task-Parallel Runtime System. In *Proceedings og the 1st International Workshop on Parallel Processing*, pages 457–462. Tata McGraw Hill, 1994.

[9] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, Aug. 1996.

[10] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manachek, and V. Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, Cambridge, Massachussetts, 1994.

[11] I. Ginzburg. *Athapascan-0b: Intégration Efficace et Portable de Multiprogrammation Légère et de Communication*. PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, France, 1997.

[12] M. Haines, D. Cronk, and P. Mehrotra. On the Design of Chant: A Talking Threads Package. In *Proceedings of Supercomputing*, pages 350–359, Washington D.C., Nov. 1994. ACM/IEEE.

[13] IEEE. *IEEE 1003.1c-1994: Standard for Information Technology — Portable Operating System Interfaces (POSIX) — Part 1: System Application Program Interface (API) — Amendment 2: Threads Extension [C Language]*. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1994.

[14] S. S. Lumetta, A. Mainwaring, and D. E. Culler. Multi-Protocol Active Messages on a Cluster of SMPs. In *Proceedings of Supercomputing'97 (CD-ROM)*, San Jose, CA, Nov. 1997. ACM SIGARCH and IEEE. University of California, Berkeley.

[15] D. S. Peranconi. Alinhamento de Seqüências Biológicas em Arquiteturas com Memória Distribuída. Master's thesis, Programa Interdisciplinar de Pós-Graduação em Computação Aplicada (PIPCA) - Unisinos, São Leopoldo, Brasil, Mar. 2005.

[16] J. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. Brooks-Cole, 1997.

[17] W. R. Stevens. *UNIX Network Programming - Networking APIs: Sockets and XTI*, volume 1. Prentice Hall PTR, Upper Saddle River, NJ, 2 edition, 1998.

[18] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990.

[19] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings the 19th Annual International Symposium on Computer Architecture,ACM SIGARCH*, volume 20, pages 256–266, Gold Coast, Australia, May 1992.

[20] D. A. Wallach, W. C. Hsieh, K. L. Johnson, M. F. Kaashoek, and W. E. Weihl. Optimistic Active Messages: A Mechanism for Scheduling Communication with Computation. *ACM SIGPLAN Notices*, 30(8):217–226, Aug. 1995.