UMA IMPLEMENTAÇÃO DO MODELO LINDA PARA PROGRAMAÇÃO PARALELA EM TRANSPUTERS

Andréa Schwertner Charão*
Rita de Cássia Pivetta Machado †
Celso Maciel da Costa‡
Wolfgang Pandikow§

Pós-Graduação em Ciência da Computação
Instituto de Informática – Universidade Federal do Rio Grande do Sul
Av. Bento Gonçalves, 9500 – Bloco IV
CEP:91501-970 - Porto Alegre - RS - Brasil
Fax:(051)336-5576

Sumário

Embora as máquinas paralelas atuais possam atingir alto desempenho, a programação neste tipo de plataforma geralmente é bastante rudimentar e intimamente ligada à sua arquitetura. Deste modo, ferramentas de mais alto nível, como a linguagem Linda, têm sido propostas para facilitar a programação em ambientes paralelos. Este trabalho apresenta a implementação de Linda sobre um sistema multiprocessador baseado em *transputers*. Seu principal objetivo é oferecer uma interface de programação mais simples e independente de máquina para o desenvolvimento de aplicações paralelas neste sistema.

Abstract

Despite the high performance achieved by parallel machines nowadays, programming on such platforms usually is a low level task, that requires a previous knowledgement of the system architecture. Therefore, higher level tools, such as the Linda language, have been proposed to make easy the programming on parallel environments. This paper presents a Linda implementation on a multiprocessor system based on transputers. The main goal of this work is to provide a simpler, machine independent interface for the development of parallel applications on this system.

*B.Sc.(UFSM-1993); Mestranda em Ciência da Computação. E-mail: andrea@inf.ufrgs.br
†B.Sc.(UFSM-1993); Mestranda em Ciência da Computação. E-mail:cassia@inf.ufrgs.br
‡Ph.D.(Grenoble-1993); Professor Adjunto do CPGCC-UFRGS. E-mail: celso@inf.ufrgs.br
§Ph.D.(Berlim-1970); Professor Visitante do CPGCC-UFRGS. E-mail: pand@inf.ufrgs.br

1 Introdução

Embora as máquinas multiprocessadoras possam alcançar altos níveis de desempenho explorando o paralelismo na execução de tarefas, este tipo de arquitetura impõe sérias dificuldades ao desenvolvimento de aplicações, especialmente quando os processadores do sistema não compartilham memória. Neste caso, a única forma de interação entre os mesmos é a troca de mensagens, e o modelo de programação utilizado no sistema geralmente baseia-se neste paradigma de comunicação. Um dos maiores problemas associados com a troca de mensagens, porém, é que as primitivas usadas para comunicação são na verdade operações de entrada e saída, de baixo nível de abstração, que dificultam o desenvolvimento de programas paralelos. Este paradigma de comunicação também torna-se inadequado para um grande número de aplicações e algoritmos onde existe a necessidade do compartilhamento de informações.

Desta forma, um mecanismo de comunicação de mais alto nível, conhecido como memória compartilhada distribuída [NIT91], tem sido proposto com o objetivo de facilitar a programação em sistemas paralelos. Através deste tipo de memória é possível o compartilhamento de dados em sistemas que não possuem memória fisicamente compartilhada. Entre as ferramentas de suporte à programação paralela, baseadas neste modelo de memória, pode-se destacar a linguagem **Linda** [GEL85]. A memória global, compartilhada por todos os processos de um programa em Linda é chamada **espaço de tuplas**. Através de um pequeno conjunto de operações definidas pela linguagem, usadas para manipular o espaço de tuplas, é possível a comunicação e a sincronização de processos.

Ferramentas de programação baseadas em memória compartilhada, como Linda, oferecem um maior nível de abstração aos mecanismos usados para o controle do paralelismo, tornando mais simples e também mais portáveis os programas desenvolvidos através destas ferramentas. Neste trabalho será apresentada a implementação do modelo Linda sobre um sistema paralelo formado por uma rede de processadores *transputer* [INM90]. Originalmente, os processadores do sistema não compartilham memória e a única forma de comunicação entre processos locais ou residentes em diferentes nodos da rede é a troca de mensagens. Além disso, o ambiente de programação disponível é bastante rudimentar, e exige que o programador conheça detalhes do funcionamento do próprio *hardware*. A implementação de Linda traz para o sistema todas as vantagens de uma interface de programação de mais alto nível, baseada em um modelo de memória compartilhada distribuída.

2 O Modelo Linda

Embora até o momento Linda tenha sido tratada como uma linguagem para programação paralela, esta ferramenta consiste, na verdade, em um conjunto res-

trito de operações que, ao serem incorporadas em uma linguagem seqüencial, permitem que a mesma seja utilizada para o desenvolvimento de programas paralelos. As operações definidas em Linda são empregadas na manipulação de uma memória global, compartilhada por todos os processos de um programa, denominada **espaço de tuplas** (*Tuple Space* — TS) [GEL85]. Através destas primitivas, usadas para incluir, remover ou apenas ler tuplas do TS, é possível a comunicação e a sincronização de processos em Linda.

2.1 Tuplas

Os elementos do TS, chamados de **tuplas** [BAL89], podem ser definidos como seqüências de valores ou dados. Tuplas não são acessadas por endereços, como é o caso de variáveis compartilhadas, mas sim por seu próprio conteúdo. Para se especificar uma tupla é necessário determinar o **valor** ou o **tipo** de cada um de seus campos de dados. O primeiro campo de uma tupla deve necessariamente apresentar um valor e geralmente é usado como chave para identificar a mesma dentro do espaço de tuplas.

Os campos de uma tupla para os quais somente é especificado o tipo são chamados **campos formais**, e geralmente funcionam como recipientes para dados. Já aqueles campos que além de um tipo, apresentam também um valor são denominados **campos reais**. Estes últimos são representados por meio de constantes ou variáveis definidas na linguagem hospedeira (linguagem na qual foram adicionadas as operações Linda). O tipo e o valor associados a um campo real são iguais ao tipo e ao valor da constante ou variável usada para representá-lo. De maneira semelhante, um campo formal é determinado por uma variável, definida na linguagem hospedeira, e o seu tipo é igual àquele definido para a respectiva variável. Admitindo-se, por exemplo, que a variável **w** tenha sido definida na linguagem (hospedeira) C como:

```
char w = 'A';
```

Então a tupla:

```
("alfanum", w, 5)
```

contém três campos **reais**: o primeiro campo, que serve como chave, é uma *string* de valor **'alfanum'**, o segundo é do tipo caracter e seu valor é **'A'** e o último campo é um número do tipo inteiro, representado pela constante **5**. Já a tupla

```
("numero", ?x)
```

possui apenas um campo real — a *string* **'numero'**, que é a sua chave. O segundo campo, representado pela variável **x**, é **formal** (o símbolo **'?'** será usado no texto para denotar um campo formal) e seu tipo é determinado pelo tipo desta variável.

2.2 Primitivas Linda

Em Linda, as primitivas definidas para manipulação do espaço de tuplas são: **out, in, read, e eval** [LEL90, AHU86]. Tais primitivas podem ser tratadas como chamadas de procedimento que recebem como parâmetro a especificação de uma tupla a ser manipulada. A descrição destas operações é apresentada a seguir.

2.2.1 A Primitiva OUT

Esta operação adiciona uma tupla no TS e é assíncrona, ou seja, não bloqueia o processo que a executa. Como exemplo, para se incluir a tupla `("XYZ", 5.7)` no TS, é necessário apenas a execução de:

```
out("XYZ", 5.7)
```

Os campos da tupla fornecidos a uma operação **out**, exceto a chave, podem também ser formais. A tupla inserida através desta operação permanece no espaço de tuplas até ser explicitamente removida. É possível incluir no TS várias cópias da mesma tupla.

2.2.2 A Primitiva IN

O objetivo desta primitiva é ler e remover uma determinada tupla do TS. A tupla a ser removida deverá apresentar campos equivalentes aos da tupla especificada pela operação (também chamada de tupla modelo). Ao contrário de **out**, a primitiva **in** é bloqueante, e caso a tupla desejada não se encontre no espaço de tuplas, o processo que executou a chamada é suspenso até que uma tupla adequada seja inserida no TS. No momento em que for encontrada uma tupla que satisfaça a operação, os valores dos seus campos reais serão atribuídos aos respectivos campos formais (se houverem) da tupla modelo. Deste modo, a operação:

```
in("XYZ", ?w)
```

irá procurar no TS uma tupla contendo dois campos, na qual o campo chave seja a string "XYZ" e cujo segundo componente apresente o mesmo tipo de dado que a variável **w**. Considerando-se que **w** tenha sido definida como um número de ponto flutuante, a tupla `("XYZ", 5.7)`, adicionada ao TS pela operação **out** mostrada na seção anterior, poderia satisfazer a operação **in** e desta forma, o valor **5.7** seria atribuído à **w**. Se existissem no TS outras tuplas que pudessem ser recuperadas pela operação, qualquer uma destas poderia ter sido escolhida.

Além da chave, que deve necessariamente possuir um valor, os demais campos da tupla modelo em uma operação **in** também podem ser reais. Neste caso, para cada campo real da tupla modelo, será necessário que a tupla selecionada do TS possua um campo real correspondente, de mesmo tipo e mesmo valor, ou um campo formal de mesmo tipo. Como exemplo, a operação:

```
in("T", 39, ?w),
```

poderá retirar do TS a tupla `("T", 39, 6.8)`, mas não a tupla `("T", 45,`

6.8). Esta maneira de se identificar uma tupla — utilizando-se outros campos, além do primeiro, é chamada de **nomeação estruturada** e geralmente é empregada para se diferenciar tuplas que apresentem a mesma chave.

2.2.3 A Primitiva READ

Esta operação funciona de forma similar à primitiva **in**, porém a tupla selecionada não é removida do espaço de tuplas. Desta forma, a operação `in("XYZ", ?w)`, mostrada na seção anterior, deve ser substituída por:

```
read("XYZ", ?w),
```

para que a tupla selecionada possa permanecer no TS.

2.2.4 A Primitiva EVAL

Assim como a operação **out**, **eval** também adiciona uma tupla no TS. Porém, antes que a tupla seja inserida, seus campos são avaliados por um processo criado implicitamente pela operação. Tuplas contendo campos formais também podem ser fornecidas à primitiva **eval**. Para ilustrar o uso desta operação, admita-se que **dif** seja uma função usada para calcular a diferença entre dois números. Então, a execução de:

```
eval("G", 2, dif(203, 67)),
```

criará um novo processo para executar a função **dif(203, 67)**, o qual irá prosseguir concorrentemente com o processo que efetuou a chamada **eval**. Após o novo processo ter calculado a diferença entre **203** e **67** (igual a **136**), a tupla resultante, `("G", 2, 136)`, será inserida no TS da mesma forma que em uma operação **out**.

3 Interface Linda Implementada

O sistema implementado dispõe de três primitivas para manipulação do espaço de tuplas: **out**, **in** e **read**. Para que tais primitivas possam ser utilizadas em um programa de usuário, é necessário que o sistema seja previamente inicializado através da função **init_linda_system**, que habilita o acesso ao TS. A primitiva **eval**, originalmente,definida no modelo Linda, não se encontra disponível nesta versão do sistema.

Cada primitiva suportada recebe argumentos que especificam a tupla a ser manipulada. O primeiro deles representa a chave da tupla, e deve necessariamente ser do tipo *string*. O segundo argumento, também do tipo *string*, serve para descrever cada um dos campos da tupla, de maneira similar à função **printf** da linguagem C, conforme será visto a seguir. Esta forma de descrição dos campos elimina a necessidade de um pré-processador para identificar os tipos dos argumentos usados numa operação Linda.

O sistema suporta basicamente quatro tipos de dados: caracter (**char**), *string*, número inteiro (**int**) e de ponto flutuante (**float**). Também é possível a manipulação de vetores formados por estes tipos básicos. A cada tipo de dado corresponde um caracter usado para compor a *string* de descrição dos campos. Para a especificação de um vetor, uma constante indicando seu número de elementos deve preceder o caracter que identifica o tipo de dado. Os caracteres permitidos atualmente são:

- **c** – para campos do tipo **char**;
- **d** – para campos do tipo **int**;
- **f** – para campos do tipo **float**;
- **s** – para campos do tipo *string*.

Além da descrição do tipo de dado, também é necessária a especificação do tipo de campo, através dos caracteres **%** ou **?**, que precedem os caracteres de descrição de dados e que indicam, respectivamente, um campo **real** ou **formal**. A fim de permitir o uso de nomeação estruturada, nas primitivas **in** e **read** é possível especificar tanto campos formais quanto reais. Na primitiva **out**, porém, somente campos reais podem ser definidos, já que campos formais têm pouca utilidade prática nesta operação [NAR89].

Após a *string* de descrição de campos, devem ser fornecidos a uma primitiva Linda argumentos que correspondam aos campos especificados. Campos reais requerem argumentos passados por valor, enquanto argumentos que correspondem a campos formais devem necessariamente ser passados por referência. Argumentos associados a vetores, no entanto, devem sempre ser passados por referência, seja qual for o tipo de campo especificado.

3.1 Exemplos

Na figura 1 são apresentados alguns exemplos que ilustram o uso das primitivas suportadas pelo sistema. Para isso, deve-se considerar a definição das seguintes variáveis:

```
int v[] = {1, 2, 3};
int t[3];
```

Inicialmente, a operação **out** insere a tupla `("numeros", 5.7, <1,2,3>)` no TS, onde os símbolos '<' e '>' servem para delimitar os elementos de um vetor. Esta tupla é, a seguir, recuperada através da primitiva **read**. Por fim, a primitiva **in** remove do TS uma tupla composta apenas pelo campo chave (neste caso, a *string* de descrição dos campos é nula). No final deste artigo é apresentado um programa simples que emprega as primitivas Linda para calcular, de forma paralela, o produto de uma matriz por um vetor.

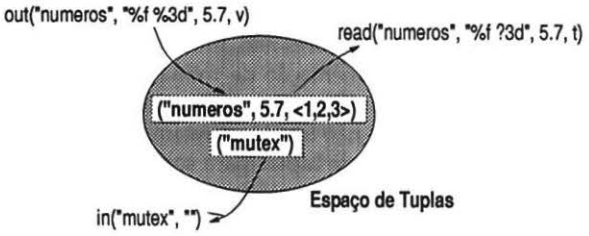

Figura 1: Exemplos de operações Linda suportadas pelo sistema.

4 Ambiente de Desenvolvimento do Sistema

Transputers oferecem várias facilidades para suporte a sistemas de processamento paralelo. Num processador deste tipo, escalonamento de processos e comunicação entre os mesmos são implementados diretamente em *hardware*[SHE87]. Além disso, cada *transputer* possui quatro *links* para conexão com outros processadores, o que permite a construção de redes de *transputers* com várias topologias. O processamento paralelo em *transputers* é baseado no modelo de processos seqüenciais comunicantes, onde processos executam concorrentemente e se comunicam uns com os outros trocando mensagens através de canais. A troca de mensagens é do tipo *rendez-vous*, e os canais de comunicação permitem a interação entre processos residentes no mesmo processador ou em processadores distintos.

A plataforma de *hardware* utilizada no desenvolvimento do sistema apresentado neste trabalho é composta por quatro *transputers* (modelo T800) ligados a um circuito de interconexão programável, formando uma rede de topologia em anel. Este circuito é conectado diretamente ao barramento de um microcomputador hospedeiro, do tipo PC/AT 386, onde é executado um programa responsável pela carga de programas paralelos na rede de *transputers*. Este programa também funciona como um servidor de arquivos, processando operações de entrada e saída que podem ser requisitadas pelo *transputer* diretamente conectado ao computador hospedeiro.

O sistema foi desenvolvido na linguagem *Parallel C* [3LL91] — uma extensão da linguagem ANSI C com funções para programação paralela em *transputers*. Tais funções implementam basicamente comunicação através de canais, sincronização através de semáforos e gerenciamento de processos e *threads*. O sistema complementa o ambiente de programação suportado pela linguagem *Parallel C*, fornecendo uma interface de mais alto nível para a comunicação e sincronização entre processos, baseada no modelo Linda de memória compartilhada.

Certas dificuldades associadas à programação paralela em *transputers* são amenizadas com o emprego das primitivas Linda. Através delas, por exemplo, diferentes paradigmas de comunicação podem ser implementados [BAL89, GEL85], o que acaba com a limitação ao estilo *rendez-vous* imposto pela arquitetura dos *transputers*.

Além disso, o mecanismo de memória compartilhada facilita a comunicação entre processos que residem em processadores não diretamente conectados, uma tarefa que originalmente exige a implementação de rotinas para o roteamento correto das mensagens. Linda também simplifica a requisição de operações de entrada e saída a partir de processadores não ligados diretamente ao sistema hospedeiro, os quais podem utilizar o espaço de tuplas para solicitar tais operações ao único *transputer* que tem acesso ao microcomputador.

5 Implementação

Para dar suporte às primitivas descritas na seção 3, o sistema é organizado em dois módulos distintos: uma **biblioteca de usuário**, onde são implementadas as operações Linda, e um **gerenciador do espaço de tuplas** (*Tuple Space Manager* — TSM), que consiste em um processo responsável pelo controle e manipulação do TS. A seguir será apresentada a organização geral do sistema e o funcionamento de cada um de seus módulos.

5.1 Organização Geral do Sistema

Como não existe memória compartilhada entre os *transputers*, o espaço de tuplas é distribuído (de forma não replicada) entre os processadores da rede, e uma instância do TSM é executada localmente em cada nodo. No entanto, para o usuário, esta distribuição é totalmente transparente e o espaço de tuplas é tratado como uma área de memória global que pode ser acessada por todos os processos de uma aplicação, não importando em que processador os mesmos residem.

A interação entre os dois módulos do sistema ocorre através da troca de mensagens, segundo um modelo cliente–servidor. Quando uma primitiva Linda é executada, os argumentos fornecidos à mesma pelo processo usuário (cliente) são enviados em uma mensagem de requisição ao TSM local (servidor). Sempre que possível a requisição é atendida localmente e os resultados da operação, se existirem, são enviados em uma mensagem de resposta ao processo cliente. Caso a requisição não possa ser satisfeita pelo TSM local, esta é enviada para os demais gerenciadores.

Requisições do processo usuário são enviadas ao TSM local através de um canal que conecta ambos os processos. Respostas às solicitações do usuário (se existirem) são enviadas pelo gerenciador através de um canal previamente fornecido na mensagem de requisição. A comunicação entre os processos gerenciadores do espaço de tuplas, por sua vez, é feita através de canais de entrada e saída, que conectam os *transputers* em uma rede de topologia em anel, conforme mostra a figura 2. As mensagens nesta rede circulam de forma unidirecional e cada TSM pode se comunicar apenas com os TSM's adjacentes, utilizando para isso dois canais de comunicação, um para a recepção de mensagens do TSM residente no

nodo anterior (canal de entrada) e o outro para o envio de mensagens ao TSM do próximo nodo (canal de saída).

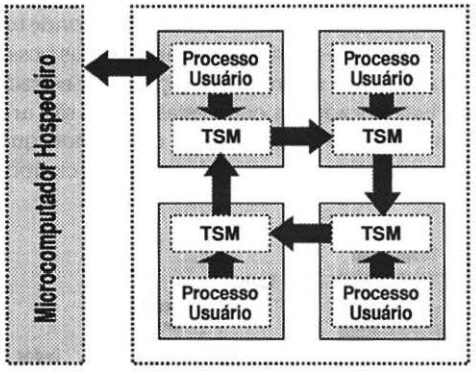

Figura 2: Organização geral do sistema.

A configuração da rede na forma de anel simplifica bastante a comunicação entre os TSM's já que cada gerenciador interage somente com os seus vizinhos. Além disso, como as mensagens circulam na rede de acordo com uma seqüência determinada pelo próprio anel, eliminam-se também os problemas que poderiam surgir com a falta de ordem no tratamento de mensagens.

5.2 A Biblioteca de Usuário

Nesta biblioteca estão disponíveis as primitivas Linda que podem ser usadas por programas em *Parallel C*. A função **init_linda_system**, implementada nesta biblioteca, serve para definir o canal de comunicação por onde são enviadas requisições ao TSM, e por isso deve ser chamada antes que qualquer operação Linda seja executada. As funções **out**, **in** e **read**, também implementadas na biblioteca de usuário, recebem argumentos que especificam uma tupla a ser manipulada, conforme descrito na seção 3. Tais argumentos são usados para compor uma mensagem de requisição que é enviada ao TSM local.

No caso de uma requisição **in** ou **read**, onde uma mensagem de resposta é esperada, um novo canal é criado dinamicamente a cada operação, e seu identificador enviado junto com a requisição. A resposta à operação é esperada através deste canal, permitindo que várias *threads* utilizem as primitivas Linda simultaneamente. Isto não seria possível se apenas um canal fixo fosse utilizado, pois uma *thread*, ao acessá-lo, poderia receber mensagens destinadas a outra *thread*.

5.3 O Gerenciador do Espaço de Tuplas

O **espaço de tuplas** gerenciado por cada TSM mantém tuplas geradas por operações **out**, e é organizado sob a forma de uma tabela *hash*, onde tuplas são localizadas a partir de suas chaves. Como é possível que várias tuplas possuam a mesma chave ou que chaves produzam um mesmo endereço, cada entrada na tabela contém ponteiros para uma lista encadeada onde estas tuplas são armazenadas. A cada tupla é associado um conjunto de descritores de campos, que especificam o tipo e o número de dados deste tipo armazenados em cada campo da tupla, conforme mostra a figura 3.

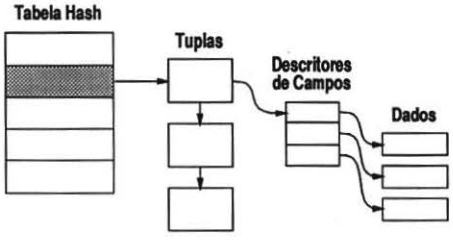

Figura 3: Estrutura do espaço de tuplas.

Outra estrutura de dados manipulada por um TSM consiste em uma **tabela de requisições pendentes**, onde são mantidas especificações de tuplas requisitadas por operações **in** ou **read** que não são satisfeitas localmente. Esta tabela é organizada e acessada de maneira semelhante ao espaço de tuplas, porém são adicionadas algumas informações a cada tupla armazenada: o tipo de requisição, a identificação do TSM requisitante e a especificação do canal de resposta. Além disso, os descritores de campos também especificam o tipo (real ou formal) de cada campo que compõe uma tupla.

Tanto o espaço de tuplas quanto a tabela de pendências são manipulados de acordo com cada tipo de requisição recebida pelo TSM. O tratamento destas requisições é apresentado a seguir.

5.3.1 A Requisição IN

Requisições **in** contêm a especificação de uma tupla modelo, cuja chave é usada pelo TSM para localizar a entrada correta na tabela *hash*. A fim de se encontrar uma tupla equivalénte, os descritores de campos de cada tupla associada a esta entrada são então comparados com aqueles contidos na tupla modelo. A equivalência entre campos formais (da tupla modelo) e reais (da tupla no TS) é feita através da comparação dos descritores de campos. Quando algum campo na tupla modelo possui valor associado (campo real), então os respectivos dados são comparados. No processamento de uma requisição **in**, três situações distintas podem ocorrer: a tupla procurada pode existir no TS local, no TS de algum outro nodo da rede

ou, ainda, pode não existir em nenhum espaço de tuplas. Se uma tupla equivalente àquela especificada na requisição é encontrada localmente, uma mensagem contendo esta tupla é enviada ao processo usuário e, a seguir, o gerenciador local remove a tupla do TS.

Quando a requisição **in** não pode ser satisfeita pelo TSM local, a tupla modelo é inserida numa tabela de requisições pendentes e, após, a requisição **in** é enviada para o gerenciador no próximo nodo. Cada TSM que recebe a requisição procede de maneira semelhante ao TSM do nodo requisitante, isto é, procura a tupla especificada localmente e, caso também não possa satisfazer a requisição, armazena-a na sua tabela de pendências e repassa-a para o TSM do próximo nodo. No entanto, se a tupla procurada é encontrada, esta é enviada numa mensagem de resposta ao TSM requisitante, podendo passar por mais de um nodo até chegar ao seu destino.

Se nenhum TSM possui a tupla especificada, a requisição **in** circula no anel até ser recebida pelo TSM que a originou, quando é então retirada do anel. Assim, uma pendência é registrada em cada TSM da rede, e a requisição só será satisfeita quando a tupla especificada for inserida em algum nodo da rede através da operação **out**.

Antes que uma mensagem de resposta alcance o TSM destino, cada gerenciador intermediário que a receber deve procurar na sua tabela de pendências uma requisição **in** equivalente. Caso seja encontrada, a pendência é removida, pois a mensagem de resposta indica que a requisição já foi atendida por outro TSM.

Quando a resposta chega ao TSM destino e a requisição **in** ainda está pendente, a resposta é enviada ao processo usuário e, após, uma mensagem solicitando a remoção das pendências restantes é enviada a cada gerenciador localizado entre o TSM que originou a requisição e aquele que forneceu a resposta. Caso não exista uma requisição pendente para a tupla contida na resposta, o que ocorre quando a requisição já foi respondida por outro gerenciador, a tupla é armazenada pelo TSM requisitante em seu próprio espaço de tuplas. Tal situação representa uma **migração** de tupla de um TS para outro.

As etapas do processamento de uma requisição **in** são mostradas esquematicamente na figura 4. Neste exemplo, a tupla procurada não existe no TS local (nodo 1), mas é encontrada no espaço de tuplas de outro nodo da rede (nodo 3).

5.3.2 A Requisição OUT

Quando uma requisição deste tipo é recebida por um TSM, sua tabela de pendências é examinada em busca de uma requisição **in** ou **read** cuja tupla modelo seja equivalente à tupla especificada na requisição **out**. Caso esta pendência seja encontrada, a tupla da requisição **out** é enviada numa mensagem de resposta ao TSM que originou a requisição pendente. Ao contrário, quando não existe pendência equivalente, a tupla é simplesmente inserida no TS local.

A figura 5 mostra o funcionamento do sistema numa situação onde uma

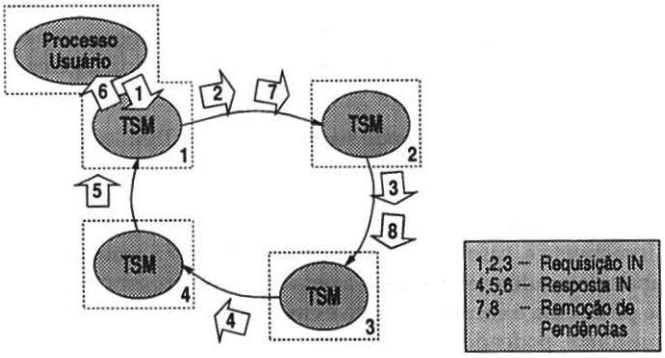

Figura 4: Processamento de uma requisição **in**.

operação **out** satisfaz uma requisição **in** pendente. Neste exemplo, a requisição **in** circula no anel até retornar ao TSM que a originou, registrando pendências em todos os nodos da rede. A resposta a esta requisição só é enviada ao processo usuário após o processamento da requisição **out**.

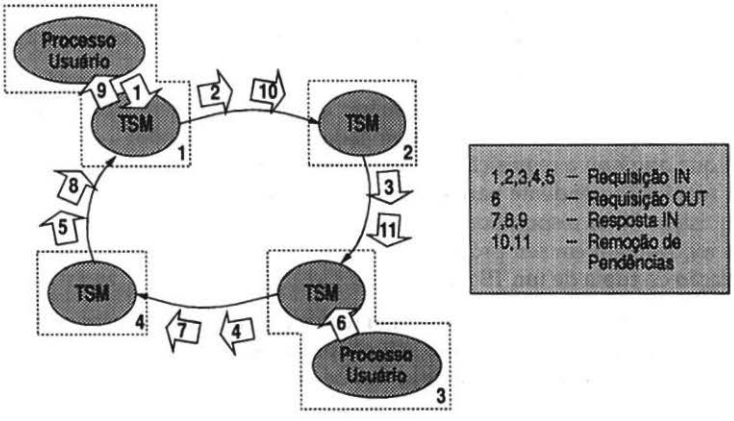

Figura 5: Processamento de uma operação **out** que satisfaz uma requisição **in** pendente.

5.3.3 A Requisição READ

Requisições **read** são tratadas por um TSM de maneira semelhante a requisições **in**. Porém, a tupla que satifaz a operação não é removida do TS e, quando uma mensagem de resposta chega ao TSM destino, ela é simplesmente descartada se a pendência equivalente não é encontrada, pois, neste caso, uma resposta anterior

já satisfez a requisição, e as pendências já foram removidas em todos os nodos da rede.

6 Desempenho do Sistema

A fim de avaliar o desempenho do sistema implementado, foram realizados alguns testes que forneceram tempos de execução das primitivas **out** e **in** em diferentes situações. Os tempos obtidos referem-se a seqüências **out/in** executadas repetidamente, num total de 1000 iterações, com tuplas compostas apenas por um campo chave (menor tamanho de tupla possível). O resultado final da cada teste representa o tempo médio calculado após o processamento de todas as iterações.

Inicialmente, cada seqüência **out/in** foi executada no mesmo processador, de modo que todas as operações **in** foram atendidas localmente. Este teste forneceu um tempo médio de $10ms$ para cada par de operações locais. Já medidas de cada operação em separado indicaram um tempo médio de $4ms$ para a operação **out**, e de $6ms$ para uma execução de **in**. A seguir, estas operações foram executadas em diferentes nodos da rede, para avaliar o custo da remoção (**in**) de uma tupla não disponível localmente. Tal experimento resultou num tempo médio de $11ms$ para operações **in** não satisfeitas localmente.

Os tempos obtidos mostram que existe um alto custo envolvido nas operações que precisam ser propagadas pela rede a fim de serem atendidas. O desempenho do sistema, mesmo para operações locais, não pode ser considerado muito alto se comparado com resultados obtidos em algumas implementações Linda existentes, como em [CAR86] e [MOR93]. Alguns destes sistemas, no entanto, são construídos sobre plataformas mais poderosas, o que justifica seu alto desempenho. Mesmo assim, algumas otimizações (discutidas na próxima seção) podem ser realizadas no sistema Linda implementado, a fim de aumentar seu desempenho.

7 Conclusões e Trabalhos Futuros

Neste artigo foi apresentada a implementação de um sistema que provê uma interface Linda para programação paralela em *transputers*. Para isso, o sistema mantém um espaço de tuplas distribuído sobre uma rede de *transputers* configurada sob forma de anel, onde cada nodo executa uma instância de um processo gerenciador do TS. A fim de permitir a manipulação de tuplas por processos usuários, o sistema fornece uma biblioteca de primitivas Linda que pode ser usada em programas escritos na linguagem *Parallel C*.

O modelo Linda de memória compartilhada simplifica o desenvolvimento de aplicações paralelas sobre a plataforma descrita neste artigo, pois fornece uma interface de programação de mais alto nível que aquela suportada pela linguagem *Parallel C*, onde processos cooperantes somente se comunicam através de troca de

mensagens.

Como atualmente o sistema não apresenta um desempenho satisfatório, algumas otimizações têm sido planejadas. Para reduzir o tempo gasto no processamento dos argumentos fornecidos a uma primitiva Linda, deverá ser construído um pré-processador para identificar os tipos dos diversos argumentos e produzir uma chamada Linda num formato que possa ser processado de forma mais eficiente pelo sistema. Além disso, deverá ser analisado o impacto da utilização de outra topologia, diferente do anel, sobre o funcionamento e o desempenho do sistema. Futuramente, também pretende-se implementar a operação **eval**, que utilizará os recursos de suporte a *threads* disponíveis no *Parallel C* para a avaliação concorrente dos campos da tupla especificada na operação.

Referências

[3LL91] 3L Ltd. **Parallel C User Guide.** Livingston: 3L Ltd., 1991, 498p.

[AHU86] AHUJA, S.; CARRIERO, N.; GELERNTER, D. *Linda and Friends*. **Computer.** New York, v.19, n.8, p.26-34, Aug. 1986.

[BAL89] BAL, H.E.; STEINER, J.G.; TANENBAUM, A.S. *Programming Languages for Distributed Computing Systems*. **ACM Computing Surveys.** New York, v.21, n.3, p.261-322, Sep. 1989.

[CAR86] CARRIERO, N.; GELERNTER, D. *The S/Net's Linda Kernel*. **ACM Trans. on Computer Systems.** New York, v.4, n.2, p.110-129, May 1986.

[GEL85] GELERNTER, D. *Generative Communication in Linda*. **ACM Trans. on Programming Languages and Systems.** New York, v.7, n.1, p.80-112, Jan. 1985.

[INM90] INMOS, Corp. **Transputer Data Book**. Bristol: INMOS, 1990.

[LEL90] LELER, W. *Linda meets Unix*. **Computer.** New York, v.23, n.2, p.43-55, Feb. 1990.

[MOR93] MORAES, S.A.S; LISTER, P.F. *The Design and Implementation of a Dinamically Distributed Linda Tuple Space*. In **Anais do XX Seminário Integrado de Software e Hardware.** Florianópolis, Set. 1993, p-503-517.

[NAR89] NAREM Jr. J.E. **An Informal Operational Semantics of C-Linda V2.3.5.** Yale University, Dec. 1989.

[NIT91] NITZBERG, B.; LO, V. *Distributed Shared Memory: A Survey of Issues and Algorithms*. **Computer.** New York, v.24, n.8, p.52-60, Aug. 1991.

[SHE87] SHEPHERD, D. **Transputer Instruction Set: A Compiler Writer's Guide.** Hertfordshire: Prentice-Hall, 1988.

Anexo

O programa a seguir, escrito em *Parallel C*, utiliza as primitivas Linda implementadas no sistema para calcular, de forma paralela, o produto de uma matriz por um vetor. Para isso, o programa é composto por um processo coordenador e por processos trabalhadores, que devem executar em processadores diferentes. Inicialmente, o coordenador insere no TS tuplas contendo as linhas da matriz e a tupla que armazena o vetor a ser multiplicado. Cada trabalhador, por sua vez, calcula um ou mais elementos do vetor resultante a partir destas tuplas existentes no TS. Os elementos calculados também são armazenados em tuplas. Por fim, o processo coordenador lê cada uma das tuplas geradas pelos trabalhadores, armazena os elementos obtidos no vetor resultante e imprime o produto calculado.

```
/* Codigo do Processo Coordenador */

#include <stdio.h>
#include <linda.h>
#define DIM 3

int mat[DIM][DIM] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
int vet[DIM] = {11, 22, 33};

void main(int argc, char *argv[], char *envp[], CHAN *in_ports[],  int ins, CHAN *out_ports[], int outs) {
        int i, valor, produto[DIM];

        init_linda_system(out_ports[0]);
        for (i = 0; i < DIM; i++)
                out("MAT", "%d%3d", i, mat[i]);          /* armazena linhas da matriz */
        out("VET", "%3d", vet);                          /* armazena vetor no TS */
        out("MULTIPLICAR", "");                          /* tupla que indica proximo elemento a calcular */
        for (i = 0; i < DIM; i++) {
                in("PROD", "?d?d", &i, &valor);          /* obtem elementos do vetor resultante */
                produto[i] = valor;
        }
        printf("PRODUTO = ");                            /* imprime vetor resultante */
        for (i = 0; i < DIM; i++) printf("%d ", produto[i]);
        printf("\n");
}

/* Codigo do Processo Trabalhador */

#include <linda.h>
#define DIM     3
#define TRUE    1

int     linha[DIM], vet[DIM];

void main(int argc, char *argv[], char *envp[], CHAN *in_ports[],  int ins, CHAN *out_ports[], int outs) {
        int i, indice, elemento;

        init_linda_system(out_ports[0]);
        read("VET", "?3d", vet);                         /* obtem vetor a ser multiplicado */
        while (TRUE) {
                in("MULTIPLICAR", "?d", &indice);        /* indice do elemento a calcular */
                if (indice < DIM) {
                        out("MULTIPLICAR", "%d", indice + 1);   /* atualiza indice */
                        read("MAT", "%d?3d", indice, linha);    /* obtem linha da matriz */
                        elemento = 0;
                        for (i = 0; i < DIM; i++)
                                elemento += linha[i]*vet[i];    /* calcula elemento */
                        out("PROD", "%d%d", indice, elemento);
                }
        }
}
```