UM NÚCLEO "LINDA" PARA O DESENVOLVIMENTO DE APLICAÇÕES DISTRIBUÍDAS EM UMA REDE UNIX

Dorgival O. Guedes Neto
Osvaldo S. F. Carvalho
Departamento de Ciência da Computação — UFMG
Caixa Postal 702 — 30161 Belo Horizonte - MG
dorgival@dcc.ufmg.br, vado@dcc.ufmg.br *

Sumário

Este artigo apresenta uma forma de se aproveitar o poder de processamento de uma rede de estações de trabalho para o desenvolvimento de aplicações distribuídas. Apresenta-se o modelo de programação **"Linda"** *proposto por David Gelernter, que se basea em quatro operações que, quando "injetadas" em uma linguagem hospedeira (no caso C), criam um dialeto para programação paralela —* **"C-Linda"** *. Essas operações manipulam objetos (tuplas) em uma memória associativa distribuída compartilhada logicamente. De posse dessa linguagem, o programador pode desenvolver aplicações que executem paralelamente em um conjunto de estações de trabalho.*

São discutidos o modelo original, as alterações propostas, os principais aspectos da implementação e o algoritmo adotado para o controle do espaço de tuplas. São avaliadas as várias possibilidades consideradas durante o projeto, bem como alterações para pesquisas futuras.

Abstract

This paper presents a way to use the processing power of a network of UNIX workstations in the development of distributed applications. The **"Linda"** *model proposed by David Gelernter is presented, which provides four operations that can be injected into a host language (in this case C) creating a parallel programming dialect —* **"C-Linda"** *. These operations deal with objects (called tuples) within a logically-shared associative memory. With this language, the programmer can develop aplications that execute in parallel in a collection of workstations.*

The original **"Linda"** *model is analised, in the same way that some proposed changes, main implementation aspects and the algorithm chosen for the control of tuple space. Some alternatives analised during the design are discussed, and changes for future research are proposed.*

1 Introdução

O aumento que vem se verificando no número de redes de estações de trabalho nas universidades brasileiras tem constituído um grande impulso para as pesquisas em áreas como computação, física e engenharia. A cada dia cresce o interesse na utilização desses recursos.

*Este trabalho conta com o apoio da Telebrás — contrato no. 416/91 e do CNPq

Em qualquer organização é comum porém encontrar períodos onde várias máquinas se encontram completamente ociosas ou com baixa utilização. Tal situação tem levado a várias pesquisas visando utilizar o poder computacional dessas máquinas ociosas como um ambiente de processamento paralelo [AB89, MC91, WL88, Tv85].

O que se busca com um ambiente para processamento paralelo para uma rede de estações de trabalho é uma forma de se utilizar o investimento em hardware disponível como um recurso paralelo que ofereça vantagens na relação custo/benefício em comparação a máquinas especializadas. Dessa forma, investimentos em nós da rede oferecem ao mesmo tempo estações de trabalho independentes e mais recursos computacionais para uma "máquina paralela".

Tendo isto em mente, o sistema foi desenvolvido visando oferecer uma interface simples e eficiente para o programador que deseje paralelizar suas aplicações. Nesse aspecto, o modelo visto por um programador utilizando **"Linda"** tem se mostrado extremamente atraente [PB90].

"Linda" já foi implementada numa grande variedade de máquinas, tais como hipercubos [Luc86], máquinas de memória compartilhada como o Encore Multimax e o Sequent Balance [Car87], redes de VAX [WL88] e redes de IBM RTs [AB89]. Uma "máquina Linda" que implementa o modelo em hardware está sendo também desenvolvida [ACGK88].

"Linda" é basicamente um conjunto de quatro primitivas que quando "injetadas" em uma linguagem alvo provê um dialeto paralelo, oferecendo ao programador a facilidade de continuar operando com uma linguagem conhecida. Já existem versões de **"Linda"** para as linguagens C, Fortran [CG88], PostScript, C++ [Lel90], Eiffel [Jel90] e Smalltalk [MK88], entre outras.

Este trabalho descreve um núcleo em tempo de execução e recursos para o desenvolvimento de aplicações paralelas em uma rede de estações de trabalho UNIX. A seção 2 apresenta o princípio de **"Linda"** , as operações definidas, o modelo de programação mais comumente utilizado e um exemplo de utilização. A seção 3 apresenta algumas possibilidades de implementação já abordadas em outros trabalhos para várias arquiteturas. Na seção 4 analisa-se então o ambiente alvo e apresenta-se o algoritmo adotado na implementação do núcleo. Finalmente, a seção 5 apresenta as principais conclusões e propostas de continuação do trabalho.

2 "Linda"

"Linda" é um mecanismo de comunicação de processos desenvolvido por David Gelernter na Universidade de Yale [CG89b, Gel82]. Ao contrário de linguagens baseadas em monitores [Hoa74] ou canais de comunicação direta entre processos como CSP [Hoa78], os processos em **"Linda"** se comunicam através de um espaço de dados compartilhado chamado espaço de tuplas onde objetos de dados denominados tuplas são depositados.

Uma tupla é um conjunto de campos com tipos associados, podendo ou não cada campo possuir um valor definido. Nesse caso o campo é denominado real, enquanto no caso de possuir apenas um tipo, sem valor explícito é denominado formal. Por exemplo, a tupla:

```
("Saudacoes",42,3.14)
```

possui três campos: uma cadeia de caracteres, um inteiro e um número de ponto flutuante. Como todos os campos possuem valores bem definidos, todos são campos reais. Já a tupla:

```
("Saudacoes",?int,?float)
```

possui também três campos, com os mesmos tipos da anterior, porém apenas o primeiro campo é real, sendo os dois outros formais.

As representações acima podem ser utilizadas para simbolizar uma tupla bem definida, ou um padrão de busca, sem uma tupla verdadeira a ele associada. A operação básica sobre

as tuplas contidas no espaço de tuplas é então o casamento com um determinado padrão, o que ocorre segundo regras bem definidas: há casamento entre uma tupla e um padrão se e somente se:

- ambos possuem o mesmo número de campos,
- campos correspondentes possuem o mesmo tipo, e
- campos correspondentes possuem valores reais iguais ou um é formal e outro real.

Segundo essas regras, há casamento entre os dois padrões acima, uma vez que:

- ambos possuem três campos;
- o primeiro campo de ambos é uma cadeia de caracteres, o segundo um inteiro e o terceiro um número de ponto flutuante;
- o primeiro campo de ambos é real, e contém o mesmo valor, "Saudacoes", enquanto os dois outros campos são reais em um padrão e formais no outro.

Com base nessas regras define-se o conjunto de operações que constitui o modelo **"Linda"** que passamos a descrever.

2.1 Primitivas básicas

out( ) deposita no espaço de tuplas uma tupla definida pelo padrão formado pelos parâmetros e o processo que a executa continua imediatamente.

in( ) uma tupla que se case com o padrão definido pelos parâmetros é retirada do espaço de tuplas. Se o padrão especificado contém campos formais com variáveis especificadas, o valor real do campo correspondente na tupla é copiado para a variável no retorno da função. Se não há uma tupla disponível que se case com o padrão, o processo é suspenso até que uma surja, quando o processo prossegue normalmente. Se há várias tuplas que se casem com o padrão, uma é escolhida aleatoriamente.

rd( ) semelhante ao `in( )` , porém a tupla não é removida do espaço de tuplas após o casamento.

eval( ) gera uma "tupla viva" no espaço de tuplas, que é a forma de se dispararem novos processos: um novo processo é criado para a avaliação das funções presentes na tupla. O processo pai, que executou o `eval( )` , prossegue sem esperar por seus filhos.

A forma de se especificar os campos reais e formais varia de implementação para implementação, bem como de acordo com restrições devidas à sintaxe da linguagem alvo. Por exemplo, na implementação em **"C-Linda"** [CG89a], supondo as seguintes declarações:

```
int   i, intf();
float f, ff();
```

as operações seriam representadas da seguinte forma:

```
out( "Tupla0",i, ?f );
```

Geraria uma tupla com um campo real contendo a cadeia de caracteres "Tupla0", um campo real contendo o valor inteiro da variável `i` e um campo formal com o tipo da variável `f`, no caso `float`.

```
out( "Resultado", intf( ) );
```

A função `intf( )` seria chamada pelo programa executante e uma tupla com um campo do tipo cadeia de caracteres com conteúdo "Resultado" e outro campo inteiro com o valor de retorno da função seria depositada no espaço de tuplas.

```
rd( "Resultado", ?i );
```

O processo que executa seria bloqueado até que uma tupla com um campo "Resultado" e um campo real inteiro estivesse disponível, quando então o valor do campo inteiro seria copiado para a variável `i` e o processo continuaria, deixando a tupla inalterada no espaço de tuplas.

```
in( "Resultado", ?i );
```

O comportamento seria semelhante ao anterior, porém a tupla seria retirada do espaço de tuplas antes que o processo fosse autorizado a prosseguir.

```
eval( "Resultado", ff( ) );
```

Um novo processo seria disparado com a criação de uma "tupla viva", o qual se encarregaria de executar a função `ff( )`. Quando essa função retornasse ao terminar, uma tupla seria gerada com os campos "Resultado" e o `float` retornado pela função.

Pode-se ver que não há nenhuma operacão que permita a alteração de uma tupla enquanto no espaço de tuplas. Essa característica, aliada à atomicidade das operações acima, garante os mecanismos de exclusão mútua e sincronização necessários a uma linguagem para programação paralela.

O espaço de tuplas pode ser visto como uma memória associativa compartilhada por todos os nodos que participem do processamento, sendo a sua implementação em um ambiente distribuído [SZ90] a principal questão a ser enfrentada na criação do núcleo. Outra forma de se encarar a programação em **"Linda"** é como a manipulação de estruturas de dados distribuídas, acessáveis por qualquer nodo que participe do processamento. O espaço de tuplas se torna um grande conjunto onde se pode inserir e retirar elementos dinamicamente, sem nenhuma forma de ordenação implícita.

2.2 Um exemplo de utilização

Para ilustrar o tema, apresenta-se a seguir uma solução para o problema de multiplicação de matrizes utilizando **"Linda"** . Essa solução é obtida utilizando-se o paradigma de programação conhecido por "Supervisor-Trabalhador" [MC91] ou "Trabalhador Replicado" [GB86] ou ainda "Paralelismo por Agenda" [CG89a].

Nesse modelo, um processo — supervisor — define tarefas a serem executadas, e vários outros processos idênticos — trabalhadores replicados — consultam a "agenda de tarefas a serem executadas", criando um resultado que é posteriormente coletado pelo supervisor. Tal estrutura tem algumas vantagens do ponto de vista do programador:

- Uma vez desenvolvido e depurado um programa com um trabalhador, o programa está pronto para ser executado por um número qualquer de processos. O programador precisa se preocupar apenas com a comunicação entre dois processos, um supervisor e um trabalhador.

- Ele se adapta facilmente ao número de processadores existentes. Ao contrário de certos algoritmos onde o número de processos é definido pelo problema, tendo então que ser mapeado para o número de processadores existentes, à vezes forçando a troca de contextos constantes devido ao fato de mais de um processo precisar ocupar um mesmo nodo, pode-se disparar tantos trabalhadores quantos sejam os nodos disponíveis, sem necessidade de alterações no algoritmo.

- Por definição, ele permite o balanceamento dinâmico de carga. Cada trabalhador repetidamente busca uma tarefa a ser executada, processa-a e gera um resultado. As tarefas assim vão se distribuindo em tempo de execução entre os processadores que estejam ociosos.

Problemas que se adaptam bem a esse tipo de estrutura são por exemplo:

Multiplicação de matrizes: Cada elemento da matriz produto é completamente independente dos demais, podendo ser calculado separadamente;

Geração de fractais de Mandelbrot: Cada ponto do conjunto de Mandelbrot é gerado por uma expressão que não referencia nenhum outro ponto, podendo os pontos ser distribuídos livremente;

Simulações de Monte Carlo: Usualmente, um modelo é simulado milhares de vezes, sendo que cada nova simulação é independente das anteriores.

Vale notar que **"Linda"** não força a utilização desse paradigma. Sendo um conjunto de operações que podem ser consideradas de alto nível de abstração, é fácil utilizá-la segundo outros padrões [CG89a, CG88].

O programa supervisor para a multiplicação de matrizes pode então ser escrito da seguinte maneira:

```
01 lmain()
02 { int    i, j, cont;
03   double val, result[DIM][DIM], ma[DIM][DIM], mb[DIM][DIM];
04
05   for ( i=0; i<DIM; i++ ) {
06     out( "LinhaA", i, <linha i da matriz ma> );
07     out( "ColunaB", i, <coluna i da matriz mb> );
08   }
09   for ( i=0; i<DIM; i++ )
10     for ( j=0; j<DIM; j++ )
11       out( "Task", i, j );
12   for ( cont=0; cont<NWORKERS; cont++)
13     eval( worker( ) );
14   for ( cont=0; cont<(DIM*DIM); cont++ ) {
15     in( "Result", ?i, ?j, ?val );
16     result[i][j] = val;
17   }
18 }
```

Os `out( )` nas linhas 06 e 07 criam uma tupla para cada linha da matriz `ma` e cada coluna da matriz `mb`, para serem utilizadas pelos trabalhadores. A forma de representação de linhas e colunas das matrizes é omitida por simplicidade.

O `out( )` na linha 11 cria uma tupla para cada elemento da matriz resultado, com o primeiro campo "Task", e os dois campos seguintes com as coordenadas de um elemento. Em seguida, o `eval( )` (linha 13) dispara um número pré-definido de trabalhadores — que poderia ser fornecido na chamada do programa, em um caso mais geral. Finalmente, na linha 15 se executa um `in( )` para recuperar cada resultado. Note-se que da forma como foi escrito, o programa não se preocupa em buscar os resultados em ordem, utilizando os dois campos inteiros da tupla lida para definir a posição do resultado dentro da matriz.

A função `worker( )` deve então obter uma tarefa a ser executada, obter a linha de `ma` e a coluna de `mb`, executar o seu produto interno e produzir a tupla resultado:

```
01 worker( )
02 {
03   int l, c, i, j;
04   double result, linha[DIM], coluna[DIM];
05
06   for (;;)
07   {
08     in( "Task", ?l, ?c );
09     rd( "LinhaA", l, ?linha );
10     rd( "ColunaB", c, ?coluna );
11     for( result=0, i=0; i<DIM; i++ )
12       result += linha[i]*coluna[i];
13     out( "Result", l, c, result );
14   }
15 }
```

Observe-se que a linha e a coluna a serem multiplicadas são obtidas com `rd( )` , de forma que continuem disponíveis no espaço de tuplas para outros trabalhadores que venham a precisar delas. Os dois `rd( )` definem um padrão com dois campos reais e um formal, de forma a obter exatamente a linha e a coluna de índices `l` e `c` respectivamente.

Como não poderia deixar de ser, esta é apenas uma de muitas possibilidades de implementação. Note-se que o volume de comunicação nessa solução é extremamente alto: cada elemento calculado exige que se leia novamente uma linha e uma coluna, e que se gere uma tupla de resultado. Outras soluções que poderiam reduzir a comunicação seriam:

- cada trabalhador leria todas as linhas de `ma` antes de iniciar as iterações;
- cada trabalhador leria as duas matrizes completamente;
- ao invés de calcular elementos isolados, cada trabalhador se ocuparia de uma linha inteira da matriz resultado a cada tarefa.

Nenhuma dessas opções implica em alterações no modelo básico, levando apenas a alterações no padrão de comunicação e na granularidade da execução.

2.3 O mecanismo de `eval( )`

A operação de `eval( )` pode ser implementada facilmente em função das demais, bastando que cada processador verifique a existência de uma "tupla descritora de `eval( )` ", retire-a e execute a função correspondente. Essa simplicidade porém é enganosa. Há ainda muitas questões relacionadas a esse mecanismo que não possuem uma solução conclusiva, e segundo Carriero [Car90] ainda são áreas abertas à pesquisa:

- Em uma rede de processadores de desempenhos heterogêneos, deve haver alguma identificação no `eval( )` para indicar que tipo de processador deve se incumbir de uma determinada tarefa?
- Hoje a maior parte dos sistemas UNIX oferece algum tipo de facilidade para criação de "processos leves" — processos que executam em um mesmo espaço de endereços, com tempos bastante baixos para trocas de contexto [Sun90]. Com isso em mente, um `eval( )` deve gerar processos leves ou pesados? O programador pode ter uma forma de definir tal coisa *a priori*?

- As funções executadas por `eval()` são definidas no mesmo programa do supervisor, compartilhando assim de variáveis globais desse programa. Qual o grau de acoplamento deve haver entre o processo original e aqueles disparados por `eval()` ? Devem enxergar apenas o estado inicial das variáveis globais, ou seu estado no instante da execução do `eval()` , em qualquer instante, ou essa possibilidade deve ser negada?

- As funções executadas por `eval()` devem poder receber parâmetros ou não? A passagem de parâmetros contraria a definição de que a comunicação entre processos deve se dar através do espaço de tuplas, mas pode agilizar mecanismos de inicialização de processos.

- Como deve ser definido o princípio de terminação distribuída? Um programa distribuído termina quando todos os processos disparados por `eval()` terminarem, ou quando o processo inicial (supervisor) terminar? Note-se que o programa de multiplicação de matrizes apresentado assume a segunda opção.

Devido a todos esse problemas, muitas implementações descritas na literatura [CG86, GB82] não implementam `eval()` . O programador deve se encarregar de desenvolver em separado os programas supervisor e trabalhadores, e dispará-los "manualmente" no início do processamento.

3 Alterações propostas

Como definido inicialmente, o modelo **"Linda"** apresenta algumas desvantagens sob uma certa forma de análise:

- Para que a busca de tuplas de qualquer padrão, por qualquer conjunto de campos seja implementada de forma eficiente, depende-se demais de um compilador para identificação de padrões de busca [CGS+89] e operações complementares fornecendo ao sistema de tempo de execução informações para a organização do espaço de tuplas;

- Um só espaço de tuplas pode levar a problemas de análise de significado de uma tupla: se dois programas diferentes manipulam tuplas de mesma "aparência", por exemplo, com tuplas
  `("AGORA",dia,mes,ano)` e `("AGORA",hora,minuto,segundo)`
  como garantir a correta execução de ambos?

- A imensa maioria das aplicações documentadas [CG89a, WL88, CG88] utiliza na definição do padrão de casamento de `in()` e `rd()` apenas um ou dois campos reais, como pode ser observado no exemplo de multiplicação de matrizes;

- As aplicações práticas não indicaram até hoje nenhuma utilidade para operações de `out()` que contenham campos formais. Tal facilidade torna mais complexa a gerência do núcleo [Lel90].

Tendo isto em mente, a implementação proposta aborda várias alterações ao modelo original:

Simplificação do compilador: Argumenta-se que a linguagem C é flexível o suficiente para permitir o desenvolvimento de um núcleo que não necessite de compiladores especiais complexos, e se possível utilize apenas recursos do pré-processador padrão da linguagem. Para isso, por exemplo, as funções a serem disparadas por `eval()` devem sempre receber um parâmetro inteiro e retornar um inteiro, e devem ser declaradas como funções para `eval()` no início do programa;

Pré-definição dos padrões das tuplas: Para uma maior documentação e para reduzir os problemas de confusão entre tuplas de padrão semelhante, os padrões devem ser definidos como `structs` da linguagem. Os processos que desejem se comunicar devem utilizar variáveis do tipo correto na chamada das operações;

Vários espaços de tuplas: Com a mesma finalidade, permite- se ao programador criar e utilizar vários espaços de tuplas independentes, aumentando a liberdade de expressão. Como no caso das funções disparadas por `eval()` , o programador deve indicar quais variáveis serão utilizadas como descritores de espaços de tuplas no programa;

Espaços de tuplas tipados: Para garantir a correção a um nível mais alto, o usuário pode definir o tipo das tuplas a serem lançadas em um espaço de tuplas.

Com essas alterações, o programa de multiplicação de matrizes já apresentado toma a seguinte forma:

```
01 typedef double row_col_t[DIM];
02 typedef struct { int row, col; double val } restask_t;
03 int ts_A, ts_B, ts_restask;
04
05 BEGIN_EVAL_DECL
06   worker
07 END_EVAL_DECL
08
09 BEGIN_TS_DECL
10   TS_INIT( ts_A, row_col_t );
11   TS_INIT( ts_B, row_col_t );
12   TS_INIT( ts_restask, restask_t );
13 END_TS_DECL
14
```

Este programa define três espaços de tuplas visíveis para os processos participantes: um para as linhas da matriz `A`, outro para as colunas de `B` e um terceiro para os descritores de tarefas e para os resultados. Os tipos utilizados para as tuplas de cada um deles são especificados nas linhas 01 e 02, enquanto na linha 03 são declaradas as variáveis que serão os descritores dos espaços. Todos os espaços de tuplas devem ser declarados dessa forma, antes do programa principal.

A seguir declara-se que a função `worker()` será executada através de `eval()` . O parâmetro e o valor de retorno são irrelevantes para esse programa.

Finalmente, os espaços de tuplas são inicializados, e os tipos de seus elementos especificado. O trecho entre as linhas 09 e 13 será posteriormente expandido para uma função que garante que cada processo criado por `eval()` terá acesso aos mesmos espaços de tuplas alocados.

```
15 lmain( )
16 { int       i, j, cont;
17   restask_t task, res;
18   row_col_t A[DIM], B[DIM]; /* A agrupada por linhas, B por colunas */
19   double    result[DIM][DIM];
20   for ( i=0; i<DIM; i++ ) {
21     out( ts_A, inttuplekey(i), A[i] );
22     out( ts_b, inttuplekey(i), B[i] );
23   }
```

```
24   for ( i=0; i<DIM; i++ )
25     for ( j=0; j<DIM; j++ ) {
26       task.row = i;
27       task.col = j;
28       out( ts_restask, tuplekey("Task"), &task );
29     }
30   for ( cont=0; cont<NWORKERS; cont++ )
31     eval( worker, i );
32   for ( cont=0; cont<(DIM*DIM); cont++ ) {
33     in( ts_restask, tuplekey("Result"), ?res );
34     result[res.row][res.col] = res.val;
35   }
36 }
```

Após declarar as variáveis que armazenarão as tuplas de tarefas, resultados e as matrizes (onde `A` será mantida por linhas e `B` por colunas), as tuplas com linhas e colunas são geradas (linhas 21 e 22) e em seguida as tarefas (linha 28). O campo `val` não é utilizado para estas tuplas, mas apenas para os resultados.

O padrão para `in()` `out()` e `rd()` é o mesmo: descritor do espaço de tuplas utilizado, chave e descritor da tupla. A chave pode ser um inteiro ou uma cadeia de caracteres, e as funções `inttuplekey()` e `tuplekey()` se encarregam das conversões necessárias. Note-se que com a utilização de espaços de tuplas isolados não há mais necessidade dos campos "LinhaA" e "ColunaB". O `eval()` (linha 31) não necessita do descritor de espaços de tuplas, apenas a função a ser executada e um parâmetro inteiro, nesse caso sem efeito.

```
37 worker( )
38 { int i, j;
39   restask_t task;
30   row_col_t rowA, colB;
41   for (;;) {
42     in( ts_restask, tuplekey("Task"), &task );
43     rd( ts_A, inttuplekey(task.row), rowA );
44     rd( ts_B, inttuplekey(task.col), col );
45     for( task.val=0, i=0; i<DIM; i++ )
46       task.val += rowA[i]*colB[i];
47     out( ts_restask, tuplekey("Result"), &task );
48   }
49 }
```

O trabalhador é bem semelhante ao original, exceto pelo fato de utilizar a variável `task` para receber a descrição da tarefa e também para gerar a tupla com o resultado para cada elemento. O supervisor ao receber uma tupla com chave "Result" recebe em uma só variável o valor de um elemento da matriz resultado e as suas coordenadas na mesma.

4 Possibilidades de implementação

As implementações de espaços de tuplas em sistemas distribuídos seguem usualmente uma das seguintes opções:

"Broadcasted `out()` local `in()` ": Cada tupla gerada é copiada para todos os nodos do sistema, o que torna a operação `rd()` bastante simples, porém `in()` exige que um nodo obtenha exclusão mútua de todos os demais nodos para poder retirar a tupla. Necessita de uma arquitetura que permita a operação de broadcast a baixo custo [CG86]. Pela replicação de tuplas ocupa uma área muito grande de memória.

"Local `out()` broadcasted `in()` ": Cada tupla gerada é armazenada apenas no nodo gerador. Nodos que executem um `in()` ou `rd()` devem gerar um "broadcast" do pedido, ao qual nodos que possuam tuplas na forma requerida respondem. Os nodos que executam um `in()` ou `rd()` devem estar preparados para múltiplas respostas [AB89].

"Hashed tuple space": Adotado normalmente em arquiteturas onde a operação de "broadcast" possui alto custo, tal como hipercubos [Luc86]. É criada uma função de hash que englobe o número e tipo dos campos, e ainda um campo definido como chave. Exige assim a definição de um campo de chave, e reduz a liberdade de definição de padrões para casamento.

"`in()` and `out()` sets": Utiliza uma replicação parcial de tuplas em subconjuntos de nodos, visando equilibrar os custos de execução de `in()` e `out()` . Exceto em casos em que a arquitetura ofereça uma divisão natural em subconjuntos [ACGK88], cria complexos problemas de gerenciamento.

5 O algoritmo adotado

Tendo em vista ser o ambiente alvo uma rede de estações de trabalho UNIX operando segundo o conjunto de protocolos TCP/IP, é importante observar algumas características desse sistema e seus protocolos:

- Uma rede local departamental pode ser formada por várias sub-redes a nível físico, usualmente segundo os padrões ethernet ou token ring, ambos permitindo "broadcast" apenas dentro de uma sub-rede.
- O protocolo que permite a utilização desse recurso no conjunto TCP/IP é o UDP, um serviço de datagrama sem confirmação, assim sendo, não confiável, embora em uma rede local com uma taxa de perdas bastante baixa.
- O protocolo que oferece um canal de transmissão confiável é o TCP, que opera sempre em conexões ponto-a-ponto, sempre com um custo adicional para o estabelecimento e liberação de conexões.

Optou-se então por um algoritmo que se vale dos recursos de "broadcast" do UDP para mensagens de controle, com um mecanismo de temporização para recuperação de possíveis perdas [Ste90] e que utiliza TCP para transporte de tuplas entre nodos quando necessário. Para não se limitar o sistema a uma só sub-rede os nós que executam em máquinas ligadas a mais de uma sub-rede cuidam da retransmissão de "broadcast" agindo assim como "gateways".

O modelo adotado foi então o de "local `out()` , broadcasted `in()` ". Quando um processo se inicia, ele abre uma conexão TCP com o processo do núcleo executando no mesmo processador, pelo qual flui a comunicação entre a aplicação e o núcleo. Sendo esta ligação permanente durante toda a duração da aplicação, o custo de conexão e desconexão pode ser desprezado.

Com as alterações propostas, um `in()` ou `rd()` é identificado simplesmente pelo descritor do espaço de tuplas alvo e pela chave, definida como um campo isolado da tupla, contendo uma cadeia de caracteres de comprimento máximo 15. Um `out()` é composto pelo descritor do espaço de tuplas, a chave atribuída à tupla e o bloco de dados que contém a estrutura da tupla. `eval()` é implementado como observado anteriormente com base nessas primitivas.

Ocorre um casamento então se há um pedido de tupla (`in()` ou `rd()`) e uma tupla (`out()`) com o mesmo descritor de espaço de tuplas e mesmo valor no campo de chave.

Na ocorrência de um `in()` ou `rd()` de um processo local, o processo do núcleo no nodo verifica se a tupla com a chave pedida existe localmente no espaço de tuplas indicado pela operação. Caso afirmativo, a entrega é feita e a tupla removida no caso do `in()` . Caso contrário, o pedido é anotado localmente e um "broadcast" é feito para todas as máquinas informando o pedido com o descritor do espaço de tuplas e a chave. Cada nodo ao receber o pedido verifica em suas tuplas se alguma o satisfaz, enviando outra mensagem UDP informando a existência da tupla. Havendo ou não a tupla requerida, o pedido é armazenado.

Ao executar um `out()` um processo entrega ao núcleo uma tupla através do protocolo TCP. Caso haja algum processo no mesmo nodo bloqueado aguardando uma tupla com aquele padrão, a entrega é imediata e os dois processos são liberados para continuar. Caso contrário, o núcleo armazena localmente a tupla e verifica se possui algum pedido armazenado de outro nodo que necessite de uma tupla com a chave dada no espaço de tuplas especificado. Caso possua, o nodo envia uma mensagem UDP ao nodo solicitante informando a existência da tupla.

Essa opção por enviar apenas uma notificação de posse da tupla desejada foi feita para evitar o caso em que vários nodos tenham tuplas que satisfaçam ao pedido e todos decidam entregá-las ao solicitante. Haveria uma grande sobrecarga devido ao estabelecimento e liberação de várias conexões TCP para que cada nodo entregasse a sua tupla, quando normalmente apenas uma seria utilizada.

Dessa forma, um nodo que receba uma mensagem de outro com a notificação de existência de uma tupla conforme solicitado verifica se a mesma ainda é necessária (não foi ainda fornecida por outro nodo). Em caso afirmativo, outra mensagem UDP é enviada em "broadcast", requisitando ao nodo de posse da tupla a entrega da mesma. Os demais nodos, ao receber a mensagem e verificar que não é a eles destinada, apagam o pedido pendente para o nodo originador da mesma.

O nodo possuidor ao receber o pedido verifica se ainda possui a tupla pedida (que pode já ter sido entregue a outro nodo ou a um processo local). Em caso afirmativo, ele inicia uma conexão TCP para o nodo requisitante *como se fosse um cliente local àquele nodo que estivesse se conectando*, e então executa um `out()` . Para o nodo requisitante tudo se passa como se outro processo local estivesse executando um `out()` e a tupla é entregue ao processo bloqueado no `in()` ou `rd()` . Caso a tupla não mais exista, uma mensagem UDP é enviada negando o pedido.

Um nodo que receba uma mensagem de negação de um pedido sabe que precisa reiniciar o processo de pesquisa de tuplas, uma vez que o nodo escolhido para responder não foi capaz de fazê-lo, e o processo se reinicia.

Com a utilização de TCP para o transporte de tuplas garante-se que tuplas nunca são perdidas. O mesmo não ocorre com as mensagens de pesquisa por uma tupla, notificação de existência, requisição e negação. Se uma dessas mensagens se perder para um ou mais nodos pode ser que o processo continue normalmente — um outro nodo receba o pedido, possua a tupla e a forneça. Há porém a possibilidade de que o processo continuasse bloqueado indefinidamente à espera de uma tupla que estivesse em um nodo não alcançado por uma mensagem UDP. Para evitar este tipo de problema, o nodo que inicia o pedido de uma tupla mantém um mecanismo de temporização: decorrido um determinado tempo sem uma resposta o processo é reiniciado, garantindo-se assim que mensagens UDP perdidas não bloqueem o núcleo.

6 Conclusões

Este trabalho apresentou os princípios utilizados no desenvolvimento de um núcleo para o desenvolvimento de aplicações distribuídas em redes de estações de trabalho operando sob sistema operacional UNIX e protocolos de rede do conjunto TCP/IP. O sistema utiliza dos protocolos TCP e UDP para a transferência de dados e controle respectivamente entre os nodos do sistema, utilizando um sistema de temporizações para evitar erros devidos à perda de mensagens UDP.

O sistema implementado vem sendo utilizado em bases experimentais no Departamento de Ciência da Computação da UFMG (DCC-UFMG) em uma rede de estações de trabalho Sun de diferentes capacidades, configurando um ambiente heterogêneo bastante interessante para o estudo e desenvolvimento de aplicações distribuídas. Um gerador de fractais do conjunto de Mandelbrot já foi desenvolvido, utilizando o paradigma dos trabalhadores replicados, apresentando "speed-ups" significativos em função do número de máquinas utilizadas.

Novas aplicações estão sendo iniciadas em conjunto com o grupo de microeletrôniça do DCC-UFMG — geração de padrões de teste para circuitos — e com o grupo de alta tensão do Departamento de Engenharia Elétrica da UFMG — simulação de comportamento de linhas de transmissão de energia sob descargas atmosféricas.

Paralelamente trabalha-se no aprimoramento do núcleo, sendo as principais linhas de pesquisa o desenvolvimento de ferramentas para depuração de programas paralelos desenvolvidos utilizando-se o núcleo, estudos sobre os vários aspectos relacionados à alocação de processadores ao longo da execução e a utilização de heurísticas para detecção de padrões de comunicação entre os nodos.

Referências

[AB89] Mauricio Arango and Donald Berndt. Tsnet: A linda implementation for networks of unix-based computers. Technical report, Yale University, August 1989.

[ACGK88] Sudhir Ahuja, Nicholas Carriero, David Gelernter, and Venkatesh Krishnaswamy. Matching language and hardware for parallel computation in the linda machine. *IEEE Transactions on Computers*, 37(8):921–929, August 1988.

[Car87] Nicholas Carriero. *Implementing Tuple Spaces in Linda.* PhD thesis, Yale University, 1987.

[Car90] Nicholas Carriero. Personal communications. Comunicações pela Internet, 1990.

[CG86] Nicholas Carriero and David Gelernter. The s/net's linda kernel. *ACM Transactions on Computer Systems*, 4(2):110–129, May 1986.

[CG88] Nicholas Carriero and David Gelernter. Applications experience with linda. In *Proceddings Symposium on Parallel Programming*, pages 173–187. ACM, July 1988.

[CG89a] Nicholas Carriero and David Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[CG89b] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[CGS+89] Nicholas Carriero, David Gelernter, Ehud Shapiro, Craig Davidson, L. V. Kalé, Kenneth M. Kahn, and Mark S. Miller. Technical correspondence - linda in context. *Communications of the ACM*, 32(10):1244–1258, October 1989.

[GB82] D. Gelernter and A. Bernstein. Distributed communication via global buffer. In *Proceedings Symposium on Principles of Distributed Computing*, pages 10–18. ACM, August 1982.

[GB86] D. Gelernter and A. Bernstein. Distributed data structures in linda. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 236–242. ACM, January 1986.

[Gel82] David Gelernter. *An integrated microcomputer network for experiments in distributed programming.* PhD thesis, SUNY at Stony Brook, 1982.

[Hoa74] C. A. R. Hoare. Monitors: An operating system concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[Hoa78] C. A. R. Hoare. Comunicating sequential processes. *Communications of the ACM*, 21(8):666,677, August 1978.

[Jel90] Robert Jellinghaus. Eiffel linda: An object-oriented linda dialect. *ACM SIGPLAN Notices*, 25(12):70–84, December 1990.

[Lel90] Wm Leler. Linda meets unix. *IEEE Computer*, pages 43–54, February 1990.

[Luc86] Steven E. Lucco. A heuristic linda kernel for hypercube multiprocessors. In *Proceedings of the Second Conference on Hypercube Multiprocessors*, pages 32–38, October 1986.

[MC91] J. N. Magee and S. C. Cheung. Parallel algorithm design for worstation clusters. *Software — Practice and Experience*, 21(3):235–250, March 1991.

[MK88] Satoshi Matsuoka and Satou Kawai. Using tuple space communication in distributed object-oriented languages. In *Proceedings of OOPSLA '88*, pages 276–284. ACM, September 1988.

[PB90] Cherri M. Pancake and Donna Bergmark. Do parallel languages respond to the needs of scientific programmer? *IEEE Computer*, pages 13–23, December 1990.

[Ste90] W. Richard Stevens. *Unix Network Programming.* Software Series. Prentice Hall, Englewood Cliffs, 1990.

[Sun90] Sun Microsystems. *Programming Overview and Utilities*, 1990.

[SZ90] Michael Stumm and Songnian Zhou. Algorithms implementing distributed shared memory. *IEEE Computer*, pages 54–64, May 1990.

[Tv85] Andrew S. Tanenbaum and Robbert van Renesse. Distributed operating systems. *ACM Computing Surveys*, 17(4):419–470, December 1985.

[WL88] Robert Whiteside and Jerrold Leichter. Using linda for supercomputing on a local area network. In *Proceedings Supercomputing '88*, pages 192–199, Orlando, Florida, November 1988. IEEE.