YALI
Uma Extensão do Modelo Linda para Programação Paralela em Redes Heterogêneas

Andréa Schwertner Charão
Instituto de Informática
Universidade Federal do Rio Grande do Sul
Porto Alegre – RS – Brasil
e-mail: andrea@inf.ufrgs.br

Celso Maciel da Costa
Instituto de Informática
Pontifícia Universidade Católica do Rio Grande do Sul
Porto Alegre – RS – Brasil
e-mail: celso@kriti.pucrs.br

Sumário

Com a disponibilidade de redes que ligam estações cada vez mais poderosas a baixos custos, o interesse em torno de ambientes que suportam a programação paralela em arquiteturas deste tipo tem aumentado muito. Este artigo descreve um ambiente destinado à programação paralela em redes heterogêneas, baseado no modelo Linda. Este ambiente, chamado YALI, suporta operações globais, que auxiliam na comunicação e sincronicação entre múltiplos processos. YALI também inclui um espaço de tuplas distribuído, baseado em *hashing*, e implementado com *threads*.
Palavras-chave: Linda, programação paralela, espaço de tuplas.

Abstract

With the availability of networks connecting powerful workstations at a low cost, increasing interest has been devoted to systems that support parallel programming in such architectures. This paper describes an environment for parallel programming in heterogeneous networks, based on the Linda model. This parallel environment, called YALI, provides some global operations, that are useful to express communication and synchronization between multiple processes. YALI also includes a multithreaded, distributed tuple space, based on hashing.
Keywords: Linda, parallel programming, tuple space.

1 Introdução

Existe, atualmente, um crescente interesse em torno de ambientes que suportam a programação paralela em redes de computadores, devido principalmente à disponibilidade de redes ligando estações e poderosos computadores pessoais, que reúnem um grande poder computacional a custos relativamente baixos. Nestes ambientes, o paradigma de programação mais frequentemente empregado é baseado em troca de mensagens, que, apesar de ser eficiente, é de difícil assimilação devido ao seu baixo nível de abstração.

Uma abordagem alternativa à troca de mensagens é permitir a comunicação entre processos através de uma memória logicamente compartilhada. Dentro desta abordagem de mais alto nível, pode-se destacar o modelo Linda[1][GEL85], onde toda interação entre processos ocorre por intermédio de uma memória compartilhada associativa, denominada espaço de tuplas. Entre as vantagens deste modelo pode-se citar a simplicidade de suas primitivas, e a possibilidade de incorporá-las a uma linguagem sequencial conhecida.

Este artigo descreve YALI, um ambiente para programação paralela em redes heterogêneas. Tendo como objetivo facilitar a programação paralela em arquiteturas deste tipo, YALI baseia-se no modelo Linda, mas possui extensões para suporte a operações globais[FOS95], usadas para comunicação e sincronização envolvendo vários processos. O paralelismo, em YALI, é baseado em processos distribuídos de modo estático, e *threads* disparadas dinamicamente. O espaço de tuplas usado em YALI é distribuído entre os processos que compõem uma aplicação paralela, e sua implementação é baseada em *threads*. O artigo está organizado da seguine maneira: a seção 2 discute sucintamente o modelo Linda e alguns aspectos de sua implementação. Na seção 3, a seguir, são apresentados os componentes do ambiente YALI. As seções 4 e 5, por fim, tratam da implementação e da avaliação do ambiente, respectivamente.

2 O Modelo Linda

O modelo Linda[GEL85] consiste em um pequeno conjunto de primitivas que manipulam uma memória global associativa denominada **espaço de tuplas** (*Tuple Space* — TS) . As primitivas Linda, quando incorporadas a uma linguagem sequencial, permitem expressar a comunicação e a sincronização entre processos, tornando-a adequada para programação paralela. Basicamente, existem quatro operações definidas em Linda: **out**, **in**, **rd** e **eval**. Todas elas são processadas atomicamente e manipulam entidades chamadas **tuplas**, que são sequências de campos de tipos específicos, aos quais podem ou não ser associados valores. Os tipos dos campos geralmente são aqueles suportados pela linguagem hospedeira de Linda, e a presença ou não de valores associados a um campo define, respectivamente, se o campo é **real** ou **formal**.

De maneira geral, as primitivas servem para inserir ou retirar tuplas do TS. Uma operação **out** insere assincronamente uma tupla no espaço, enquanto **eval** dispara um novo processo para processar os campos de uma tupla, depositando a tupla resultado no TS.

[1] Linda é marca registrada de Scientific Computing Associates, Inc.

A primitiva **in**, ao contrário, serve para remover uma tupla do espaço. A tupla fornecida como argumento à primitiva **in** é chamada *template*, e pode conter campos reais, que formam uma espécie de chave para busca da tupla, e campos formais, representados por variáveis que devem receber algum valor depois que uma tupla equivalente é encontrada no TS. Caso tal tupla não exista no TS, o processo que executa **in** é bloqueado até que uma tupla seja depositada. De maneira semelhante atua a primitiva **rd**, que também serve para a recuperação de tuplas, porém sem removê-las do TS. Existem também duas variações não-bloqueantes das primitivas **in** e **rd**, chamadas, respectivamente, **inp** e **rdp**[NAR89].

2.1 Aspectos de Implementação do Modelo

Apesar da simplicidade associada ao modelo Linda, sua implementação eficiente depende de alguns fatores. O espaço de tuplas, uma das principais características do modelo, é facilmente implementado em arquiteturas paralelas com memória compartilhada. No entanto, em arquiteturas desprovidas deste recurso, como é o caso das redes de computadores, é necessária a implementação de mecanismos que permitam o compartilhamento lógico do TS. Para isso, muitos sistemas utilizam um processo gerenciador do espaço de tuplas, que processa requisições de operações Linda. Este processo pode estar centralizado em somente um nodo da rede (como em Glenda[SEY93] e p4-Linda[BUT93]), instanciado em alguns nodos (D-Linda[PIN91]) ou mesmo replicado em toda a rede (POSYBL[SCH91]). Embora um gerenciador centralizado seja mais simples de implementar, sistemas Linda que adotam esta solução geralmente não são escaláveis. Espaços de tuplas distribuídos, onde vários gerenciadores estão envolvidos, suportam melhor o desenvolvimento de aplicações com grande número de processos espalhados sobre vários nodos.

Ao implementar um espaço de tuplas distribuído, deve-se adotar alguma política de distribuição das tuplas entre os diversos gerenciadores instanciados em diversos nodos. As políticas mais frequentemente empregadas são as seguintes:

- Distribuição uniforme[GEL85]: segundo este esquema, cada tupla gerada através de **out** é armazenada em um conjunto pré-determinado de nodos, denominado *out_set*. Tuplas requisitadas através de **in**, por sua vez, são procuradas num conjunto de nodos denominado *in_set*. Deve-se garantir que cada *in_set* tenha uma intersecção não vazia com os conjuntos *out_set*. Esta política admite inúmeras possibilidades. Em S/Net[CAR86], por exemplo, todos os nodos fazem parte do *out_set*, e o *in_set* consiste apenas no nodo que executa **in** ou **rd**, o que caracteriza um espaço de tuplas totalmente replicado. O contrário também é possível, onde o *out_set* contém apenas o nodo que executa **out**, e o *in_set* engloba todos os nodos da rede, como em POSYBL[SCH91].

- *Hashing*: neste esquema, o nodo onde uma tupla deve ser armazenada ou procurada é determinado através de uma função de *hashing* aplicada a uma chave extraída da tupla. Esta chave pode ser determinada de diversas maneiras, e tuplas com diferentes chaves são armazenadas em nodos distintos. Um exemplo de implementação baseada em *hashing* é C-Linda[CAR94].

Outra questão associada à implementação de Linda diz respeito à interface oferecida ao usuário. Muitas implementações definem uma biblioteca de primitivas Linda, que se encarregam da interação com os gerenciadores de espaços de tuplas. Esta solução é empregada em várias implementações, como POSYBL[SCH91] e Glenda[SEY93]. Uma alternativa ao uso de uma biblioteca é a construção de um novo compilador para uma linguagem sequencial extendida com as primitivas Linda, como em C-Linda[CAR94]. Esta solução permite desenvolver uma interface mais simples para o usuário, porém sua implementação é mais trabalhosa.

3 O Ambiente YALI

YALI (*Yet Another Linda Implementation*) é um ambiente para programação paralela em redes heterogêneas, baseado no modelo Linda. Além das primitivas propostas neste modelo, que são incorporadas à linguagem C, YALI suporta também algumas operações globais, usadas para comunicação e sincronização envolvendo múltiplos processos, que são necessárias em vários tipos de aplicações, mas geralmente trabalhosas de expressar em Linda.

A fim de garantir flexibilidade ao programador, YALI permite estruturar uma aplicação paralela tanto num modelo SPMD (*Single Program Multiple Data*), onde um mesmo programa executa em cada nodo da rede (embora processando trechos de código diferentes), como também segundo o modelo de processos comunicantes ou MPMD (*Multiple Program Multiple Data*), onde vários processos diferentes formam a aplicação paralela.

O espaço de tuplas utilizado em YALI é distribuído segundo uma política baseada em *hashing*. Ao contrário da maioria das implementações Linda, YALI não emprega um processo especial para gerenciamento do espaço de tuplas, fazendo com que cada processo seja responsável pelo gerenciamento de uma porção do espaço global.

Para dar suporte ao desenvolvimento e execução de aplicações, YALI conta com três componentes distintos: uma biblioteca de funções (YaliLib), um pré-processador (YaliPP), e um programa de inicialização de aplicações paralelas (YaliStart).

3.1 A Biblioteca YaliLib

As funções desta biblioteca constituem a interface com o programador e, em sua maioria, implementam as operações sobre o espaço de tuplas previstas em Linda. Cinco destas operações têm o comportamento idêntico àquele proposto no modelo Linda, e por isso não serão discutidas em detalhe. São elas: Y_OUT, Y_IN, Y_RD, Y_INP, Y_RDP. As primitivas complementares ou que não seguem estritamente o modelo serão apresentadas a seguir.

- Y_EVAL: embora proposta pelo modelo Linda, esta primitiva tem uma semântica um pouco diferente em YALI, pois não está automaticamente associada à inserção de tuplas. Neste ambiente, Y_EVAL serve para disparar uma nova *thread*, que de-

verá executar uma função especificada. Esta função deve ser implementada pelo processo que a invoca através de Y_EVAL.

- Y_GLOBAL: esta primitiva permite associar um nome global a uma função implementada por um determinado processo. Através desta associação, qualquer processo pode disparar a execução da função, bastando, para isso, conhecer seu nome global.
- Y_GLOBEVAL: usada em conjunto com Y_GLOBAL, esta função atua como uma chamada remota de procedimento, fazendo com que a função associada ao nome global especificado seja processada por uma nova *thread*, disparada pelo processo que implementa a função.
- Y_REDUCE: esta primitiva é semelhante a Y_IN, porém permite remover um certo número de tuplas que satisfazem um determinado *template* e, adicionalmente, executar operações simples sobre os diversos valores coletados para os campos formais deste *template*. As operações são pré-definidas em YALI, recebem somente um argumento e produzem um resultado do mesmo tipo do argumento. Exemplos de operações são SUM, que retorna a soma dos valores dos campos, e MIN, que retorna o menor dos valroes encontrados. Y_REDUCE é uma primitiva bloqueante, isto é, o processo (ou *thread*) que a executa é interrompido até que o número especificado de tuplas seja encontrado.
- Y_BARRIER: esta primitiva é usada para sincronizar a execução de um grupo de processos. Cada processo que a utiliza permanece bloqueado até que o número especificado de processos tenha também executado Y_BARRIER.
- Y_ID: retorna a identificação do processo. O identificador é um número inteiro de 0 a (N-1), sendo N o número de processos da aplicação. Cada processo tem um identificador único, que pode ser usado para selecionar o código a ser executado em programas SPMD.
- Y_NPROC: esta primitiva retorna o número de processos que compõem uma aplicação paralela.

Para ilustrar a utilização de algumas primitivas YALI, a figura 1 mostra a implementação de uma aplicação paralela simples, composta por dois tipos de processos. um mestre, que gera tarefas, e um trabalhador, que as executa. Deve-se notar, nos *templates* utilizados com Y_IN e Y_REDUCE, a distinção entre campos reais e formais: estes últimos são precedidos pelo caracter '?'.

3.2 O Pré-Processador e o Programa de Inicialização

O pré-processador YaliPP tem o propósito de converter as chamadas YALI para um formato utilizado internamente pela biblioteca. Além disso, ele mantém uma tabela de símbolos para identificar automaticamente os tipos dos campos das tuplas fornecidas como parâmetro para Y_OUT, Y_IN, Y_RD, Y_INP, Y_RDP e Y_REDUCE, o que facilita a descrição dos campos pelo programador.

Processo Mestre

```
#define N 10

yali_main() {

  int i, result;

  for (i = 0; i < N; i++) {
    y_out("work", i);
    y_globeval("worker");
  }

  y_reduce(N, "partial_result", sum(?result));
  y_barrier("end", N);
}
```

Processo Trabalhador

```
#define N 10

yali_main() {

  y_global("worker", (void *(*)())worker());
  y_barrier("end", N+1);

}
worker() {

  int work, result;

  y_in("work", ?work);
  /* processa tarefa */
  y_out("partial_result", result);
}
```

Figura 1: Exemplo de aplicação YALI.

A inicialização de uma aplicação YALI é feita através de um programa chamado YaliStart. Este programa recebe do usuário a especificação dos processos da aplicação paralela e, opcionalmente, das máquinas onde a aplicação será executada. Esta especificação pode ser feita de três modos. No primeiro, o usuário fornece na linha de comando as informações para execução da aplicação. Este método deve ser utilizado para aplicações simples, com poucos processos diferentes (modelo SPMD, por exemplo). No segundo modo, mais genérico e flexível, utiliza-se um arquivo de configuração. Alternativamente, para usuários pouco experientes, a configuração pode ser especificada interativamente.

O programa de inicialização é responsável por distribuir os processos sobre a rede, e garantir que estes poderão se comunicar uns com os outros. A criação de processos só pode ser feita durante a inicialização, não sendo permitida a criação dinâmica de processos, mas apenas de *threads*, através de Y_EVAL ou Y_GLOBEVAL.

4 Implementação do Ambiente

A seguir serão descritos alguns detalhes sobre a implementação do espaço de tuplas e das primitivas YALI, juntamente com os procedimentos de inicialização e término de aplicações.

4.1 Arquitetura do Espaço de Tuplas

Como mencionado na seção anterior, o espaço de tuplas em YALI é distribuído segundo uma política de *hashing*, e cada processo que compõe uma aplicação é responsável por gerenciar uma porção deste espaço global. Esta arquitetura torna mais eficiente a comunicação entre processos YALI, já que mensagens para manutenção do TS são trocadas diretamente, sem o intermédio de um processo gerenciador de espaço de tuplas.

Esta arquitetura do espaço de tuplas é implementada através de duas *threads* especiais

em cada processo, como pode ser visto na figura 2. Uma das *threads*, chamada *thread* **gerenciadora**, processa operações YALI requisitadas tanto local (por uma *thread* do próprio processo) como remotamente (por *threads* de outros processos no mesmo nodo ou em nodos diferentes). Esta *thread* permanece bloqueada até que uma requisição seja inserida num *buffer* global de requisições (ver figura 2). A segunda *thread*, denominada *thread* **de comunicação**, permanece bloqueada até que uma requisição de operação seja recebida através da rede. Ela é responsável por inserir a requisição no *buffer* global, sinalizando à *thread* gerenciadora que existe nova requisição a processar.

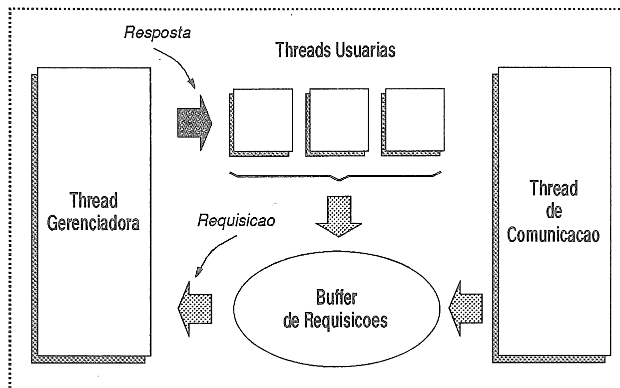

Figura 2: Processo YALI.

YALI permite que várias *threads* utilizem primitivas bloqueantes ao mesmo tempo, porque são usados semáforos diferentes para bloquear cada *thread*. As *threads* usuárias se comunicam com a *thread* gerenciadora depositando requisições no *buffer* global, como pode ser visto na figura 2. Toda requisição contém o endereço de memória onde deve ser colocada a resposta correspondente. Para tratar adequadamente cada requisição, a *thread* gerenciadora mantém as seguintes estruturas de dados:

- o espaço de tuplas propriamente dito, implementado como uma tabela *hash*;
- uma tabela de requisições Y_IN, Y_RD e Y_REDUCE pendentes, onde são armazenados *templates* cujas tuplas equivalentes não foram produzidas;
- uma tabela para gerência de barreiras, onde são armazenadas informações sobre processos bloqueados numa determinada barreira;
- uma tabela para gerência de funções globais, onde é mantida a identificação dos processos que implementam determinadas funções tornadas globais e
- uma tabela com informações sobre os processos que compõem a aplicação (identificação, conexões abertas, etc.).

As *threads* de comunicação trocam mensagens através de protocolos suportados em sistemas Unix. As mensagens constituem requisições ou respostas de operações, geral-

mente contendo tuplas ou *templates*. Processos no mesmo nodo se comunicam através de datagramas ou conexões no domínio Unix, enquanto processos remotos se comunicam através dos protocolos UDP ou TCP. Mensagens com até 2Kbytes são transferidas através de datagramas ou UDP, e mensagens maiores causam o estabelecimento dinâmico de uma conexão (Unix ou TCP) entre os processos comunicantes. Uma conexão, depois de estabelecida, é usada para transmissão de quaisquer mensagens entre os dois processos envolvidos, aproveitando melhor o custo da conexão. Para suportar a comunicação em redes heterogêneas, toda mensagem inclui uma identificação da arquitetura onde foi originada. Quando as arquiteturas são diferentes, o receptor se encarrega da decodificação da mensagem.

4.2 Implementação das Primitivas

Como a política de distribuição de tuplas é baseada em *hashing*, toda primitiva de manipulação de tuplas gera uma mensagem de requisição ao processo que mantém a tupla especificada. A identificação do processo é obtida pela aplicação da função de *hashing* sobre uma chave formada pela descrição de cada campo da tupla. Esta política de distribuição é altamente escalável, pois o número máximo de mensagens trocadas não aumenta com o número de processos. Para a implementação de Y_OUT, por exemplo, é necessário no máximo uma mensagem para enviar a tupla ao processo determinado através de *hashing*. Da mesma maneira, Y_IN, Y_RD, Y_INP e Y_RDP exigem, no máximo, uma mensagem de requisição e outra de resposta, caso a tupla solicitada não seja mantida pelo próprio processo. No caso de Y_IN e Y_RD, suas implementações envolvem ainda o registro de uma pendência para cada requisição que não for satisfeita. Quando a tupla que satisfaz uma pendência é gerada com Y_OUT, a requisição é respondida, e a pendência removida. Já as implementações de Y_INP e Y_RDP não requerem a manutenção de pendências, pois as primitivas não são bloqueantes.

As únicas primitivas que não são implementadas com auxílio de *hashing* são Y_EVAL, Y_ID e Y_NPROC. Y_EVAL apenas causa o disparo de uma nova *thread*, enquanto Y_ID e Y_NPROC retornam valores recebidos do processo YaliStart durante a inicialização da aplicação. A seguir será discutida a implementação das primitivas restantes.

- Y_GLOBAL: para tornar o nome de uma função globalmente acessível, o endereço da função e a identificação do processo que a implementa são enviados para um determinado processo da aplicação, onde serão mantidos numa tabela de funções globais. Para decidir em qual processo estas informações devem ser armazenadas, uma função de *hashing* é aplicada ao nome global da função.
- Y_GLOBEVAL: a função de *hashing* é aplicada ao nome global fornecido na operação, produzindo a identificação de um processo que mantém as informações necessárias à execução da função global (identificação do processo que a implementa e seu endereço neste processo). Uma mensagem de requisição é então enviada a este processo intermediário e, caso o nome global já tenha sido registrado, a requisição é enviada ao processo que implementa a função. Este processo, então, dispara uma *thread* para executar a função solicitada. Se a função ainda não foi registrada, isto

é, nenhum processo executou Y_GLOBAL, a requisição fica pendente no processo intermediário até que isto ocorra.

- Y_REDUCE: uma função de *hashing* é usada para descobrir qual processo mantém tuplas que satisfazem o *template* especificado. Uma requisição Y_REDUCE é enviada a este processo, contendo o número de tuplas que devem ser removidas, além das operações a serem aplicadas nos campos das tuplas coletadas. Caso não existam tuplas suficientes, pendências são registradas pelo processo (tantas quantas forem as tuplas faltantes). Quando todas as tuplas estiverem disponíveis, as operações de combinação de campos são aplicadas, e a tupla resultante é enviada ao processo que executou Y_REDUCE.
- Y_BARRIER: esta primitiva também utiliza uma função de *hashing* para determinar o processo que deve gerenciar a barreira especificada. Uma requisição Y_BARRIER é enviada a este processo, que pode tratá-la de duas maneiras. Caso seja a primeira requisição para uma determinada barreira, um contador é inicializado com o número de processos que devem se sincronizar. Caso contrário, este contador é decrementado e, quando for igual a zero, o processo que gerencia a barreira envia mensagens a todos os processos bloqueados, liberando-os para execução.

4.3 Inicialização e Término de Aplicações

Antes que os processos de uma aplicação YALI possam se comunicar através do espaço de tuplas, é necessária a execução de um procedimento de inicialização, cujas etapas são descritas a seguir.

- Inicialmente, YaliStart dispara os processos da aplicação, enviando sua própria identificação como um parâmetro para estes processos;
- cada processo cria seus canais (*sockets*) de comunicação, e envia as respectivas identificações a YaliStart;
- YaliStart reúne identificações de canais de todos os processos, enviando uma lista com estas informações a cada processo da aplicação;
- cada processo, após receber esta lista, cria as *threads* necessárias para implementação do espaço de tuplas;
- por fim, é criada uma *thread* para executar a função principal de cada processo, que deve ter sido codificada pelo usuário com o nome YALI_MAIN.

As primeiras três etapas do procedimento de inicialização são necessárias porque os canais de comunicação são criados dinamicamente, e suas identificações não são conhecidas até o momento da execução. O processamento da função principal através de uma *thread* tem o propósito de garantir que, mesmo após seu término, as *threads* que implementam o espaço de tuplas ainda permaneçam executando. Quando a função principal termina, YaliStart é notificado e, quando todos os processos encerram suas funções principais, a aplicação YALI é finalmente encerrada.

5 Avaliação do Ambiente

O propósito desta seção é fornecer uma avaliação preliminar do ambiente YALI, tanto em termos qualitativos como quantitativos. Para permitir uma avaliação geral das características do ambiente, foi organizada uma tabela comparativa entre YALI e algumas implementações Linda citadas na seção 2.1. Em complemento às informações contidas na tabela 1, deve-se mencionar que as operações globais suportadas por p4-Linda são, na verdade, herdadas da biblioteca p4[BOY87], na qual está baseada esta implementação. Também deve-se observar que C-Linda, ao contrário das outras implementações, é um produto comercial, que inclui não somente um pré-processador, mas também um compilador que permite várias otimizações na implementação do modelo Linda. Entre as implementações comparadas, a maioria delas emprega um processo intermediário para gerenciamento do espaço de tuplas, com exceção de YALI e C-Linda.

Tabela 1: Comparação entre YALI e outras implementações Linda.

Sistemas	**Processo Gerenciador do TS**	**Pré-Processador**	**Política de Distribuição**	**Operações Globais**
C-Linda	não	sim	*hashing*	não
p4-Linda	sim	não	centralizada	sim
Glenda	sim	sim	centralizada	não
POSYBL	sim	não	uniforme	não
YALI	não	sim	*hashing*	sim

Para uma avaliação quantitativa do ambiente foi implementado uma aplicação conhecida como *ping/pong*, onde um processo deposita uma tupla "ping" e espera por uma tupla "pong", enquanto outro retira a tupla "ping" e insere a tupla "pong". Um total de 100 iterações *ping/pong* foram executadas, com tuplas de diferentes tamanhos, fornecendo os resultados contidos na tabela 2. Observando-se esta tabela, nota-se um aumento significativo do tempo para tuplas com mais de 1000 bytes, o que pode ser explicado pela necessidade do estabelecimento de uma conexão quando as tuplas excedem 2Kbytes. Os dados contidos na tabela 2 foram obtidos numa rede de estações Sun, modelo SPARCClassic, num período de baixa utilização da rede.

6 Conclusão e Trabalhos Futuros

Este artigo apresentou a interface e implementação de um ambiente para programação paralela em redes heterogêneas, baseado no modelo Linda. Entre as principais características deste ambiente está o suporte a operações globais, que permitem a comunicação e a sincronização entre um grupo de processos. Outra característica importante consiste na implementação baseada em *threads*, que permite embutir a funcionalidade do espaço de tuplas em cada processo de uma aplicação paralela, diminuindo o volume de comunicação entre processos.

Tabela 2: Tempos médios de cada sequência *ping/pong*.

Tamanho da Tupla (em bytes)	**Tempo Médio** (em milissegundos)
100	9.6
400	10.8
1000	13.3
4000	23.2
10000	57.6

O projeto do ambiente teve uma preocupação muito forte em oferecer uma interface simples e flexível, ao mesmo tempo permitindo a execução eficiente de aplicações paralelas. Mesmo assim, várias otimizações têm sido planejadas. Entre elas estão o desenvolvimento de uma interface gráfica para a configuração das aplicações e a utilização de informações sobre a carga das máquinas para auxiliar na distribuição eficiente dos processos durante a inicialização das aplicações.

A implementação do ambiente ainda está em evolução e, atualmente, está disponível em duas plataformas: SPARC/Solaris e i386/Mach. Está previsto o porte para outras arquiteturas e sistemas que suportam *threads*, como o sistema UNICOS que executa no Cray Y-MP.

Referências

[BOY87] BOYLE, J. et al. *Portable Programs for Parallel Processors*. Hold, Rinehart, and Winston, 1987.

[BUT93] BUTLER, R.M.; LUSK, E. *p4-Linda: A Portable Implementation of Linda*, in Proc. 2nd Int. Symposium on High-Performance Distributing Computing, IEEE Computer Society Press, 1993.

[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.

[CAR94] CARRIERO, N. et al. *The Linda alternative do message-passing systems*. Parallel Computing, n.20, 1994.

[FOS95] FOSTER, I. *Designing and Building Parallel Programs*. Addison Wesley, 1995.

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

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

[PIN91] PINAKIS, J. *The Design and Implementation of a Distributed Linda Tuple Space*, in Proc. of the Dept. of Computer Science Research Conference, University of Western Australia, 1991.

[SCH91] SCHOINAS, G. *Issues on the implementation of PrOgramming SYstem for distriButed appLications*. Department of Computer Science, University of Crete, Greece, 1991.

[SEY93] SEYFARTH, B. et al. *Glenda Installation and Use*. University of Southern Mississippi, 1993.