YALI
Uma Extensão do Modelo Linda com Suporte a Operações Globais

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

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

Sumário

Este artigo descreve YALI, um ambiente destinado à programação paralela em redes heterogêneas. Com o objetivo de oferecer uma interface simples e flexível, YALI implementa o modelo Linda, que destaca-se por permitir a interação entre processos através de uma memória compartilhada chamada Espaço de Tuplas. Uma das principais características de YALI é o suporte a operações globais, que facilitam e otimizam a comunicação e a sincronização entre múltiplos processos, e que são normalmente trabalhosas de expressar no modelo Linda original.
Palavras-chave: Linda, programação paralela, operações globais.

Sumário

This paper describes YALI, an environment for parallel programming in heterogeneous networks. In order to provide a simple and flexible interface, YALI implements the Linda model, which main contribution is to allow process interaction through a kind of shared memory called Tuple Space. One of the YALI's main features is the inclusion of global operations in Linda. Such operations are helpful to efficiently express communication and synchronization between multiple processes, and they are in general hard to code in the original Linda model.
Keywords: Linda, parallel programming, global operations.

1 Introdução

Entre as ferramentas que têm surgido para suportar a programação paralela em redes de computadores, poucas não se baseiam no paradigma de troca de mensagens. Apesar de eficiente, este paradigma é de difícil assimilação, já que não esconde do programador a distribuição inerente às redes. Uma alternativa de mais alto nível que a troca de mensagens é proposta pelo modelo Linda[1][GEL85, CAR94], onde toda interação entre processos ocorre por intermédio de uma memória associativa logicamente compartilhada, denominada Espaço de Tuplas. Entre as vantagens de Linda, destaca-se a simplicidade de suas primitivas, e a possibilidade de incorporá-las a uma linguagem sequencial conhecida.

Por representar um paradigma de mais alto nível de abstração, o modelo Linda foi escolhido como base para o desenvolvimento de YALI (*Yet Another Linda Implementation*), um ambiente para programação paralela em redes heterogêneas. Um dos principais objetivos deste ambiente é oferecer uma interface simples e flexível para programação paralela em arquiteturas deste tipo. Para isso, YALI implementa o modelo Linda e incorpora primitivas para criação dinâmica de *threads* e suporte a operações globais[FOS95]. A manipulação de *threads* torna mais flexível a expressão do paralelismo em YALI, enquanto as operações globais facilitam e otimizam a comunicação e sincronização entre um grupo de processos. Operações deste tipo são trabalhosas de expressar no modelo Linda original, e não são encontradas em outros ambientes que implementam o modelo. O artigo está organizado da seguinte maneira: a seção 2 discute sucintamente o modelo Linda, e na seção 3, a seguir, são apresentados os recursos do ambiente YALI. As seções 4 e 5 tratam, respectivamente, da implementação e da avaliação do ambiente.

2 O Modelo Linda

O modelo Linda[GEL85] consiste em um pequeno conjunto de primitivas que permitem estender uma linguagem sequencial, tornando-a adequada para programação paralela. O paradigma de programação suportado por este modelo baseia-se no compartilhamento de uma memória global associativa denominada **espaço de tuplas** (*Tuple Space* — TS). A unidade de armazenamento nesta memória – as tuplas – são sequências de campos de tipos bem definidos. Os tipos de campos permitidos são, geralmente, aqueles suportados pela própria linguagem sequencial hospedeira de Linda.

Basicamente, existem quatro operações definidas em Linda: **out**, **in**, **rd** e **eval**. Todas elas são processadas atomicamente e servem para inserir ou recuperar tuplas do TS. Uma operação **out** insere assincronamente uma tupla no espaço, enquanto **eval** dispara um novo processo para executar uma função cujo resultado é depositado em uma tupla no TS. 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, ou campos **formais**, representados por variáveis que devem recebem algum valor quando uma tupla equivalente ao *template* é encontrada no TS. Caso não seja encontrada uma tupla equivalente, o processo que executa **in** é bloqueado até que a tupla seja depositada. A primitiva **rd**, por fim, também serve para a recuperação de tuplas, mas a tupla que satisfaz o *template* não é removida do TS.

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

2.1 Implementações do Modelo Linda em Redes

O modelo Linda, apesar de envolver uma memória compartilhada, pode ser implementado em arquiteturas desprovidas deste recurso, como é o caso das redes de computadores. Entre os sistemas que implementam Linda em um ambiente de rede, pode-se destacar Glenda[SEY93], p4-Linda[BUT93], POSYBL[SEY93] e Network-Linda[CAR94]. Em Glenda e p4-Linda, o compartilhamento lógico do espaço de tuplas se dá através de um processo servidor centralizado, que armazena tuplas e processa requisições de operações Linda provenientes de processos clientes em qualquer nodo da rede. Já em POSYBL, um processo gerenciador é instanciado em todos os nodos da rede, e cada um destes processos fica responsável por armazenar tuplas geradas por clientes locais. Quando uma tupla não é encontrada localmente, o gerenciador se comunica com os processos gerenciadores em outros nodos, a fim de localizar, se possível, a tupla requisitada. O sistema Network-Linda, por sua vez, é um produto comercial, que implementa o espaço de tuplas sem o intermédio de um processo servidor independente. Em Network-Linda, uma porção do espaço de tuplas é gerenciada por cada processo da aplicação.

Outra questão importante associada à implementação do modelo é a interface oferecida ao usuário. Embora as primitivas Linda sejam geralmente implementadas em uma biblioteca, alguns sistemas ainda incluem um pré-processador ou pré-compilador, que transforma um programa fonte contendo primitivas Linda em um programa puramente na linguagem hospedeira (geralmente C). Esta solução permite desenvolver uma interface mais simples para o usuário, e também possibilita a análise de programas em tempo de compilação, a fim de otimizar os acessos ao espaço de tuplas.

3 O Ambiente YALI

YALI implementa o modelo Linda incorporado à linguagem C, e inclui extensões para criação dinâmica de *threads* e suporte a operações globais. Operações deste tipo são necessárias em vários tipos de aplicações, mas são geralmente trabalhosas de expressar em Linda. Para dar suporte ao desenvolvimento e execução de aplicações, YALI conta com três componentes: uma biblioteca de funções (YaliLib), um pré-processador (YaliPP), e um programa de inicialização de aplicações paralelas (YaliStart).

3.1 Paralelismo em YALI

Em YALI, processos são distribuídos estaticamente sobre uma rede de computadores através do programa YaliStart. A fim de garantir flexibilidade ao programador, YALI permite estruturar uma aplicação paralela tanto num modelo SPMD (*Single Program Multiple Data*), onde uma instância de 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 distintos formam a aplicação paralela.

O paralelismo também pode ser expresso em YALI através de *threads* que são disparadas dinamicamente. Para isso, a biblioteca YaliLib oferece as seguintes primitivas:

- Y_EVAL: embora proposta pelo modelo Linda, esta primitiva tem uma semântica diferente em YALI, pois não está automaticamente associada à inserção de tuplas. Nes

te ambiente, Y_EVAL serve para disparar uma nova *thread*, que deverá executar uma função especificada. Esta função deve ser implementada pelo mesmo processo que a invoca através de Y_EVAL.

- Y_GLOBAL: um processo utiliza esta primitiva para associar um nome global a uma determinada função, permitindo que qualquer processo que conheça este nome possa disparar a execução desta função.

- Y_GLOBEVAL: usada em conjunto com Y_GLOBAL, esta primitiva recebe como argumento um nome global de função, e dispara sua execução através de uma nova *thread* no processo que a implementa. Esta primitiva atua de maneira semelhante a uma chamada remota de procedimento, onde o servidor é o processo que executa Y_GLOBAL.

3.2 Comunicação e Sincronização em YALI

Em YALI, assim como em Linda, a comunicação entre processos é feita indiretamente, por intermédio do espaço de tuplas. Para isso, um processo usa Y_OUT para depositar no TS uma tupla contendo dados, enquanto outro processo posteriormente a recupera usando Y_IN ou Y_RD. A comunicação é anônima, pois nenhuma informação sobre a identificação ou localização de processos é necessária para a troca de dados entre eles, e também é ortogonal, já que os processos envolvidos, tanto emissores como receptores, não precisam conhecer a identificação de outros processos. Esta propriedade normalmente não é verificada em sistemas baseados em troca de mensagens, onde os emissores identificam o processo receptor, mas dificilmente ocorre o contrário.

A sincronização entre processos para coordenar o acesso a tuplas compartilhadas é garantida pela atomicidade no processamento de Y_OUT, Y_IN e Y_RD. Já que não existe uma primitiva para alterar os campos de uma tupla, um processo deve removê-la, modificá-la e inseri-la novamente no TS, o que garante a integridade dos dados compartilhados.

3.3 Operações Globais

Aplicações paralelas frequentemente necessitam expressar a interação entre múltiplos processos. Por isso, muitos ambientes de programação oferecem operações globais, capazes de facilitar e otimizar este tipo de interação. Em ambientes baseados em troca de mensagens, as operações globais mais comuns implementam comunicação um-para-todos, um-para-muitos ou muitos-para-um. Alguns ambientes também suportam operações de redução, que servem para reunir um conjunto de dados distribuídos, e barreiras, utilizadas para sincronização de um grupo de processos. Estas duas últimas operações, que não estão ligadas estritamente à troca de mensagens, podem ser incorporadas ao modelo Linda sem afetar o seu nível de abstração. YALI suporta estas operações globais através das primitivas descritas a seguir:

- Y_REDUCE: esta primitiva é semelhante a Y_IN, porém serve para recuperar não uma, mas várias tuplas que satisfaçam um mesmo *template*, depositadas no TS por diferentes processos. Y_REDUCE permite também combinar os valores coletados para determinados campos formais, usando para isso algum operador pré-definido. Exemplos de operadores permitidos em YALI são SUM, que retorna a soma dos valores de um campo, e MIN, que retorna o menor entre os valores encontrados.

- Y_BARRIER: implementa a sincronização global de processos. Y_BARRIER recebe como argumento um nome global de barreira, e sua função é bloquear todos os processos que atingem tal barreira até que o número especificado de processos esteja sincronizado.

3.4 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 e Y_REDUCE, o que facilita a enumeração dos campos pelo programador.

O programa de inicialização é responsável por distribuir processos sobre a rede, e garantir que estes poderão se comunicar uns com os outros. 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.

3.5 Exemplo

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 '?'.

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);

}
worker() {

  int work, result;

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

Figura 1: Exemplo de aplicação YALI.

4 Implementação do Ambiente

O espaço de tuplas utilizado em YALI é distribuído sobre os processos que compõem uma aplicação paralela. Ao contrário da maioria das implementações de Linda em redes, como p4-Linda, Glenda e POSYBL, YALI não emprega um processo especial para gerenciamento do espaço de tuplas, fazendo com que cada processo da aplicação seja responsável por gerenciar uma porção do espaço global. Esta arquitetura torna mais eficiente a comunicação entre processos YALI, já que mensagens para manutenção do TS são trocadas diretamente, sem passar por um processo intermediário que gerencia o espaço de tuplas. Para decidir quais tuplas devem ser armazenadas em quais processos, YALI utiliza uma política de distribuição baseada em *hashing*. A descrição dos campos de uma tupla serve de chave para uma função de *hash*, cujo resultado é a identificação do processo que deve armazenar a tupla. A seguir serão descritos alguns detalhes sobre a implementação do espaço de tuplas e das primitivas YALI.

4.1 Organização do Espaço de Tuplas

Para embutir a funcionalidade do espaço de tuplas em cada processo, YALI utiliza duas *threads* especiais em cada processo, como pode ser visto na figura 2. Uma das *threads*, chamada **thread gerenciadora**, processa operações YALI requisitadas tanto localmente, 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.

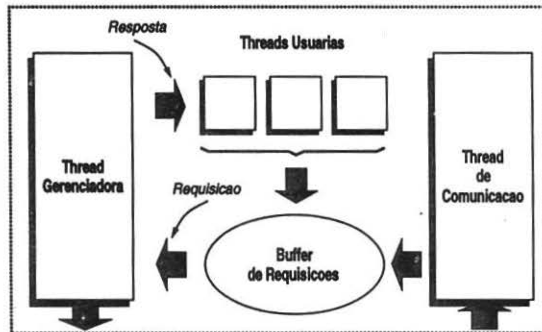

Figura 2: Processo YALI.

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.

As *threads* de comunicação trocam mensagens através de protocolos suportados em sis-

temas Unix. As mensagens constituem requisições ou respostas de operações, geralmente 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, reduzindo a sobrecarga existente no estabelecimento de uma nova 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 tupla gera uma mensagem de requisição ao processo que mantém a tupla especificada. Esta política de distribuição foi escolhida por ser altamente extensível, já que o número máximo de mensagens necessárias para implementação das primitivas não aumenta conforme o número de processos da aplicação. Para a implementação de Y_OUT, por exemplo, é necessário no máximo uma mensagem para enviar a tupla ao processo determinado através da função de *hash*. Da mesma maneira, Y_IN e Y_RD exigem, no máximo, uma mensagem de requisição e outra de resposta, caso a tupla solicitada não seja mantida pelo próprio processo. A implementação destas primitivas envolve 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. 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 (servidor) são enviados para um determinado processo da aplicação, onde são mantidos numa tabela de funções globais. Para decidir em qual processo estas informações devem ser armazenadas, uma função de *hash* é aplicada ao nome global da função.

- Y_GLOBEVAL: a função de *hash* é aplicada ao nome global fornecido na operação, produzindo a identificação de um processo intermediário que mantém as informações necessárias à execução da função global (identificação do processo servidor 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 servidor. 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 *hash* 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 então 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.

Com relação à implementação de Y_REDUCE e Y_BARRIER, deve-se notar que estas operações globais poderiam ser implementadas pelo programador somente utilizando as primitivas do modelo Linda. A operação de redução, por exemplo, é equivalente a uma sequência de chamadas Y_IN, seguida de operações para combinação dos campos. No entanto, esta implementação seria menos eficiente, já que exigiria um número maior de operações e, consequentemente, mais mensagens trocadas entre processos. Outra vantagem da existência de Y_REDUCE é que as operações de combinação de campos são executadas pelo processo determinado por *hashing*, o que permite distribuir a carga associada à execução de uma redução. Um mecanismo simples de sincronização de barreira também poderia ser implementado com primitivas Linda, como mostra [CAR89], porém implementações mais robustas seriam mais complexas e, exigindo maior número de operações, seriam menos eficientes que Y_BARRIER.

5 Avaliação do Ambiente

Para permitir uma avaliação geral das características do ambiente, foi organizada uma tabela comparativa entre YALI e outras implementações de Linda em redes, comentadas na seção2.1. Para auxiliar na comparação dos sistemas, vale mencionar, em primeiro lugar, que a existência de um processo intermediário para gerenciamento do TS inevitavelmente aumenta a quantidade de mensagens necessárias para implementação das primitivas Linda. Em segundo lugar, sistemas que não incluem um pré-processador geralmente exigem que o usuário indique o tipo de cada campo utilizado numa chamada Linda, o que dificulta a utilização das primitivas. Por fim, um espaço de tuplas centralizado é uma solução simples de ser implementada, mas pode apresentar problemas de sobrecarga no processo servidor quando o número de processos clientes aumenta.

Tabela 1: Comparação entre YALI e outras implementações Linda.

Sistemas	Processo Gerenciador do TS	Pré-Processador	Espaço de Tuplas	Operações Globais	Heterogeneidade
C-Linda	não	sim	distribuído	não	sim
p4-Linda	sim	não	centralizado	não	sim
Glenda	sim	sim	centralizado	não	sim
POSYBL	sim	não	distribuído	não	não
YALI	não	sim	distribuído	sim	sim

A fim de permitir também uma avaliação quantitativa do ambiente, o algoritmo de Mandelbrot para geração de fractais foi implementado em uma versão preliminar de YALI,

que ainda não suporta *threads*. A implementação distribuída deste algoritmo seguiu o paradigma mestre/trabalhador, onde um processo mestre deposita tuplas contendo um bloco de imagem a calcular, e vários processos trabalhadores devolvem tuplas contendo informações para apresentação do bloco calculado. Esta aplicação foi executada com diferentes números de processos trabalhadores executando em nodos diferentes da rede, para vários tamanhos de imagem, e o *speedup* para esta aplicação é apresentado na figura 3. Observando-se esta figura, nota-se que o *speedup* não decai significativamente com o aumento do número de nodos, o que se deve à política de *hashing* adotada em YALI.

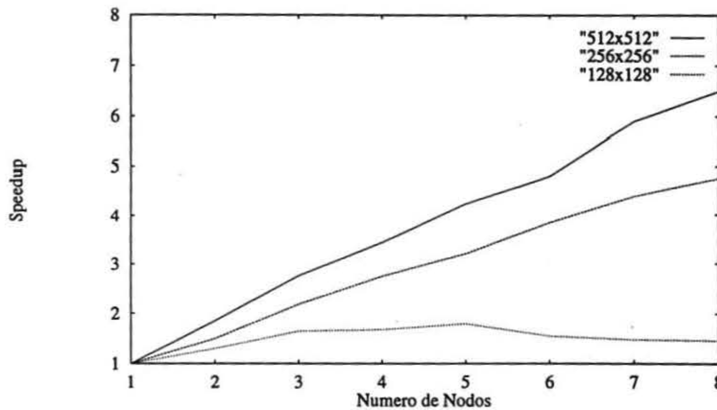

Figura 3: *Speedup* da implementação distribuída do algoritmo de Mandelbrot.

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 características marcantes deste ambiente estão o suporte a operações globais e a possibilidade de criação dinâmica de *threads* através de nomes globais de funções. As primitivas que implementam esta nova funcionalidade preservam a ortogonalidade do modelo Linda, mantendo o nível de abstração original do modelo.

Com relação à implementação, uma característica importante consiste no uso de múltiplas *threads*, que permitem embutir a funcionalidade do espaço de tuplas em cada processo de uma aplicação paralela, diminuindo o volume de comunicação entre processos.

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 sendo realizada 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

[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.

[CAR89] CARRIERO, N.; GELERNTER, D. *How to Write Parallel Programs: A Guide to the Perplexed*. ACM Computing Surveys, v.21, n.3, Sep. 1989.

[CAR94] CARRIERO, N. et al. *The Linda alternative to 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.

[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.