X Simpósio Brasileiro de Arquitetura de Computadores

# Penelope - Um Modelo de Escalonador Hieráquico para o Sistema PLoSys

Ana Paula Bluhm Centeno* Cláudio Fernando Resin Geyer†

Instituto de Informática - UFRGS
Caixa Postal 15064
91501-970 Porto Alegre-RS
Fone (051) 316-6802

**Resumo**

Em sistemas paralelos os algoritmos de escalonamento utilizados têm fundamental importância no ganho de desempenho da execução, pois além de designar processos a processadores da melhor forma possível devem manter o *overhead* de escalonamento baixo.

Este trabalho tem por objetivo apresentar o Penelope, um modelo de escalonador distribuído hierárquico para o sistema PLoSys . Com o Penelope tentamos diminuir o número e o tamanho das mensagens trocadas entre os processadores e manter a localidade dos mesmos.

**Palavras-Chave:** Escalonamento, Programação em Lógica, Sistemas Distribuídos e Paralelismo

**Abstract**

Scheduling algorithms play a fundamental role in the seek for faster executions in parallel systems because, besides designating processes to processors as best as it's possible, they must also keep scheduling overheads at a low level.

The aim of this work is shows the Penelope, a model of the hierarquical distributed scheduler to PLoSys. The Penelope tries keep low the number and size of the messages between the processors.

**Keywords:** Scheduling, Logic Programming, Distributed Systems and Paralelism

*Mestranda do CPGCC/UFRGS. E-mail: anapaula@inf.ufrgs.br
†Professor Doutor da UFRGS. E-mail: geyer@inf.ufrgs.br

# 1 Introdução

O sistema PLoSys, utilizado no Instituto de Informática da UFRGS, explora o paralelismo OU implícito nos programas Prolog em ambientes de memória fisicamente distribuída. Nestes ambiente utiliza-se troca de mensagem como única forma de comunicação entre os elementos de processamento.

A vantagem obtida quando paralelizamos programas Prolog é desenvolver aplicações em uma linguagem de alto nível e que podem ser executadas mais rápido em paralelo, sem intervenção do usuário final.

A exploração do paralelismo pode ser limitada pela falta de paralelismo no programa, desbalanceamento de carga e *overheads*. Estes *overheads* surgem do suporte ao paralelismo, da latência de comunicação e do escalonamento para balancear a carga (escalonamento deve ser feito de forma dinâmica devido a natureza irregular das computações Prolog).

O artigo se concentra em minimizar os *overheads* de escalonamento ao mesmo tempo em que emprega técnicas de balanceamento de carga que garantem que para a maior parte das aplicações, os processadores estarão executando uma quantidade semelhante de trabalho, ou seja, a execução vai estar balanceada.

O artigo está organizado em seis seções. Na seção 2 o sistema PLoSys é descrito em detalhes. Na seção 3 descrevemos o modelo de escalonador hierárquico Penelope. Na seção 4 descrevemos questões de implementação relevantes ao trabalho. Na seção 5 fazemos algumas comparações com trabalhos relacionados e por fim a seção 6 conclui o trabalho apresentado.

# 2 O Sistema PLoSys - Parallel Logic System

O PLoSys [8] explora o paralelismo OU em programas Prolog. A arquitetura utilizada para a execução dos programas é uma máquina paralela com memória distribuída. Segundo [2], a execução de um programa Prolog pode ser vista como uma exaustiva procura paralela por todas soluções possíveis. A execução começa em um processador com a evolução de uma meta inicial. Os outros processadores pedem por trabalho tentando receber uma parte inexplorada da árvore de execução Prolog. O espaço de procura de cada processador é representado pelos pontos de escolha na pilha Local. A profundidade desta pilha representa o número de pontos de escolha que esperam para serem executados pelo processador. O PLoSys foi inicialmente desenvolvido no LMC-IMAG na Universidade de Grenoble na França, e implementado usando a máquina de inferência WAMCC [5]. A comunicação é feita através de uma biblioteca desenvolvida sobre MPI no LMC-IMAG, o Athapascan [3].

## 2.1 O Trabalhador

No PLoSys, um programa Prolog é executado em paralelo por diversos trabalhadores. Cada trabalhador inclui três *threads*: uma exportadora, uma importadora e uma máquina de inferência Prolog. A carga de um trabalhador consiste de alternativas não exploradas de um ou mais pontos de escolha. Cada trabalhador envia sua carga ao escalonador através de chamadas da máquina de inferência. O estado de carga que um nodo pode assumir são: *idle*: o trabalhador não tem qualquer tarefa Prolog para computar; *quiet*: o trabalhador está ativo mas não tem trabalho suficiente para dividir com um *idle*; *overloaded*: o trabalhador está com carga de trabalho acima do limite, tendo assim trabalho suficiente

para dividir com um trabalhador *idle*. O limite entre *quiet* e *overloaded* é uma constante de $k$ pontos de escolha (estimada experimentalmente).

### 2.2 O Escalonador

A estratégia de escalonamento é centralizada. O escalonador mantém um estado global aproximado da carga do sistema usando as informações de carga enviadas pelos trabalhadores.

O PLoSys utiliza cópia total das pilhas da WAM para realizar a transferência de trabalho de um trabalhador para outro, e também escolhe o ponto de escolha que estiver mais alto na árvore de execução, seguindo a heurística que "quanto mais alto está o trabalho, maior é sua granulosidade". Para isso, cada ponto de escolha é marcado com a data da sua criação, a qual corresponde à profundidade na árvore de busca.

No modelo de escalonamento centralizado utilizado pelo PLoSys apenas um trabalhador coleta as informações do sistema e faz as decisões de escalonamento, ou seja, apenas um processador ou máquina regula carga de todos os processos. Um problema para sistemas que usam escalonamento centralizado é este único processador que faz o escalonamento torna-se um gargalo, pois a comunicação para manter um estado de carga aproximado do sistema quando tem-se muitos processadores é alta.

Então, para evitar este gargalo desenvolveu-se o Penelope, um modelo de escalonador hierárquico para o sistema PLoSys.

## 3 Penelope - O Modelo do Escalonador Hierárquico para Sistemas OU de Programação em Lógica

Essa seção apresenta o escalonador distribuído modelado para o sistema PLoSys, que explora o paralelismo OU em programas em lógica em ambientes de memória distribuída. A este escalonador foi dado o nome de Penolope.

### 3.1 Considerações Iniciais

Para apresentar o Penolope devemos primeiro ressaltar as principais características que regeram sua modelagem. Segundo [6] as principais características que um escalonador distribuído deve possuir são: diminuir o número de mensagens trocadas entre os processadores, transmitir mensagens pequenas, manter comunicação com processadores que tem alguma afinidade e por fim o escalonador deve tomar decisões rápidas.

Adotamos que o Penolope teria uma abordagem hierárquica, para que nem todos os processadores precisassem manter comunicação com todos os outros do sistema, já que possuímos uma rede de estações de trabalho, tentamos com isso diminuir o número de mensagens total do sistema.

Para diminuir o tamanho das mensagens trocadas entre os processadores, optamos por implementar o modelo de cópia incremental ao invés de cópia total das pilhas da WAM. Em Prolog paralelo distribuído sempre que um trabalhador importar trabalho de um outro, deve trazer para suas pilhas todo o contexto da tarefa além da própria tarefa para ser executada. Com a cópia incremental não é necessário copiar totalmente as pilhas da WAM referentes ao contexto da tarefa, mas somente a parte não comum entre os dois trabalhadores. O algoritmo utilizado foi proposto por [4].

O importante para a cópia incremental é que um processador sempre tente, primeiro, pegar trabalho de um outro que tenha passado pelo mesmo caminho que ele até um certo ponto. Este mesmo caminho significa que ambos podem ter em suas pilhas várias informações comuns, fazendo que a quantidade de informações das mensagens diminua.

Para manter somente a comunicação entre processadores com alguma afinidade, fizemos com que as informações fluíssem das folhas para a raiz na árvore de execução Prolog. Com isso nossa comunicação é tida como *bottom-up*.

Como cada processador do sistema possui um escalonador e uma máquina abstrata o escalonamento é distrubído hierárquico. O escalonador é responsável pela distribuição de tarefas Prolog entre os processadores e a reorganização dos mesmos em caso de sobrecarga ou subutilização. A máquinaé módulo responsável pela execução dos programas Prolog.

Como as informações fluem das folhas para a raiz, cada escalonador mantém informações referentes aos seus descendentes ($i$,$o$) e filhos propriamente ditos ($s$), além de manter uma referência do seu trabalhador pai, ou seja, do trabalhador de quem importou trabalho. Estas características podem ser exemplificadas pela figura 1, onde Px são os processadores do sistema localizados na árvore de execução Prolog e os pontos cheios são os pontos de escolha.

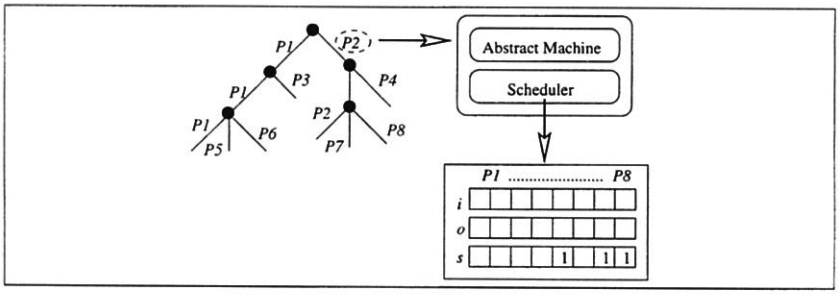


Figura 1: Árvore de Execução, Processadores e Conjunto de Dados de um Escalonador

## 3.2 Caracterização da Política de Escalonamento

Para melhor descrever as decisões tomadas para o Penolope, utilizaremos os quatro componentes de decisão adotadas por [13], as quais veremos a seguir: **Política de Transferência**: determina quando um trabalhador está apto a participar da transferência de trabalho; **Política de Seleção**: determina qual tarefa deve ser transferida; **Política de Localização**: determina o trabalhador para o qual a tarefa será transferida; **Política de Informação**: deve manter atualizado o estado de carga geral do sistema.

### 3.2.1 Política de Transferência

A política de transferência determina quando um trabalhador está apto a participar da transferência de trabalho. Para determinar a política de transferência do modelo precisamos definir os estados que um trabalhador pode assumir. Hoje estes estados são: *idle*, *quiet* e *overloaded*. Os limites para a mudança de estado de um trabalhador são calculados através da quantidade de pontos de escolha criados e ainda não explorados.

No Penelope os estados que poderão ser assumidos pelos trabalhadores são (análogos ao PLoSys):

- ***idle***: o trabalhador não tem qualquer tarefa Prolog para computar;
- ***silent***: o trabalhador está trabalhando mas não tem trabalho suficiente para dividir com um trabalhador em estado *idle*;
- ***overloaded***: o trabalhador está com carga de trabalho acima do limite, tendo assim trabalho suficiente para dividir com um trabalhador *idle*.

Trabalhadores em estado *idle* ou *overloaded* estão capacitados a participar de uma transferência de trabalho, um nenhuma tarefa Prolog a computar e outro com tarefas em excesso, respectivamente.

A transição de estado dos trabalhadores processadores é delimitada pelos limites, *low* e *high* e pode ser ilustrada pela figura 2. Um trabalhador que esteja entre os limites é considerado *silent*. Estes limites têm considerável efeito no desempenho do sistema e várias propostas para o cálculo podem ser encontradas em [10], [14] e [15].

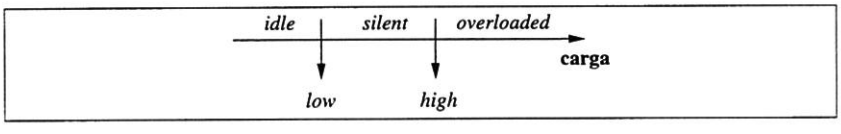


Figura 2: Transição de Estado dos Trabalhadores

Para calcular os limites utilizaremos a complexidade dos pontos de escolha que ainda não foram explorados. Esta complexidade é calculada a partir de informações disponibilizadas pelo módulo GRANLOG [1] e utilizadas pelo PLoSys para calcular a complexidade de cada ponto de escolha ainda não explorado [7].

### 3.2.2 Política de Seleção

A decisão de qual tarefa deve ser exportada em caso de sobrecarga, tem grande influência no desempenho do sistema. Quando exportamos uma tarefa em um sistema de memória distribuída o custo é muito maior do que se exportássemos a mesma tarefa em um sistema de memória compartilhada. Em sistemas de memória distribuída a exportação é implementada com troca de mensagens, enquanto que em um sistema de memória compartilhada apenas variáveis em comum são acessadas.

Logo, para sistemas de memória distribuída como o PLoSys, é mais conveniente exportar tarefas com maior granulosidade, ou seja, tarefas que mantenha o processador ocupado por mais tempo exigindo menos comunicação.

Já em sistemas que utilizam memória compartilhada, como o Muse [9] e o Aurora [12] o custo de transferência de tarefa é baixo. Nestes sistemas são exportados os pontos de escolha mais recentes na árvore.

O escalonador atual do PLoSys exporta o ponto de escolha inexplorado mais antigo, ou seja, o que estiver mais alto na árvore de busca, seguindo a heurística de que "quanto mais alto está o trabalho, maior é sua granulosidade", ou seja, maior seu volume de trabalho.

No Penolope esta decisão é baseada na complexidade dos pontos de escolha. Calculando o custo de execução dos pontos de escolha inexplorados, podemos decidir com maior precisão qual ponto de escolha exportar.

A informação de complexidade de cada ponto de escolha será utilizada para decidir quantos pontos de escolha exportar, a partir da base da pilha Local, onde estão os pontos de escolha inexplorados. Considera-se sempre que o custo de execução dos pontos de escolha que continuarão no trabalhador exportador.

### 3.2.3 Política de Localização

Para determinar o trabalhador para o qual uma tarefa será transferida encontra-se na literatura três políticas de localização distribuídas: *sender-initiated*, iniciadas pelo exportador; *receiver-initiated*, iniciadas pelo importador ou *symmetrically-initiated*, iniciadas pelo importador e/ou exportador.

O Penolope adota a política *symmetrically-initiated* para a política de localização, já que ambos trabalhadores (*idle*, *overloaded*) procuram por trabalho. Com esta política em nenhum momento um trabalhador *idle* fica esperando que um trabalhador *overloaded* o encontre e vice-versa, perdendo tempo de processamento.

### 3.2.4 Política de Informação

O custo para que cada trabalhador mantenha o estado global do sistema atualizado é alto. Então, para manter o estado global do sistema sem degradá-lo com a utilização de *broadcasts*, e como utilizamos o fluxo de informações *bottom-up* optamos pela manutenção dos conjuntos de dados *idle*, *overloaed* e *sons*. Cada conjunto de dados contém todos os trabalhadores do sistema, mas com informações apenas dos seus descendentes.

Como foi dito anteriormente cada escalonador possuirá um conjunto de dados *idle* e outro *overloaded*, compostos por todos os processadores do sistema, mas apenas estarão setados os processadores que lhe enviaram mensagem dizendo que estão em estado *idle* ou *overloaded* respectivamente. Cada escalonador somente recebe mensagens dos seus descendentes, devido ao fluxo de informações ser *bottom-up*. Já o conjunto de dados *sons* de um escalonador mantém setados apenas os processadores filhos deste processador, ou seja, os processadores que importaram trabalho dele e ainda continuam neste trabalho.

Cada escalonador ainda deve manter a informação de quem é seu processador pai, ou seja, o trabalhador de quem importou trabalho.

Na próxima seção serão descritos os algoritmos utilizados para o gerenciamento dos conjuntos de dados, fluxo de informação *bottom-up* e casamento dos trabalhadores sobrecarregados com trabalhadores ociosos.

## 3.3 Distribuição de Trabalho: situação inicial

Em um primeiro momento apenas um processador estará executando uma primeira meta do programa Prolog. Este primeiro processador necessita possuir, além da máquina abstrata e do escalonador, um outro escalonador Mestre, responsável pelo início e término da computação. Isto é necessário para que, quando o processador raiz (o que iniciou a computação) ficar *idle*, possa situar sua máquina abstrata em um outro ponto da árvore de execução Prolog sem que deixe de ser o processador raiz da árvore e não disperdice processamento.

Este escalonador Mestre possuirá no início da execução todo seu conjunto de dados *idle* setado, menos ele próprio. Os outros conjuntos de dados não possuirão dados armazenados. Um exemplo da árvore de execução, processadores trabalhadores e conjuntos de dados pode ser observado na figura 1.

Ao iniciar a computação o trabalhador está em estado *silent*, mas logo poderá encontrar-se *overloaded*. Neste momento sua atitude é descrita pelo algoritmo da figura 3.

```
------------------------------------------------------------
Px = overloaded

  if bits setados em seu bitmap idle >= 1
     exporta trabalho para o primeiro trabalhador encontrado
  else
    envia mensagem para seu pai avisando que esta overloaded
------------------------------------------------------------
```

Figura 3: Algoritmo utilizado quando um processador torna-se *overloaded*

## 3.4 Busca de Trabalho: durante a execução

Quando um trabalhador estiver em estado *idle*, seja por falha ou sucesso na execução, este deve procurar por trabalhadores sobrecarregados que possam lhe enviar trabalho. Devemos estar atentos para o fato de que este escalonador visa manter a localidade entre os processadores. Deste modo será necessário ao trabalhador que ficar em estado *idle* esperar pelo menos um tempo igual ou maior que o tempo de movimento antes de mover-se para outro ponto da árvore, se não conseguir encontrar trabalho entre os seus descendentes. Este tempo de espera deriva-se do fato de que os trabalhadores próximos a ele têm a possibilidade de ficarem *overloaded* antes que este tempo acabe, evitando custos excessivos. Para tal o escalonador deve seguir o algoritmo da figura 4.

```
------------------------------------------------------------
Px = idle

  if bits setados em seu bitmap overloaded >= 1
    importa trabalho do primeiro trabalhador encontrado
  else
    espera tempo >= tempo de movimentacao
      importa trabalho do primeiro trabalhador encontrado
    else
      envia mensagem para seu pai avisando que esta idle
 ------------------------------------------------------------
```

Figura 4: Algoritmo utilizado quando um processador torna-se *idle*

## 3.5 Publicando Trabalho

A publicação de trabalho é feita sempre quando um trabalhador não consegue encontrar trabalhadores em estado *idle* dentre seus descendentes.

Sempre que um trabalhador torna-se *overloaded* deve tentar encontrar um trabalhador *idle* para compartilhar do seu trabalho, visando que nenhum processador fique ocioso por muito tempo e que a computação termine em um tempo menor. A idéia é que o trabalhador fique em estado *overloaded* o menor tempo possível e ainda possa compartilhar seu trabalho com um trabalhador que já tenha parte de suas pilhas copiadas, tornando a mensagens de envio de trabalho menor, mantendo a localidade da execução.

Se o trabalhador *overloaded* não conseguir um trabalhador descendente *idle* para exportar trabalho, este envia uma mensagem ao seu pai avisando que está com sobrecarga de

trabalho. O pai deste modo poderá fazer o "casamento" com um trabalhador *idle* se este existir dentre seus descendentes. Caso contrário reporta mensagens para seu pai (ver algoritmos das figuras 3 e 4). O mesmo ocorre para trabalhadores *idle* que não encontraram trabalhadores *overloaded* entre seus descendentes.

Na verdade não ocorre publicação de trabalho, mas sim publicação dos trabalhadores *idle* e *overloaded*

### 3.6 Comunicação com Outros Trabalhadores

Sempre que um escalonador não consegue encontrar trabalho para seu trabalhador que está em estado *idle*, ele reporta a mensagem para seu processador pai, para que este verifique entre seus descendentes (ver figura 5).

```
-------------------------------------------------------------
Px recebe mensagem de descendente idle Py

  if Px == overloaded
    exporta trabalho para o descendente idle Py
  else
    if bits setados em bitmap overloaded >= 1
      realiza casamento com o primeiro trabalhador encontrado
      atualiza bitmaps idle e overloaded
    else
      envia mensagem para seu pai avisando que Py = idle
-------------------------------------------------------------
```

Figura 5: Algoritmo utilizado quando um processador recebe mensagem de um *idle*

Somente se não for possível encontrar trabalho entre seus descendentes é que o escalonador reporta a mensagem para seu pai. Caso ele encontre um processador *overloaded*, deve realizar o "casamento" entre os dois processadores.

Um processador pode também receber mensagens de processadores em estado *overloaded*, que estão procurando processadores *idle* para exportar trabalho (ver algoritmo da figura 6).

```
-------------------------------------------------------------
Px recebe mensagem de descendente overloaded Py

  if Px == idle
    importa trabalho do descendente idle Py
  else
    if bits setados em bitmap idle >= 1
      realiza casamento com o primeiro trabalhador encontrado
      atualiza bitmaps idle e overloaded
    else
      envia mensagem para seu pai avisando que Py = overloaded
-------------------------------------------------------------
```

Figura 6: Algoritmo utilizado quando um processador recebe uma mensagem de descendentes em estado *overloaded*

## 4 Questões de Implementação

Como questões de implementação optamos por duas *threads* em cada processador, para implementar a máquina abstrata e o escalonador Penelope. Utilizamos *threads* pelo fato

de sistemas baseados em *threads* se mostrarem com melhor escalabilidade do que sistemas baseados em processos, de acordo com [11].

No processador que dá início à computação são utilizadas três *threads* para implementar a máquina abstrada, o escalonador Penelope e o escalonador Mestre. O que difere os algoritmos do escalonador Penelope e do Mestre, é que ao mestre apenas serão necessários os algoritmos de recebimento de mensagens devido ao fato de este não possuir uma máquina abstrata associada para computar trabalho Prolog.

Os conjuntos de dados *idle*, *overloaded* e *sons* utilizados por cada escalonador para manter informações sobre seus descendentes são implementados através de *bitmaps*. Cada escalonador deve ainda manter a informação sobre seu trabalhador exportador (seu pai na árvore de execução Prolog), que é mantido por um registrador.

# 5 Trabalhos relacionados

O escalonador Dharma melhora o *overhead* de escalonamento escalonando em um nível de abstração mais elevado que os outros tipos de escalonadores. Para isso necessita manter a ordem na árvore de busca, o que lhe causa degradação.

No Dharma, o trabalhador pode atravessar a árvore de execução se encontrar trabalho do outro lado. Já no Muse um trabalhador pode mover-se apenas no seu ramo. A vantagem do Muse é que não tem degradação do desempenho mantendo a topologia da árvore. Mas pode ser desvantajoso se um trabalhador está sobrecarregado enquanto outro ocioso não tem condições nem de saber seu estado nem chegar até ele.

Já o Penelope tenta minimizar o *overhead* de escalonamento mantendo conjuntos de dados em todos os escalonadores e cada escalonador mantém apenas informações sobre seus descendentes, estando apto a fazer o "casamento" entre dois deles, com poucas mensagens.

Todos os processadores do sistema podem atravessar a árvore de execução à medida que vão tornando-se *idle*. Posicionam-se em ramos sobrecarregados por intermédio de seus antecessores, eliminando a desvantagem do Muse (um trabalhador ficar sobrecarregado enquanto um *idle* está disponível).

# 6 Conclusões

Com o Penelope tentamos diminuir o *overhead* de escalonamento e a desvantagem do gargalo inerente a sistemas que utilizam escalonamento centralizado. Para diminuir o *overhead* optamos por decisões rápidas. Ainda diminuimos o número e tamanho das mensagens utilizando cópia incremental e uma proposta hierárquica com fluxo de mensagens *bottom-up*.

Como a implementação do Penelope não está completa, não podemos ter dados concretos. Mas avaliações analíticas nos fazem confiantes.

# Referências

[1] Jorge L. V. Barbosa. Granlog: Um modelo para análise automática de granulosidade na programação em lógica. Master's thesis, CPGCC-UFRGS, 1996.

[2] E. Morel; S. Kannat; A. Carissimi; J. Briat. Task scheduling for parallel execution of logic programs on distributed memory architectures. Technical report, LMC-IMAG, 1996. Available from: http://amon.imag.fr/Les.Groupes/PLoSys.

[3] M. Christaller. Athapascan-0a sur pvm3: Définition et mode d'emploi. Technical report, IMAG, Grenoble, 1995.

[4] Patrícia K. Vargas. Exploração de paralelismo ou em uma linguagem em lógica com restrições. Master's thesis, CPGCC-UFRGS, Porto Alegre, 1998.

[5] P. Codognet; D. Diaz. Wamcc: Compiling prolog to c. In *12th International Conference on Logic Programming*. Tokyo, 1995.

[6] Inês C. Dutra. Comunicação por e-mail, Março 1998.

[7] Débora N. Ferrari. Uma proposta de integração granlog-plosys. Technical report, UFRGS, Porto Alegre, 1998.

[8] J. Chassin; E. Morel; J. Briat; C. Geyer. Side-effects in plosys or-parallel prolog on distributed memory machines. Technical report, LMC-IMAG, 1996.

[9] Khayiri Ali; Roland Karlsson. Scheduling or-parallelism in muse. In *International Conference on Logic Programming*, pages 807–821, 1990.

[10] T. Kunz. The influence of different workload descriptions on a heuristic load balancing scheme. *IEEE Transactions on Software Engineering*, 17(7):725–730, July 1991.

[11] Luís M. B. Lopes; Fernando M. Silva. Thread- and process-based implementations of the psystem parallel programming environment. *Software - Practice and Experience*, 27(3):329–351, March 1997.

[12] R. Y. Sindaha. Branch-level scheduling in aurora: The dharma scheduler. In *International Symposium for Logic Programming*, pages 403–419, 1993.

[13] N. G. Shivaratri; M. Singhal. *Advanced Concepts in Operating Systems: Distributed, Database and Multiprocessor Operating Systems*. MIT Press, New York, 1994.

[14] P. Krueger; N. G. Shivaratri; M. Singhal. Load distributing for locally distributed systems. *IEEE Computer*, pages 33–34, 1992.

[15] S. Zhou. A trace-driven simulation study of dynamic load balancing. *IEEE Transactions on Software Engineering*, SE-14(9):1327–1341, September 1988.