Paralelismo em Prolog: Conceitos e Sistemas

Parallelism in Prolog: Concepts and Systems

João Fabrício Filho [1] [2]
Anderson Faustino da Silva [2]

Data de submissão: 27/10/2015, *Data de aceite: 25/04/2016*

Resumo: Paralelismo é uma área de estudo que cresce a cada dia, devido à redução do custo e popularização de máquinas com arquiteturas paralelas. Nesse contexto, as linguagens lógicas, sobretudo o PROLOG, apresenta uma alternativa viável e prática de paralelismo. A exploração desse paralelismo pode ser realizada de diferentes formas, e há inúmeros desafios nessa tarefa. Este tutorial visa apresentar os principais conceitos de paralelismo em PROLOG, os desafios enfrentados quando se busca a paralelização nessa linguagem e o estado-da-arte do desenvolvimento de sistemas que dão suporte à paralelização em linguagens lógicas. São apresentados sistemas baseados em paralelismo implícito implementados em diferentes plataformas. Ao final é realizada uma comparação entre os sistemas apresentados e os modelos neles implementados.

***P*alavras-chave:** prolog, paralelismo, paralelismo-E, paralelismo-OU, paralelismo implícito

Abstract: Parallelism is a study area that grows up each day, caused by the cost reduction and popularizing of machines with parallels architecture. In this context, the logical languages, especially PROLOG, show a feasible and practical alternative of parallelism. This exploitation can be accomplished of different ways, and are there several challenges on this task. This survey aims to show the main concepts of parallelism in PROLOG, the faced challenges when aims to do parallelism in this language and the state-of-art of systems development to give parallelism support in logical languages. Systems with basis on implicit parallelism developed in different platforms are presented. At the end, is accomplished a comparison between the presented systems and the implemented models by they.

***K*eywords:** prolog, parallelism, AND-Parallelism, OR-Parallelism, implicit parallelism

[1] Universidade Tecnológica Federal do Paraná - Câmpus Campo Mourão Via Rosalina Maria dos Santos, 1233 CEP 87301-899 - Campo Mourão/PR - Brasil. {joaof@utfpr.edu.br}
[2] Departamento de Informática, Universidade Estadual de Maringá, Avenida Colombo, 5790 - Bloco C56 CEP 87020-900 - Maringá/PR - Brasil. {pg48335@uem.br}{anderson@din.uem.br}

1 Introdução

Os limites físicos dos componentes eletrônicos levaram ao desenvolvimento de máquinas com mais de um núcleo de processamento, e à consequente enfatização do estudo de técnicas de programação paralela, ou paralelização. A paralelização mostrou-se uma alternativa completamente viável para a maximização do processamento.

O sistema declarativo das linguagens lógicas facilita a exploração do multiprocessamento da linguagem, já que a sincronização de dados é o maior desafio ao se paralelizar um código. O paralelismo é um meio de aumentar a eficiência da execução dos códigos de programação lógica, já que se aproveita da utilização de outros núcleos de processamento, existentes em máquinas modernas.

PROLOG é a mais difundida linguagem de programação lógica e tem como princípio a cláusula de Horn [24], sendo um programa composto por regras no formato H:- $\alpha_1, \alpha_2, ..., \alpha_n$, no qual H é a cabeça do programa lógico e α_i é a iésima parte que define o corpo da cláusula. A cabeça e os alvos podem possuir zero ou mais argumentos, que podem ser termos simples (inteiros, *strings*, átomos) ou complexos (listas e estruturas).

Assim como em qualquer outro tipo de linguagem, há dois meios de paralelização de linguagens lógicas, explícita e implicitamente. Na paralelização explícita são adicionados comandos na linguagem, com os quais o usuário pode escolher quais funcionalidades serão utilizadas. Na paralelização implícita, não há diferença para o usuário em relação à programação sequencial, o interpretador procura por maneiras de paralelizar o código automaticamente, sem afetar o significado semântico.

Este tutorial tem como objetivo fazer uma apresentação da área de paralelismo em PROLOG, mostrando os principais conceitos, uma breve descrição do estado-da-arte com os principais sistemas desenvolvidos e suas funcionalidades básicas.

As principais contribuições deste trabalho são listadas a seguir:

- Apresenta conceitos básicos da área de paralelismo em PROLOG, indicado para iniciar pesquisadores nessa área;
- Lista os principais sistemas que desenvolvem paralelismo implícito ou explícito em PROLOG e suas principais funcionalidades;
- Realiza uma comparação entre esses sistemas paralelos, destacando o tipo de paralelismo implementado em cada um;
- Aborda conceitos relativamente recentes para a área, como MapReduce e *multithreading*, além dos sistemas que implementam esses conceitos;

- Realiza a conexão dos conceitos apresentados com o desenvolvimento dos sistemas, facilitando ao pesquisador a visualização das funcionalidades dos sistemas.

O restante deste tutorial está organizado da seguinte forma: a Seção 2 apresenta os conceitos básicos de paralelismo em PROLOG para melhor entendimento da área, a Seção 3 lista e discorre sobre ferramentas que implementam paralelismo em PROLOG e suas principais funcionalidades, e por fim a Seção 4 apresenta as conclusões deste trabalho.

2 Conceitos Básicos

As duas principais estratégias de exploração de paralelismo em PROLOG foram propostas por Conery e Kibler [8], e são chamadas de paralelismo-E e paralelismo-OU. Cada uma das estratégias traz estruturas de análise diferentes e modelos diferentes de implementação, os quais são abordados nas próximas seções.

Conceitos como *overhead* e granularidade estão estritamente ligados, já que definindo bem a granularidade de um sistema diminui-se o *ovehead* e ganha-se desempenho. Recentemente, abordagens com alto volume de dados são exigidas para processamento paralelo, e o modelo MapReduce [16] pode suprir essa exigência. Por último, as máquinas multi-núcleo trouxeram a necessidade de suporte à paralelismo por meio do *multithreading*, que há plataformas que adotam por padrão na linguagem.

2.1 Paralelismo-E

O paralelismo-E [8] pode ocorrer quando há alguma separação na descrição de uma regra. Esse tipo de paralelismo permite a exploração da programação concorrente trazendo uma solução final para o alvo original.

Se houver uma regra definida por:

$a(X, Y)$:- $b(Y)$, $c(X)$, $d(Y, X)$.

o interpretador pode considerar o paralelismo-E no momento de executar uma consulta que leve à $a(X, Y)$ conforme o fluxo representado na Figura 1.

É comum na literatura a divisão do paralelismo-E em dois tipos: Paralelismo-E Dependente e Paralelismo-E Independente.

Ocorre o Paralelismo-E Dependente quando um ou mais alvos executados em paralelo compartilham dados entre si, esses alvos são chamados de alvos dependentes. A execução dos alvos dependentes ocorre com um dado de ligação comum entre os dois alvos, e há cooperação entre os processos diferentes. Na regra do exemplo citado anteriormente, há o

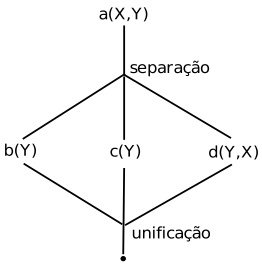

Figura 1. Fluxo de execução para a cláusula $a(X, Y)$ explorando o paralelismo-E. A separação define-se como a criação e atribuição das tarefas para diferentes fluxos de execução e a unificação como a junção dos fluxos de execução.

Paralelismo-E Dependente na execução de $b(Y)$ e $d(Y, X)$ por causa do dado Y compartilhado entre essas cláusulas.

O Paralelismo-E Independente ocorre quando os alvos paralelizados na execução não têm relações entre si, sem compartilhamento de dados. A execução desse tipo de paralelismo pode ocorrer independentemente do início ou final da execução de qualquer uma das outras regras, a junção para a solução final ocorrerá quando todos os alvos forem executados. O Paralelismo-E Independente ocorre em diversas aplicações, principalmente em problemas do tipo dividir-para-conquistar, nos quais são realizadas chamadas recursivas independentes que podem ser executadas em paralelo. Na regra introduzida no exemplo anterior, ocorre o Paralelismo-E Independente nas execuções simultâneas de $b(Y)$ e $c(X)$, que não possuem dados compartilhados entre si.

Existem alguns modelos de paralelismo na literatura que exploram o Paralelismo-E Independente. O modelo abstrato de Conery [9] foi o primeiro a ser proposto, em 1983, e explorava o Paralelismo-E Independente por meio de um grafo de fluxo de dados, no qual poderia existir uma relação produtor/consumidor entre literais se houvesse um dado não instanciado em comum. Um algoritmo de ordenamento em tempo de execução determina qual literal deverá ser executado antes e quais podem ser executados em paralelo. Há mais dois motores de execução nesse modelo além do algoritmo de ordenamento, a *execução para frente* manipula mensagens e atribui relação de produtor à literais que podem ser executados como resultado, e a *execução para trás* verifica falhas e tarefas que precisam ser refeitas. Apesar da verificação de dependência ocorrer de forma simples, há um substancial *overhead* em tempo de execução.

O modelo APEX (*And Parallel EXecution*) [28] consiste de dois algoritmos: *execução para frente* e *execução para trás*. A execução para frente pode ser vista como um esquema conceitual de passagem de *token*. A cada dado na execução do programa, um *token* é criado

e, se um literal produz um dado, ele possui o *token* desse dado. Um literal pode ser executado se ele receber os *tokens* de todos os seus dados não instanciadas do ambiente de compartilhamento atual. O algoritmo de execução para trás realiza verificação de falhas e realiza *backtracking* para refazer tarefas.

No modelo RAP (*Restricted And Parallelism* - Paralelismo-E Restrito) [18] um algoritmo de tipagem monitora os termos e predicados durante o tempo de execução do código, para manter uma indicação dos tipos de dados e construir um grafo de execução de expressões, que determina a independência entre termos e cria procedimentos de paralelização para termos independentes.

2.2 Paralelismo-OU

A máquina PROLOG possui rotinas que analisam o código antes de executá-lo. Rotinas não determinísticas são aquelas que possuem mais de uma cláusula que pode ser executada por certa sequência de argumentos. Um ponto de escolha corresponde a uma marca da busca, necessária para a restauração do estado anterior da computação. No momento em que a busca encontra uma falha e é necessário retornar, o controle passa então para o último ponto de escolha que continuará a busca pela cláusula correta. Esse retorno da busca é chamado *backtracking*, e é isso que o paralelismo-OU [8] explora na execução do código PROLOG.

Basicamente, o paralelismo-OU é possível quando existe mais de um corpo para descrever a mesma regra. O que ocorre não é paralelismo da execução, mas uma paralelização da busca pela solução por diferentes alternativas. O exemplo abaixo descreve a definição da cláusula a:

$a(X, Y)$:- $b(Y)$.
$a(X, Y)$:- $c(X)$.
$a(X, Y)$:- $d(Y, X)$.

Nesse exemplo, ao se fazer uma consulta que leve à '$a(X, Y)$', a busca pode ser realizada em paralelo efetuando as consultas por $b(Y)$, $c(X)$ e $d(Y, X)$. Por sua vez, essas consultas executam em paralelo as consultas pelas suas regras. A estrutura de uma árvore de busca é construída ao realizar o paralelismo-OU, a árvore construída no exemplo citado é representada na Figura 2. O nó raiz da árvore contém a consulta que deu origem à busca e a lista de alvos associados. Em cada um dos outros nós estão os resultados da união do primeiro alvo do nó pai, com a cabeça do programa lógico.

Diferentemente do paralelismo-E, não há junção após encontrar as definições que satisfazem a consulta realizada, o que há é a necessidade de encerrar as outras ramificações da busca, que devem ser finalizadas para evitar processamento desnecessário, já que a busca terminou.

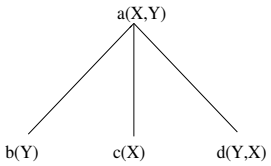

Figura 2. Árvore de busca paralela OU estruturada para a consulta de 'a.'.

Embora paralelismos E e OU sejam duas formas diferentes de explorar paralelismo em linguagens lógicas, com a exploração de ambos de forma adequada uma plataforma consegue melhor desempenho do que com a exploração de cada tipo isoladamente [29].

2.3 Efeitos colaterais

Os programas lógicos utilizam predicados chamados extra-lógicos, que não são parte da programação lógica, mas são necessários para o desenvolvimento na linguagem. São chamados de efeitos colaterais os resultados e a execução desses predicados na programação lógica. São exemplos de predicados extra-lógicos: `read`, `write`, `assert`, `retract` e `cut`.

O suporte a predicados extra-lógicos é uma das barreiras para dar suporte paralelo à linguagem PROLOG por completo. Tal suporte pode trazer vantagens, como a execução de programas existentes sem a necessidade de alteração no código. E há melhoria de desempenho no paralelismo de tais programas, que inicialmente podem ter sido desenvolvidos para ambientes sequenciais.

Dentre os predicados extra-lógicos, vale destacar a função do *corte* (ou `cut`, representado pelo símbolo !), que faz com que o programa descarte alternativas desnecessárias para uma regra. No paralelismo-OU, a execução paralela das alternativas pode fazer com que haja execução desnecessária de alternativas que já poderiam ser descartadas, o que é definido como trabalho especulativo [20].

2.4 Granularidade

A granularidade diz respeito à partilha do trabalho a ser realizado pelos processos do programa [21]. A criação e o escalonamento desses processos exige um *overhead*, e a redução desse *overhead* é o desafio ao escolher a melhor técnica de granularidade.

A granularidade é um dos maiores problemas no paralelismo implícito. Isso porque paralelizar não implica diretamente em aumento de desempenho. Há uma relação entre

overhead e fluxos de execução criados, pois a cada criação de um fluxo diferente um custo é acrescido na computação, assim deve haver uma relação em que a paralelização da tarefa compense o custo gerado pela sua criação. Portanto, deve-se encontrar a melhor relação entre o número de fluxos e custo da execução do processo para maximizar a eficiência do código.

No início da década de 1990, o trabalho de Tick *et al.* [37] atribuiu pesos quantificadores para definir a granularidade dos processos por meio da complexidade de tempo. A atribuição ocorre em tempo de compilação e por meio da conversão do código-fonte em um grafo cíclico de chamada, no qual cada nó corresponde a um procedimento e cada aresta a uma chamada de procedimento em potencial. O cálculo do peso de cada procedimento é um procedimento com recursão em calda.

O trabalho de Debray *et al.* [17] utilizou técnicas para estimar limites superiores da complexidade da computação para determinar a granularidade da paralelização. As técnicas utilizadas se basearam no tamanho dos dados e na dependência entre eles. Nesse trabalho foram calculados tamanhos relativos de variáveis compartilhadas por meio de um grafo de dependência de dados, e a informação dos tamanhos para determinar equações de representação do custo computacional dos predicados. As equações são avaliadas assim que o tamanho dos dados é conhecido, em tempo de execução. Resultados experimentais demonstram que essa técnica pode obter ganho de desempenho significativo, porém, em alguns casos pode gerar *overhead* pelas demasiadas informações processadas em tempo de execução.

Resultados mostraram que, para o paralelismo-OU, as decisões de granularidade não dependiam somente de análises de complexidade de tempo. Em 1997, o trabalho de King *et al.* [27] propôs uma técnica de análise de complexidade que englobava modelos de predicados recursivos, já que as análises anteriores não eram consideradas satisfatórias para esses tipos de problemas. Nessa técnica, considerou-se a computação em todos os tamanhos e complexidade de dados.

O trabalho de Xirogiannis [44] propôs uma técnica de análise de granularidade com foco em plataformas distribuídas com paralelismo implícito. A maior parte da análise é realizada em tempo de compilação, com alguns processamentos em tempo de execução, com aplicação experimental em paralelismo-E e paralelismo-OU.

Em 1998, o trabalho de Shen [31] tomou como base resultados que mostraram que a estimativa de complexidade não é um métrica ideal no que diz respeito à granularidade para processos criados dinamicamente, e propôs uma métrica que considera os principais *overheads* incorridos na execução do código. Nessa métrica, chamada *distance*, considera-se também os *overheads* indiretos, que são causados por tarefas paralelas que podem ser criadas dinamicamente por outras tarefas paralelas. A granularidade definida pela métrica *distance* foi experimentada em métodos executados em tempo de compilação e em tempo de execução do código PROLOG e mostrou bons resultados, especialmente no método em

tempo de execução.

2.5 Escalonamento

O escalonamento diz respeito à execução e ao gerenciamento da carga de trabalho dos processos paralelos [35]. Para se ter um ganho eficiente de desempenho, a ordem das execuções e a distribuição correta das cargas de trabalho são itens importantes.

O escalonamento pode ser dirigido por motor ou dirigido por tarefas. É dirigido por motor quando o motor de execução examina as tarefas, e dirigido por tarefas quando as tarefas examinam motores ociosos (ou menos carregados) para executá-las. Um dos primeiros escalonadores dirigidos por motor foi desenvolvido no sistema Andorra-I [14], no qual um escalonador de primeiro nível gerencia dois escalonadores de segundo nível, um responsável pelo paralelismo-E e outro responsável pelo paralelismo-OU. Os escalonadores particionam os motores em times flexíveis que distribuem o trabalho aos processos paralelos. O escalonamento dirigido por motor é utilizado no controle da execução de processos do paralelismo-OU especulativo e se mostrou eficiente nos sistemas Muse [1] e Aurora [29].

Três tipos de escalonadores implementados na literatura merecem destaque por apresentarem bons resultados: os escalonadores *Manchester*, *Bristol* e *Darma*.

O *escalonador Manchester* [6] tenta realizar uma combinação entre trabalho disponível e processos, assim que possível. Este escalonador não foi projetado para realizar trabalho especulativo eficientemente, assim não é bem encaixado no paralelismo-OU e em casos ocasionados por efeitos colaterais.

O *escalonador Bristol* [5] tenta minimizar o *overhead de sincronização* estendendo a região global do escalonador, criando várias instâncias para cada nó e é flexível para adaptar algoritmos de compartilhamento de tarefas. O escalonador foi baseado no ambiente de cópia implementado no sistema Muse [1], e foi aprimorado para o escalonamento eficiente de trabalho especulativo no paralelismo-OU, direcionado para o sistema Aurora [29].

O *Escalonador Darma* [32] também possui foco para o paralelismo especulativo, e adequado ao sistema Aurora, tentando diminuir o trabalho perdido que o sistema possuía, causado pela não poda de buscas eliminadas, como quando se tem o operador corte.

2.6 MapReduce

MapReduce [16] é um paradigma de programação projetado para processar grande volume de dados, com uma função *map* para proceder com um par chave/valor e gerar uma chave intermediária e a função *reduce* para fundir as chaves intermediárias com seus respectivos valores.

Proposto inicialmente para ser aplicado em linguagens funcionais, o modelo também pode ser adaptado à linguagens lógicas. O trabalho de Srinivasan *et al.* [33] teve como foco experimentar uma abordagem MapReduce em linguagens lógicas com paralelização de dados e tarefas e verificar sua viabilidade. Os resultados alcançados por esse trabalho evidenciaram que a abordagem é factível quando há uma quantidade considerável de dados para o problema e quando há possibilidade de diminuição de *overheads* dinâmicos.

O trabalho de Côrte-Real *et al.* [10] implementou o modelo MapReduce para PROLOG em alto nível, com o objetivo de obter uma ferramenta que distribuísse transparentemente os trabalhos não somente em grande volume de dados, mas em tarefas de processamento de dados paralelos.

Baseado na ideia do paradigma mestre-escravo, a tentativa de obter desempenho consiste em reduzir o tempo de processamento de dados distribuindo pelos recursos computacionais disponíveis, e são dispostos três níveis hierárquicos na arquitetura do trabalho de Côrte-Real *et al.* [10]: mestres globais, mestres locais e escravos. Os mestres globais controlam o fluxo de dados e o escalonamento de alto nível, enquanto os mestres locais distribuem as tarefas e controlam os acessos dos escravos, que fazem a computação propriamente dita.

Além da arquitetura, a execução do modelo MapReduce de Côrte-Real *et al.* [10] disponibiliza um modelo flexível que pode ser executado em máquinas distribuídas para grande volume de dados.

Na implementação MapReduce são adicionados predicados à linguagem PROLOG padrão, que possibilitam a utilização do paradigma pelos usuários, por meio da plataforma YAP. Foram obtidos resultados de duas implementações, memória compartilhada e passagem de mensagem, e no geral, foram obtidos *speedups* lineares.

Um modelo híbrido de MapReduce [11] permitiu a aplicação do modelo em memórias compartilhadas e distribuídas com máquinas com diferentes configurações.

2.7 Multithreading

Multithreading consiste na disponibilização de bibliotecas ou comandos-padrão na linguagem para criação e controle de fluxos paralelos explicitamente. Várias plataformas PROLOG [23, 7, 13, 26, 34, 43] disponibilizam essa funcionalidade devido tanto à popularização das máquinas paralelas quanto à disponibilização dessa funcionalidade em outras linguagens, como C e Java.

Ao contrário dos paralelismos E e OU, em *multithreading* os códigos sequenciais necessitam alterações para utilizar o paralelismo, uma vez que a funcionalidade adiciona primitivas na linguagem padrão. A mudança pode não trazer portabilidade, mas pode fazer com que os processos sejam mapeados diretamente para diferentes processadores, além de

poderem ser controlados e reutilizados.

O trabalho de Tarau [36] teve foco na construção de uma API, dentro da plataforma Bin PROLOG, na qual fluxos diferentes são criados para cada processador da máquina, otimizando a utilização do *hardware*, e desacoplando as primitivas de alto nível de *multithreading* das operações realizadas no motor de execução para adaptar os processos e fluxos ao *hardware* em questão.

O conceito de *multithreading* também é utilizado no sistema ThOr [12], em que tal conceito é aplicado em um modelo de paralelismo implícito, que é abordado com maiores detalhes na Seção 3.10.

3 Sistemas Paralelos

Vários sistemas com implementação de paralelismo exploram as linguagens lógicas. Até o final da década de 1980, a exploração do paralelismo explícito nas chamadas *linguagens de mudança persistida* recebia uma atenção considerável. Embora fossem linguagens baseadas no PROLOG, o alteravam de forma abrupta para o usuário.

Este tutorial não foca nesse tipo de linguagem, mas em ferramentas que exploram o paralelismo em PROLOG alterando-o o mínimo possível. Esta seção lista alguns sistemas e apresenta um breve resumo sobre suas funcionalidades.

3.1 PEPSys

O *Parallel ECRC PROLOG System* (PEPSys) foi apresentado em 1988 por Baron *et al.* [4], com o objetivo de avaliar novas soluções para os problemas de paralelização em programação lógica. Apesar de implementar facilidades para os paralelismos E e OU, é um sistema baseado em paralelismo explícito, no qual o controle é do usuário e são adicionados comandos à linguagem PROLOG.

Para execução do paralelismo-E, as cláusulas da regra a serem paralelizadas são separadas explicitamente por um terminal ("#"). No paralelismo-OU é necessária uma declaração de propriedades do predicado para facilitar a busca pela definição correta.

Em teoria, o paralelismo explícito dessas técnicas reduz o *overhead*, pois ninguém melhor do que o desenvolvedor para conhecer o código, escolher e saber o que é necessário paralelizar. O problema da definição da granularidade do programa assim inexiste para o sistema, ficando a critério do desenvolvedor a quantidade de computação escolhida para cada processo.

Por adicionar comandos na linguagem, programas escritos em PROLOG nativo pre-

cisam ser reescritos para utilizarem o PEPSys, o que é uma desvantagem quando se procura portabilidade e dinamicidade, pois um código escrito para uma máquina com n núcleos pode não ser tão eficiente para uma máquina com $2n$ núcleos. Além disso, existe a introdução de complexidade para o desenvolvimento de novos programas. A inexperiência do usuário também pode trazer problemas na execução do sistema, pois a granularidade do código fica a seu critério.

3.2 Aurora

Aurora [29] implementa o paralelismo-OU na linguagem PROLOG com foco em arquiteturas de memória compartilhada. O objetivo do sistema era explorar implicitamente o paralelismo-OU, para causar o mínimo de impacto possível ao usuário ao utilizar o sistema.

O sistema foi desenvolvido com base na ideia de que a exploração transparente do paralelismo-OU é mais produtiva e mais fácil de implementar do que do paralelismo-E, explorado por sistemas que implementavam o paralelismo explícito estendendo o PROLOG.

O compartilhamento de dados é feito por meio de arranjos de ligação, uma estrutura privada de cada processo, na qual os processos armazenam ligações condicionais. Para facilitar e diminuir o tempo de acesso, a estrutura é construída como um vetor compartilhado, tendo como índice um número, que representa a quantidade de variáveis criadas no ramo atual.

A exploração somente do paralelismo-OU simplifica o problema da granularidade do sistema, que pode utilizar granularidade alta, que se mostrou mais efetiva para esse tipo de paralelismo.

O compartilhamento dos dados tomou como base o modelo SRI, proposto por Warren [41], e foi construído adaptando-se à plataforma SICStus PROLOG [7].

Aurora ainda dá suporte a anotações no código-fonte em que o usuário pode optar por não paralelizar certa porção do código, fazendo com que o sistema não atue nessa porção.

O sistema conseguiu bons resultados, que evidenciaram uma redução considerável do *overhead* dos processos comparados aos sistemas PROLOG da época. Porém, a exploração somente do paralelismo-OU não traz o melhor desempenho possível que se possa ter com paralelização de PROLOG. Para obter o melhor desempenho deve-se implementar todos os tipos de paralelismo reduzindo ao máximo o *overhead* criado.

3.3 Muse

Multi Sequential PROLOG Engines (Muse) é uma abordagem apresentada em [1] que inicialmente explorou o paralelismo-OU em uma variante da linguagem PROLOG, em arqui-

teturas de memória compartilhada. Essa variante consiste na exploração do paralelismo-OU em uma extensão do SICStus PROLOG [7].

A implementação de Muse foi baseada no desenvolvimento de um sistema para paralelismo-OU chamado BC-Machine [2], focado em arquitetura de memória distribuída. A grande contribuição do sistema Muse foi o desenvolvimento do ambiente de cópia incremental, desenvolvido para minimizar o *overhead de sincronização*, fazendo com que cada instância dentro da mesma ramificação de busca possua uma parte consistente de dados e outra com modificações locais, e a sincronização dos dados é feita copiando apenas a diferença entre o local e global. Tal ambiente foi base para o escalonador Bristol [5] e para outro sistema que explora o paralelismo-OU em PROLOG, o YapOr [30].

O sistema permite ainda a utilização de anotações pelo usuário para forçar paralelismo. Tais anotações podem auxiliar no tratamento que o sistema dá ao código, porém podem deixá-lo estático e não-portável, uma vez que as anotações são específicas deste sistema.

No caso de outras funcionalidades, como corte e efeitos colaterais, o tratamento que o sistema Muse dá é desabilitar o paralelismo em seus escopos, não explorando a especulação de busca.

Muse utiliza uma abordagem em que não é necessário um coletor de lixo global, porém em casos como o da cópia incremental é preciso um processo local de coletor de lixo assíncrono.

3.4 Andorra-I

Andorra-I é um sistema proposto por Costa *et al.* [14] para explorar o paralelismo-E dependente e o paralelismo-OU, ambos de forma implícita, dando suporte à linguagem PROLOG padrão e aos programas escritos em linguagens de mudança persistida.

Este sistema implementa o modelo Andorra básico, um modelo de execução para programas lógicos que utiliza a ideia de que as cláusulas devem ser executadas assim que possível. Há dois componentes principais: o motor de execução e o preprocessador determinístico. Enquanto o motor de execução interpreta os comandos na linguagem-fonte, o preprocessador verifica possíveis ramificações para uma busca paralela.

A computação determinística pode ter vários predicados executados em paralelo, compondo o paralelismo-E independente. Quando não há alvo determinado (computação não-determinística) um alvo é escolhido e um ponto de escolha é criado para o alvo, e os vários pontos de escolha podem ser executados em paralelo, compondo o paralelismo-OU.

Quanto ao escalonamento implementado, o escalonador Bristol [5], que foi desenvolvido para o sistema Aurora, foi adaptado ao sistema Andorra-I.

3.5 &-Prolog

O sistema &-Prolog foi apresentado em 1991 por Hermenegildo e Greene [22] com o objetivo de prover um ambiente no qual o usuário pode explorar a execução paralela de programas lógicos automaticamente, ou adicionando anotações de divisão de tarefas entre fluxos.

A implementação se dá por meio do paralelismo implícito, com código PROLOG nativo, e explícito, no qual a adição de terminais ao código-fonte permite ao usuário explorar o paralelismo-E.

A análise do código em tempo de compilação se dá pelo Grafo de Expressão Condicional (GEC), que possui a forma ($i\_cond \implies goal_1 \ \& \ goal_2 \ \& \ ... \ \& \ goal_n$), na qual i_cond representa uma condição que, se satisfeita, permite a independência de $goal_i$, para todo $i < n$.

O sistema &-Prolog funciona basicamente com quatro módulos: anotador, analisador de efeitos colaterais, analisador global e compilador de baixo nível.

O anotador realiza a análise de dependência no código de entrada, verificando quais tarefas podem ser executadas em paralelo. O analisador de efeitos colaterais verifica se o predicado atual contém ou chama efeitos colaterais. O analisado global interpreta o código no domínio abstrato do sistema e infere sobre possíveis substituições ao longo do programa. Por fim, o compilador de baixo nível funciona como uma extensão do SICStus PROLOG, produzindo código intermediário do programa de entrada.

3.6 Opera

O projeto Opera, apresentado no trabalho de Werner *et al.* [42], explora o paralelismo-E restrito e o paralelismo-OU multi-sequencial de forma implícita, apesar de disponibilizar minimamente primitivas opcionais ao usuário para paralelização.

O paralelismo-OU multi-sequencial é implementado sem compartilhamento de variáveis, caso duas ou mais operações precisem do mesmo dado, o dado é copiado para uma nova instância na memória. O autor explorou tal alternativa visando arquiteturas de memória compartilhada, nas quais o acesso à mesma região de memória de diferentes processos pode ser computacionalmente custoso. O paralelismo-E restrito é empregado por meio de análises de estruturas em tempo de compilação.

O sistema constrói uma estrutura de análise de granularidade, na qual busca a máxima granularidade, utilizando uma heurística de que, quanto mais alto nível for o trabalho, maior sua granularidade. O *overhead* de criação de um processo aumenta quando considera-se que os dados são copiados a cada criação.

O foco na arquitetura de memória distribuída com paralelismos E e OU traz uma vantagem ao sistema Opera. A opção de primitivas para paralelismo explícito é um atrativo para os usuários. Porém, tais primitivas podem deixar o código estático e não portável.

3.7 YapOr

YapOr é um sistema originalmente publicado por Rocha *et al.* [30] que explora o paralelismo-OU em uma extensão do ambiente YAP PROLOG [13].

O ambiente YAP PROLOG é baseado na WAM [40], e portanto possui várias otimizações que não são presentes na versão original do SICStus PROLOG, ambiente dos sistemas Muse e Aurora. Assim, um sistema com base na plataforma YAP tem mais condições de ganho de desempenho do que em sistemas com base em plataformas que não possuam essas otimizações.

A implementação do paralelismo-OU no YapOr é baseada no ambiente de cópia utilizado no sistema Muse, que, segundo os autores, é mais simples, elegante e adequado à complexidade da plataforma YAP. Porém, a organização da memória e a implementação de travas são as diferenças entre o desenvolvimento de YapOr e Muse.

A comparação com Muse é inevitável, visto que o desenvolvimento de YapOr foi baseado em Muse. No trabalho em que o YapOr foi apresentado [30], há uma seção em que foram comparados resultados de execução de um mesmo conjunto de programas nos dois ambientes. Apesar de na época o ambiente Muse estar mais maduro, YapOr obteve melhor desempenho.

A implementação do corte no YapOr é realizada por meio da execução especulativa, o envio de sinais pelas raízes de busca para os nós finaliza a busca. A busca especulativa se inicia ao encontrar as alternativas, mas ao receber o sinal os nós são podados. Tal implementação minimiza o *overhead* na busca, mas por outro lado, ainda inicia busca desnecessárias.

3.8 PALS

PALS [39] é um sistema publicado em 2001 que explora o paralelismo-OU em arquiteturas de memória distribuída, construído com base no ambiente de cópia presente em sistemas como Muse [1] e YapOr [30].

Além do ambiente de cópia, a cópia incremental também presente nos sistemas citados foi implementada no sistema PALS, se mostrando eficiente para exploração do paralelismo-OU. Resultados da execução do sistema, que tem como base a plataforma ALS PROLOG [3], mostraram que o ambiente de cópia é adequado à arquitetura de memória distribuída, reduzindo *overheads* significativamente. Tal redução pode ser causada pela minimização das

atualizações de memórias, que possuem custo significativo nessa arquitetura.

No que diz respeito ao escalonamento, o sistema utiliza uma técnica chamada de divisão de pilha, tida como uma evolução das técnicas utilizadas no ambiente de cópia de Muse e YapOr. Na divisão de pilha, a distribuição de tarefas se realiza no momento da separação, evitando estruturas de dados centralizadas para escalonar.

3.9 PAN

PAN [45] é um ambiente distribuído de propósito geral e portável para executar programas lógicos em paralelo, explorando as paralelizações dos tipos dividir-e-conquistar e especulativa.

Originalmente publicado em 2002, o sistema combina SICStus PROLOG [7] e PVM [15] para gerar uma arquitetura distribuída. A exploração do paralelismo se dá por meio de análises em tempo de compilação, sem interferência do usuário (paralelismo implícito). De um modo grosseiro, o sistema PAN cria processos estaticamente, que se comunicam por meio de mensagens síncronas e assíncronas, utilizando primitivas adicionadas à linguagem PROLOG para controlar a passagem de mensagem e habilitar a sincronização.

O sistema PAN dá suporte ao paralelismo-OU e ao paralelismo-E independente, e, apesar de a comunicação da arquitetura exigir adição de primitivas, o paralelismo se dá de forma automática, com a linguagem PROLOG padrão.

3.10 ThOr

Com o crescimento e popularização das máquinas multinúcleos, houve necessidade de adequar o paralelismo implícito à melhor distribuição dos fluxos entre os núcleos disponíveis.

Threads and Or-Parallelism Unified (ThOr) [12] é uma extensão do sistema YapOr, com o objetivo de integrar o paralelismo-OU com a biblioteca de *multithreading* da plataforma YAP. A biblioteca de *multithreading* do YAP consiste basicamente em uma interface alto nível para POSIX Threads [25].

A exploração de *multithreading* junto com o paralelismo-OU permite maior escalabilidade do sistema, com uma distribuição mais uniforme das tarefas pelos núcleos de processadores disponíveis na máquina.

Melhorias foram realizadas no sistema recentemente [19] e, no geral, conseguiram resultados mais eficientes do que YapOr.

3.11 Uma comparação

Os sistemas apresentados exploram os paralelismos E e OU de forma implícita em sua maioria, isso permite que o usuário não tenha preocupação quanto às oportunidades de paralelismo do programa, o sistema de execução paraleliza seu programa automaticamente.

Alguns sistemas [1, 29, 42] disponibilizam alternativas de anotações adicionais à linguagem padrão, apesar de denominados sistemas de paralelismo implícito. Tais anotações podem disponibilizar mais funcionalidades ao usuário, porém o código pode ficar estático e não-portável.

O paralelismo-E independente e modelos que exploram esse paralelismo são os mais implementados devido à facilidade que proporcionam na interpretação do código.

A arquitetura de memória compartilhada é a mais explorada na área de paralelismo em PROLOG, porém há sistemas que disponibilizam essa alternativa nas arquiteturas de memória distribuída.

A Tabela 1 apresenta uma comparação dos sistemas apresentados nesta seção. Na coluna 'arquitetura' entede-se por 'MD' arquiteturas de memória distribuída e 'MC' arquiteturas de memória compartilhada.

Tabela 1. Síntese dos sistemas apresentados que implementam paralelismo em PROLOG

sistema	ano	plataforma	paralelismo	implementação	arquitetura
PEPSys	1988	CProlog	explícito	E e OU	MD
Aurora	1990	SICStus	implícito*	OU	MC
Muse	1990	SICStus	implícito*	OU	MC
Andorra-I	1991	Andorra	ambos	E e OU	MC
&-Prolog	1991	Quintus	ambos	RAP	MC
Opera	1992	MAP	implícito*	RAP e OU	MC
YapOr	1999	YAP	ambos	OU	MC
PALS	2001	ALS	implícito	OU	MD
PAN	2002	SICStus/PVM	implícito	E e OU	MD
ThOr	2010	YAP	ambos	OU	MC

* com suporte a anotações

O paralelismo-OU é mais explorado que o paralelismo-E na literatura, pois os resultados apresentados apresentam melhor desempenho em relação a seus custos. No contexto de

paralelismo-OU em arquiteturas de memória compartilhada, YapOr e Muse apresentaram os melhores resultados [38].

A maximização da exploração do paralelismo ocorre somente quando explorados ambos os tipos de paralelismo, mas apesar disso, diversos sistemas [30, 1, 29, 39] apresentam bons resultados explorando apenas um tipo de paralelismo.

O ambiente de cópia implementado inicialmente no sistema Muse, adaptado aos sistemas YapOr e PALS, foi uma grande contribuição e permitiu a redução principalmente do *overhead* de sincronização entre processos.

Atualmente, com a popularização de máquinas com arquitetura de vários núcleos, o *multithreading* tornou-se necessário em plataformas que necessitam expandir seus ganhos de desempenho nessas máquinas, havendo sistemas que implementaram esse tipo de solução [12].

4 Conclusão

O paralelismo está presente na computação como um paradigma de implementação, e é uma área com frutíferos estudos. Nesse contexto, as linguagens lógicas têm mostrado um gancho potencial para a exploração de paralelismo.

A exploração do paralelismo implícito em PROLOG mostra uma maneira de fazer paralelismo sem alterações para o usuário, além de dar suporte a todas as aplicações pré-existentes na linguagem.

Vários conceitos foram abordados neste tutorial, que exemplificam a complexidade da área de estudo e dos problemas enfrentados. A minimização do *overhead* ao se paralelizar as aplicações PROLOG pode ser realizada com diferentes estratégias, dependendo do tipo de aplicação e de paralelismo desenvolvido.

Uma análise dos sistemas atuais que exploram o paralelismo indica que há sistemas consolidados que exploram um tipo de paralelismo (caso de YapOr, Muse e Aurora, que exploram somente o paralelismo-OU) ou em certa arquitetura (PAN e Opera, que executam em arquiteturas de memória distribuída e compartilhada, respectivamente).

Com a popularização de máquinas com vários núcleos computacionais, PROLOG é uma alternativa elegante para a exploração de tal paralelismo.

Contribuição dos autores:

Os dois autores contribuíram igualmente para a elaboração de todas as seções.

Referências

[1] ALI, K., AND KARLSSON, R. The muse approach to or-parallel prolog. *International Journal of Parallel Programming 19*, 2 (1990), 129–162.

[2] ALI, K. M. Or parallel execution of prolog on bc-machine. *SICS Research Report* (1988).

[3] APPLIED LOGIC SYSTEMS INC. Als prolog manual. *Cambridge, MA* (1997).

[4] BARON, U., DE KERGOMMEAUX, J. C., HAILPERIN, M., RATCLIFFE, M., ROBERT, P., SYRE, J.-C., AND WESTPHAL, H. The parallel ecrc prolog system pepsys: An overview and evaluation results. In *FGCS* (1988), pp. 841–850.

[5] BEAUMONT, A., RAMAN, S. M., SZEREDI, P., AND WARREN, D. H. D. Flexible scheduling of or-parallelism is aurora: The bristol scheduler. In *Proceedings on Parallel Architectures and Languages Europe : Volume II: Parallel Languages: Volume II: Parallel Languages* (New York, NY, USA, 1991), PARLE '91, Springer-Verlag New York, Inc., pp. 403–420.

[6] CALDERWOOD, A., AND SZEREDI, P. Scheduling Or-parallelism in Aurora: The Manchester Scheduler. In *International Conference on Logic Programming/Joint International Conference and Symposium on Logic Programming* (1989), pp. 419–435.

[7] CARLSSON, M., AND MILDNER, P. Sicstus prologthe first 25 years. *Theory and Practice of Logic Programming 12* (1 2012), 35–66.

[8] CONERY, J. S., AND KIBLER, D. F. Parallel interpretation of logic programs. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture* (New York, NY, USA, 1981), FPCA '81, ACM, pp. 163–170.

[9] CONERY, J. S., AND KIBLER, D. F. And parallelism in logic programs. In *In Proceedings of the International Joint Conference on AI* (Los Altos, CA, 1983), A. Bundy, Ed., pp. 539–543.

[10] CÔRTE-REAL, J., DUTRA, I., AND ROCHA, R. Prolog programming with a mapreduce parallel construct. In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming* (New York, NY, USA, 2013), PPDP '13, ACM, pp. 285–296.

[11] CÔRTE-REAL, J., DUTRA, I., AND ROCHA, R. A hybrid mapreduce model for prolog. In *Integrated Circuits (ISIC), 2014 14th International Symposium on* (Dec 2014), pp. 340–343.

[12] COSTA, V. S., DUTRA, I., AND ROCHA, R. Threads and or-parallelism unified. *Theory and Practice of Logic Programming 10* (7 2010), 417–432.

[13] COSTA, V. S., ROCHA, R., AND DAMAS, L. The yap prolog system. *Theory and Practice of Logic Programming 12* (1 2012), 5–34.

[14] COSTA, V. S., WARREN, D. H. D., AND YANG, R. Andorra i: A parallel prolog system that transparently exploits both and-and or-parallelism. *SIGPLAN Not. 26*, 7 (Apr. 1991), 83–93.

[15] CUNHA, J. C., AND MARQUES, R. F. Pvm-prolog: A prolog interface to pvm. In *In Proceedings of the 1st Austrian-Hungarian Workshop on Distributed and Parallel Systems, DAPSYS'96* (1996), pp. 173–181.

[16] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[17] DEBRAY, S. K., LIN, N.-W., AND HERMNEGILDO, M. Task granularity analysis in logic programs. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1990), PLDI '90, ACM, pp. 174–188.

[18] DEGROOT, D. Restricted and-parallelism and side effects. In *Procs. of the 1987 Int'l Symp. on Logic Programming (ISLP 87)* (1987), IEEE Computer Society, pp. 80–89.

[19] DUTRA, I., ROCHA, R., COSTA, V., SILVA, F., AND SANTOS, J. Scheduling or-parallelism in yapor and thor on multi-core machines. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International* (May 2012), pp. 1581–1590.

[20] GUPTA, G., AND COSTA, V. S. Cuts and side-effects in and-or parallel prolog. *The Journal of Logic Programming 27*, 1 (1996), 45 – 71.

[21] GUPTA, G., PONTELLI, E., ALI, K. A., CARLSSON, M., AND HERMENEGILDO, M. V. Parallel execution of prolog programs: A survey. *ACM Trans. Program. Lang. Syst. 23*, 4 (July 2001), 472–602.

[22] HERMENEGILDO, M., AND GREENE, K. The &-prolog system: Exploiting independent and-parallelism. *New Generation Computing 9*, 3-4 (1991), 233–256.

[23] HERMENEGILDO, M. V., BUENO, F., CARRO, M., LÓPEZ-GARCÍA, P., MERA, E., MORALES, J. F., AND PUEBLA, G. An overview of ciao and its design philosophy. *Theory and Practice of Logic Programming 12* (1 2012), 219–252.

[24] HORN, A. On sentences which are true of direct unions of algebras. *The Journal of Symbolic Logic 16* (3 1951), 14–21.

[25] IEEE PORTABLE APPLICATIONS STANDARDS COMMITTEE AND OTHERS. Ieee std 1003.1 c-1995, threads extensions, 1995.

[26] INTELLIGENT SYSTEMS LABORATORY. *Quintus Prolog user's manual*. Swedish Institute of Computer Science, Kista, Sweden, December 2003.

[27] KING, A., SHEN, K., AND BENOY, F. Lower-bound time-complexity analysis of logic programs. In *ILPS* (1997), vol. 97, pp. 261–275.

[28] LIN, Y.-J., AND KUMAR, V. And-parallel execution of logic programs on a shared-memory multiprocessor. *J. Log. Program. 10*, 2 (Jan. 1991), 155–178.

[29] LUSK, E., BUTLER, R., DISZ, T., OLSON, R., OVERBEEK, R., STEVENS, R., WARREN, D., CALDERWOOD, A., SZEREDI, P., HARIDI, S., BRAND, P., CARLSSON, M., CIEPIELEWSKI, A., AND HAUSMAN, B. The aurora or-parallel prolog system. *New Generation Computing 7*, 2-3 (1990), 243–271.

[30] ROCHA, R., SILVA, F., AND COSTA, V. S. YapOr: an Or-Parallel Prolog System Based on Environment Copying. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence (EPIA 1999)* (Évora, Portugal, September 1999), P. Barahona and J. Alferes, Eds., no. 1695 in LNAI, Springer, pp. 178–192.

[31] SHEN, K., COSTA, V. S., AND KING, A. Distance: A new metric for controlling granularity for parallel execution. In *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming* (Cambridge, MA, USA, 1998), JICSLP'98, MIT Press, pp. 85–99.

[32] SINDAHA, R. The dharma scheduler-definitive scheduling in aurora on multiprocessors architecture. In *Parallel and Distributed Processing, 1992. Proceedings of the Fourth IEEE Symposium on* (Dec 1992), pp. 296–303.

[33] SRINIVASAN, A., FARUQUIE, T., AND JOSHI, S. Data and task parallelism in ilp using mapreduce. *Machine Learning 86*, 1 (2012), 141–168.

[34] SWIFT, T., AND WARREN, D. S. Xsb: Extending prolog with tabled logic programming. *Theory and Practice of Logic Programming 12* (1 2012), 157–187.

[35] TANENBAUM, A. S. *Modern Operating Systems*, 3rd ed. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[36] TARAU, P. Concurrent programming constructs in multi-engine prolog: Parallelism just for the cores (and not more!). In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming* (New York, NY, USA, 2011), DAMP '11, ACM, pp. 55–64.

[37] TICK, E. Compile-time granularity analysis for parallel logic programming languages. *New Generation Computing 7*, 2-3 (1990), 325–337.

[38] VIEIRA, R., ROCHA, R., AND SILVA, F. Or-Parallel Prolog Execution on Multicores Based on Stack Splitting. In *Proceedings of the 7th International Workshop on Declarative Aspects and Applications of Multicore Programming (DAMP 2012)* (Philadelphia, Pennsylvania, USA, January 2012), V. S. Costa, Ed., ACM Digital Library, pp. 1–9.

[39] VILLAVERDE, K., PONTELLI, E., GUO, H., AND GUPTA, G. Pals: An or-parallel implementation of prolog on beowulf architectures. In *Logic Programming*, P. Codognet, Ed., vol. 2237 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2001, pp. 27–42.

[40] WARREN, D. H. D. An abstract prolog instruction set. Tech. Rep. 309, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Oct 1983.

[41] WARREN, D. H. D. The sri model for or-parallel execution of prolog: Abstract design and implementation issues. In *inProceedings of the 1987 Symposium on Logic Programming* (1987), pp. 92–102.

[42] WERNER, O., YAMIN, A. C., BARBOSA, J. L. V., AND GEYER, C. F. R. Opera project: An approach towards parallelism exploitation on logic programming. In *WLP* (1994), pp. 20–23.

[43] WIELEMAKER, J., SCHRIJVERS, T., TRISKA, M., AND LAGER, T. SWI-Prolog. *Theory and Practice of Logic Programming 12*, 1-2 (2012), 67–96.

[44] XIROGIANNIS, G. Granularity control for distributed execution of logic programs. In *Distributed Computing Systems, 1998. Proceedings. 18th International Conference on* (1998), IEEE, pp. 230–237.

[45] XIROGIANNIS, G., AND TAYLOR, H. Pan: A portable, parallel prolog: Its design, realisation and performance. *New Generation Computing 20*, 4 (2002), 373–399.