Um Sistema Baseado na Cópia de Ambientes para a Execução de Prolog em Paralelo

Ricardo Jorge Gomes Lopes da Rocha

Dissertação submetida à Universidade do Minho
para obtenção do grau de Mestre em Informática
na especialidade de Ciências da Computação

Departamento de Informática
Universidade do Minho
Julho de 1996

Dedicado aos meus pais.

Agradecimentos

Desejo agradecer a colaboração e o apoio prestado por todos aqueles que em diferentes momentos e a diversos níveis contribuíram para uma melhor prossecução deste trabalho. Em particular, desejo agradecer e enaltecer todo o trabalho de orientação do Prof. Doutor Fernando Silva, pelo modo como a todos os níveis acompanhou e orientou todo o trabalho e pela disponibilidade, preocupação e apoio constante que me concedeu.

Gostaria também de agradecer ao Prof. Doutor Luís Damas, ao Prof. Doutor Vítor Santos Costa, ao Luís Lopes, e ao Eduardo Correia pela disponibilidade e pela paciente colaboração em vários momentos deste trabalho.

Um agradecimento também especial ao Prof. Doutor José Maia Neves, pela atenção e disponibilidade demonstrada na cooperação entre o Departamento de Informática da Universidade do Minho e o Laboratório de Inteligência Artificial e de Ciência de Computadores da Universidade do Porto (LIACC).

Quero exprimir a minha gratidão ao LIACC e ao projecto PROLOPPE (PRAXIS 3/3.1/TIT/24/94) pelas condições oferecidas e pelas facilidades concedidas para o desenvolvimento deste trabalho.

Quero agradecer o importante apoio financeiro que usufruí no âmbito do Programa PRAXIS XXI, pela concessão da bolsa de estudo com a referência BM/2356/94.

Quero testemunhar a minha amizade ao Ricardo Nuno e ao Michel, que sempre me acompanharam ao longo de todo este trabalho.

Quero carinhosamente agradecer aos meus pais José e Noémia, à Silvia e à Rute pelo seu constante apoio e por estarem sempre ao meu lado nos momentos mais difíceis.

Por fim, quero apresentar uma palavra de reconhecimento a todos aqueles que ao longo destes anos e aos diversos níveis, de algum modo contribuíram para a minha formação como indivíduo.

Resumo

A implementação eficiente de Prolog bem como a adequação da linguagem para aplicações no domínio da Inteligência Artificial, são factores que muito contribuíram para a sua popularidade. Apesar do muito esforço no aumento de performance dos sistemas sequenciais de Prolog, surgiu a necessidade de novas propostas de implementação alicerçadas nas modernas arquitecturas paralelas, onde um vasto conjunto de unidades de processamento cooperam entre si para acelerar a execução de um programa. Através dessa aceleração, as implementações paralelas de Prolog podem aumentar a performance dos programas existentes e expandir a classe dos problemas que podem ser resolvidos nesta linguagem.

Nesta tese descreve-se o desenho e implementação do YapOr, um sistema paralelo de execução de Prolog, que explora paralelismo-Ou implícito a partir da plataforma Yap de execução sequencial. O YapOr utiliza o modelo de cópia de ambientes para exploração de paralelismo-Ou e implementa muitos dos conceitos introduzidos pelo sistema Muse.

No desenvolvimento deste sistema foi essencialmente importante desenhar as estruturas de dados de suporte a todo o processo paralelo, implementar o mecanismo de cópia incremental de ambientes, desenvolver uma organização de memória capaz de responder com elevada eficácia às necessidades do processo paralelo e em particular às do mecanismo de cópia incremental, implementar as estratégias de distribuição de trabalho, desenhar uma *interface* entre o distribuidor de trabalho e o emulador de instruções do Yap, implementar eficientemente o conjunto de aspectos envolvidos no processo de partilha de trabalho, tratar o predicado de corte de alternativas e desenvolver um esquema de suporte às soluções que potencialmente podem corresponder a trabalho especulativo.

O YapOr suporta a execução paralela de uma ampla classe de programas escritos em Prolog, obteve elevados índices de performance na execução paralela de um largo conjunto de programas de teste, e mostrou um excelente comportamento quando comparado com o sistema Muse.

Abstract

Prolog is a popular programming language that is particularly important in Artificial Intelligence applications. Traditionally, Prolog has been implemented in the common, sequential general-purpose computers. More recently, Prolog implementations have also been proposed for parallel architectures where several processors work together to speedup the execution of a program. By giving better speedups, parallel implementations of Prolog should enable better performance for current problems, and expand the range of applications we can solve with Prolog.

This work addresses the issues of the design, implementation and performance evaluation of YapOr, an Or-parallel Prolog system. YapOr extends Yap's sequential execution model to exploit implicit Or-parallelism in Prolog programs. It is based on the environment copy model and implements most of the ideas introduced in Muse system.

To develop YapOr, it was necessary to solve some important issues, such as, design of data structures to support parallel process, implement the incremental copy technique, develop a memory organisation able to answer with efficiency to the parallel process and to the incremental copy in particular, implement the scheduling strategies, design an interface between the scheduler and the engine, implement the sharing work process, handle cut predicate with care and develop a scheme to support solutions that potentially correspond to speculative work.

An initial evaluation of the performance of YapOr showed that it achieves very good performance on a large set of benchmark programs. It also showed that YapOr compares favourably with Muse.

Índice

Índice de Figuras

Índice de Tabelas

Capítulo 1

Introdução

O Prolog tornou-se na mais popular linguagem de programação lógica graças às suas implementações eficientes. Os sistemas actuais de Prolog tornaram-se eficientes com o desenvolvimento e progresso da WAM, um modelo de execução sequencial.

Recentemente, as máquinas paralelas começaram a emergir comercialmente. Tudo indica que possivelmente estas máquinas se tornarão na maior fonte de poder de computação. Todavia, o desenvolvimento de algoritmos paralelos é actualmente o maior obstáculo a uma ampla aceitação das máquinas paralelas. A programação lógica, pelo paralelismo implícito na avaliação das expressões lógicas, liberta o programador do peso de gerir paralelismo explícito. Por conseguinte, a programação lógica oferece um grande potencial para tornar as máquinas paralelas não mais difíceis de programar do que as sequenciais.

Os programas lógicos possuem essencialmente dois tipos de paralelismo implícito: paralelismo-Ou e paralelismo-E. Num primeiro passo, o paralelismo-Ou é aquele que parece ser mais fácil e mais produtivo de explorar de forma transparente. Em princípio, o paralelismo-Ou deveria ser simples de implementar, já que as várias ramificações da árvore de procura são independentes umas das outras, o que requer pouca comunicação entre os vários agentes do processo paralelo. Todavia, na prática o paralelismo-Ou é de difícil implementação. Os principais problemas na sua implementação são: a eficiente representação dos múltiplos ambientes que coexistem simultaneamente na execução de um programa e a eficiente distribuição do paralelismo existente pelos agentes do sistema.

O âmbito deste trabalho enquadra-se no desenho e implementação de sistemas capazes de explorar, de forma implícita, paralelismo-Ou em programas lógicos.

1.1 Motivação

1.1.1 Computação Paralela

Arquitecturas com múltiplos processadores podem cooperar na resolução simultânea de um dado problema e potencialmente diminuir o tempo de resposta. Além disso, como um largo número de problemas são inerentemente paralelos, podem ser implementados de uma forma mais natural e eficiente em arquitecturas que suportam modelos de execução paralela. No entanto, extrair a máxima performance de um modelo constituído por um conjunto de unidades de processamento é tudo menos trivial.

Existem dois problemas fundamentais no desenvolvimentos de tais modelos. O primeiro diz respeito ao modo como o paralelismo existente num programa pode ser reconhecido pelo modelo paralelo. O segundo diz respeito à forma como esse paralelismo deve ser eficientemente distribuído pelas diferentes unidades de processamento do modelo.

O primeiro problema pode ser resolvido ou através do desenvolvimento de compiladores capazes de detectar eficazmente as partes paralelas dum programa (*paralelismo implícito*), ou por acção directa do programador que através de instruções adequadas explícita as partes paralelas do programa (*paralelismo explícito*).

A resolução do segundo problema assenta essencialmente no desenvolvimento de algoritmos eficientes de distribuição das partes paralelas dum programa, que consigam aumentar substancialmente a performance do programa, quando executado em paralelo em arquitecturas multiprocessador.

1.1.2 Programação Lógica e Prolog

As linguagens de programação lógica encerram em si próprias um conjunto de vantagens [Car90]:

Semântica declarativa simples. Um programa lógico é simplesmente um conjunto de cláusulas de predicados lógicos.

Semântica procedimental simples. Um programa lógico é simplesmente uma colecção de procedimentos recursivos. As cláusulas são tentadas pela ordem que são escritas e os objectivos[1] de cada cláusula são executados da esquerda para a direita.

Grande poder expressivo. Os programas lógicos podem ser vistos como especificações executáveis, e não obstante a semântica procedimental simples é possível desenhar programas algoritmicamente eficientes.

Não determinismo próprio. Como em geral várias cláusulas podem unificar com um objectivo, os problemas envolvendo procura podem facilmente ser programados neste tipo de linguagens.

O Prolog tornou-se na mais popular linguagem de programação lógica graças às suas implementações eficientes. O Prolog é um simples provador de teoremas baseado em lógica de primeira ordem, que dado um programa (também chamado de teoria) e uma questão, tenta satisfazer a questão usando apenas o programa. Em caso de sucesso, as instanciações das variáveis da questão são apresentadas como resultado final. O Prolog contempla fundamentalmente os seguintes aspectos [Kar92a]:

- As variáveis são variáveis lógicas que só podem ser instanciadas uma única vez.
- As variáveis não têm tipo até serem instanciadas.
- As variáveis são instanciadas via unificação (operação de equiparar variáveis, que encontra a instância comum mais geral entre dois conjuntos de variáveis).
- Em caso de falha da unificação, a execução retrocede e tenta encontrar uma outra forma de satisfazer a questão original. Em caso de retrocesso o Prolog investiga as diferentes alternativas (cláusulas) pela ordem em que estão listadas no programa.

A primeira implementação de Prolog foi um interpretador escrito por Robert Kowalski e Alain Colmerauer [Kow79] no início dos anos 70 que surgiu de um estudo para compreender linguagem natural. Em 1977, David Warren tornou o Prolog numa linguagem viável através do desenvolvimento do primeiro compilador [War77]. Os sistemas actuais de Prolog tornaram-se eficientes com o desenvolvimento e progresso do modelo sequencial de execução apresentado em 1983 por David Warren, conhecido por WAM (*Warren Abstract Machine*) [War83][2].

[1] Em toda a tese, a palavra 'objectivo' traduz a palavra '*goal*', habitualmente utilizada neste contexto na literatura inglesa.

[2] Em [War83], Warren apresenta os aspectos mais importantes da WAM e releva os detalhes para segundo plano. Muitos aspectos importantes não estão focados e poucos estão devidamente clarificados, o

Recentemente, as máquinas paralelas começaram a emergir comercialmente, e tudo indica que possivelmente estas máquinas rapidamente se tornarão na maior fonte de poder de computação mercê de uma equilibrada relação custo-desempenho. Todavia, o desenvolvimento de algoritmos paralelos é actualmente bastante difícil. Este é o maior obstáculo a uma ampla aceitação das máquinas paralelas.

A programação lógica, pelo paralelismo implícito na avaliação das expressões lógicas, liberta o programador do peso de gerir paralelismo explícito. Por conseguinte, a programação lógica além de oferecer um grande potencial para tornar os computadores paralelos não mais difíceis de programar do que os sequenciais, permite que o *software* possa migrar transparentemente entre máquinas sequenciais e paralelas.

Nos programas lógicos podem ser identificados essencialmente quatro tipos de paralelismo implícito [GACH96]:

Paralelismo na Unificação. Aparece quando os argumentos de um objectivo são unificados com os da cabeça de uma cláusula com o mesmo nome e aridade. Os diferentes termos de um argumento podem ser unificados em paralelo bem como os diferentes subtermos de um mesmo termo. Este paralelismo é de granularidade baixa e não têm sido o foco da investigação em programação lógica.

Paralelismo-Ou. Aparece quando mais do que uma cláusula define um predicado e uma chamada do predicado unifica com a cabeça de mais do que uma cláusula. O paralelismo-Ou é um modo eficiente de procurar as soluções de uma questão, através da exploração paralela de soluções das diferentes alternativas.

Paralelismo-E Independente. Aparece quando existe mais do que um objectivo presente numa questão ou no corpo de uma cláusula, e as instanciações das variáveis desses objectivos, num dado momento da computação, são tais que dois ou mais objectivos são independentes um do outro, isto é, a intersecção do conjunto das variáveis não instanciadas dos objectivos é vazio. A execução paralela destes objectivos dá lugar a paralelismo-E independente.

Paralelismo-E Dependente. Aparece quando dois ou mais objectivos no corpo de uma cláusula têm variáveis comuns e são executados em paralelo. Existem duas formas de explorar este paralelismo: (i) Os dois objectivos podem ser executados

que torna a compreensão do modelo bastante difícil ao leitor mediano. Em [AK91] e [Kog91, caps. 17-18] podemos encontrar descrições mais explícitas e detalhadas do modelo.

independentemente até que um deles instancie uma variável comum. Podem também ser executados independentemente até que ambos terminem, para então aí comparar a compatibilidade das atribuições a variáveis comuns (*back unification*). Esta versão é semelhante ao paralelismo-E independente. (ii) Logo que uma variável comum seja instanciada por um dos objectivos (designado por *produtor*) é lida como argumento de entrada para o outro objectivo (designado por *consumidor*) e o paralelismo pode continuar a ser explorado.

Estes quatro tipos de paralelismo são amplamente ortogonais, isto é, qualquer um deles pode ser explorado sem afectar a exploração de outro. Por conseguinte, é possível explorar os quatro em simultâneo num único sistema. Todavia, nenhum sistema paralelo eficiente foi ainda construído que suporte simultaneamente estas quatro formas de paralelismo. Um sistema que explore o máximo de paralelismo dos programas lógicos enquanto mantém a melhor performance possível, é o objectivo fundamental dos investigadores em programação lógica paralela.

1.1.3 Paralelismo-Ou

Existem várias razões para num primeiro passo nos restringirmos ao paralelismo-Ou. A razão fundamental é que o paralelismo-Ou parece ser mais fácil e mais produtivo de explorar, de forma transparente, do que o paralelismo-E. As principais vantagens de explorar paralelismo-Ou são as seguintes [LBD$^+$88]:

Generalidade. É relativamente linear explorar paralelismo-Ou sem restringir o poder da linguagem de programação lógica. Em particular, conseguimos conservar a vantagem que temos no Prolog de gerar todas as soluções para um objectivo.

Simplicidade. É possível explorar paralelismo-Ou sem solicitar nenhuma anotação extra a nível da programação e sem necessitar de alguma análise complexa no tempo de compilação.

Proximidade do Prolog. É possível explorar paralelismo-Ou num modelo de execução bastante próximo do modelo sequencial. Isto significa que se torna mais fácil manter a semântica da linguagem, e que podemos tomar total vantagem da tecnologia existente de implementação sequencial, para atingir velocidades absolutas bastante altas por processador.

Granularidade. O paralelismo-Ou têm um potencial, pelo menos para um grande classe de programas, de paralelismo de granularidade alta. O tamanho da granularidade de uma computação paralela refere-se à quantidade de trabalho que pode ser executado sem a interacção de outros pedaços de trabalho processados em simultâneo. É mais fácil de explorar paralelismo quando efectivamente a granularidade é alta.

Aplicações. Uma significante quantidade de paralelismo-Ou ocorre num extenso conjunto de aplicações, especialmente na área específica de Inteligência Artificial. O paralelismo-Ou manifesta-se principalmente em programas de procura como é a prova de teoremas, a análise gramatical de linguagem natural, a resposta a questões sob uma base de dados e o emprego de regras em sistemas periciais.

Em princípio, o paralelismo-Ou deveria ser simples de implementar já que as várias ramificações da árvore de procura, de um objectivo, são independentes umas das outras, o que requer pouca comunicação entre os vários agentes do processo paralelo.

Todavia, na prática, o paralelismo-Ou revela-se de difícil implementação devido essencialmente à partilha de nós da árvore de procura. Dadas duas ramificações, existe uma série de nós comuns a ambas. Uma variável criada nesses nós comuns, pode ser instanciada de forma diferente nas duas ramificações. Os ambientes das duas ramificações têm que ser organizados de modo a que as diferentes instanciações aplicadas em cada ramificação possam ser facilmente discerníveis. Se uma atribuição a uma variável, criada nos nós comuns, é feita pelo menos até o último nó comum, então essa atribuição é a mesma nas duas ramificações (*atribuição incondicional*). Todavia, se essa atribuição é feita depois da criação do último nó comum, deve ser apenas vista na ramificação onde foi feita a atribuição (*atribuição condicional*).

Posto isto, o principal problema na implementação de paralelismo-Ou é a eficiente representação dos múltiplos ambientes que coexistem simultaneamente na execução de um programa. O principal problema na gestão de ambientes múltiplos é o acesso às atribuições condicionais e a eficiente representação das mesmas, já que as atribuições incondicionais podem ser tratadas como na execução sequencial de programas lógicos.

O problema da gestão de múltiplos ambientes é essencialmente solucionado por mecanismos onde cada ramificação tem uma área privada onde guarda as suas atribuições condicionais. Existem variados modelos que implementam estes mecanismos das mais diversas maneiras [GJ93]. No entanto, cada modelo tem custos associados que se fazem notar no tempo de criação de um nó, no tempo de acesso às variáveis, no tempo de instanciação de uma variável e no tempo de mudança de contexto para a execução de uma nova ramificação.

Foi mostrado em [GJ93] que é impossível evitar simultaneamente todos os custos e executar todas as operações citadas em tempo constante. Por outras palavras, o custo da gestão de múltiplos ambientes não pode ser completamente evitado. No entanto, pode ser significativamente reduzido através dum cuidado desenho do distribuidor de trabalho, cuja função fundamental é distribuir de uma forma eficiente e cuidada as ramificações, por explorar, da árvore de procura pelos diversos agentes envolvidos no processamento paralelo.

1.1.4 Modelos e Sistemas de Execução de Paralelismo-Ou

Dos modelos de execução de paralelismo-Ou existem dois que são considerados como referências, porque servem de suporte aos dois sistemas mais robustos e amadurecidos de paralelismo-Ou, os sistemas Aurora [LBD$^+$88, Car90] e Muse [AK90b, AK90a, Kar92a]. Esses modelos são:

Binding Arrays. No modelo de *binding arrays* cada agente (processador ou processo) possui uma estrutura de dados auxiliar designada por *binding array*. Cada variável condicional, ao longo de uma ramificação, é desde a raiz numerada sequencialmente. Para fazer esta numeração, quando uma variável condicional é criada é etiquetada com o valor dum contador, isto é, o valor do contador é nela guardado (este valor corresponde à posição da variável no *binding array*). De seguida o contador é incrementado. Quando uma variável condicional é instanciada, na área privada do processador responsável pela instanciação e na posição do *binding array* correspondente à variável, é guardado o valor da atribuição. Além disso, o endereço da variável condicional juntamente com a atribuição condicional são guardados na trilha. Se mais tarde for necessário utilizar a atribuição de uma variável, a etiqueta da variável é usada para indexar o *binding array*, para assim obter essa atribuição. As atribuições de todas as variáveis, condicionais e incondicionais são acessíveis em tempo constante. Para garantir a consistência do modelo, quando um agente muda de uma ramificação para outra, actualiza o seu *binding array* desinstalando as atribuições guardadas na trilha referentes à ramificação que deixou e instala as atribuições correctas relativas à nova ramificação.

Cópia de Ambientes. No modelo de cópia de ambientes, cada agente mantém uma cópia separada do seu ambiente, no qual pode escrever sem causar conflito de atribuições. Neste modelo nem tão pouco as atribuições incondicionais são partilhadas. Um agente suspenso quando pega numa alternativa não explorada dum nó criado por um outro agente, copia todas as pilhas de execução deste. Esta cópia de pilhas é eficientemente concretizada através do mecanismo de *cópia incremental*. A ideia da cópia incremental é baseada no facto de que um agente suspenso pode já ter percorrido parte do caminho entre o nó raiz e o nó mais profundo comum a ambos os agentes, e por conseguinte não necessita de copiar a percentagem das pilhas referente a essa parte. Após este processo de cópia, cada agente pode comportar-se exactamente como um sistema sequencial, requerendo apenas ligeiras sincronizações com os outros agentes.

Um sistema paralelo que suporta todos os predicados extralógicos, metalógicos, efeitos colaterais e que produz o mesmo efeito que a execução sequencial, da esquerda para a direita e de cima para baixo na selecção das cláusulas, é designado como um sistema que suporta a *semântica sequencial do Prolog*. Um dos maiores problemas inerente ao suporte desta semântica é o introduzido pelo predicado de corte de alternativas. A execução do predicado de corte de alternativas, a que corresponde o corte das alternativas situadas à direita da ramificação de alcance do predicado, pode invalidar parte da computação de um outro agente. Esta computação desnecessária é habitualmente denominada *trabalho especulativo* [Cie92]. Para se dar suporte à semântica sequencial do Prolog de uma forma eficiente é então necessário uma maior elaboração no desenho dos distribuidores de trabalho.

O sistema **Aurora** utiliza o modelo de execução de *binding arrays* e foi construído adaptando o sistema sequencial de Prolog SICStus Prolog 0.6 [CW88]. Dá suporte à semântica sequencial do Prolog e o maior custo do sistema verifica-se no tempo de mudança de contexto para a execução de uma nova alternativa. Os três distribuidores de trabalho mais conhecidos utilizados no sistema Aurora são:

Manchester Scheduler. Distribui trabalho logo que seja possível com uma estratégia de distribuição o mais perto possível do topo da árvore. Não foi desenhado para tratar com cuidado o trabalho especulativo.

The Bristol Scheduler. Tenta minimizar os custos do distribuidor de trabalho estendendo ao máximo a região partilhada: são partilhadas sequências de nós em lugar de nós singulares; o trabalho é tomado do mais profundo nó vivo de uma ramificação.

Uma versão mais recente aborda com eficiência o problema do trabalho especulativo através de uma estratégia denominada *actively seeking the least speculative work*. Usa esta estratégia quando existe trabalho disponível não especulativo e a anterior quando todo o trabalho disponível é especulativo.

The Dharma Scheduler. Foi desenhado para tratar eficientemente o problema do trabalho especulativo. Resolve o problema de encontrar rapidamente o mais à esquerda, logo o menos especulativo, pedaço de trabalho disponível, através de ligações directas entre as extremidades de cada ramificação.

O sistema **Muse** utiliza o modelo de execução de cópia de ambientes e foi inicialmente construído estendendo o sistema sequencial de Prolog SICStus Prolog 0.6. Dá suporte à semântica sequencial do Prolog e tal como no sistema Aurora o maior custo do sistema verifica-se no tempo de mudança de contexto para a execução de uma nova alternativa. O Muse é composto por um conjunto de agentes, que não são mais do que WAMs estendidas. Cada um deles possui um espaço local de endereçamento, e algum espaço global é partilhado por todos. A cópia incremental é utilizada eficientemente mercê da organização das pilhas da WAM. Suporta com eficiência o problema de trabalho especulativo. O distribuidor de trabalho usa duas estratégias: uma para lidar com o trabalho especulativo (*actively seeking the leftmost available work strategy* [AK92]) e outra para lidar com os agentes suspensos à procura de trabalho (distribui trabalho pelo nó mais profundo e com mais alternativas por explorar). Controla em tempo de execução a granularidade de cada tarefa a ser executada evitando a partilha de pequenos pedaços de trabalho.

Segundo alguns estudos [AKM92, AK92, CCS94], em situações análogas o sistema Muse quando comparado com o sistema Aurora apresenta índices de performance superiores.

1.2 Objectivos

O principal objectivo deste trabalho consiste no desenho cuidado e na implementação eficiente do YapOr, um sistema paralelo de execução de Prolog que explora paralelismo-Ou implícito a partir da plataforma Yap de execução sequencial [DSCRA89]. O YapOr utiliza o modelo de cópia de ambientes, baseia-se em muitos dos conceitos introduzidos pelo sistema Muse [Kar92b], e visa conseguir obter altos desempenhos para a execução paralela de Prolog de forma a acelerar a execução dos programas executados sobre o sistema.

Subjacente ao objectivo principal, pretende-se que o novo sistema execute uma ampla classe de programas escritos em Prolog, isto é, pretende-se que o YapOr suporte o maior número de predicados *standard* do Prolog. Destes é essencialmente importante tratar o uso do predicado de corte de alternativas.

As principais diferenças do sistema YapOr para o sistema Muse devem-se à forma particular como foram solucionados alguns dos conceitos do modelo de cópia de ambientes, comum aos dois sistemas. Estas diferenças resultam, por um lado, da ausência de referências específicas e/ou detalhadas dos aspectos envolvidos, e por outro, do facto de a estrutura do Yap não ser completamente compatível com as descrições de referência. O conjunto desses aspectos é enumerado de seguida.

- Organização de memória.
- Mecanismos de fecho para permitir a execução atómica de conjuntos de operações.
- Diferente divisão do mecanismo normal de retrocesso (*backtracking*) pelas instruções `fail`, `retry_me` e `trust_me` entre o Yap e as descrições de referência.
- Adaptação dos procedimentos e sincronizações das fases do processo de partilha de trabalho. No Yap, ao contrário do SICStus, os pontos de escolha[3] e os ambientes são tratados na mesma pilha de execução.
- Cálculo das áreas (das pilhas de execução) a copiar no processo de partilha de trabalho.
- Sincronizações necessárias ao processo de solicitação e partilha de trabalho.
- Mensagens entre agentes para tratar eficientemente o predicado de corte de alternativas.
- Esquema de suporte às soluções que vão sendo encontradas pelo sistema e que potencialmente podem corresponder a trabalho especulativo.
- Esquema de suporte à exploração contínua de todas as soluções de um objectivo.

Todos estes aspectos são explicados em detalhe nos capítulos seguintes.

[3] Em toda a tese, a designação 'ponto de escolha' traduz a designação '*choicepoint*', habitualmente utilizada neste contexto na literatura inglesa.

O bom comportamento dos programas de teste utilizados, bem como os bons índices de performance por eles obtidos quando executados no sistema YapOr, foram e são motivadores. O objectivo de conseguir obter altos desempenhos para a execução paralela de Prolog e de executar uma ampla classe de programas escritos em Prolog, foi assim satisfatoriamente atingido. Os resultados alcançados comprovaram que o modelo de cópia de ambientes se adapta eficientemente à exploração de paralelismo-Ou, justificando assim a sua escolha para modelo de execução paralela do YapOr.

1.3 Estrutura da Tese

A estrutura desta tese ambiciona essencialmente permitir uma agradável leitura e uma fácil compreensão da sequência de aspectos a considerar, de forma a motivar o mais possível o leitor. A tese está dividida em sete capítulos principais que reflectem, de certo modo, as fases do trabalho. Segue-se uma breve descrição do conteúdo de cada capítulo.

Cap. 2: O Sistema Muse. Este capítulo apresenta as estratégias e os conceitos básicos que servem de suporte à concepção do sistema Muse. Apresenta o modelo de execução de cópia de ambientes e o algoritmo de cópia incremental. Introduz as estratégias do distribuidor de trabalho para lidar eficientemente com os agentes sem trabalho.

Cap. 3: Extensão do Yap para suportar o YapOr. Este capítulo apresenta o conjunto de extensões e alterações que foram introduzidas no Yap de modo a suportar o YapOr. Inclui a mutação da organização de memória do Yap para a nova organização do YapOr, o como e o porquê da organização de memória do YapOr, a adaptação dos pontos de escolha à exploração paralela, as novas pseudo-instruções de ligação entre a execução sequencial e a paralela, o cálculo e a manutenção da carga de um agente, a influência da área de código e do processo de indexação no paralelismo, e como o YapOr procura todas as soluções.

Cap. 4: Pormenores de Implementação. Este capítulo aprofunda os aspectos mais importantes do sistema através de descrições mais explícitas da implementação. Descreve detalhadamente a adaptação do emulador de instruções e o retrocesso dos agentes para nós vivos partilhados. Apresenta minuciosamente todos os aspectos relativos ao processo de partilha de trabalho entre dois agentes e finalmente descreve o modo de evitar a partilha de pequenos pedaços de trabalho.

Cap. 5: O Predicado de Corte de Alternativas. Este capítulo apresenta o enquadramento do predicado de corte de alternativas no sistema YapOr. Apresenta a semântica do predicado de corte nos sistemas sequenciais e o problema do trabalho especulativo nos sistemas paralelos. Refere as estruturas de suporte adicionadas ao YapOr e o conjunto de soluções implementadas para uma correcta e eficiente manutenção da semântica do predicado de corte.

Cap. 6: Análise de Resultados. Este capítulo avalia o comportamento do YapOr face aos resultados obtidos. Apresenta o conjunto de programas de teste utilizados e o custo associado à adaptação do sistema sequencial Yap ao sistema paralelo YapOr. Analisa a performance do sistema face ao incremento do número de agentes e compara-a com o sistema Muse. Apresenta alguns dos potenciais factores que contribuem para uma menor performance do sistema e analisa o comportamento do YapOr face ao problema do trabalho especulativo.

Cap. 7: Conclusões e Trabalho Futuro. Este capítulo apresenta um conjunto de conclusões finais sobre o trabalho realizado. Relembra o trabalho numa perspectiva global, sublinha alguns dos principais problemas e dificuldades, e enumera a forma particular como foram solucionados alguns aspectos. Resume o comportamento do sistema face aos resultados alcançados, apresenta um conjunto de perspectivas de trabalho futuro e termina com uma breve nota de fecho.

Nos capítulos 4 e 5 é apresentado o pseudo-código de algumas implementações importantes. De forma a permitir ao leitor uma mais fácil compreensão dos conceitos apresentados, nunca é incluído o código referente a possíveis optimizações e/ou sincronizações no pseudo-código relativo a cada implementação, a menos que essa inclusão se revele fundamental.

Capítulo 2

O Sistema Muse

Neste capítulo, apresentam-se os conceitos básicos que servem de suporte à concepção do sistema Muse. A secção 2.1 introduz o modelo computacional de cópia de ambientes, enquanto que a secção 2.2 descreve as estratégias utilizadas para uma melhor disposição dos vários agentes envolvidos no processo paralelo. A globalidade dos conceitos aqui apresentados, foi igualmente adoptada para servir de suporte à concepção do sistema YapOr.

2.1 O Modelo de Cópia de Ambientes

Esta secção descreve sumariamente o modelo de cópia de ambientes [AK90b], utilizado nos sistemas Muse e YapOr.

2.1.1 Introdução de Conceitos

Para implementar este modelo é necessário a existência de um sistema multiprocessador, sob o qual irão operar as unidades fundamentais do modelo, os *agentes*. O termo agentes pode ser representado a nível do sistema tanto por processadores como por simples processos. Cada agente possui um idêntico espaço local de endereçamento e partilha com os restantes agentes um certo espaço global de endereçamento. Um agente comporta-se como se de uma máquina de execução sequencial de Prolog se tratasse, o que naturalmente implica que tenha de contemplar as pilhas de execução da WAM: *local stack*, *heap stack* e *trail stack*, as quais designaremos por *pilha local*, *pilha de termos* e *trilha* respectivamente. Estas pilhas não são partilhadas entre os agentes, já que se situam no espaço local de endereçamento de cada agente. A área de código da WAM é guardada no espaço partilhado, de forma a ser acessível por todos os agentes.

Em Prolog, cada ponto de escolha está associado à ideia abstracta de um nó da árvore de procura, a partir do qual se pode ter acesso a diferentes alternativas de executar um mesmo predicado. Um predicado é composto por um conjunto de cláusulas e cada cláusula define um modo diferente de executar esse predicado.

Um nó é considerado *privado* se é apenas acessível pelo agente que o criou, tal como acontece na execução sequencial de Prolog. É considerado *partilhado* se é acessível por mais do que um agente, isto é, mais do que um agente contém esse nó na sua ramificação corrente (o conceito abstracto de ramificação corrente da árvore de procura é em termos práticos concretizado pelo conjunto de todos os pontos de escolha criados na pilha local). Sendo assim, todo o nó, numa dada altura ou é privado ou se encontra partilhado, o que divide a árvore de procura de cada agente em duas regiões, por analogia referidas como, a *região privada* e a *região partilhada* da árvore de procura. Outro conceito relativo aos nós é o facto de poderem estar *mortos* ou *vivos*. Um nó é referido como nó vivo se ainda existirem alternativas não exploradas a partir dele, e como nó morto quando essas alternativas já não existem. Consequentemente, todo o nó privado é um nó vivo.

2.1.2 Modelo de Execução Básico

Os procedimentos básicos deste modelo podem resumir-se ao seguinte:

1. Antes do início da execução de um programa em Prolog, todos os agentes se encontram *suspensos*[1]. Logo após o seu início, apenas um dos agentes, seja ele P, começa por explorar as diversas alternativas disponíveis. Enquanto isso, todos os outros agentes continuam suspensos até que P crie algum ponto de escolha na sua pilha local, que corresponde ao facto de um dado predicado conter mais do que uma alternativa de execução.

2. Após tal suceder, um dos outros agentes, seja ele Q, solicita trabalho a P de modo a cooperar na exploração das alternativas pendentes.

[1] Um agente está suspenso, sempre que não está a contribuir para a execução do programa.

3. Seguidamente P permite que Q tome parte do seu trabalho, partilhando com ele os seus nós privados. Para que tal aconteça:

   - Para cada nó privado, P cria no espaço global uma *estrutura partilhada* para a qual move informação relativa às alternativas inexploradas e aos agentes envolvidos na operação, colocando no nó privado um apontador referenciando a estrutura entretanto criada (ver figura 2.1).

   - P copia todo o seu estado para Q, que corresponde a copiar a sua pilha local, pilha de termos e trilha. A partir desse momento P e Q encontram-se no mesmo estado computacional.

4. P prossegue a sua execução, a partir do local onde tinha sido interrompido, enquanto que Q, para não executar o mesmo trabalho que P, simula uma falha que conduz à execução do mecanismo normal de retrocesso[2] do Prolog. Como o nó em questão é do tipo partilhado, este mecanismo toma a próxima alternativa a explorar, não do ponto de escolha corrente mas sim da estrutura partilhada correspondente, para de seguida a processar exactamente como se de uma computação sequencial se tratasse.

5. Sempre que continue a existir um outro agente suspenso X, e se alguma alternativa se encontra por explorar num outro agente Y, então X deve solicitar esse mesmo trabalho a Y da mesma forma que Q o havia feito em relação a P.

6. Sempre que um agente Z não tenha mais nenhuma alternativa para explorar na sua ramificação corrente, retorna para o estado de suspenso e volta a procurar por um agente *ocupado*[3] passível de ser solicitado para partilhar o seu trabalho.

7. Quando todo o trabalho tiver sido explorado, a execução termina e todos os agentes voltam ao estado de suspensos.

2.1.3 Cópia Incremental

Numa operação de partilha de trabalho, a cópia de todo o estado de um agente para outro é uma actividade que acarreta um custo elevado para o desempenho do sistema. Sendo

[2] Em toda a tese, a designação 'normal de retrocesso' traduz a designação '*backtracking*', habitualmente utilizada neste contexto na literatura inglesa.

[3] Um agente é considerado ocupado quando se encontra a explorar uma dada alternativa. Um agente ocupado é um agente potencial para dispor de alternativas por explorar.

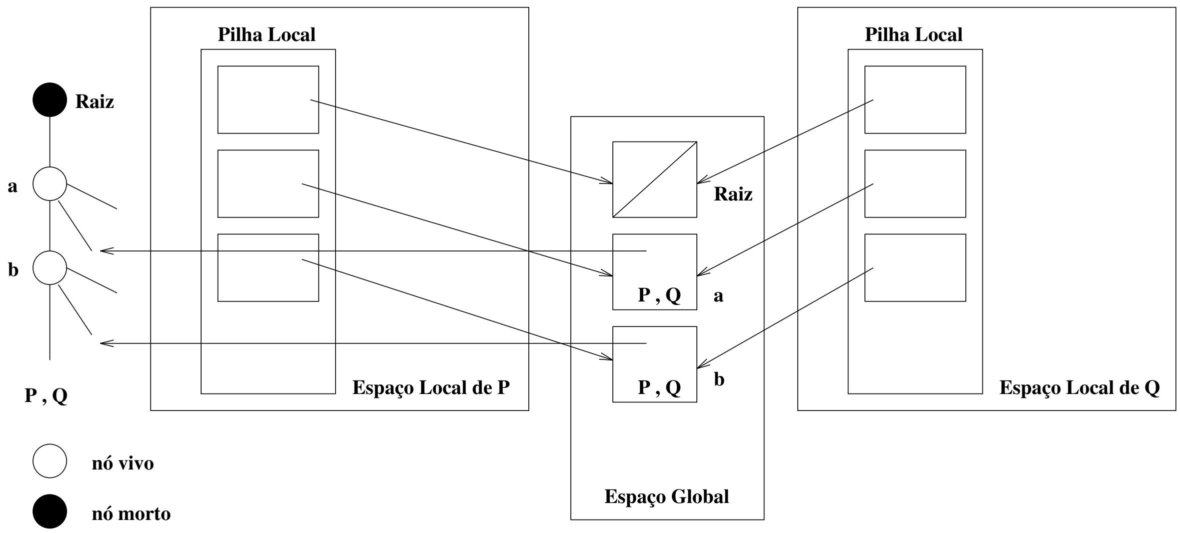


Figura 2.1: Relação entre pontos de escolha e estruturas partilhadas.

assim, é bastante importante o uso de mecanismos que permitam reduzir substancialmente esse custo. Um desses mecanismos é a cópia incremental do ambiente que caracteriza o estado de um agente.

O objectivo final da cópia de ambientes é posicionar os agentes envolvidos na operação de partilha, sejam eles P e Q, no mesmo nó da árvore de procura, a que corresponde um idêntico estado computacional. A ideia da cópia incremental é fazer com que Q mantenha a parte do seu estado que é consistente com o estado de P, de modo a copiar apenas a diferença entre ambos os estados.

Em Prolog, a criação de um ponto de escolha tem como objectivo principal guardar o estado corrente da computação. Sendo assim, para as variáveis criadas antes do ponto de escolha e às quais venha a ser atribuído um dado valor, guarda-se uma referência na trilha, de modo a que numa fase posterior seja possível recuperar o estado da computação. No mecanismo normal de retrocesso para um nó da árvore de procura, além de se recuperar o estado da computação guardado no ponto de escolha correspondente é também necessário desreferenciar as atribuições às variáveis cujas referências foram guardadas na trilha.

A questão que se coloca a seguir é de como implementar de forma eficiente a ideia atrás descrita da cópia incremental? Suponhamos que o agente Q não encontra trabalho disponível na sua árvore de procura, isto é, o conjunto de nós desde o nó raiz até ao seu nó corrente estão todos mortos, e que existe um agente P com nós vivos (ver Figura 2.2). Sendo assim, Q pode solicitar trabalho a P. Antes disso, Q deve começar por retroceder para o primeiro nó pertença também de P, de modo a que o seu estado fique consistente com parte do de P. De seguida, P pode partilhar os seus nós privados, criando para isso as necessárias estruturas partilhadas, e copiar para Q apenas as partes da pilha local, pilha de termos e trilha que reflictam a diferença entre ambos os estados. Essa diferença é calculada através da informação guardada no nó comum encontrado por Q e nos segmentos de topo da pilha local, pilha de termos e trilha de P. Além disso é necessário que as referências guardadas na trilha de P, que referenciam variáveis criadas na parte comum a ambos os agentes, sejam recuperadas no agente Q de modo a garantir a igualdade de estados pretendida.

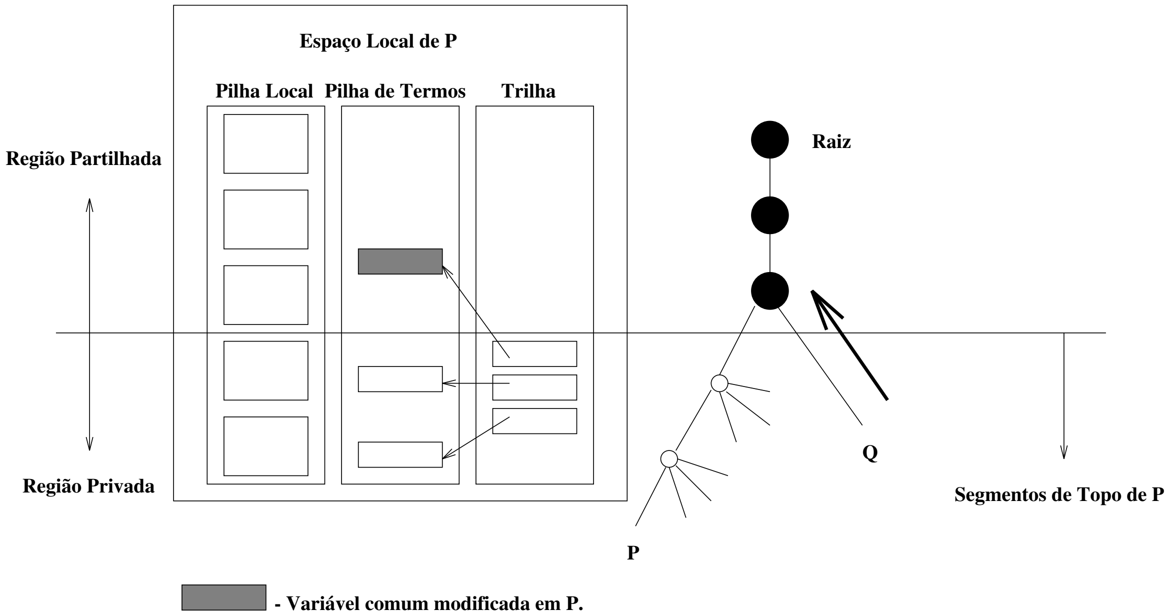


Figura 2.2: Alguns dos aspectos fundamentais da cópia incremental.

2.2 Distribuição de Trabalho

Nesta secção, descrevem-se as estratégias que o distribuidor de trabalho usa para melhor posicionar os diversos agentes pela árvore de procura na busca de soluções [AK90a], evitando-se tanto quanto possível referências ao modo como foram implementadas as estratégias descritas, no caso particular do sistema YapOr.

2.2.1 Ideias Gerais

Podemos dividir o período de execução de um certo agente na exploração das alternativas de um determinado objectivo em dois modos: em modo de procura e em modo de execução. Um agente entra em modo de procura sempre que se encontra na região partilhada da árvore na procura de uma nova alternativa. Um agente encontra-se em modo de execução sempre que deixa o modo de procura. Em modo de execução, todo o agente se comporta exactamente como se de um sistema de Prolog sequencial se tratasse, exceptuando o facto de ser capaz de responder a solicitações dos outros agentes.

As duas principais funções do distribuidor de trabalho são: manter correcta a semântica sequencial do Prolog, e dentro do possível atribuir novas *tarefas*[4] aos agentes que vão ficando suspensos, de modo a minimizar os factores que contribuem para a perda de desempenho do sistema[5]. Alguns dos principais factores conducentes à degradação do desempenho por parte do sistema são: a cópia de partes do estado de um outro agente, tornar partilhados os nós privados, e pegar em novas tarefas da região partilhada.

As principais linhas estratégicas que o distribuidor de trabalho deverá seguir para minimizar a contribuição desses factores são:

- Na operação de partilha de trabalho, o agente ocupado deve partilhar a totalidade dos nós privados que possui. Isto maximiza a quantidade de trabalho partilhada e permite que os agentes que terminam a exploração de uma tarefa possam rapidamente encontrar novas tarefas na região tornada partilhada.

- O distribuidor de trabalho deve seleccionar para partilhar trabalho, o agente ocupado que possua uma *maior carga* de trabalho e que se encontre *mais perto* do agente suspenso. A carga é uma medida que indica o número de alternativas por explorar dos nós privados. A escolha de um agente com uma carga elevada permite maximizar a quantidade de trabalho partilhado e evita por um maior período de tempo uma possível nova situação de suspensão. O estar mais perto diz respeito às posições relativas dos agentes na árvore de procura, o que minimiza a quantidade de partes do estado do agente ocupado a copiar, já que as partes comuns a ambos os agentes é à partida maior.

[4] Uma tarefa é uma porção contínua de trabalho executada na região privada da árvore. A intervenção do distribuidor de trabalho verifica-se sempre que uma tarefa é concluída.

[5] Na literatura inglesa estes factores são habitualmente designados por '*overheads*'.

- Quando o distribuidor de trabalho não encontra trabalho disponível no sistema, deve posicionar o agente suspenso na melhor posição possível da árvore de procura, de tal modo que uma futura operação de partilha acarrete o mínimo custo ao sistema.

- Um agente suspenso deve colocar-se antecipadamente na posição devida da árvore de procura antes de solicitar trabalho a um agente ocupado. Isto permite ao agente ocupado despender o menor tempo possível na operação de partilha.

Posto isto, a ideia básica do algoritmo do distribuidor de trabalho pode resumir-se ao seguinte: sempre que um agente entra na região partilhada da árvore de procura, tenta tomar a alternativa de trabalho por explorar mais próxima. Se não existe tal alternativa, torna-se suspenso e tenta seleccionar um outro agente ocupado, com excesso de carga, para partilhar trabalho. Se não existe tal agente, coloca-se na melhor posição possível da árvore de procura de modo a dispor de novo trabalho logo que seja possível.

2.2.2 Procura de Trabalho na Região Partilhada

Pretende-se que a procura de trabalho disponível na região partilhada seja realizada da forma mais eficaz possível. Tomar a alternativa, ainda por explorar, mais próxima da posição actual do agente na árvore de procura, parece ser uma ideia razoável. Isto corresponde a encontrar a nova tarefa no nó vivo mais profundo da ramificação corrente. Para tal, um mecanismo eficiente para decidir da existência ou não de um nó vivo na ramificação corrente e de como o poderemos encontrar, é forçosamente necessário.

O conjunto dos pontos de escolha que vão sendo criados na pilha local, representam no seu todo a posição relativa de um dado agente na árvore de procura global, desde a raiz, a que corresponde o ponto de escolha de topo, até o mais profundo e corrente ponto de escolha. Posto isto, é fácil concluir que, a menos da partilha de trabalho, um agente apenas se pode movimentar na referida árvore através da criação de novos pontos de escolha ou ascendendo por retrocesso aos nós mais antigos. Um agente apenas ascende na árvore para tomar trabalho de um nó vivo ou para se posicionar para uma operação de partilha de trabalho.

As estruturas partilhadas, criadas durante as operações de partilha de trabalho, possuem informação sobre as alternativas inexploradas e sobre os agentes que partilham o nó. Além disso, para evitar acessos simultâneos à estrutura, que poderiam levar a inconsistências no sistema, possuem um mecanismo de fecho. O mecanismo de decisão da existência ou não de um nó vivo na ramificação corrente, introduz um novo campo nas estruturas partilhadas que será designado por `nó_vivo_mais_próximo`. Este campo poderá conter ou uma referência para o nó vivo mais próximo ou um valor por defeito `null` indicando a inexistência de tais nós.

Quando um agente finaliza uma tarefa, verifica se o nó partilhado mais profundo está vivo. Se sim, fecha o acesso à estrutura partilhada correspondente e toma a seguinte alternativa disponível. De seguida, actualiza essa estrutura com a próxima alternativa que pode ser tomada nesse mesmo nó, liberta o acesso e inicia a exploração da tarefa tomada. Se o nó estiver morto, o agente consulta o campo `nó_vivo_mais_próximo` da estrutura. Se esse campo não contém o valor por defeito `null`, o agente percorre a cadeia de referências sucessivas dos campos `nó_vivo_mais_próximo`, até encontrar realmente um nó que se encontre vivo, sem que para isso efectue qualquer operação de fecho e sem que saia da sua posição corrente na árvore.

Se não existe nenhum nó vivo, então todos os nós da ramificação corrente devem ficar com o valor `null` nos respectivos campos `nó_vivo_mais_próximo`. De seguida, o agente mover-se-á para o nó mais próximo em que se encontrem outros agentes ocupados, e aí tentará solicitar um deles para que consigo partilhe novo trabalho, conforme o descrito na secção 2.2.3. Se por outro lado encontrou um nó vivo a, todos os nós mais profundos do que a passarão a referenciar no campo `nó_vivo_mais_próximo` o nó a (ver figura 2.3). Ao mesmo tempo o agente posicionar-se-á da forma mais rápida possível no nó a para tomar uma nova alternativa. Se entretanto ao chegar a a, todo o trabalho aí disponível tiver sido tomado por outros agentes, deve proceder de modo idêntico repetindo para isso o procedimento descrito.

2.2.3 Procura de Agentes Ocupados

Outra das principais funções do distribuidor de trabalho é a de seleccionar agentes ocupados para partilhar trabalho com os agentes suspensos. Como foi referido na secção 2.2.1, uma boa heurística para essa selecção parece ser a da escolha do agente que se encontra mais perto do agente suspenso e que simultaneamente possui maior carga. Nesta secção, pretende-se apresentar um mecanismo eficiente para emparelhar agentes suspensos com ocupados que sirva de suporte a esta heurística.

A ideia básica deste mecanismo é a seguinte: quando um agente Q passa para o estado

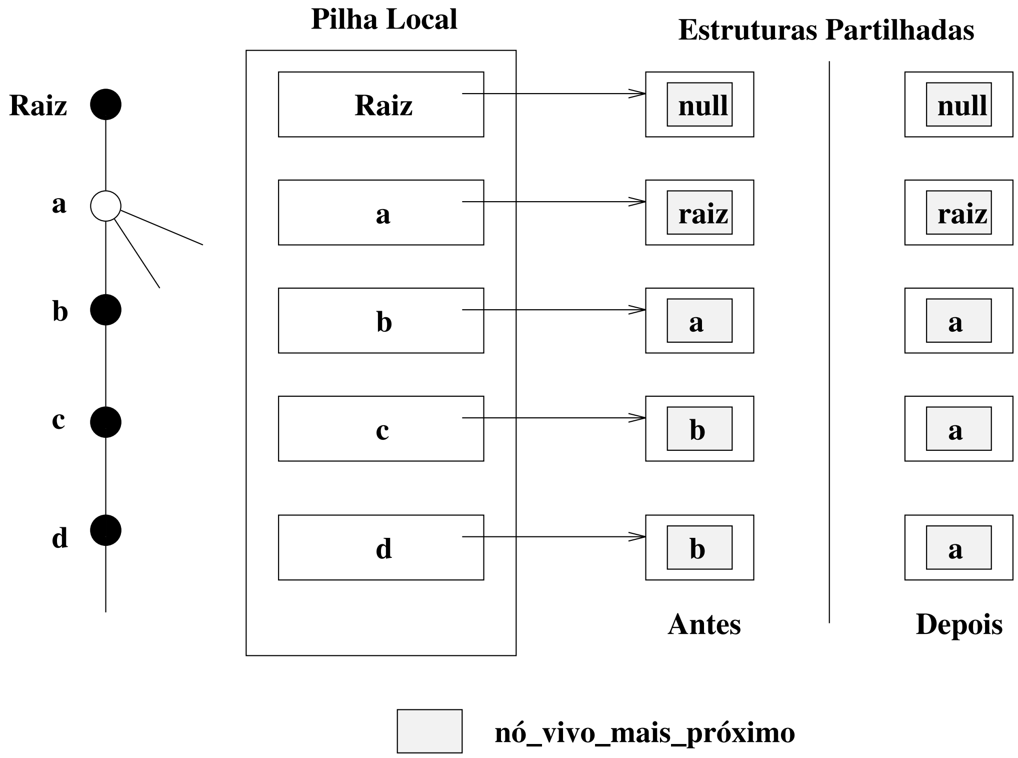


Figura 2.3: Actualização do campo `nó_vivo_mais_próximo`.

de suspenso, começa por determinar a partir da subárvore do nó corrente[6], o conjunto de agentes ocupados que se encontram mais próximo de si do que de outros agentes suspensos. Seguidamente, selecciona desse conjunto o agente P que possuir maior carga e solicita-lhe trabalho.

Se Q não encontrar nenhum agente disponível na fase anterior, então determina o conjunto de agentes ocupados fora da subárvore do nó corrente. De entre esses, selecciona P, aquele que no momento possua maior carga. Antes de lhe dirigir a consequente solicitação para partilhar trabalho, retrocede na árvore de procura até ao primeiro nó que contenha P.

A secção 2.2.3.1 descreve o modo de seleccionar P a partir da subárvore de Q, enquanto que a secção 2.2.3.2 descreve o modo de seleccionar P fora da subárvore de Q. Por fim, a secção 2.2.4 descreve as heurísticas que Q deve seguir para se posicionar num nó mais apropriado, quando não consegue encontrar no sistema nenhum agente ocupado com excesso de carga.

[6] Por subárvore do nó corrente, entenda-se a subárvore que tem por raiz esse nó corrente e que contém o conjunto das ramificações dos agentes que partilham esse nó e que se encontram nesse momento a explorar um nó mais profundo que esse.

Para concretizar estas estratégias é necessário adicionar ao sistema a seguinte informação:

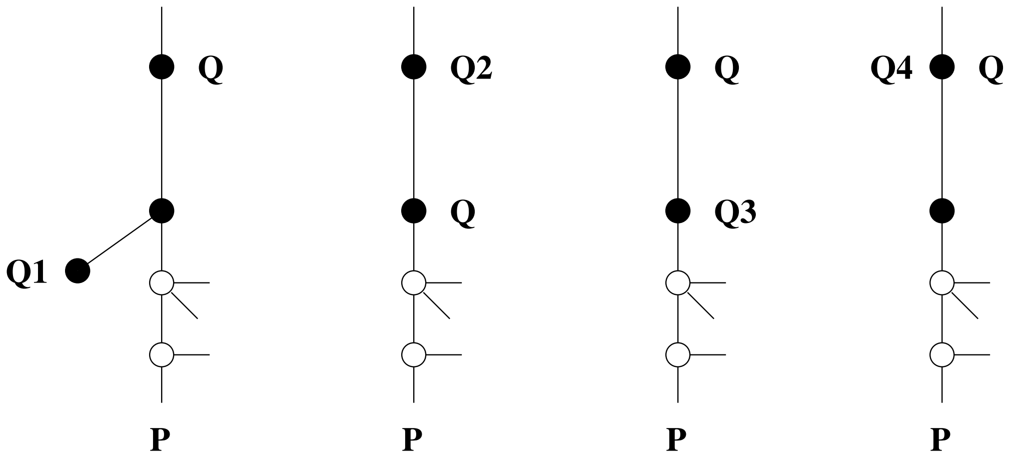
- Um registo local `carga`, associado a cada agente, contendo a quantidade de alternativas por explorar dos nós privados.
- Um registo local `nó_partilhado_corrente`, associado a cada agente, contendo uma referência para o nó partilhado mais profundo da sua actual ramificação da árvore de procura. Este nó não coincide necessariamente com o nó corrente, desde que este último seja privado.
- Um *bitmap* global `agentes_suspensos`, contendo os agentes do sistema correntemente no estado de suspensos.

A informação que as estruturas partilhadas possuem sobre os agentes que partilham um dado nó, passará a ser designada como `bitmap_de_agentes`.

2.2.3.1 Procura na Subárvore do Nó Corrente

Inicialmente, o agente suspenso Q, determina o conjunto de agentes ocupados, a partir da subárvore do nó corrente. Para isso, determina o conjunto de agentes ocupados (`bitmap_ocupados`) e suspensos (`bitmap_suspensos`) a partir da subárvore corrente, com base no *bitmap* global `agentes_suspensos` e no `bitmap_de_agentes` relativo ao `nó_partilhado_corrente`. De seguida, Q restringe o `bitmap_ocupados` aos agentes ocupados que estão mais próximos de si do que de outros agentes suspensos, do seguinte modo: para cada agente Q_i em `bitmap_suspensos`, se `nó_partilhado_corrente` de Q_i é mais profundo que `nó_partilhado_corrente` de Q, então remove de `bitmap_ocupados` os agentes no `bitmap_de_agentes` relativo ao `nó_partilhado_corrente` de Q_i. Por fim, Q escolhe de entre os restantes agentes em `bitmap_ocupados` aqueles que tiverem excesso de carga, isto é, pelo menos uma alternativa por explorar nos nós privados, e de entre esses aquele cuja carga seja a maior, consultando para isso os respectivos registos locais `carga`.

Na figura 2.4, Q representa o nosso agente suspenso, P representa um agente ocupado e os vários Q_i representam outros agentes suspensos. Utilizando o mecanismo anteriormente descrito, Q irá solicitar partilha de trabalho a P nas situações (a), (b) e (d), mas não em (c). Na situação (a), apesar do `nó_partilhado_corrente` de Q_1 ser mais profundo que o de Q, P não está na subárvore de Q_1 e daí não ficar excluído do `bitmap_ocupados` de Q. Contudo, caso Q_1 retrocedesse um nó e se colocasse na ramificação da árvore de procura de P, estaria melhor colocado que Q para partilhar o trabalho de P, já que a parte do estado de Q_1 consistente com o estado de P é maior do que a de Q. No entanto, permite-se que Q solicite trabalho a P sem levar em conta essa possibilidade porque em situações semelhantes mais complexas, seria custoso avaliar quando é que o agente Q_1 poderia realmente seleccionar o agente P para partilhar trabalho. Na situação (b) não existem agentes suspensos entre P e Q. Na situação (c) existe um outro agente suspenso entre P e Q. Q_3 é o agente suspenso mais perto de P, e daí ser ele quem deve solicitar partilha de trabalho a P. Na situação (d), tanto Q como Q_4 estão em igualdade de circunstância para solicitar trabalho de P, e daí que ambos o possam vir a fazer. Suponhamos que Q é quem primeiro o faz. Sendo assim, inviabiliza desde logo a possibilidade de que qualquer um outro agente o possa também fazer, dado que o agente P apenas pode responder a uma solicitação para partilha de trabalho de cada vez.



(a) Q solicita trabalho a P. **(b) Q solicita trabalho a P.** **(c) Q não solicita trabalho a P.** **(d) Q solicita trabalho a P.**

Figura 2.4: Solicitação de trabalho a agentes ocupados na subárvore do nó corrente.

2.2.3.2 Procura Fora da Subárvore do Nó Corrente

Quando Q não consegue encontrar nenhum agente ocupado com excesso de carga na subárvore do nó corrente, inicia fora dessa mesma subárvore a procura por um outro agente P nessas condições. Para evitar que outros agentes suspensos na mesma situação que Q possam seleccionar simultaneamente o agente P, colocam-se P e Q invisíveis para os restantes agentes. O suporte a este mecanismo é dado pelo *bitmap* global, `agentes_invisíveis`. Antes de retroceder na árvore de procura até o primeiro nó que contenha P, Q deve assinalar que pretende solicitar trabalho a P, marcando para isso os *bits* que correspondem a P e Q no *bitmap* `agentes_invisíveis`. Quando Q parar de retroceder, os dois *bits* voltam ao estado inicial. O *bitmap* `agentes_invisíveis` é usado apenas para evitar que outros agentes em circunstâncias semelhantes às de Q, percam tempo a retroceder para nós que contenham P, não podendo depois solicitar-lhe trabalho. No entanto, P pode ser solicitado para partilhar trabalho por um outro agente que não se encontre nessas condições. Por exemplo, P pode estar na subárvore de um outro agente suspenso.

O agente suspenso Q para determinar o conjunto de agentes ocupados fora da subárvore do nó corrente, que não se encontram mais próximo de outros agentes suspensos começa por determinar o conjunto de agentes ocupados e visíveis (`bitmap_ocupados_visíveis`) e o conjunto de agentes suspensos e visíveis (`bitmap_suspensos_visíveis`) fora da sua subárvore. Para isso, consulta os *bitmaps* globais `agentes_suspensos` e `agentes_invisíveis` e o `bitmap_de_agentes` relativo ao `nó_partilhado_corrente`. De seguida, remove do `bitmap_ocupados_visíveis` os agentes que estejam nas subárvores dos agentes do `bitmap_suspensos_visíveis`, isto é, para cada agente Q_i em `agentes_suspensos_visíveis`, remove de `bitmap_ocupados_visíveis` os agentes no `bitmap_de_agentes` relativo ao `nó_partilhado_corrente` de Q_i.

De notar que se o próprio agente Q estiver na subárvore de um outro agente suspenso Q_j, não deve seleccionar nenhum agente para solicitar partilha de trabalho enquanto Q_j se encontre suspenso. Isto porque Q_j está em melhor posição na árvore para ser ele a solicitar partilha de trabalho a qualquer um dos outros agentes. Posto isto, Q deve verificar antecipadamente se está na subárvore de algum dos agentes do `bitmap_suspensos_visíveis`. Uma forma de o fazer eficientemente é adicionar-se ao `bitmap_ocupados_visíveis` antes de remover algum dos agentes, e verificar se no final dessa operação continua ou não no *bitmap*. Se Q continuar no conjunto então pode prosseguir, caso contrário não. De entre os restantes agentes em `bitmap_ocupados_visíveis`, selecciona P, aquele que no momento tenha uma maior carga. Antes de lhe dirigir a consequente solicitação para partilha de trabalho, Q deve marcar os *bits* que correspondem a P e Q[7] no *bitmap* `agentes_invisíveis`, retroceder na árvore de procura até o primeiro nó que contenha o agente P e libertar os *bits* anteriormente assinalados.

[7] O *bit* correspondente ao agente Q é também marcado para permitir que os agentes suspensos na subárvore de Q possam simultaneamente seleccionar outros agentes ocupados.

Na figura 2.5, a aplicação da estratégia anteriormente descrita, permite que Q possa solicitar partilha de trabalho a P apenas na situação (c). Na situação (a), P está na subárvore de Q_1 e daí ficar excluído do `bitmap_ocupados_visíveis` de Q. Na situação (b), Q está na subárvore de Q_2, ou seja, Q_2 está mais perto de P do que Q. Na situação (c), nenhuma das condicionantes anteriores se verifica e tanto Q como Q_3 podem solicitar trabalho a P. Supondo que Q é quem primeiro encontra P visível, desde logo é ele quem toma vantagem para solicitar trabalho a P. Quando Q_3 posteriormente for consultar P, este já não se encontra visível.

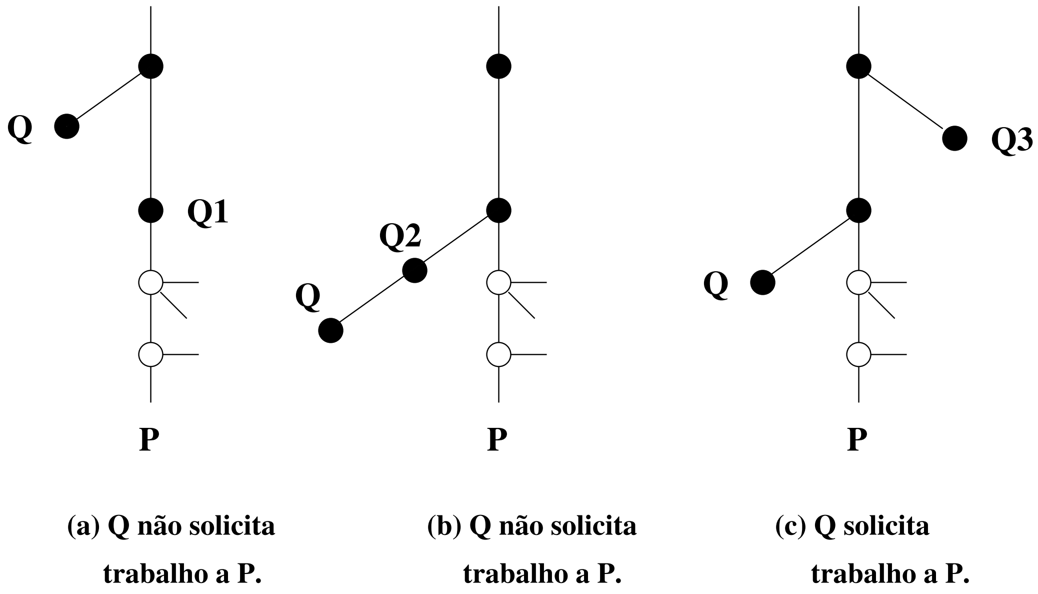


Figura 2.5: Solicitação de trabalho a agentes ocupados fora da subárvore do nó corrente.

2.2.4 Inexistência de Trabalho

A terceira componente do algoritmo básico do distribuidor de trabalho enunciado na secção 2.2.1, diz respeito à resolução da situação em que um dado agente suspenso, nem encontra trabalho disponível na sua região partilhada, nem encontra agentes ocupados com excesso de carga. Os agentes suspensos nestas circunstâncias, deverão colocar-se em nós que permitam que logo que novo trabalho seja gerado, este possa ser partilhado de modo a acarretar o mínimo custo possível. Na secção 2.2.4.1 são discutidas algumas heurísticas para tornar este esquema exequível.

O excesso de carga no modelo de cópia de ambientes é quantificado como sendo o número de alternativas privadas por explorar. Com este modo de medição, pode acontecer que por vezes exista trabalho por explorar em nós partilhados que não seja visível pelos agentes suspensos. No entanto, estas alternativas continuam a poder ser exploradas pelos agentes ocupados que a elas têm acesso e que ao não gerarem novo trabalho privado, nunca permitirão aos agentes suspensos poder delas tomar parte. Nesta situação, os agentes suspensos poderão permanecer suspensos para sempre, enquanto não exista excesso de carga num dado agente. Na secção 2.2.4.2 descreve-se um modo de detectar esta situação de forma a permitir que os agentes suspensos partilhem esse trabalho.

2.2.4.1 Melhor Posicionamento

A principal finalidade das heurísticas que se seguem é posicionar os agentes suspensos pela árvore de procura de forma a que, quando um agente ocupado gere novas alternativas privadas, um dos agentes suspensos possa solicitar a partilha desse trabalho da forma mais eficiente possível. Um agente suspenso Q deixa o seu nó corrente (isto é, retrocede na árvore) para se colocar numa melhor posição, apenas se uma das duas situações seguintes ocorrer:

1. Existem agentes ocupados fora da subárvore de Q e Q não está na subárvore de nenhum outro agente suspenso.
2. Todos os agentes na subárvore corrente estão suspensos.

Na primeira situação, cada agente suspenso irá impedir o retrocesso de todos os outros agentes suspensos da sua subárvore. Apenas o agente de topo dessa sequência deverá retroceder para o primeiro nó onde se encontrem todos os agentes ocupados do sistema, que não se encontrem mais próximos de outros agentes suspensos. Quando um agente suspenso se encontra mais próximo de um agente que pode gerar trabalho, a perda de desempenho inerente a uma possível operação de partilha de trabalho diminui. Deslocando alguns agentes suspensos para nós comuns aos agentes ocupados, permite que uma futura partilha de trabalho se concretize o mais cedo possível. Além disso, o efeito bloqueador de retrocesso referido anteriormente, permite que parte dos agentes suspensos não percam as boas posições que possam ocupar na árvore de procura.

Na segunda situação, permite-se que o agente suspenso retroceda até atingir um nó que possua pelo menos um agente ocupado. Daqui resulta o facto de se libertar de uma posição onde a possibilidade de poder vir a ser gerado novo trabalho era nula, em contraste com a nova posição onde isso passa a ser possível. Uma outra vantagem é que a nova posição coloca o agente suspenso mais perto dos agentes ocupados que se encontram fora da sua subárvore corrente, para o caso de lhes pretender solicitar trabalho.

A figura 2.6 representa uma situação em que as heurísticas aqui descritas são aplicadas. Das situações anteriormente referidas, os agentes Q_1 e Q_3 enquadram-se na primeira, Q_4 enquadra-se na segunda e Q_2 não se enquadra em nenhuma delas. Tanto para Q_1 como para Q_3, os agentes ocupados do sistema que não estão mais perto de outros agentes suspensos são P_2 e P_3. Sendo assim, ambos os agentes irão retroceder para o nó de topo da figura, o qual é o primeiro comum aos dois agentes ocupados. O agente Q_4 retrocede dois nós até atingir o nó em que também se encontra o agente ocupado P_3. O agente Q_2 mantém a sua posição e fica numa situação privilegiada caso P_1 gere novo trabalho. De referir ainda que os nós anteriormente ocupados por Q_3 e Q_4 foram removidos da figura porque deixaram de possuir qualquer agente do sistema.

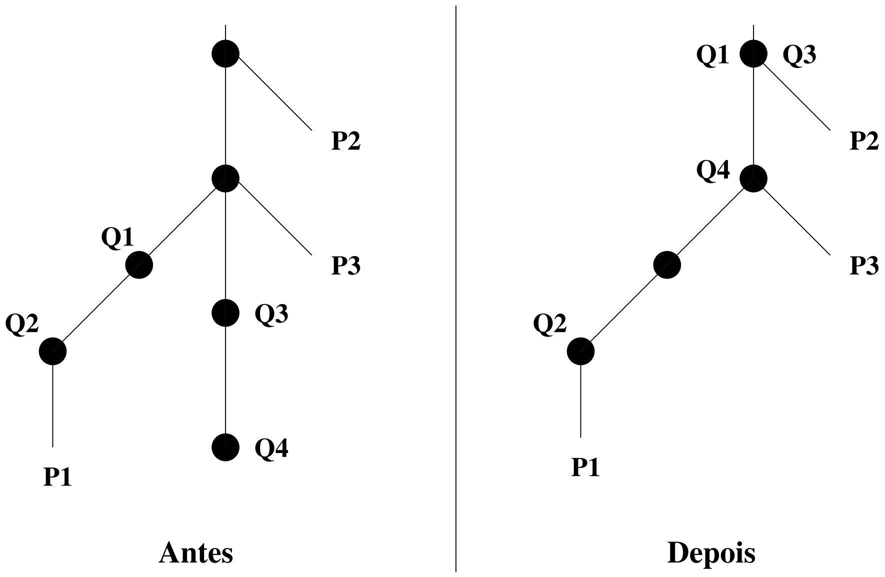


Figura 2.6: Melhor posicionamento na árvore de procura.

2.2.4.2 Procura de Trabalho Invisível

A estratégia para encontrar trabalho que é invisível para os agentes suspensos e conseguir com que este venha a ser partilhado com eles, consiste no seguinte: quando um agente suspenso Q, não consegue encontrar trabalho por explorar no sistema e existem agentes ocupados na sua subárvore, deve através de solicitações de partilha de trabalho a esses agentes, tentar ter acesso a trabalho invisível, pois são estes os potenciais agentes para conseguir esse acesso. Q deverá apenas fazer solicitações aos agentes que tenham estado ocupados por um certo período de tempo e que não tenham sido ainda solicitados por um outro agente do sistema. Se algum deles encontrar trabalho inexplorado na sua árvore, então deve-o partilhar com Q.

Para suportar esta ideia, é necessário um *bitmap* global `agentes_podem_aceder_trabalho`, que contém os agentes que podem ter acesso a trabalho partilhado num dado momento. Inicialmente encontra-se vazio, com todos os *bits* limpos. Em cada operação de partilha de trabalho, os dois *bits* correspondentes aos dois agentes envolvidos nessa partilha são marcados. Quando um agente ocupado é questionado para partilhar trabalho e rejeita essa solicitação, significa que não tem acesso a trabalho por explorar e dessa forma deve limpar o seu *bit* em `agentes_podem_aceder_trabalho`.

Existe também um *bitmap* local a cada agente, `bitmap_continua_ocupado`, que contém os agentes que se mantêm ocupados dentro da subárvore do agente suspenso por um certo período de tempo. Quando Q se coloca suspenso, inicializa o seu `bitmap_continua_ocupado` com os agentes que se encontram ocupados dentro da sua subárvore. Enquanto existirem agentes ocupados na sua subárvore, Q vai actualizando o `bitmap_continua_ocupado`, removendo os agentes que entretanto se colocaram suspensos. Depois de k[8] iterações investigando os agentes ocupados na sua subárvore, Q possui um conjunto potencial de agentes aos quais poderá solicitar partilha de trabalho invisível. Os *bits* que estejam marcados simultaneamente no `bitmap_continua_ocupado` e no *bitmap* global `agentes_podem_aceder_trabalho` correspondem a esses agentes. Q poderá então fazer solicitações a qualquer um desses agentes, mas apenas o fará àqueles que não se encontrem mais próximo de um outro agente suspenso. Caso o agente solicitado não tenha acesso a trabalho invisível, recusa a solicitação para partilha de trabalho e limpa o seu *bit* no `agentes_podem_aceder_trabalho`.

Esta estratégia permite que todo o agente ocupado que não gere mais trabalho privado, após uma operação de partilha de trabalho e que possua nós vivos na sua ramificação corrente, possa ser questionado para partilhar trabalho pelo menos uma vez.

[8] No YapOr, por defeito k é instanciado com o valor 10. No entanto é possível redefinir esse valor no início do sistema.

Capítulo 3

Extensão do Yap para suportar o YapOr

Este capítulo expõe de uma forma pormenorizada, o conjunto de extensões e consequentes alterações que foram necessárias introduzir no Yap de modo a suportar o sistema implementado, o YapOr.

3.1 Organização de Memória

Começaremos por descrever a forma como está organizada a memória do sistema Yap e quais foram as alterações efectuadas para suportar eficientemente o modelo de cópia de ambientes do sistema YapOr. Descreve-se ainda a forma como os diferentes agentes inicializam e lidam com a memória e o modo como o processo de cópia incremental se enquadra na disposição de memória do YapOr.

3.1.1 Organização de Memória no Yap

Na linha habitual dos compiladores de Prolog, o sistema Yap estrutura a sua memória em quatro áreas fundamentais de trabalho [SC88]: heap, trilha, global e local (ver figura 3.1).

Heap Corresponde à área de memória onde é guardada informação estática cuja vida não depende da execução do programa. Inclui informação sobre a base de dados interna e sobre os átomos, os predicados e o código a eles associado.

Trilha Guarda-se uma referência na trilha das variáveis que vão sendo instanciadas. Esta informação é necessária ao mecanismo normal de retrocesso para que seja possível recuperar um estado anterior da computação.

Global Serve para guardar informação relativa às variáveis que vão sendo criadas e aos termos que vão sendo utilizados.

Local Nesta área é mantida informação relativa aos pontos de escolha e aos ambientes. Um ambiente serve para guardar informação relativa às variáveis que ocorrem mais do que uma vez nos objectivos do corpo de uma cláusula.

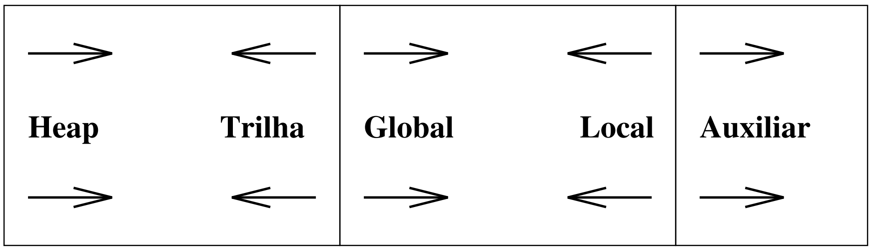


Figura 3.1: Disposição das áreas de memória no Yap.

As áreas definidas como heap e global, correspondem respectivamente às que anteriormente foram referidas como área de código e pilha de termos. Além destas quatro áreas e à semelhança da WAM, existe também uma área auxiliar usada pelo ambiente de suporte, em particular para o compilador, analisador sintáctico, base de dados interna e algoritmos de indexação e de unificação de estruturas.

A solução de colocar as quatro áreas principais a crescerem em direcções opostas, visa obter um melhor aproveitamento de memória (bem patente no facto de não existir um limite predefinido de crescimento, já que uma pilha pode crescer tanto quanto a pilha oposta não o tenha feito) e reduzir o número de testes necessários para avalizar se as áreas com crescimento oposto chocam.

3.1.2 Organização de Memória no YapOr

Conforme se pode observar pela figura 3.2, na organização de memória do YapOr destacam--se dois grandes espaços de endereçamento: o espaço global e o conjunto dos espaços locais.

- O **espaço global** serve em grande parte de suporte ao paralelismo do sistema, pois é nele que estão as principais estruturas de apoio ao mesmo. Este espaço está dividido em quatro partes principais, a saber: área de código; área de informação global, que serve de suporte aos diversos tipos de sincronização entre os agentes; área de criação das estruturas partilhadas; área onde se vão guardando as soluções do objectivo em exploração.
- O grande **espaço local** é composto por um conjunto de subespaços locais, cada um deles representando um dos agentes do sistema. Cada um destes subespaços, além das pilhas da WAM contém uma área de informação local na qual se encontram os dados do agente que são necessários ao decurso do processo paralelo.

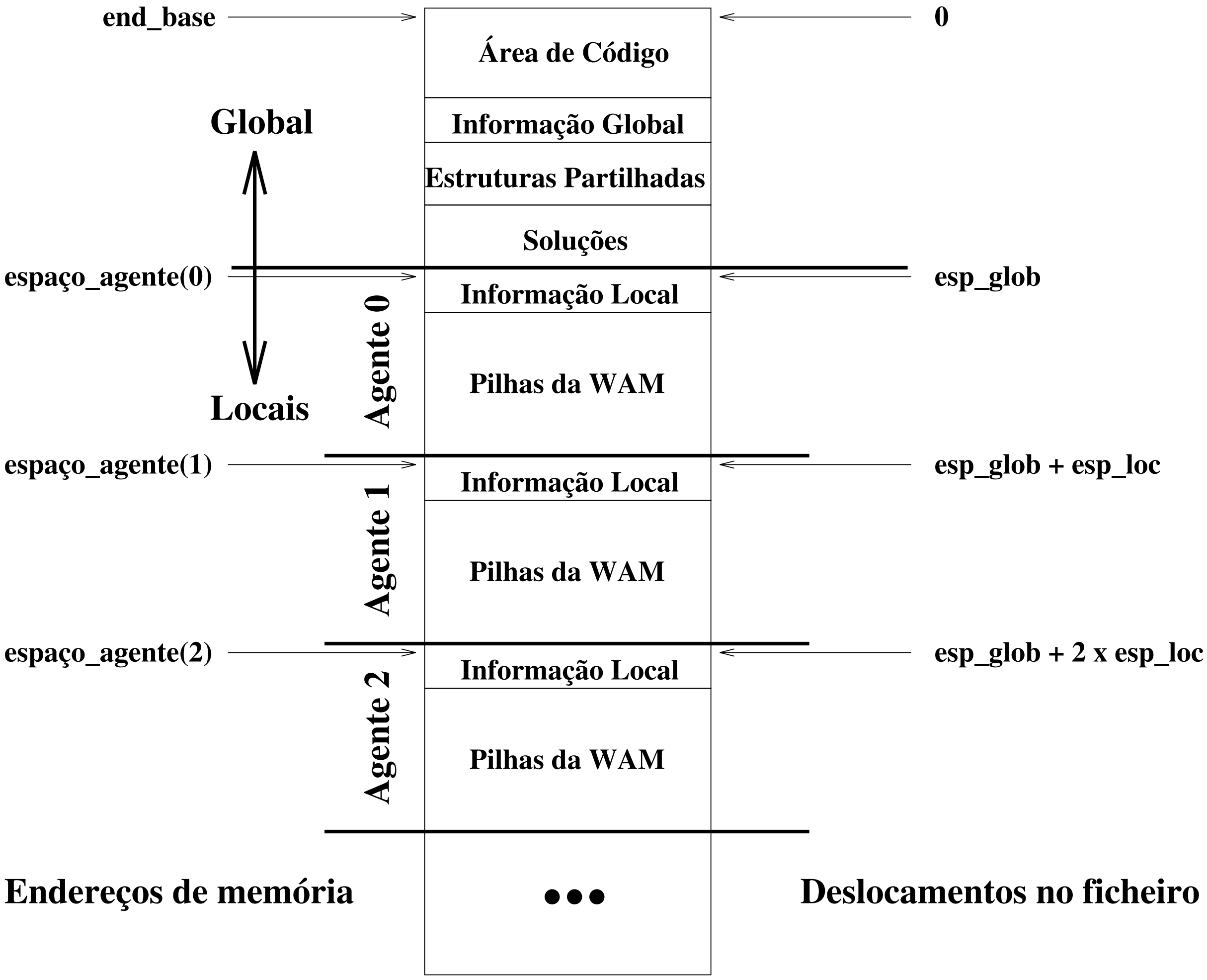


Figura 3.2: Organização de memória no YapOr.

A estrutura das pilhas da WAM é idêntica à do próprio Yap, a menos da área de código que, como já foi referido, passou a estar isolada no espaço global, o que levou a que a trilha passasse para o fundo do conjunto opondo-se à pilha auxiliar (ver figura 3.3). Os espaços locais de cada agente funcionam assim como subsistemas de execução de Prolog sequencial.

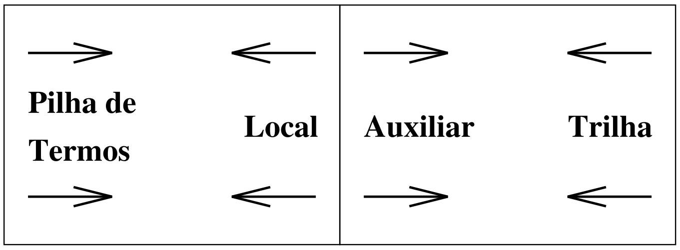


Figura 3.3: Disposição das pilhas da WAM dos espaços locais.

Para se garantir uma maior eficiência do funcionamento do esquema de cópia incremental, tirou-se partido da grande permissividade e funcionalidade que nos fornece o uso do mecanismo de mapeamento de memória *mmap*[1] [Ste92, páginas 407-413], para implementar a disposição de memória anteriormente referida. Este mecanismo permite mapear conjuntos contíguos de endereços de memória numa zona, também ela contígua, de um dado ficheiro em disco. Posto isto, uma operação de leitura ou de escrita de dados nos endereços mapeados, corresponde de facto a uma leitura e a uma escrita dos mesmos nas respectivas posições da zona mapeada do ficheiro. Através da chamada à função *fork*[2] [Ste92, páginas 188-193], o conjunto de endereços de memória anteriormente mapeado é herdado pelo processo filho, o que corresponde implicitamente a tornar esse conjunto de endereços partilhado por ambos os processos.

No processo de inicialização do YapOr, uma das tarefas cruciais ao funcionamento posterior do sistema é o mapeamento inicial da memória. Seja `fich`, o ficheiro onde será mapeada toda a memória e suponhamos que o espaço de endereçamento global e cada espaço de endereçamento local ocupam respectivamente `esp_glob` *bytes* e `esp_loc` *bytes*. O agente 0 (aquele que dá início ao funcionamento do sistema) começa por mapear o espaço de endereçamento global num endereço predefinido, `end_base`, no início do ficheiro `fich`. De seguida, cada espaço local respeitante a cada um dos agentes do sistema, é mapeado no endereço posterior ao último anteriormente mapeado na posição seguinte do ficheiro `fich`

[1] O mecanismo de mapeamento de memória *mmap* passou a fazer parte integral das implementações de UNIX, desde as distribuições SVR4 e 4.3+BSD.

[2] A função *fork* pertence às bibliotecas das implementações UNIX e permite criar novos processos.

(ver figura 3.4). Ao mesmo tempo que se efectua esta tarefa, o agente 0 inicializa as variáveis `espaço_agente()` e `offset_agente()`, que referenciam respectivamente o endereço de memória de início do espaço local de cada agente e o deslocamento em *bytes* entre os endereços de memória de início do espaço local do agente 0 e o espaço local de cada agente.

```
mapear (end_base, esp_glob, fich, 0);
for (i = 0 to número de agentes) {
  espaço_agente(i) = end_base + esp_glob + i * esp_loc;
  offset_agente(i) = espaço_agente(i) - espaço_agente(0);
  mapear (espaço_agente(i), esp_loc, fich, esp_glob + i * esp_loc);
}
```

Figura 3.4: Mapeamento inicial da memória pelo agente 0.

Na figura 3.2 está representada a relação entre os endereços de memória `end_base`, `espaço_agente(0)`, `espaço_agente(1)` e `espaço_agente(2)` e os correspondentes deslocamentos `0`, `esp_glob`, `esp_glob + esp_loc` e `esp_glob + 2`×`esp_loc` no ficheiro `fich`.

Numa segunda fase, o agente 0 através de chamadas sucessivas à função *fork*, tantas quantas o número de agentes pretendidos menos uma, dá lugar à criação de uma série de novos processos, os quais correspondem aos restantes agentes do sistema. Por conseguinte, no YapOr a cada agente do sistema corresponde um simples processo do sistema operativo e não um processador dedicado. No entanto, para se obter boas performances é desejável que em tempo de execução paralelo o sistema operativo distribua cada um dos processos por cada um dos diferentes processadores da máquina.

Por sua vez, cada novo agente remapeia todos os espaços locais de memória, de modo a que todo o espaço local a cada agente, dum ponto de vista individual tenha início no mesmo endereço, isto é, se para um determinado agente o seu espaço local tem início no endereço `loc_end`, então do ponto de vista de um qualquer outro agente, o seu correspondente espaço local também deve ter início em `loc_end` (ver figura 3.5). No entanto, tudo isto é conseguido sem que as zonas relativas a cada agente no ficheiro `fich`, anteriormente mapeadas pelo agente 0, sejam alteradas. Para isso, cada um dos novos agentes apenas mapeia endereços de memória diferentes nas zonas pretendidas do ficheiro `fich`.

No final desta operação, do ponto de vista do agente 1, o início do seu espaço local seria no endereço colocado em `espaço_agente(1)`, idêntico ao valor de `end_base + esp_glob`, mapeado na posição `esp_glob + esp_loc` do ficheiro `fich`, enquanto que poderia aceder

```
for (i = 0 to número de agentes) {
  espaço_agente(i) = end_base + esp_glob + (i-p mod número de agentes) * esp_loc;
  remapear(espaço_agente(i), esp_loc, fich, esp_glob + i * esp_loc);
}
for (i = 0 to número de agentes) {
  offset_agente(i) = espaço_agente(i) - espaço_agente(p);
}
```

Figura 3.5: Remapeamento dos espaços locais de memória pelo agente P.

ao início do espaço local do agente 2 através da consulta do endereço colocado em `espaço_agente(2)`, idêntico ao valor de `end_base + esp_glob + esp_loc`, mapeado na posição `esp_glob + 2`×`esp_loc` do ficheiro `fich`. Por outro lado, do ponto de vista do agente 2, o início do seu espaço local seria também no endereço `end_base + esp_glob` mapeado na posição `esp_glob + 2`×`esp_loc` do ficheiro `fich`, a mesma posição que permite o acesso do agente 1 ao espaço local do agente 2.

Qual é então a vantagem de todo este esquema de mapeamento de memória, relativamente à mecânica da cópia incremental? É a simplificação efectiva do mecanismo de cópia de porções das pilhas de execução da WAM de um dado agente para outro. Utilizando a função *memcpy*[3] [Ste92, páginas 411-413], podemos facilmente transferir os dados pretendidos de um agente para outro. Na figura 3.6, o agente X copia para o seu espaço local uma porção de memória do espaço local do agente Y através do seguinte procedimento:

```
memcpy (Base, Base + offset_agente(Y), número de bytes a copiar);
```

enquanto que o agente Y move do seu espaço local para o espaço local do agente X uma outra porção de memória através de:

```
memcpy (Base + offset_agente(X), Base, número de bytes a copiar);
```

Como o objectivo fundamental da cópia incremental é obter um mesmo estado computacional, as cópias de porções de memória fazem-se de um dado endereço do espaço local de um agente para o mesmo endereço relativo no outro agente. Nas posições de memória referentes às pilhas de execução da WAM, são por vezes colocados endereços relativos aos

[3] A função *memcpy* pertence às bibliotecas das implementações UNIX e permite copiar uma dada porção de memória para uma outra zona da mesma.

próprios endereços em que as pilhas estão a funcionar. A grande vantagem deste esquema de mapeamento de memória assenta pois neste aspecto. Após uma operação de cópia, não é preciso ajustar os endereços, das posições de memória que possuem referências de endereços das próprias pilhas, de modo a fazerem sentido no espaço local do outro agente, porque do ponto de vista de cada agente, o endereço base sob o qual estão a trabalhar é o mesmo.

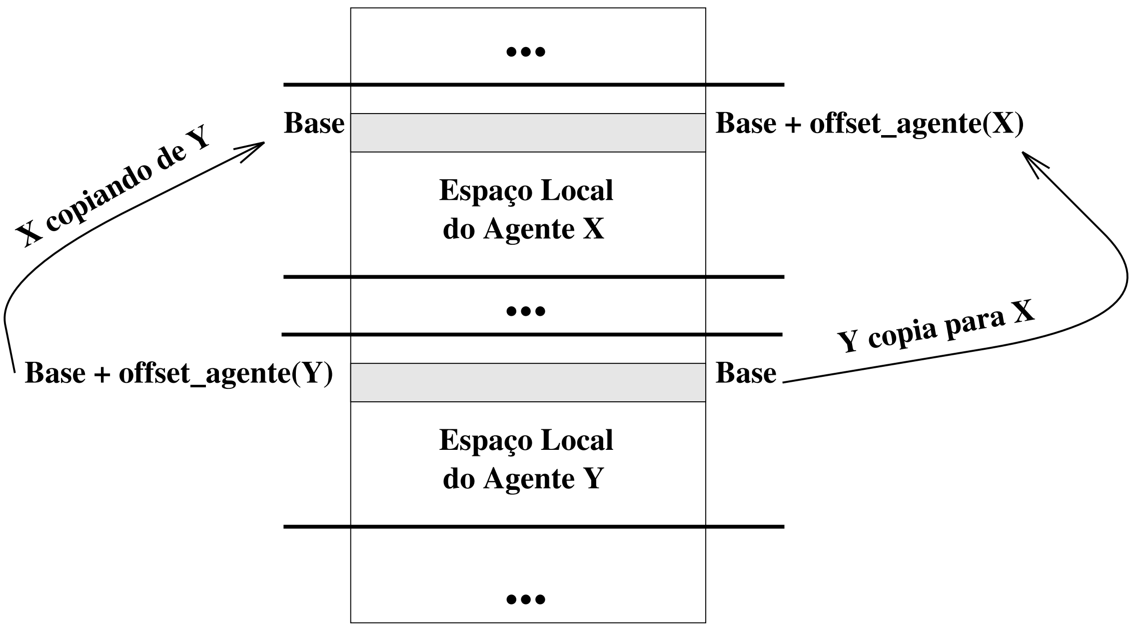


Figura 3.6: Cópia de porções de memória entre agentes.

3.2 Pontos de Escolha

Em Prolog, o principal objectivo da criação de um ponto de escolha é guardar o estado computacional do momento, de forma a que numa fase posterior da computação e por retrocesso, esse estado possa ser recuperado para a exploração de outras alternativas. Com a introdução da exploração paralela de alternativas, os pontos de escolha passaram a poder ser partilhados entre os vários agentes e daí tornou-se necessário adaptá-los de forma a responderem eficientemente a esta nova realidade.

Das questões essenciais relacionadas com esta nova realidade, destacam-se as seguintes:

- Quais os novos campos a introduzir nos pontos de escolha?
- Como é feita a ligação entre os pontos de escolha e as estruturas partilhadas a eles associadas?
- Quando e como é criado o ponto de escolha de topo?

A figura 3.7 representa a estrutura dos novos pontos de escolha. Os sete primeiros campos são herdados do Yap, enquanto que os restantes dois foram introduzidos no YapOr. Num dos novos campos guarda-se o apontador para a estrutura partilhada associada ao ponto de escolha, enquanto que no outro mantém-se uma referência actualizada do registo local *carga* (ver secção 3.4).

CP_N		**Aridade do Predicado**
CP_TR		**Apontador para a Trilha**
CP_ALT		**Próxima Alternativa**
CP_H		**Apontador para a Pilha de Termos**
CP_B		**Ponto de Escolha Anterior**
CP_E		**Ambiente Anterior**
CP_CP		**Continuação do Código do Programa**
CP_EP		**Apontador para a Estrutura Partilhada**
CP_ALE		**Alternativas Locais por Explorar**

Figura 3.7: Estrutura dos pontos de escolha no YapOr.

Uma das operações fundamentais da partilha de trabalho é tornar os pontos de escolha privados em partilhados. A figura 3.8 esquematiza a relação entre um desses pontos de escolha antes e depois dessa operação, bem como a ligação resultante com a correspondente estrutura partilhada entretanto criada. Os campos `CP_ALT` e `CP_EP` do ponto de escolha passam a apontar respectivamente para a pseudo-instrução `procura_trabalho` (ver secção 3.3) e para a estrutura partilhada criada. Por sua vez, a estrutura partilhada é inicializada do seguinte modo: o campo `próxima_alternativa` fica com o apontador `ALT` que se encontrava anteriormente no campo `CP_ALT` do ponto de escolha (ou seja, o controle das alternativas por explorar passa para a estrutura partilhada); no campo `bitmap_de_agentes` são marcados os *bits* correspondentes aos dois agentes envolvidos na operação de partilha; o campo `nó_vivo_mais_próximo` fica a referenciar o ponto de escolha anterior; o campo `fecho_de_acesso` fica livre para permitir o acesso à estrutura.

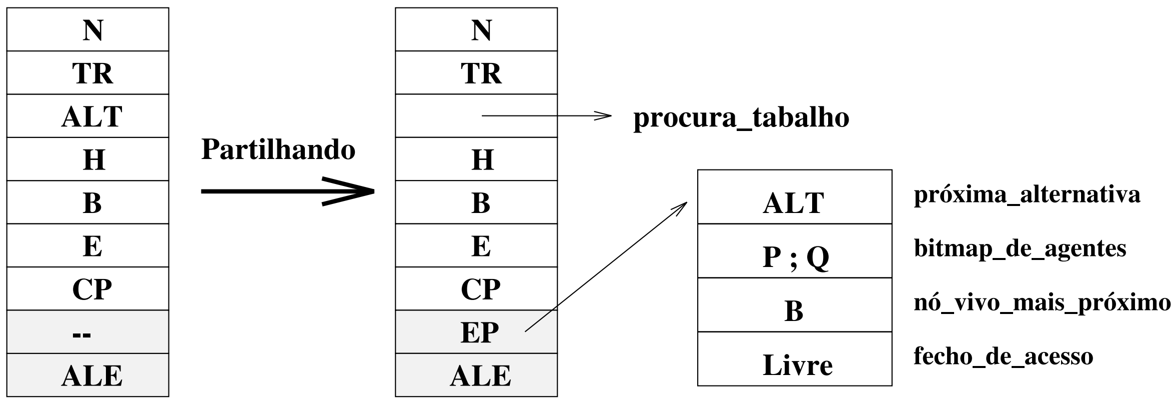


Figura 3.8: Partilha de um ponto de escolha.

De referir ainda, que numa operação de partilha de trabalho entre P e Q, podem existir alguns pontos de escolha que já tenham sido anteriormente partilhados, aos quais P tem acesso e Q ainda não. Neste caso, P apenas necessita de marcar o *bit* correspondente a Q no `bitmap_de_agentes` das estruturas partilhadas desses pontos de escolha.

O ponto de escolha de topo é criado pelo agente 0 e precede a criação dos restantes agentes do sistema. Este ponto de escolha é desde logo inicializado como se de um ponto de escolha partilhado se tratasse. A estrutura partilhada a ele associada é inicializada do seguinte modo: o campo `próxima_alternativa` fica com um valor por defeito `null`, que indica a ausência de alternativas por explorar; no campo `bitmap_de_agentes` são marcados os *bits* correspondentes a todos os agentes do sistema; o campo `nó_vivo_mais_próximo` fica com o valor de defeito `null` (rever secção 2.2.2); o campo `fecho_de_acesso` fica livre. Jamais durante a execução dos diversos objectivos estes valores são alterados.

O ponto de escolha de topo serve essencialmente de referência como topo da cadeia de pontos de escolha.

3.3 Novas Pseudo-Instruções

No YapOr foram introduzidas unicamente duas novas instruções. Uma delas foi, a já referida, `procura_trabalho` e a outra foi `procura_trabalho_primeira_vez`. A função fundamental destas instruções é estabelecer a necessária ligação entre o precedente processo sequencial e as extensões de suporte ao novo sistema, ao nível da execução do código de um dado programa.

No Yap, cada predicado é composto por um conjunto de cláusulas. Após ser compilada, cada cláusula tem associada a si uma sequência de instruções WAM que representam, a nível abstracto, o conjunto de operações a executar. Estas cláusulas são ligadas entre si através de uma lista ligada de apontadores, ou seja, a primeira cláusula de um dado predicado tem uma referência para o local onde se encontra a segunda, a segunda para a terceira e assim sucessivamente. Sempre que um predicado é chamado para ser executado e este contém mais do que uma cláusula passível de ser executada[4], um ponto de escolha deve ser criado. Nesse ponto de escolha, além do estado computacional corrente é guardada uma referência para a próxima alternativa a explorar. Sempre que o mecanismo normal de retrocesso é executado, o anterior estado computacional é restabelecido e a alternativa guardada passa a ser processada, sem que antes, no seu lugar seja colocada uma referência para a alternativa que lhe sucederá. Caso seja tomada a última alternativa o ponto de escolha correspondente deve ser removido.

As novas instruções são apelidadas de pseudo-instruções, porque na verdade elas nunca são geradas no processo anteriormente referido, como fazendo parte de um conjunto predefinido de instruções resultantes de um dado código de programa. Independentemente disso, estas são introduzidas à medida que vão sendo necessárias no progresso do próprio processo paralelo.

A instrução `procura_trabalho`, como foi referido na secção 3.2, é introduzida no campo `CP_ALT` de um ponto de escolha quando este está a ser partilhado. Deste modo, sempre que um agente retrocede para um ponto de escolha partilhado, em lugar de tomar a próxima alternativa por explorar, é forçado a executar a pseudo-instrução `procura_trabalho`. A execução desta instrução permite obter de um modo sincronizado com os outros agentes, que também partilham a estrutura partilhada correspondente, o acesso a uma nova alternativa. Todo este processo funciona sem que tenha sido necessário alterar a mecânica herdada do Yap, inerente ao mecanismo normal de retrocesso.

No início do sistema e sempre que a exploração de um determinado objectivo termina, todos os agentes à excepção do agente 0, são convidados a executar a instrução `procura_trabalho_primeira_vez`. Esta instrução coloca os diferentes agentes num estado de espera por uma sinalização predefinida a ser enviada pelo agente 0, cuja função é indicar o início da exploração de um novo objectivo. Ao invés, o agente 0 além de sinalizar os

[4]Independentemente de se considerar o processo de indexação ou não. A indexação é uma optimização, que nos permite limitar o número de alternativas passíveis de serem executadas, mediante o tipo de argumentos de chamada do predicado.

vários agentes do início da exploração de um novo objectivo, é responsável por fornecer as diversas soluções resultantes da execução desse objectivo, e por gerir a *interface* com o utilizador até que este ordene a exploração de um outro objectivo.

3.4 Carga de um Agente

O registo local *carga* associado a cada agente dá-nos uma medida da quantidade de alternativas privadas por explorar que o agente possui. Para tornar possível a computação desse valor em tempo de execução, foi necessário introduzir algumas alterações na estrutura base do Yap. Ao já referido campo `CP_ALE` que passou a fazer parte dos pontos de escolha, junta-se a extensão do código de algumas das instruções WAM. O objectivo desta secção é expor como é realmente calculado e mantido o valor do registo local *carga*, em função das alterações referidas.

Partindo de uma dada alternativa `Alt` de um certo predicado e mediante a estrutura implementada no Yap, a única forma de obter o número de alternativas por explorar para além de `Alt`, é percorrer a sequência de referências entre alternativas, partindo de `Alt` até atingir a última delas. Por conseguinte, para calcular de um modo eficiente o valor da carga de um dado agente num dado momento, foi necessário introduzir algo de novo à estrutura existente.

As instruções que contêm referências para outras alternativas, estão directamente relacionadas com a criação e manuseamento de pontos de escolha. Todas elas tiveram então, que ser ligeiramente alteradas de modo a responder a esta necessidade.

Da figura 3.9 pode depreender-se a funcionalidade e finalidade do novo campo `ape` (alternativas por explorar) introduzido nas referidas instruções. O código relativo a este novo campo é gerado para as instruções já mencionadas, tanto na fase normal de tradução de um programa em Prolog para código WAM, como na fase posterior de optimização por indexação. Como a pseudo-instrução `procura_trabalho` está unicamente associada a nós partilhados, o seu campo `ape` possui sempre o valor 0.

Já vimos que nesta implementação do YapOr, estamos apenas interessados em quantificar o número de alternativas privadas por explorar. A quantidade de alternativas partilhadas por explorar, varia constantemente com o decorrer do processo paralelo por intervenção dos outros agentes. Manter uma medida correcta dessa quantidade acarretaria um enorme

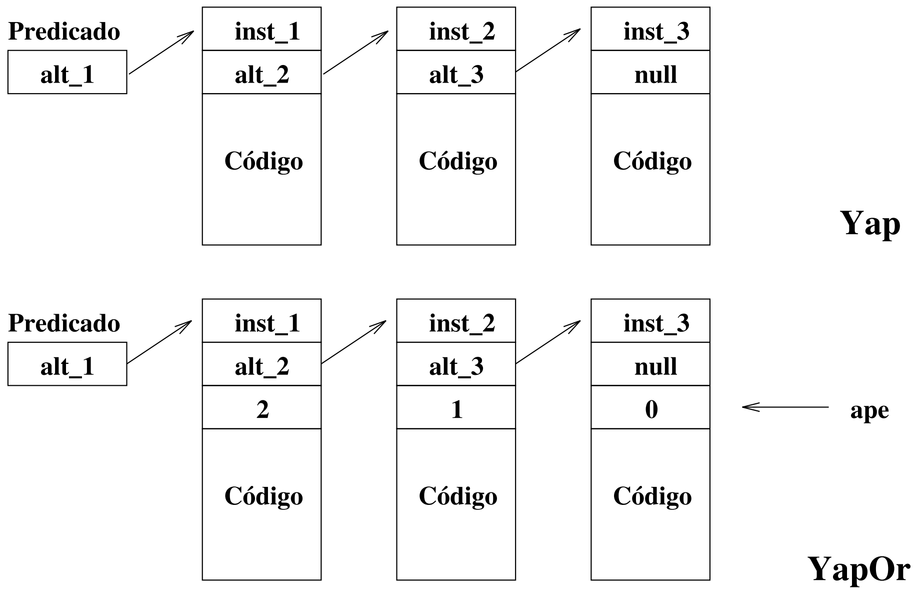


Figura 3.9: O novo campo `ape` na estrutura de dados dos predicados.

custo ao sistema e daí a justificação da sua não inclusão. Apesar disso, a carga privada de um dado agente também varia muito frequentemente. Como este valor é utilizado apenas para a obtenção de uma melhor selecção de agentes no processo de procura de trabalho, existe um compromisso entre o valor correcto de carga e a eficiência do processo paralelo, isto é, existe sempre um valor aproximado reflectindo a carga de um dado agente num dado instante, sendo que o valor exacto dessa quantidade é apenas calculado quando da criação de um novo ponto de escolha.

Para tornar possível esse cálculo, é guardado no campo `CP_ALE` de cada ponto de escolha um valor de carga relativo às alternativas privadas por explorar nos pontos de escolha precedentes. Esse valor corresponde à soma do campo `CP_ALE` do ponto de escolha imediatamente anterior com o valor do campo `ape` da alternativa colocada em `CP_ALT` desse mesmo ponto de escolha mais um (ver figura 3.10). A não inclusão de valores relativos ao ponto de escolha corrente tem como objectivo, evitar a actualização constante desse valor sempre que o mecanismo normal de retrocesso tiver lugar. Finalmente, o valor a colocar no registo local `carga` é calculado como sendo a soma do valor calculado para `CP_ALE` mais o valor do campo `ape` da alternativa em execução. Este valor é alterado sempre que o agente retrocede na árvore de procura. No entanto, por razões de eficiência, como já foi referido, apenas é realmente actualizado quando da criação de um novo ponto de escolha.

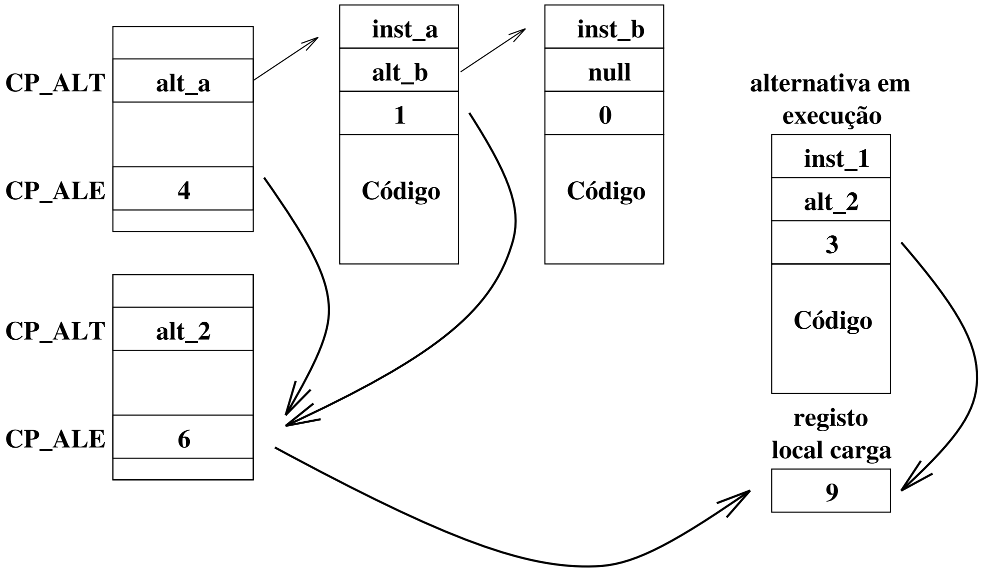


Figura 3.10: Cálculo da carga privada de um agente.

3.5 A Área de Código e o Processo de Indexação

A área de código do YapOr insere-se no espaço global de endereçamento, sendo por isso partilhada por todos os agentes do sistema. Uma das primeiras prioridades foi sincronizar o acesso a esta área de forma a manter a coerência dos dados. O agente 0 é o responsável por quase todas as modificações efectuadas nesta área, as quais, vulgarmente se inserem em actividades exteriores ao processo paralelo em si. A única operação efectuada em tempo de execução paralela, que pode ser executada por qualquer um dos agentes do sistema, é a indexação de um dado predicado. Um predicado é indexado apenas uma única vez quando da sua primeira chamada. Esta actividade além de relativamente rápida é realizada escassas vezes e daí que a sua influência na performance do processo paralelo seja desprezável.

Sempre que uma nova cláusula é introduzida no sistema, esta é transformada de modo a que por ele possa ser tratada. Inicialmente e após uma série de subfases é compilada para uma sequência de instruções WAM, as quais são guardadas na área de código. Esta sequência além do código respeitante à própria cláusula, inclui um pequeno cabeçalho. Este cabeçalho é utilizado na fase seguinte para estabelecer a ligação, através de uma lista ligada de apontadores (rever figura 3.9), com as cláusulas já existentes no sistema relativas ao mesmo predicado.

Antes da primeira chamada de execução, um qualquer predicado não está associado ao seu verdadeiro código compilado, mas sim a uma pseudo-instrução de indexação. Quando o predicado é invocado pela primeira vez, o sistema é forçado a executar essa pseudo-instrução dando assim lugar ao processo de indexação. O código de indexação entretanto gerado é também ele colocado na área de código e o predicado em questão fica a ele associado. Finalmente, todas as futuras invocações desse predicado passam a executar esse mesmo código. Mediante o tipo de argumentos de chamada do predicado, o código de indexação limita o número de cláusulas a considerar para a execução do verdadeiro código compilado.

No acesso ao processo de indexação de um determinado predicado foi necessário introduzir um mecanismo de fecho. Este mecanismo garante que apenas o primeiro agente a invocar um determinado predicado fica responsável pela sua indexação, o que evita possíveis indexações simultâneas por parte de outros agentes. Todos os outros agentes só poderão executar esse predicado após a conclusão do processo de indexação.

No processo de indexação herdado do Yap, foi necessário evitar a geração de um conjunto de instruções daí resultantes que se tornaram incompatíveis com o eficiente funcionamento do YapOr. Esse conjunto comporta instruções unicamente relacionadas com a optimização do processo natural de criação de pontos de escolha. Por vezes, um novo ponto de escolha coincide em muitos dos seus campos com os valores guardados no ponto de escolha precedente. As referidas instruções permitem optimizar esta situação, optando por expandir o ponto de escolha precedente com os novos campos que são diferentes, em lugar da criação de um novo ponto de escolha. Esta situação tornou-se insustentável no processo de partilha de pontos de escolha, porque levava a inconsistências relativamente à fiel representação da árvore de procura entre diferentes agentes, e daí a sua exclusão. No entanto, é possível idealizar um esquema que permita continuar a suportar este tipo de instruções, mas que acarretaria para o sistema custos mais elevados do que os créditos introduzidos por tais instruções.

3.6 Procura de Todas as Soluções

O sistema Yap, após a invocação de um novo objectivo, termina a sua computação logo que a primeira solução é encontrada. Poderá continuar à procura das restantes soluções, de uma forma sequencial, caso o utilizador assim o ordene. Por seu lado, o sistema YapOr, após a invocação de um novo objectivo, só termina a sua exploração quando percorrer toda a sua árvore de procura e encontrar todas as soluções possíveis.

Após as inicializações fundamentais ao funcionamento do sistema, o agente 0 inicia por defeito a execução do predicado '`$live`', que se encontra definido no ficheiro `boot.yap`. Este ficheiro é carregado no início do sistema e contém definida toda a *interface* com o utilizador. Para tornar possível a exploração contínua de toda a árvore de procura na execução de um objectivo, foi necessário adaptar parte do código de `boot.yap`.

A figura 3.11 dá-nos a conhecer a parte essencial desse novo código. Os predicados `solução`, `sucesso` e `insucesso` não são predicados normais de Prolog porque na realidade estão implementados em C[5]. À medida que o utilizador vai introduzindo novos comandos, através da referida *interface*, o agente 0 vai processando-os. Se o utilizador indica a execução de um objectivo, então o agente 0 atinge a execução do código da figura 3.11, pelo predicado `yapor`. Por defeito, apenas o agente 0 explora esse objectivo. Para um objectivo ser explorado em paralelo, por todos os agentes do sistema, é necessário introduzir o predicado `par`, na linha de comandos, precedendo o objectivo. Por exemplo, o comando para a exploração em paralelo do objectivo `a(X)` seria '`?- par, a(X).`'.

```
yapor(Objectivo, Variáveis) :- yapor_pergunta(Objectivo, Variáveis), fail.

yapor_pergunta(Objectivo, [ ])        :- !, yapor_sucesso(Objectivo).
yapor_pergunta(Objectivo, Variáveis) :- call(Objectivo), solução(Variáveis).

yapor_sucesso(Objectivo) :- call (Objectivo), !, sucesso.
yapor_sucesso(_)         :- insucesso.
```

Figura 3.11: Código Prolog adaptado em `boot.yap`.

No corpo do predicado `yapor` invoca-se inicialmente o predicado `yapor_pergunta` e duas situações podem ocorrer: ou o objectivo em questão não contém qualquer variável ou então contém uma ou mais variáveis.

- No caso de não conter variáveis trata-se apenas de verificar do seu sucesso ou insucesso. Isto é concretizado na prática através do predicado `yapor_sucesso` que pode originar ou a execução de `sucesso`, o qual guardará uma referência indicando sucesso, ou de `insucesso`, o qual guardará uma referência de não sucesso. No

[5]O Yap permite definir em C o código a executar por certos predicados de Prolog. Para isso, na fase de inicialização do sistema, basta executar a função em C `InitCPred()` e indicar como argumentos o predicado em Prolog e a função em C que deve ficar associada ao predicado. Posteriormente, sempre que o predicado for invocado, o sistema passa o controle de execução para a referida função.

caso paralelo, basta que um dos agentes guarde uma referência de sucesso para se considerar o sucesso do objectivo.

- Se pelo contrário, o objectivo possui variáveis a serem instanciadas é necessário guardar as soluções que vão sendo encontradas. O agente 0 começa por executar o predicado `call` e dá início à exploração do objectivo em questão. No caso paralelo é desde logo executado o predicado `par`. Este predicado dá início ao processo paralelo e sinaliza todos os outros agentes desse início. Sempre que um agente encontra uma solução, o que equivale a executar o predicado `call` com sucesso, executa o predicado `solução`. Este predicado cria uma nova estrutura na área de soluções da zona global de memória (rever figura 3.3), e coloca aí a solução encontrada. De seguida, o agente volta ao predicado `yapor` e executa a instrução `fail`. Esta instrução obriga o agente a retomar a exploração das alternativas pendentes nos seus pontos de escolha. Quando toda a árvore de procura tiver sido explorada, todos os agentes, à excepção do agente 0, são forçados a executar a pseudo-instrução `procura_trabalho_primeira_vez` (rever secção 3.3) e voltam ao estado inicial. Entretanto o agente 0 como não encontra também ele mais alternativas, prossegue a sua execução mediante o código de `boot.yap`, até regressar ao estado de partida.

No final deste ciclo, o agente 0, responde com sucesso ou insucesso (caso se encontre na primeira situação) ou fornece o conjunto de soluções encontradas pelos diversos agentes (caso se encontre na segunda situação). Finalmente, volta a estar disponível para processar novos comandos do utilizador.

Capítulo 4

Pormenores de Implementação

Este capítulo aprofunda alguns dos conceitos, esquemas e mecanismos anteriormente referidos, através de uma mais explícita descrição da implementação.

4.1 O Emulador de Instruções

O código em Prolog das diferentes cláusulas é, como já vimos, transformado numa sequência de instruções WAM. Para tornar essa sequência executável é necessário a existência de um emulador dessas instruções. No YapOr esse emulador é herdado do Yap e adaptado à nova concepção. O objectivo desta secção é descrever essa adaptação.

4.1.1 O Emulador no Yap

No sistema Yap, o emulador de instruções é implementado como um enorme `switch` dos diversos tipos de instrução dentro dum ciclo infinito, como a figura 4.1 apresenta. O `switch` mediante o valor do `program_counter` selecciona um dos `case`, a que se segue a execução do código correspondente. Aí, são efectuados todos os procedimentos inerentes ao tipo de instrução seleccionada, actualizado o `program_counter` com a instrução seguinte da sequência do programa e por fim executado '`goto ciclo_wam`', o que nos leva de volta ao início do ciclo.

As instruções WAM `call`, `fail`, `try_me`, `retry_me` e `trust_me` são aquelas que estão

```
 B = ponto de escolha de topo;
 program_counter = primeira instrução;
ciclo_wam:
 switch (program_counter) {
   case call:  ...     goto ciclo_wam;
  falha:
   case fail:  ...     goto ciclo_wam;
   case try_me:  ...   goto ciclo_wam;
   case retry_me:  ... goto ciclo_wam;
   case trust_me:  ... goto ciclo_wam;
   ...
 }
```

Figura 4.1: O emulador de instruções do Yap.

directamente relacionadas com o processo paralelo. Na figura 4.2 encontra-se o pseudo--código das mesmas, segundo o implementado no Yap.

O código relativo à instrução `fail` também pode ser executado, sem ser por acção directa do `switch`, através da instrução de salto '`goto falha`'. Esta instrução de salto encontra-se no código de algumas instruções WAM que por vezes falham na sua execução, como por exemplo, nas instruções pertencentes ao processo de unificação.

Se o predicado `pred` associado à execução de uma instrução `call` tem mais do que uma alternativa a ele associada, então o `program_counter` ao ser instanciado com a '`primeira alternativa de acordo com pred`' provoca de seguida a execução de uma instrução do tipo `try_me` e só depois se segue o verdadeiro código dessa alternativa. Caso `pred` contenha apenas uma única alternativa, o `program_counter` é desde logo instanciado com a execução desse código.

A instrução `fail` começa por recuperar as variáveis de topo da trilha, a que se segue a execução da próxima alternativa guardada no ponto de escolha corrente. A instrução de topo, da sequência de instruções referentes à nova alternativa, ou contém uma referência para a alternativa que se lhe segue ou contém um valor por defeito `null` indicando a ausência de mais alternativas (rever figura 3.9). Na figura 4.2 as instruções `retry_me` e `trust_me` representam respectivamente o conjunto das instruções desses dois tipos. Em ambas é inicialmente recuperado o estado guardado no ponto de escolha corrente e enquanto que, em `retry_me` é também guardada a referência para a próxima alternativa, em `trust_me` é removido o ponto de escolha corrente (a sua manutenção deixa de fazer sentido devido à ausência de alternativas por explorar).

```
 B = ponto de escolha de topo;
 program_counter = primeira instrução;
ciclo_wam:
 switch (program_counter) {
   case call:
     pred = predicado relativo ao call;
     program_counter = primeira alternativa de acordo com pred;
     goto ciclo_wam;
  falha:
   case fail:
     usar a trilha para desreferenciar as atribuições às variáveis aí guardadas;
     program_counter = alternativa guardada no ponto de escolha corrente;
     goto ciclo_wam;
   case try_me:
     alt = próxima alternativa a explorar;
     criar novo ponto de escolha;
     guardar o estado corrente e alt no ponto de escolha;
     program_counter = próxima instrução;
     goto ciclo_wam;
   case retry_me:
     restaurar o estado guardado no ponto de escolha corrente;
     alt = próxima alternativa a explorar;
     guardar alt no ponto de escolha corrente;
     program_counter = próxima instrução;
     goto ciclo_wam;
   case trust_me:
     restaurar o estado guardado no ponto de escolha corrente;
     remover esse ponto de escolha;
     program_counter = próxima instrução;
     goto ciclo_wam;
   ...
 }
```

Figura 4.2: A emulação das instruções directamente relacionadas com o processo paralelo.

4.1.2 O Emulador no YapOr

Na figura 4.3 encontra-se o pseudo-código das adaptações fundamentais efectuadas no emulador de instruções do Yap para o suporte do paralelismo. Esse pseudo-código sugere as seguintes observações:

- O registo `B` contém uma referência do ponto de escolha corrente, enquanto que o registo `nó_partilhado_corrente` contém uma referência do ponto de escolha partilhado mais profundo (rever secção 2.2.3).
- A função `inicializações_YapOr()` implementa o conjunto de inicializações do processo paralelo (ver figura 4.4).

```
 inicializações_YapOr();                                              /* novo */
 program_counter = primeira instrução;
ciclo_wam:
 switch (program_counter) {
   case call:
     pred = predicado relativo ao call;
     verificar_solicitações();                                        /* novo */
     ...
  falha_partilhada:                                                   /* novo */
     B = nó_partilhado_corrente;
  falha:
   case fail:  ...
   case try_me:
     alt = próxima alternativa a explorar;
     criar novo ponto de escolha;
     actualizar_carga(try_me);                                        /* novo */
     ...
   case retry_me:  ...
   case trust_me:  ...
   case procura_trabalho:                                             /* novo */
     estrutura_partilhada_corrente = nó_partilhado_corrente->CP_EP;
     fechar_acesso (estrutura_partilhada_corrente);
     if (nó morto) {
       libertar_acesso (estrutura_partilhada_corrente);
       em_espera (instalação);
       if (distribuidor_de_trabalho())
         goto falha_partilhada;                          /* conseguiu trabalho */
       finalizações;
       if (agente 0) return;
       program_counter = procura_trabalho_primeira_vez;
       goto ciclo_wam;
     }
     else toma próxima alternativa;                                 /* nó vivo */
   case procura_trabalho_primeira_vez:                                /* novo */
     em_espera (sinal de início por parte do agente 0);
     if (distribuidor_de_trabalho())  /* procura trabalho pela primeira vez */
       goto falha_partilhada;
     finalizações;
     program_counter = procura_trabalho_primeira_vez;
     goto ciclo_wam;
   ...
 }
```

Figura 4.3: O emulador de instruções adaptado à execução paralela.

- Na instrução `call` introduziu-se a função `verificar_solicitações()` que é responsável pela verificação da existência de novas solicitações para partilha de trabalho (ver figura 4.4).

- A instrução `fail` foi mantida inalterável.

- Na instrução `try_me` introduziu-se a função `actualizar_carga()` que é responsável pelo cálculo da carga privada de cada agente (ver figura 4.4).

- O tratamento dado às instruções `retry_me` e `trust_me` bem como ao '`else toma próxima alternativa`' da instrução `procura_trabalho` está descrito em pormenor na secção 4.2.

- O pseudo-código das pseudo-instruções `procura_trabalho_primeira_vez` e `procura_trabalho` é introduzido no emulador de instruções. O procedimento '`em_espera (instalação)`' da instrução `procura_trabalho` diz respeito a sincronizações relativas ao processo de partilha de trabalho, tratado em pormenor na secção 4.3. A função `distribuidor_de_trabalho()` presente nas duas pseudo--instruções é a responsável por encontrar novo trabalho para exploração (ver figura 4.5). Caso este seja conseguido (caso em que a função retorna o valor 1) simula-se uma falha através de '`goto falha_partilhada`', a que se segue a actualização do ponto de escolha corrente e a execução da instrução `fail`. O caso contrário (caso em que a função retorna o valor 0), corresponde ao fim da exploração do objectivo em questão, a que se segue um conjunto de procedimentos finais.

A figura 4.4 apresenta o pseudo-código de três das funções referidas na figura 4.3.

```
inicializações_YapOr() {
  inicialização e tratamento da memória partilhada;
  conjunto de outras inicializações;
  B = criar ponto de escolha de topo;
  criar os outros agentes do sistema;
}

verificar_solicitações() {
  if (alguma solicitação para partilha de trabalho) P_partilha();
}

actualizar_carga(inst) {
  B->CP_ALE = B->CP_B->CP_ALE + B->CP_B->CP_ALT->ape + 1;
  carga = B->CP_ALE + inst->ape;
}
```

Figura 4.4: Três das funções introduzidas no emulador de instruções.

Por fim, a figura 4.5 apresenta a função `distribuidor_de_trabalho()`. Esta função é central a todo o processo paralelo, pois é nela que estão implementadas as estratégias descritas na secção 2.2, como são: a procura de trabalho na região partilhada da árvore de procura; a procura de agentes ocupados dentro e fora da subárvore do nó corrente; um melhor posicionamento na árvore de procura na inexistência de trabalho; a procura de trabalho invisível.

```
distribuidor_de_trabalho() {
  if (existem nós vivos na árvore de procura) {
    retroceder (primeiro nó vivo);
    return 1;
  }
  if (alguma solicitação para partilha de trabalho) recusar solicitação;
  estado_do_agente = suspenso;
  contador = 1;
  while(TRUE) {                               /* ciclo de procura de trabalho */
    if (todos os agentes suspensos) return 0;             /* fim do objectivo */
    b = agentes ocupados mais próximos na subárvore do nó corrente;
    if (algum agente em b com excesso de carga) {
      P = agente com maior carga em b;
      if (Q_partilha(P)) {
        estado_do_agente = ocupado;
        return 1;
      }
    }
    b = agentes ocupados mais próximos fora da subárvore do nó corrente;
    if (algum agente em b com excesso de carga) {
      P = agente com maior carga em b;
      retroceder (primeiro nó com o agente P);
      if (Q_partilha(P)) {
        estado_do_agente = ocupado;
        return 1;
      }
    }
    if (existe uma melhor posição na árvore de procura)
      retroceder (melhor posição);
    if (contador++ == k) {     /* Procura trabalho invisível após k iterações */
      b = agentes com trabalho invisível mais próximos na subárvore corrente;
      if (algum agente em b){
        P = qualquer agente de b;
        if (Q_partilha(P)) {
          estado_do_agente = ocupado;
          return 1;
        }
      }
      contador = 1;
    }
  }
}
```

Figura 4.5: O pseudo-código do distribuidor de trabalho.

As funções `P_partilha()` e `Q_partilha()` introduzidas nas duas últimas figuras, implementam o processo de partilha de trabalho entre dois agentes. Esse processo, bem como as tarefas que `P_partilha()` e `Q_partilha()` efectuam, onde se inclui a solicitação para partilha de trabalho, a aceitação/recusa dessa solicitação, a partilha de pontos de escolha privados, a cópia das pilhas de execução e as várias sincronizações do processo, são minuciosamente descritos na secção 4.3.

4.2 Retrocesso para Nós Vivos Partilhados

Quando um agente retrocede para um nó partilhado duas situações podem acontecer:

- o nó está morto e nesse caso o agente invoca o distribuidor de trabalho, seguindo-se o processo já anteriormente descrito na procura de trabalho ainda por explorar (rever secções 2.2 e 4.1).
- o nó está vivo e nesse caso o agente toma a próxima alternativa por explorar do nó. Em resultado da acção do agente, pode acontecer que o nó se mantenha no estado de vivo ou que passe para o estado de morto. Interessa-nos distinguir estas duas situações para actuar de forma apropriada.

Quando o mecanismo normal de retrocesso é executado sobre um nó partilhado, a instrução `procura_trabalho` é tomada do campo `CP_ALT` do ponto de escolha mais profundo e em seguida executada. A figura 4.6 descreve a sequência de procedimentos que se seguem.

De início é impedido o acesso de outros agentes à estrutura partilhada associada ao `nó_partilhado_corrente`. De seguida testa-se o nó em causa de modo a verificar o seu estado. No caso de este se encontrar vivo é tomada a alternativa colocada no campo `próxima_alternativa` da `estrutura_partilhada_corrente`, a que se segue a sua execução.

Como foi referido na secção 4.1, as instruções `retry_me` e `trust_me` representam o conjunto dos dois tipos de instruções que podem ser executadas de seguida. No caso da execução da instrução `retry_me` será invocada a função `nova_alternativa()`, enquanto que no caso da instrução `trust_me` será invocada a função `última_alternativa()`.

Na função `nova_alternativa()` além do campo `próxima_alternativa` da estrutura `estrutura_partilhada_corrente` ser actualizado com o valor da próxima alternativa a

```
ciclo_wam:
  switch (program_counter) {
    case retry_me:
      restaurar o estado guardado no ponto de escolha corrente;
      alt = próxima alternativa a explorar;
      if (nó corrente partilhado) nova_alternativa (alt);          /* novo */
      else guardar alt no ponto de escolha corrente;           /* anterior */
      ...
    case trust_me:
      if (nó corrente partilhado) última_alternativa();            /* novo */
      ...
    case procura_trabalho:
      estrutura_partilhada_corrente = nó_partilhado_corrente->CP_EP;
      fechar_acesso (estrutura_partilhada_corrente);
      if (nó morto) {  ...  }
      else {
        program_counter = estrutura_partilhada_corrente->próxima_alternativa;
        goto ciclo_wam;
      }
    ...
  }

nova_alternativa(nova_alt) {
  estrutura_partilhada_corrente->próxima_alternativa = nova_alt;
  libertar_acesso (estrutura_partilhada_corrente);
}

última_alternativa() {
  estrutura_partilhada_corrente->próxima_alternativa = null;
  retirar_agente (estrutura_partilhada_corrente->bitmap_de_agentes);
  if (nenhum agente em estrutura_partilhada_corrente->bitmap_de_agentes)
    remover_estrutura (estrutura_partilhada_corrente);
  else libertar_acesso (estrutura_partilhada_corrente);
}
```

Figura 4.6: As instruções `procura_trabalho`, `retry_me` e `trust_me` revisitadas.

explorar, é também libertado o fecho de acesso à estrutura. No caso de nós privados, a próxima alternativa continua a ser colocada no campo `CP_ALT` do ponto de escolha corrente.

A função `última_alternativa()` começa por colocar no campo `próxima_alternativa` da estrutura `estrutura_partilhada_corrente` o valor `null`, de seguida exclui do campo `bitmap_de_agentes` o agente em causa e finalmente ou remove a estrutura partilhada ou liberta o fecho de acesso à mesma, em função de ser ou não o último agente associado à estrutura. Após a execução de uma instrução do tipo `trust_me`, o ponto de escolha corrente é removido da pilha local do agente. Em consequência disso, o agente deixa de contar com esse nó na sua árvore de procura, o que justifica a exclusão do agente do campo `bitmap_de_agentes` da estrutura partilhada.

4.3 Partilha de Trabalho

É através do processo de partilha de trabalho entre dois agentes que se torna realmente possível a exploração paralela de um dado objectivo.

4.3.1 Solicitação para Partilha de Trabalho

Quando um agente suspenso Q selecciona um agente ocupado P para solicitar partilha de trabalho, começa por verificar se nenhum outro agente o fez antecipadamente. Caso exista um agente nessas condições, à espera de uma resposta de P, Q deve desistir dessa tentativa e voltar ao ciclo de procura do distribuidor de trabalho. Caso contrário, Q solicita partilha de trabalho a P e fica à espera da resposta deste. Por seu lado, logo que P execute a instrução `call`, verifica da existência de novas solicitações. Ao constatar que possui uma solicitação de Q, deve decidir da sua aceitação ou não. As condições que poderão levar P a rejeitar uma solicitação são as seguintes:

- P não faz parte da subárvore de Q, ou seja, por algum motivo P retrocedeu para um nó que precede o nó onde Q se posicionou para pedir trabalho a P. Nestas circunstâncias o estado computacional de Q não é consistente com parte do estado de P, o que inviabiliza a operação de partilha.
- P e Q encontram-se no mesmo nó da árvore. Neste caso não faz sentido partilhar trabalho porque ambos se encontram nas mesmas circunstâncias.
- P possui uma carga nula[1] e o campo `nó_vivo_mais_próximo` da estrutura partilhada associada ao seu ponto de escolha corrente contém o valor por defeito. Neste caso, P não possui qualquer tipo de trabalho disponível à excepção da actual alternativa cuja execução interrompeu.

Se uma destas condições for válida, então a solicitação é rejeitada. P envia uma mensagem de recusa a Q e retoma a execução da instrução `call`. Q ao receber essa recusa, retorna ao ciclo de procura do distribuidor de trabalho, tal como na situação anterior. Pelo contrário, se nenhuma condição for válida, então a solicitação é aceite. P envia uma mensagem de aceitação a Q e ambos dão início ao processo de partilha de trabalho. De referir ainda

[1] Apenas possível, porque Q seleccionou P com o objectivo de obter trabalho invisível deste.

que, se entretanto P ficar suspenso antes de executar uma instrução `call`, deve recusar a solicitação pendente de Q.

4.3.2 Fases do Processo de Partilha

O processo de partilha de trabalho pode dividir-se em quatro fases principais, a saber:

Fase de preparação onde se calculam os segmentos a copiar entre os agentes e se inicializam algumas variáveis utilizadas para a sincronização do processo.

Fase de partilha onde os pontos de escolha privados de P são partilhados.

Fase de cópia onde se efectua a cópia propriamente dita, isto é, onde os segmentos anteriormente calculados são copiados para o espaço local de Q.

Fase de instalação onde os segmentos mantidos em Q são actualizados com as atribuições condicionais de P às variáveis criadas nesses segmentos.

É obvio que a execução destas fases pode ser dividida pelos dois agentes envolvidos no processo, de modo a que seja concluída no menor espaço de tempo possível. Conseguir uma eficiente divisão dessa execução não é certamente uma tarefa fácil. De entre os vários aspectos a considerar, destacam-se os seguintes:

- O agente P deve prosseguir com a execução da alternativa interrompida o mais breve possível.
- A fase de partilha é executada exclusivamente por P.
- A cópia da pilha local só pode efectuar-se depois da fase de partilha estar concluída.
- As fases de partilha e de cópia podem efectuar-se em simultâneo, exceptuando o caso anterior.
- A instalação só pode ter lugar depois de todos os segmentos terem sido copiados.
- P só pode prosseguir com a execução interrompida após as fases de partilha e de cópia estarem concluídas.
- P não pode executar o mecanismo normal de retrocesso sobre um nó partilhado antes que a instalação seja concluída.

A ideia do processo resume-se ao seguinte: após a fase de preparação, P começa por executar a fase de partilha enquanto que Q executa a de cópia. Q copia os segmentos de P pela seguinte ordem: trilha, pilha de termos e pilha local. Esta última só depois de P ter concluído a fase de partilha. Depois da cópia estar completa Q inicia a instalação. Se P terminar a fase de partilha e Q ainda não tiver concluído a de cópia, então P ajuda Q na cópia dos segmentos ainda em falta, mas pela ordem inversa à enumerada para Q. De seguida, P prossegue a execução interrompida e caso venha a retroceder para um nó partilhado antes que Q termine a fase de instalação, deve esperar pela conclusão da mesma.

4.3.3 Cálculo das Áreas a Copiar

O cálculo das áreas a copiar é uma tarefa vital para o correcto funcionamento do processo de partilha de trabalho. No YapOr e baseando-se no mecanismo de cópia incremental, P é o responsável por esse cálculo, o qual se apresenta esquematizado na figura 4.7.

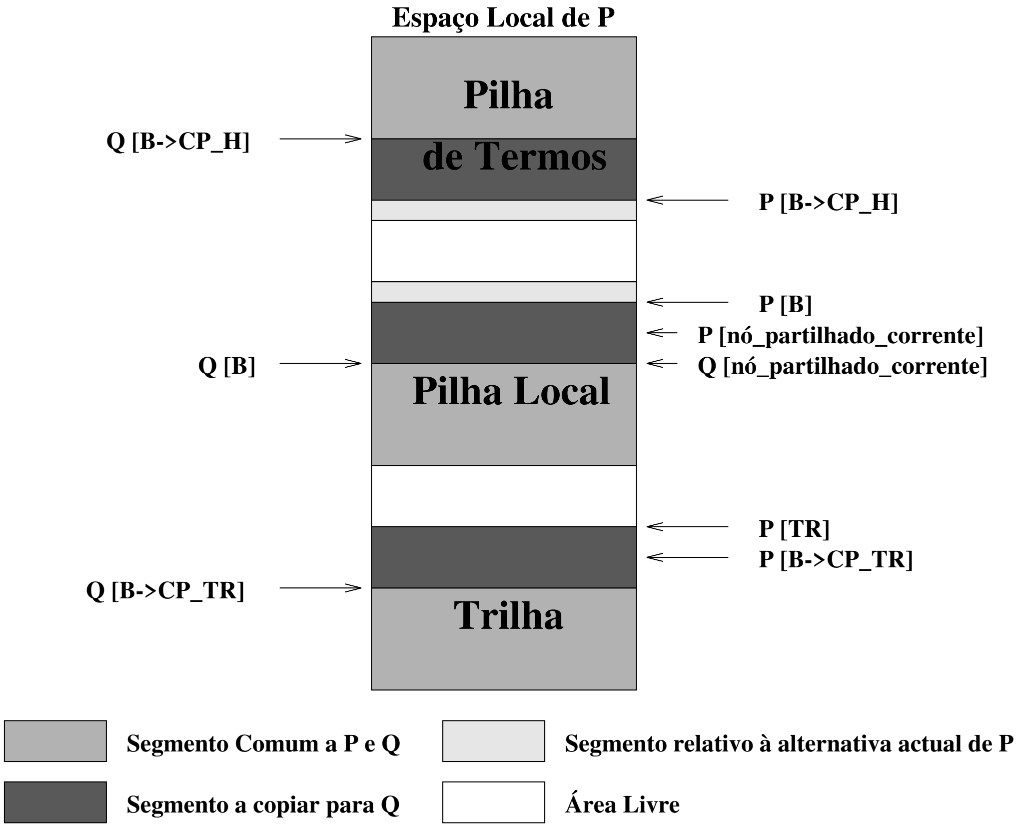


Figura 4.7: Áreas de memória envolvidas no processo de partilha de trabalho.

Q`[B->CP_H]` e Q`[B->CP_TR]` representam os apontadores guardados nos campos análogos do ponto de escolha corrente de Q, cuja referência é dada por Q`[B]`. Por seu lado, Q`[nó_partilhado_corrente]` representa o apontador para o ponto de escolha partilhado mais profundo de Q, que nesta situação coincide sempre com Q`[B]`. No que diz respeito ao agente P, P`[TR]` representa o apontador para o actual topo da trilha que não coincide necessariamente com o guardado em P`[B->CP_TR]`, já que entretanto novas atribuições condicionais podem ter sido feitas. De observar ainda que a referência em P`[nó_partilhado_corrente]` não coincide com a de P`[B]` nem com a de Q`[nó_partilhado_corrente]`. No primeiro caso porque P possui excesso de carga, o que corresponde à existência de nós privados. No segundo caso porque nas últimas operações de partilha de trabalho em que P e Q estiveram envolvidos, o nó mais profundo a ser partilhado não foi o mesmo.

4.3.4 Sincronizações do Processo de Partilha

Após o cálculo das áreas a copiar, o agente P começa por partilhar os seus pontos de escolha privados e por actualizar, de modo a contemplarem o agente Q, aqueles pontos de escolha que já se encontram partilhados mas aos quais Q ainda não tem acesso (rever secção 3.2). Na figura 4.7 esses pontos de escolha são aqueles que se situam entre P`[nó_partilhado_corrente]` e Q`[nó_partilhado_corrente]`. Em simultâneo, Q começa a execução da fase de cópia.

A figura 4.8 apresenta as sincronizações necessárias entre os dois agentes durante todo o processo de partilha de trabalho. A sequência descrita apenas pretende representar a ordem e a dependência dos vários eventos em função das sincronizações necessárias e não corresponde a uma ordem preestabelecida de como ocorre todo o processo. Por exemplo, Q pode receber o sinal '`partilha_ok`' estando ainda a copiar a pilha de termos, o que levaria P a copiar a pilha local e Q a não esperar por essa sincronização e a não copiar essa mesma pilha.

A figura 4.9 descreve o esquema de sincronização utilizado na fase de cópia, cuja função essencial é evitar cópias repetidas e/ou simultâneas dos mesmos segmentos de memória. O agente Q executa a função `Q_fase_de_cópia()` no início da fase de cópia, enquanto que o agente P executa a função `P_ajudar_na_cópia()` após ter concluído a fase de partilha. Na fase de preparação as variáveis `Q_fase` e `P_fase` são inicializadas com as sincronizações `Q_início` e `P_início` respectivamente. A ordenação das cinco sincronizações, conforme

```
      Agente P                                 Agente Q
-----------------------------------------------------------------
.                                        .
Cálculo dos segmentos                    Em espera (aceitação)
.                       ---aceitação-->  .
Fase de partilha                         Cópia da trilha?
.                                        Cópia da pilha de termos?
.                                        Em espera (partilha_ok)
.                       --partilha_ok->  .
.                                        Cópia da pilha local?
Ajudar Q nas cópias?                     .
.                       ----pronto---->  .
Em espera (instalação)                   .
.                       <--instalação--  .
Executar Prolog                          Em espera(pronto)
.                                        .
Retrocesso nó part.?                     Fase de instalação
Em espera (pronto)                       .
.                       <---pronto----   .
.                                        Executar Prolog
.                                        .
```

Figura 4.8: Sincronizações entre P e Q durante o processo de partilha de trabalho.

apresentado no topo da figura, permite-nos de uma forma simples e eficaz implementar este esquema do modo pretendido.

4.3.5 Fase de Instalação

O agente Q é o responsável pela fase de instalação, onde a partir do segmento de trilha copiado de P actualiza nos segmentos da pilha de termos e pilha local mantidos inalteráveis, o conjunto de atribuições condicionais do agente P a variáveis desses mesmos segmentos. A função `instalação()` é a responsável pela instalação dessas atribuições nos segmentos em causa (ver figura 4.10).

Nesta função, por `antes_da_partilha()` pretende-se referir o valor do argumento antes do início do processo de partilha, isto é, o valor que se observa na figura 4.7. Por `depois_da_partilha()` pretende-se referir o valor do argumento depois da fase de cópia. O valor de `depois_da_partilha()` no caso do argumento Q`[B->CP_TR]`, coincide com o valor de P`[B->CP_TR]` da figura 4.7.

A função `instalação()` percorre todo o novo segmento da trilha, à excepção da parte compreendida entre P`[TR]` e P`[B->CP_TR]` da figura 4.7. Se as referências aí guardadas

```
Sincronizações: Q_início < trilha < pilha_de_termos < pilha_local < P_início

Q_fase_de_cópia() {
  if (P_fase > trilha) {
    Q_fase = trilha;
    copia_trilha_de_P();
  } else return;
  if (P_fase > pilha_de_termos) {
    Q_fase = pilha_de_termos;
    copia_pilha_de_termos_de_P();
  } else return;
  em_espera (partilha_ok);
  if (P_fase > pilha_local) {
    Q_fase = pilha_local;
    copia_pilha_local_de_P();
  }
}

P_ajudar_na_cópia() {
  if (Q_fase < pilha_local) {
    P_fase = pilha_local;
    copia_pilha_local_para_Q();
  } else return;
  if (Q_fase < pilha_de_termos) {
    P_fase = pilha_de_termos;
    copia_pilha_de_termos_para_Q();
  } else return;
  if (Q_fase < trilha) {
    P_fase = trilha;
    copia_trilha_para_Q();
  }
}
```

Figura 4.9: Sincronizações entre P e Q durante a fase de cópia.

```
instalação() {
  aux = antes_da_partilha (Q[B->CP_TR]);
  while (--aux > depois_da_partilha (Q[B->CP_TR]))
    if (*aux > antes_da_partilha (Q[B]) || *aux < antes_da_partilha (Q[B->CP_H]))
      **aux = *(*aux + offset_agente[P]);
}
```

Figura 4.10: Função responsável pela fase de instalação.

forem de variáveis pertencentes aos segmentos da pilha de termos ou da pilha local mantidos intactos na operação de partilha, então é para aí copiado o valor correspondente à mesma posição das pilhas do agente P. A parte da trilha que não é percorrida na operação anterior também o poderia ser, mas tal não é necessário porque no final da operação de partilha o agente Q ao simular uma falha, provoca a execução do mecanismo normal de retrocesso que desreferencia as variáveis que se encontram nessa parte da trilha, isto é, as atribuições dessas variáveis voltam ao estado de auto-referenciação[2].

4.4 Atraso na Divulgação do Excesso de Carga

Grande parte dos programas em Prolog contêm predicados em que as tarefas a eles associadas são relativamente pequenas, isto é, o tempo de execução das mesmas é bastante curto. Evitar que pedaços de trabalho deste género sejam partilhados entre os diversos agentes é fundamental para não degradar a performance do sistema, já que o tempo despendido na operação de partilha de trabalho entre dois agentes não é compensado no tempo de execução deste tipo de tarefas, mesmo que partilhadas.

Na implementação do YapOr tenta-se aliviar ligeiramente essa permissividade, sem que isso introduza custos adicionais. A única desvantagem verifica-se em programas que normalmente apresentavam grandes fontes de paralelismo, só interrompidas casualmente devido a pequenas tarefas e que passaram a apresentar um ligeiro decréscimo nesse paralelismo, embora uniformemente distribuído por todo o tempo de execução. Apesar disso, o processo paralelo em geral é substancialmente melhorado.

A ideia deste esquema é retardar a actualização do registo local `carga` de um dado agente por um certo período de tempo, sempre que este possa passar de zero para um valor positivo. Na prática, passar para um valor positivo corresponde a comunicar aos outros agentes que se possui excesso de carga, de onde resulta um estado de permissividade face a futuras solicitações para partilha de trabalho. Pretende-se assim, que a um agente com excesso de carga esteja associada a garantia de uma compensação positiva para a performance do sistema numa possível operação de partilha de trabalho.

Quando um ponto de escolha é criado, o registo `carga` não é desde logo actualizado se o seu valor anterior for nulo. Em seu lugar, é inicializado o contador `contador_de_atraso` com um valor por defeito[3], que corresponde ao número de instruções `call` que devem ser executadas antes do valor de carga do agente ser actualizado (ver figura 4.11).

A partir daqui, cada execução da instrução `call` provoca um decremento sucessivo de uma

[2]Em Prolog, um esquema habitualmente utilizado para indicar que uma variável ainda não foi instanciada é colocar um apontador a apontar para ela própria.

[3]No YapOr, esse valor é de 5 unidades. No entanto é possível redefinir esse valor no início do sistema.

```
actualizar_carga() {
  B->CP_ALE = B->CP_B->CP_ALE + B->CP_B->CP_ALT->ape + 1;          /* anterior */
  if (carga == 0) {                                                    /* novo */
    if (contador_de_atraso == 0)
      contador_de_atraso = VALOR_POR_DEFEITO;
  } else carga = B->CP_ALE + inst->ape;                            /* anterior */
}

verificar_solicitações() {
  if (contador_de_atraso > 0)                                          /* novo */
    if (--contador_de_atraso == 0)
      carga = B->CP_ALE + B->CP_ALT->ape + 1;
  if (alguma solicitação para partilha de trabalho) P_partilha(); /* anterior */
}
```

Figura 4.11: Extensões necessárias à implementação do esquema de atraso na divulgação do excesso de carga.

unidade no valor de `contador_de_atraso`. Só quando este atingir zero é que o registo `carga` é actualizado. Posteriormente, sempre que um novo ponto de escolha é criado, o registo `carga` passa a ser normalmente actualizado. Só após uma operação de partilha de trabalho é que o registo local `carga` retoma o valor zero e o esquema aqui descrito volta a ter lugar.

Capítulo 5

O Predicado de Corte de Alternativas

Este capítulo apresenta a semântica inerente ao uso do predicado de corte de alternativas em programas de Prolog e o seu particular enquadramento nos sistemas de paralelismo-Ou. Descreve com detalhe as estruturas de suporte à implementação do predicado de corte, bem como o conjunto de soluções encontradas para a correcta manutenção da sua semântica.

5.1 Semântica do Predicado de Corte

O predicado de corte é um predicado interno do sistema, vulgarmente representado pelo símbolo '`!`', e funciona como um operador explícito de corte de alternativas, cuja execução se verifica no contexto normal da sequência de execução dos objectivos de uma dada cláusula. A execução deste predicado resulta no corte de todas as ramificações da árvore de procura associadas a alternativas colocadas à direita da ramificação de alcance do corte. A ramificação de alcance do corte situa-se entre o nó corrente e o nó associado ao predicado de cabeça da cláusula que contém o corte. O predicado de corte é um operador assimétrico porque só corta alternativas colocadas à sua direita, o que implica a existência de um esquema de ordenação das alternativas. O suporte a este esquema é dado pelo campo `ape` existente no cabeçalho das alternativas.

A figura 5.1 apresenta a semântica do predicado de corte no contexto da árvore de procura do objectivo '`a(X), z(X).`'. A execução inicia-se pelo predicado `a(X)`, que por sua vez invoca `b(X)`. O objectivo `b(X)` sucede na sua primeira alternativa '`b(0).`', seguindo-se a execução do predicado de corte que tem como alcance o nó raiz. A execução prossegue com o predicado `z(X)` com `X` já instanciado com o valor `0`.

A execução de `!(raiz)` corta as alternativas '`b(1).`', '`b(2).`', '`b(3).`', '`a(X):-c(X).`' e '`a(X):-d(X).`'. Em consequência desse corte são removidos os nós associados aos predicados `a` e `b`.

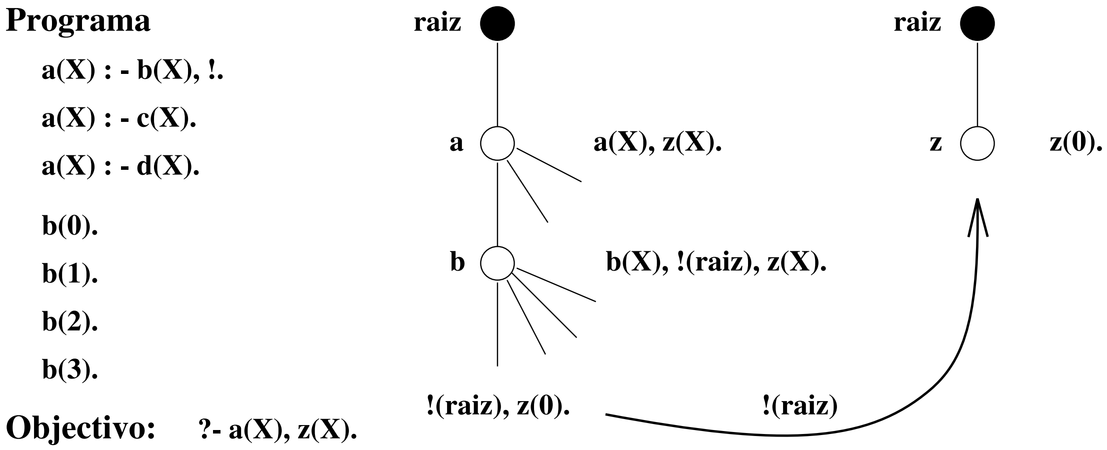


Figura 5.1: A semântica do predicado de corte.

5.2 O Corte nos Sistemas de Paralelismo-Ou

Nos sistemas sequenciais o predicado de corte, corta alternativas que ainda não haviam sido iniciadas. Nos sistemas paralelos e em particular nos sistemas de paralelismo-Ou, o predicado de corte pode por vezes cortar alternativas que já foram iniciadas ou que já foram totalmente processadas. Sendo assim, a implementação do predicado de corte num sistema de paralelismo-Ou implica desde logo a resolução de dois grandes problemas:

1. Como tratar a operação de corte quando as alternativas a cortar se situam na região partilhada da árvore de procura. Daqui resultam as seguintes complicações:

   - Outros agentes podem ser afectados pela execução da operação de corte.
   - Uma operação de corte pode ser inviabilizada pela execução de uma outra operação de corte efectuada mais à esquerda.
   - Nem sempre uma operação de corte pode ser completamente executada.

- As soluções que vão sendo encontradas nem sempre correspondem a soluções válidas.

2. Como evitar a exploração de alternativas cuja execução se pode tornar inútil por acção de uma operação de corte. O explorar de alternativas que potencialmente podem tornar-se inúteis é vulgarmente referido como *trabalho especulativo*. A execução destas tarefas representam computações especulativas e podem resultar num esforço desnecessário quando comparadas com a execução num sistema sequencial, onde o trabalho especulativo nunca é executado já que as ramificações da árvore de procura são processadas da esquerda para a direita. Posto isto, *"trabalho que nunca será executado num sistema sequencial"* [Cie92] parece-nos ser uma boa definição para trabalho especulativo.

Quando um agente executa uma operação de corte e não se situa o mais à esquerda na subárvore de alcance do corte, corta dentro do possível o maior número de ramificações dessa subárvore associadas a alternativas colocadas à direita da ramificação de alcance do corte. O corte das ramificações em falta fica pendente no nó da subárvore onde existe um agente mais à esquerda. Logo que a execução das alternativas mais à esquerda termine, o corte das ramificações em falta deve prosseguir até que todas estejam cortadas. A secção 5.4 descreve em pormenor a implementação desta operação no YapOr.

5.2.1 Trabalho Especulativo

O tratamento do trabalho especulativo pode ser dividido em duas vertentes principais:

1. Eliminar e cortar o mais cedo possível o conjunto das alternativas cuja exploração se tornou entretanto especulativa por acção de uma operação de corte. Esse conjunto deve ser tal, que as operações de corte que possam ser executadas mais à esquerda na árvore de procura e que inviabilizem a operação de corte corrente, mantenham o carácter especulativo do conjunto.

2. Evitar tanto quanto possível a exploração de alternativas que potencialmente correspondam a trabalho especulativo, através da introdução de novos esquemas de distribuição de trabalho que minimizem a possibilidade de computações especulativas.

No sistema YapOr foi apenas dado tratamento à primeira vertente do trabalho especulativo, por se mostrar de fácil implementação face à estrutura já elaborada, uma vez que não implicava grandes alterações. O mesmo não aconteceria com a implementação da segunda vertente, que provocaria uma remodelação na estrutura do algoritmo do distribuidor de trabalho, que se tornaria inviável face ao tempo disponível para a conclusão do trabalho.

Cada alternativa possui um diferente carácter especulativo mediante o posicionamento na respectiva árvore de procura. À medida que nos movemos em direcção às alternativas colocadas nas ramificações mais profundas e mais à esquerda, o carácter especulativo das mesmas diminui. À medida que nos movemos em direcção às alternativas colocadas nas ramificações de topo e mais à direita esse carácter especulativo aumenta. Para obter sucesso no tratamento da segunda vertente atrás referida é necessário considerar esquemas que favoreçam a exploração de tarefas posicionadas o mais próximo possível do canto inferior esquerdo da árvore de procura.

Tal como no YapOr, as primeiras versões do sistema Muse apenas consideraram o tratamento da primeira vertente do trabalho especulativo. Só posteriormente surgiu uma versão que passou também a considerar a segunda vertente [AK92]. A estratégia fundamental para tratar essa segunda vertente segundo [AK92], baseia-se essencialmente em concentrar o maior número possível de agentes nas ramificações com um menor carácter especulativo, de modo a simular tanto quanto possível a execução sequencial de Prolog, sempre no mais à esquerda e profundo nó. A árvore de procura, durante a execução de um dado objectivo, é dividida em duas regiões: a região à direita que contém trabalho suspenso e a região à esquerda que contém trabalho activo. A ideia básica é a seguinte: sempre que um dado agente é o mais à direita de todos, suspende a sua execução e vai tomar o trabalho disponível colocado o mais à esquerda na região activa. Sempre que um agente termine uma tarefa e não exista mais trabalho na região activa, retoma da região suspensa o trabalho suspenso o mais à esquerda, activa-o e inicia a sua exploração. Esta estratégia foi baptizada como *actively seeking the leftmost available work strategy* e visa minimizar a execução de trabalho inútil.

5.3 Estruturas de suporte ao Predicado de Corte

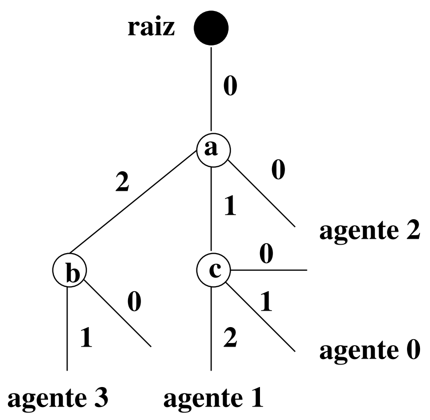
O suporte ao predicado de corte, como implicitamente foi dado a notar, requer a existência de um mecanismo eficiente de representação do posicionamento dos diferentes agentes na árvore de procura, de modo a que seja possível: verificar quando é que um dado agente é o mais à esquerda numa dada subárvore; identificar num certo nó quais os agentes colocados em ramificações à esquerda e/ou à direita da ramificação corrente de um dado agente.

Num sistema paralelo em que é permitido trabalho especulativo, o encontrar de uma nova solução não garante desde logo a validade da mesma face a possíveis operações de corte que a tornem inválida. Para precaver situações deste tipo, é necessário guardar as soluções de um modo prático e eficiente que permita distinguir, no final da execução de um objectivo, quais as soluções realmente válidas.

As secções que se seguem descrevem as estruturas que servem de suporte às soluções encontradas para estes problemas.

5.3.1 Representação da Árvore de Procura

No YapOr, a representação da árvore de procura foi implementada através de uma matriz de duas entradas, `ramificação_corrente(x,y)`, colocada no espaço global de endereçamento. Para cada par `(x,y)` da matriz `ramificação_corrente` encontra-se uma referência da alternativa tomada pelo agente `x` no nó partilhado, da sua ramificação corrente, de profundidade `y` (ver figura 5.2). Se uma dada alternativa contém no seu campo `ape` o valor `v`, então é referida como a alternativa `v`.



agente 0	agente 1	agente 2	agente 3
0	0	0	0
1	1	0	2
1	2	--	1
--	--	--	--

Figura 5.2: A estrutura de suporte à representação da ramificação corrente de cada agente.

Nas estruturas partilhadas foi adicionado um novo campo designado por `profundidade`. Este campo contém uma referência para a profundidade do nó partilhado, a que corresponde a estrutura partilhada. A estrutura partilhada associada ao nó raiz contém por defeito no campo `profundidade` o valor zero. Na figura 5.2, os nós partilhados `a`, `b` e `c` têm respectivamente profundidades 1, 2 e 2.

Através dos campos `bitmap_de_agentes` e `profundidade` da estrutura partilhada associada a um dado nó conseguimos, pela consulta da matriz `ramificação_corrente`, identificar os agentes presentes nesse nó que se encontram em ramificações à esquerda e/ou à direita da ramificação corrente de um dado agente.

5.3.2 O Mais à Esquerda

Sempre que um dado agente executa uma operação de corte ou encontra uma nova solução, necessita de tomar conhecimento do nó partilhado mais profundo onde existe pelo menos um outro agente numa ramificação mais à esquerda.

Um algoritmo para conhecer esse nó pode ser o seguinte: quando um agente P quer saber o nó onde deixa de ser o mais à esquerda, começa pelo seu nó partilhado mais profundo (`nó_partilhado_corrente`) e para cada agente que partilhe esse nó (membro do campo `bitmap_de_agentes` da estrutura partilhada associada ao nó `nó_partilhado_corrente`) consulta, através da matriz `ramificação_corrente`, a alternativa tomada por cada um deles nesse nó. Se o agente P encontra alguma alternativa tomada por algum dos outros agentes com um valor superior ao valor da sua alternativa, então P deixa de ser o mais à esquerda nesse nó. Caso contrário, P deve repetir sucessivamente a mesma operação no nó seguinte da sua ramificação, até encontrar um nó onde deixe de ser o mais à esquerda ou até atingir o nó raiz.

Na figura 5.2, o agente 3 é o mais à esquerda de toda a árvore de procura. Os agentes 0, 1 e 2 deixam de ser os mais à esquerda nos nós `c`, `a` e `a` respectivamente.

5.3.2.1 Redução do Número de Consultas

No algoritmo anterior sempre que um agente P verifica que é o mais à esquerda num dado nó N e se prepara para repetir no nó M (seguinte da sua ramificação) a mesma operação, não necessita de voltar a consultar todo o agente Q já consultado no nó N. Isto acontece porque se no nó N o agente Q se encontra à direita de P, então mesmo que Q retroceda na árvore de procura para o nó M, nunca poderá tomar uma alternativa superior à tomada por P nesse nó. No entanto, Q pode conseguir trabalho de um outro agente W situado à esquerda de P no nó M. Nesse caso, apesar de Q passar a situar-se à esquerda de P em

M, P deixa na mesma de ser o mais à esquerda no nó M apenas pela consulta do agente W. Os agentes a consultar em M obtêm-se da diferença entre os `bitmap_de_agentes` da estrutura partilhada associada ao nó M e da estrutura associada ao nó N.

Por exemplo, na figura 5.2, o agente 1 consulta a matriz `ramificação_corrente` para o agente 0 no nó `c` e para os agentes 2 e 3 no nó `a`. Com esta optimização, cada agente é no máximo consultado uma única vez.

5.3.2.2 Redução do Número de Nós a Investigar

Nesta secção apresenta-se uma outra optimização do algoritmo anterior. Esta segunda optimização tem como objectivo essencial reduzir o número de nós investigados pelo agente P, durante a execução do algoritmo.

Para tal às estruturas partilhadas foi adicionado um novo campo designado por `nó_à_esquerda_mais_próximo`, o qual poderá conter ou uma referência para o nó partilhado mais próximo onde existe uma ramificação mais à esquerda ou um valor por defeito `null` indicando a inexistência de tais nós. A estrutura partilhada associada ao nó raiz no seu campo `nó_à_esquerda_mais_próximo` contém o valor `null`.

Posto isto, o ciclo do algoritmo anterior em lugar de se repetir sob o nó seguinte da ramificação, passa a repetir-se sob o nó colocado no campo `nó_à_esquerda_mais_próximo`, até que o agente em causa encontre um nó onde deixe de ser o mais à esquerda ou até que o campo `nó_à_esquerda_mais_próximo` contenha o valor `null` (neste caso o agente é o mais à esquerda na sua ramificação).

Se um determinado agente deixa de ser o mais à esquerda num dado nó N, actualiza o campo `nó_à_esquerda_mais_próximo` das estruturas partilhadas dos nós mais profundos que N com uma referência para o nó N. Se por outro lado um determinado agente é o mais à esquerda na sua ramificação, actualiza o campo `nó_à_esquerda_mais_próximo` das estruturas partilhadas atrás referidas com o valor por defeito `null`.

As estruturas partilhadas são criadas durante as operações de partilha de trabalho. O primeiro nó partilhado onde pode existir uma ramificação mais à esquerda é o nó `nó_partilhado_corrente`. Se o agente que vai partilhar trabalho é o mais à esquerda nesse nó, então coloca no campo `nó_à_esquerda_mais_próximo` de todas as estruturas partilhadas que criar, a referência do campo `nó_à_esquerda_mais_próximo` da estrutura partilhada associada ao nó `nó_partilhado_corrente`. Caso contrário, coloca a referência do nó `nó_partilhado_corrente`.

A figura 5.3 apresenta o pseudo-código do algoritmo para encontrar o nó partilhado mais profundo onde existe pelo menos um outro agente numa ramificação mais à esquerda, usando as duas optimizações atrás referidas.

```
deixa_de_ser_o_mais_à_esquerda() {
  inserir(b, eu próprio);
  n = nó_partilhado_corrente;
  while (n != null) {
    b = n->CP_EP->bitmap_de_agentes - b;
    número_alt = ramificação_corrente(eu próprio, n->CP_EP->profundidade);
    for (cada agente w membro de b)
      if (ramificação_corrente(w, n->CP_EP->profundidade) > número_alt) {
        actualizar_campos_nó_à_esquerda_mais_próximo(n);
        return n;
      }
    b = n->CP_EP->bitmap_de_agentes;
    n = n->CP_EP->nó_à_esquerda_mais_próximo;
  }
  actualizar_campos_nó_à_esquerda_mais_próximo(null);
  return raiz;
}
```

Figura 5.3: Encontrando o nó onde o agente deixa de ser o mais à esquerda.

A função `actualizar_campos_nó_à_esquerda_mais_próximo` implementa a actualização dos campos `nó_à_esquerda_mais_próximo` segundo o esquema descrito.

5.3.3 Soluções Pendentes

Como foi anteriormente referido, o encontrar de uma nova solução não garante, desde logo, a validade da mesma face a possíveis operações de corte que a tornem inválida.

Para lidar com este problema foi introduzido no sistema uma nova estrutura, de soluções pendentes, e um novo campo designado por `soluções_pendentes` foi adicionado às estruturas partilhadas. Este campo poderá conter ou uma referência para uma sequência de estruturas de soluções pendentes ou um valor por defeito `null` indicando a ausência de uma tal sequência. As estruturas de soluções pendentes servem para guardar de uma forma agrupada as soluções que vão sendo encontradas numa mesma ramificação a partir do nó ao qual se encontram associadas (ver figura 5.4).

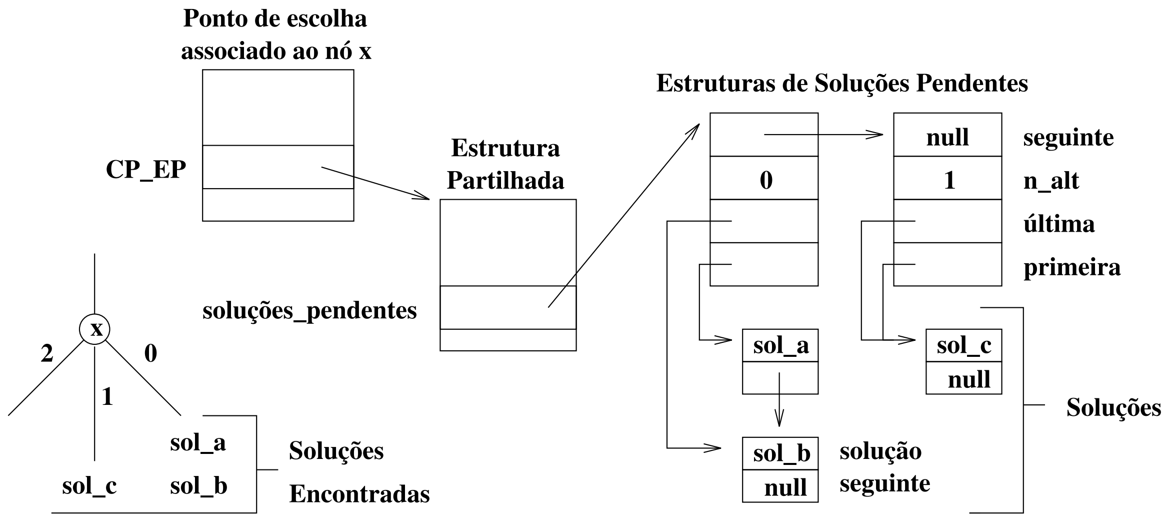


Figura 5.4: O contexto das estruturas de soluções pendentes.

De forma a que possíveis operações de corte possam eliminar as soluções encontradas em ramificações associadas a alternativas colocadas à direita da ramificação de alcance do corte, todo o agente que encontre uma nova solução deve colocá-la pendente no primeiro nó partilhado onde deixe de ser o mais à esquerda. Na figura 5.5 encontra-se o pseudo-código que implementa esta ideia.

Uma solução colocada pendente na estrutura partilhada associada ao nó raiz é desde logo uma solução válida do objectivo em execução.

Na figura 5.5 a função `colocar_nova_solução()` implementa a sequência de estruturas de soluções pendentes, ordenada de forma crescente mediante o valor do campo `n_alt`. Esta ordenação facilita a colocação de soluções relativas às ramificações das alternativas de menor valor e fica a dever-se à maior probabilidade de encontrar em primeiro lugar as soluções das alternativas de maior valor, isto porque as alternativas de maior valor são tomadas primeiro para exploração.

Quando a última alternativa de um dado nó é tomada para exploração, provoca a execução de uma instrução do tipo `trust_me`. Esta instrução, exclui o agente em causa do campo `bitmap_de_agentes` da correspondente estrutura partilhada no caso do nó ser partilhado, e remove o ponto de escolha corrente da pilha local do agente. Estes procedimentos retiram explicitamente o nó em causa da ramificação corrente do agente, quando implicitamente este ainda lhe pertence. Para permitir que o predicado de corte funcione correctamente, estes procedimentos deixaram de ter lugar.

```
nova_solução(solução) {
  ...
  n = deixa_de_ser_o_mais_à_esquerda();
  fechar_acesso (n->CP_EP);
  colocar_nova_solução(n->CP_EP, solução);
  libertar_acesso (n->CP_EP);
}

colocar_nova_solução(est_part, solução) {
  número_alt = ramificação_corrente(eu próprio, est_part->profundidade);
  aux_sol = &(est_part->soluções_pendentes);
  while (*aux_sol != null) {
    if ((*aux_sol)->n_alt >= número_alt) break;
    aux_sol = &((*aux_sol)->seguinte);
  }
  if (*aux_sol != null && (*aux_sol)->n_alt == número_alt) {
    (*aux_sol)->última->seguinte = solução;
    (*aux_sol)->última = solução;
  } else {
    criar nova estrutura de soluções pendentes nova_sol;
    nova_sol->seguinte = *aux_sol;
    nova_sol->n_alt = número_alt;
    nova_sol->primeira = solução;
    nova_sol->última = solução;
    *aux_sol = nova_sol;
  }
}
```

Figura 5.5: Encontrar e colocar novas soluções.

5.4 O Predicado de Corte no YapOr

No YapOr, o predicado de corte foi implementado de forma a evitar modificações profundas na concepção anterior do sistema, ou seja, foi como que adicionada uma nova camada em torno do sistema que o tornou capaz de executar uma classe maior de programas escritos em Prolog.

5.4.1 Tipos de Corte

De um modo geral, a execução de uma instrução do tipo `corte N` resulta no corte de todas as ramificações da árvore de procura situadas em nós abaixo do nó `N` e associadas a alternativas colocadas à direita da ramificação corrente do agente que efectua a operação de corte.

Numa operação de corte podemos distinguir dois tipos de corte de alternativas (ver figura 5.6):

1. Cortar outras alternativas (cláusulas) do predicado corrente.
2. Cortar alternativas resultantes da invocação dos objectivos do corpo da cláusula corrente.

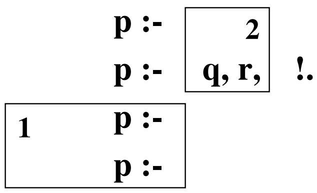


Figura 5.6: Os dois tipos de corte de alternativas.

Para uma eficiente detecção do primeiro tipo de corte, nas instruções de topo da sequência de instruções resultante da compilação de uma dada cláusula foi adicionado um novo campo, designado por `corte`, cujo objectivo fundamental é assinalar as cláusulas que contém o predicado de corte (ver figura 5.7).

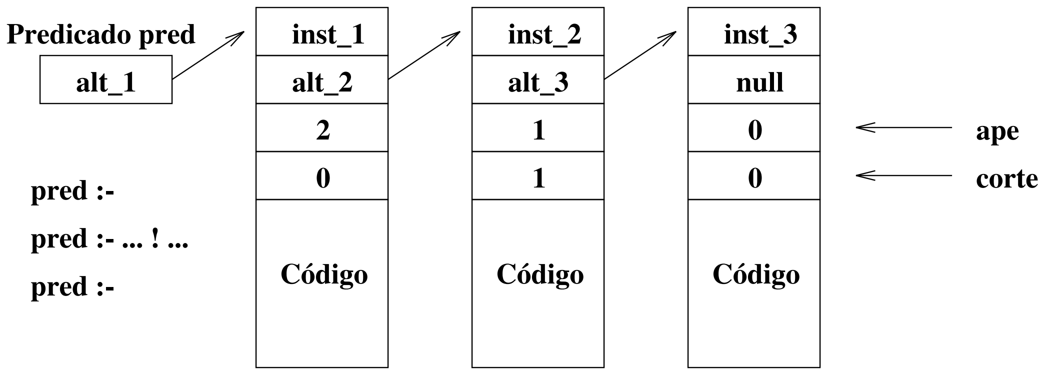


Figura 5.7: O novo campo `corte` na estrutura de dados dos predicados.

5.4.2 Um Exemplo

Para ilustrar como a detecção e o corte de alternativas se concretiza eficazmente é apresentado na figura 5.8 um pequeno exemplo.

Para uma melhor compreensão do exemplo, o processo de indexação foi ignorado, ou seja, assume-se que a invocação de um predicado composto por mais do que uma cláusula implica sempre a criação de um novo ponto de escolha. O símbolo '`!`' colocado no ramo de uma alternativa, indica que a mesma contém pelo menos um predicado de corte e que consequentemente pode executar um corte do tipo (1). O objectivo '`!(b1), !(raiz).`' prossegue a sequência de execução apresentada na figura. Num sistema sequencial, a execução de `!(b1)` cortaria as alternativas `b2-0` e `a2-0` e a execução de `!(raiz)` cortaria as alternativas `b1-1`, `b1-0` e `a1-0`.

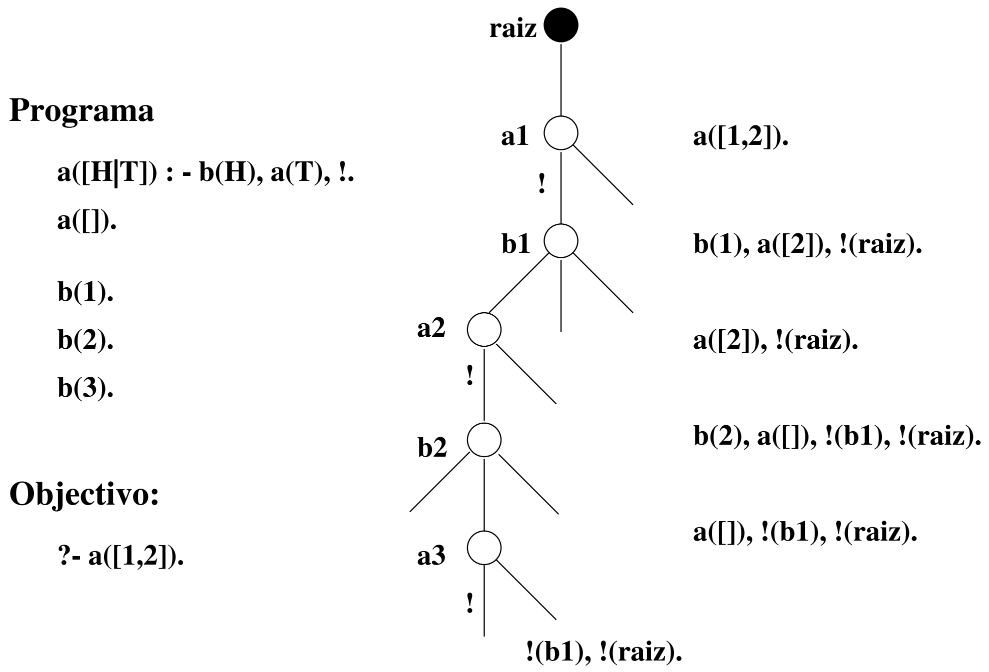


Figura 5.8: A execução de um objectivo com predicados de corte.

Vamos assumir que numa certa execução paralela o agente P se encontra na ramificação `[a1-1,b1-2,a2-1,b2-1,a3-0]`. Em toda a árvore de procura existem apenas duas alternativas posicionadas à sua esquerda (`a3-1` e `b2-2`). Se existir algum agente na alternativa `a3-1`, então o agente P não pode executar nenhum corte de alternativas devido à marca de corte existente nessa alternativa. Uma possível execução da operação de corte da alternativa `a3-1` invalidaria qualquer procedimento executado em `a3-0`. Sendo assim, o agente P deixa pendente no nó `a3` uma marca do corte que não foi capaz de efectuar. Logo que não exista qualquer agente em `a3-1`, a alternativa `a3-0` torna-se a mais à esquerda e o corte deixado pendente é executado.

Vamos agora assumir que não existe nenhum agente em `a3-1`, mas que existe algum em `b2-2`. A alternativa `b2-2` não pode executar nenhuma operação de corte do tipo (1), mas no entanto pode executar uma do tipo (2), já que o objectivo por executar no nó `b2` contém duas possíveis operações de corte (`!(b1)` e `!(raiz)`). O agente P ao executar inicialmente `!(b1)` corta as alternativas `b2-0` e `a2-0`. Este corte de alternativas é seguro porque toda a operação de corte efectuada por um agente colocado em `b2-2`, também terá de cortar as alternativas cortadas por P. Ao mesmo tempo P deixa no nó `b2` uma marca do corte que efectuou. Esta marca é sempre colocada no primeiro nó onde P deixa de ser o mais à esquerda.

Ao prosseguir com a execução de `!(raiz)`, P deveria cortar as alternativas `b1-1`, `b1-0` e `a1-0`. Este corte de alternativas é no entanto uma operação bastante perigosa. Um agente em `b2-2` ao executar a operação de corte anterior (`!(b1)`), corta a ramificação do agente P mas não corta as alternativas `b1-1`, `b1-0` ou `a1-0`. Posto isto, nada nos garante que esse mesmo agente venha posteriormente a efectuar o corte dessas três alternativas. A marca de corte deixada anteriormente por P no nó `b2` alerta-nos para essa possibilidade. Sendo assim, em vez de P executar a operação de corte imediatamente, deixa pendente no nó `b2` uma nova marca, em substituição da anterior, do corte que não pode efectuar.

5.4.3 Procedimentos da Operação de Corte

A operação de corte no sistema YapOr pode ser executada por cada uma das seguintes instruções: `cut`, `cut_t`, `cut_e`, `comit_b_x` e `comit_b_y`. Estas cinco instruções foram herdadas do sistema Yap e executam basicamente a mesma sequência de procedimentos. As ligeiras diferenças que se verificam entre elas reflectem os diferentes contextos em que se enquadram. Para facilitar toda a descrição que se segue vamos utilizar como referência das cinco instruções a instrução `cut`.

Num sistema paralelo, o efeito de executar a instrução `cut Obj`, por parte de um dado agente P, é desde logo diferente mediante o facto de a ramificação de alcance do corte se posicionar totalmente na região privada de P ou não. Em caso afirmativo, a operação de corte é processada exactamente como numa implementação sequencial de Prolog (ver figura 5.9). Em caso negativo, duas novas situações podem acontecer: ou a ramificação de alcance do corte é a mais à esquerda ou não é a mais à esquerda. A figura 5.10 apresenta o pseudo-código envolvido nestas duas últimas situações.

```
case cut:
  nó_objectivo = nó objectivo previamente guardado;
  if (nó_partilhado_corrente é mais profundo que nó_objectivo) {   /* novo */
    if (!corte_de_alternativas(nó_objectivo)) goto falha_partilhada;
  }
  else B = nó_objectivo;            /* operação de corte na região privada */
  program_counter = próxima instrução;
  goto ciclo_wam;
```

Figura 5.9: A instrução `cut` no emulador de instruções.

Na situação em que a ramificação de alcance do predicado de corte é a mais à esquerda, o agente P pode executar completamente a operação de corte de modo a posicionar-se no nó `Obj`. Para isso, para todos os nós mais profundos que o nó `Obj`, remove da sua pilha local os pontos de escolha a eles associados e retira-se de membro de todas as estruturas partilhadas a eles também associados. Em simultâneo recolhe dessas estruturas partilhadas, todas as soluções pendentes encontradas em ramificações não cortadas pela operação de corte e recoloca-as pendentes no nó onde deixa de ser o mais à esquerda. Além disso, a todos os agentes colocados em ramificações à direita da ramificação de alcance do corte envia uma mensagem para retrocederem para o nó `Obj`. Por fim, continua a sua execução com a instrução seguinte da sequência do programa.

Se a ramificação de alcance do predicado de corte não é a mais à esquerda, então é porque existe um nó `N` mais profundo do que o nó `Obj` em que o agente P deixa de ser o mais à esquerda. Sendo assim, P só pode executar parcialmente a operação de corte. Para isso posiciona-se no nó `N` tal como na situação anterior. As soluções pendentes recolhidas em nós mais profundos que `N` são recolocadas pendentes no nó `N`. Aos agentes cortados pela operação de corte, até ao nó `N` inclusive, é enviada uma mensagem para retrocederem para o nó anterior a `N`. De seguida, na estrutura partilhada associada ao nó `N` é colocada uma marca indicando a execução de uma operação de corte. Essa marca é composta pelo nó `Obj` e pela alternativa tomada em `N` pelo agente P, cujas referências são guardadas respectivamente nos campos `corte_pendente_nó_objectivo` e `corte_pendente_alternativa` (adicionados às estruturas partilhadas de modo a servirem de suporte às marcas de corte). Finalmente e segundo o esquema introduzido na secção 5.4.2, o agente P prossegue com o processo de corte das restantes alternativas, enquanto se revele seguro.

A figura 5.11 apresenta o reflexo da execução de duas operações de corte sob uma subárvore de procura, segundo o esquema de corte enunciado. A instrução `!(a)` da esquerda só pode ser executada até o nó `d` devido à existência de uma marca de corte na alternativa `d-2` que

```
corte_de_alternativas(nó_objectivo) {
  if (mensagem para retroceder para o nó l) {  /* outro corte mais à esquerda */
    retroceder (l);
    return 0;
  }
  m = deixa_de_ser_o_mais_à_esquerda();
  n = nó_mais_profundo(nó_objectivo, m);
  retroceder (n);
  cortar_alternativas_até_o_nó (n);
  sol = null;
  for (cada nó l mais profundo que n)
    sol += soluções_pendentes_não_cortadas_pela_operação_de_corte();
  if (nó_objectivo == n) {      /* ramificação de alcance é a mais à esquerda */
    for (cada agente w a explorar uma alternativa abrangida pelo corte)
      mensagem_para_retroceder_para_o_nó(w, n);
    if (sol != null) colocar_soluções_pendentes(sol, m);
  } else {                  /* ramificação de alcance não é a mais à esquerda */
    cortar_alternativas_mais_à_direita_no_nó (n);
    for (cada agente w a explorar uma alternativa abrangida pelo corte)
      mensagem_para_retroceder_para_o_nó(w, anterior(n));
    if (sol != null) colocar_soluções_pendentes(sol, n);
    if (algum corte pendente até n) cont_corte = 0;
    else cont_corte = 1;
    n->CP_EP->corte_pendente_nó_objectivo = nó_objectivo;
    n->CP_EP->corte_pendente_alternativa = alternativa tomada em n;
    if (cont_corte && campo corte das alternativas à esquerda em n == 0) {
      n = anterior(n);
      while (n mais profundo do que nó_objectivo) {
        cortar_alternativas_mais_à_direita_no_nó (n);
        for (cada agente w mais à direita que eu próprio em n)
          mensagem_para_retroceder_para_o_nó(w, anterior(n));
        if (algum campo corte das alternativas à esquerda em n == 1) break;
        n = anterior(n);
      }
    }
  }
  B = nó_partilhado_corrente;
  return 1;
}
```

Figura 5.10: O pseudo-código da operação de corte abrangendo a região partilhada.

condiciona o corte da alternativa `b-0`. A instrução `!(a)` da direita pode ser totalmente executada. Como a operação de corte da esquerda ficou condicionada, é colocada no nó `f` a respectiva marca de corte. Posteriormente, esta operação poderá continuar de modo a cortar a alternativa `b-0`. Para prevenir futuras operações de corte da mesma ramificação, é colocada uma marca fictícia de corte no nó `e`. Esta marca é designada por fictícia, porque não se destina a continuar uma operação de corte que ficou condicionada.

No processo anterior o agente P envia a outros agentes mensagens de retrocesso para um

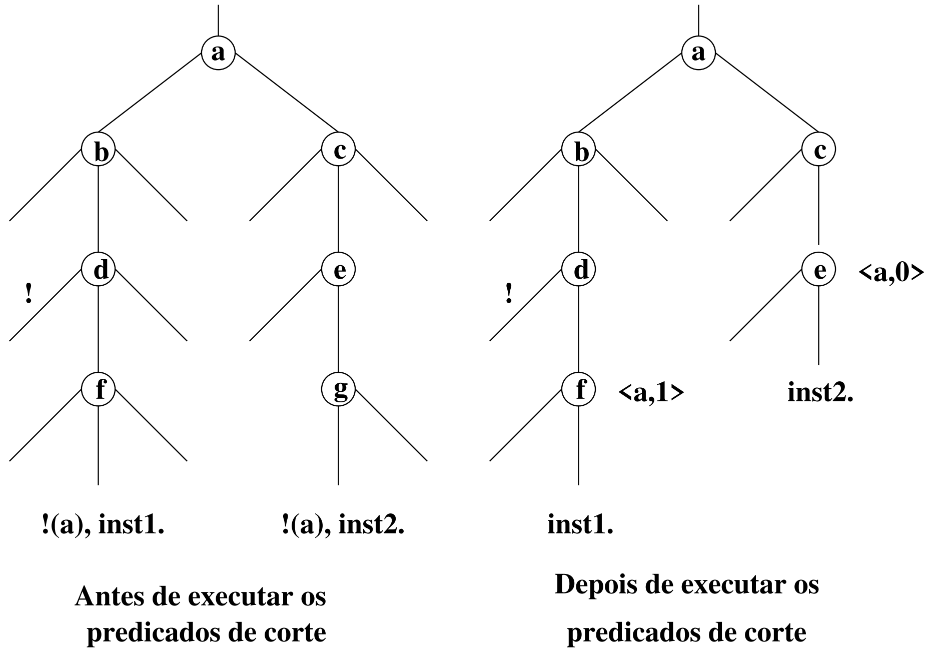


Figura 5.11: A operação de corte segundo o esquema enunciado.

dado nó. A recepção desse tipo de mensagens é confirmada antes da execução de uma instrução `call` (ver figura 5.12) e no início de uma operação de corte abrangendo a região partilhada (rever topo da figura 5.10).

```
case call:
  if (mensagem para retroceder para o nó l) {        /* novo */
    retroceder (l);
    goto falha_partilhada;
  }
  ...                                              /* anterior */
```

Figura 5.12: A instrução `call` adaptada para o suporte do predicado de corte.

Uma marca de corte deixada pendente num dado nó partilhado, deve ser retomada para execução quando o último agente colocado à esquerda dessa marca se prepara para abandonar o nó. Esse agente fica responsável por retomar essa execução e processa-a a partir desse nó como se de uma operação normal de corte se tratasse. Esta situação acontece quando o agente retrocede na árvore de procura ou para tomar uma nova alternativa por explorar ou para se posicionar de forma a solicitar trabalho a um outro agente.

Capítulo 6

Análise de Resultados

Neste capítulo são avaliados alguns aspectos importantes do sistema YapOr, nomeadamente aqueles relacionados com a performance e os custos do sistema. Para isso, analisam-se vários resultados obtidos na execução de um largo conjunto de programas de teste. Todos os resultados apresentados, foram obtidos numa máquina paralela de memória partilhada da Sun, modelo SparcCenter 2000 com 8 processadores, 250 Mbytes de memória RAM, 256 Kbytes e 1 Mbyte de memória *cache* de primeiro nível e segundo nível respectivamente.

6.1 Programas de Teste

O conjunto de programas de teste seleccionado é composto por programas habitualmente utilizados em estudos de análise de performance dos sistemas Muse e Aurora [AK90b, AK92, Sze89]. O código Prolog destes programas encontra-se detalhado no apêndice A. Segue-se uma breve descrição de cada um deles.

- **8-queens2** uma forma ingénua (geração e teste) de resolver o problema de colocar 8 rainhas num tabuleiro de xadrez.
- **8-queens1** um algoritmo mais eficiente para resolver o problema das 8 rainhas, que analisa o estado do tabuleiro em cada passo.
- **5cubes** um programa para empilhar 5 cubos coloridos de forma a que num dado lado não apareçam duas cores repetidas. Este programa não possui predicados de corte.

- **puzzle** um algoritmo para colocar os números de 1 a 19 em forma de hexágono de modo a que a soma de todas as diagonais seja igual. Este programa não possui predicados de corte e possui bastante trabalho especulativo.
- **salt-mustard** resolve o problema "sal e mostarda" através da verificação de um conjunto de restrições.
- **farmer** um pequeno programa para resolver o problema do lavrador que pretende atravessar o rio com um lobo, uma cabra e uma couve.
- **house** resolve o problema "quem possui a zebra" através da verificação de um conjunto de restrições. Este programa não possui predicados de corte.
- **parse4** é uma pergunta para análise gramatical de linguagem natural.
- **parse5** é uma outra pergunta mais complexa para análise gramatical de linguagem natural.
- **db4** corresponde à pergunta de *parse4*, mas para questionar uma base de dados.
- **db5** corresponde à pergunta de *parse5*, mas para questionar uma base de dados.

Todos os programas de teste procuram todas as soluções possíveis.

6.2 Análise de Performance

Para fazer uma análise de performance do sistema YapOr, começou por se ponderar o custo associado à adaptação do sistema sequencial Yap ao sistema paralelo YapOr, de seguida avaliou-se o comportamento do sistema face ao incremento do número de agentes e por fim, comparou-se o sistema YapOr com resultados análogos obtidos no sistema Muse.

6.2.1 Custo do Modelo Paralelo

A tabela 6.1 apresenta os tempos de execução (em milisegundos) de todos os programas de teste, quando executados no sistema Yap e no sistema YapOr com 1 agente. Os tempos apresentados correspondem aos menores tempos obtidos num conjunto de 10 execuções por programa. O tempo de uma execução nos programas *farmer*100*, *house*10*, *db4*10* e *db5*10* corresponde ao total da soma de uma série de execuções parciais, tantas quantas o valor do respectivo factor multiplicativo.

Programas de Teste	**Sistema**		
	YapOr	**Yap**	**YapOr / Yap**
8-queens1	1708.7	1608.2	1.062
8-queens2	4246.9	3629.8	1.170
5cubes	2881.6	2752.2	1.047
puzzle	33596.4	28614.9	1.174
salt-mustard	126.6	108.6	1.166
farmer*100	515.3	497.6	1.036
house*10	313.3	280.2	1.118
parse4	166.5	147.3	1.130
parse5	675.5	556.0	1.215
db4*10	370.1	321.3	1.152
db5*10	440.5	387.1	1.138
Σ	45041.4	38903.2	1.158
Média			1.128

Tabela 6.1: Desempenho do YapOr (com 1 agente) relativamente ao Yap.

A última coluna da tabela 6.1 mostra a proporção entre os tempos obtidos nos dois sistemas, a que corresponde o custo do modelo paralelo, isto é, o custo associado à adaptação do modelo sequencial do Yap ao modelo paralelo do YapOr. As duas últimas linhas, correspondem respectivamente ao somatório de todos os tempos de execução e à média de todas as proporções.

Para uma *justa* comparação entre os dois sistemas, compatibilizou-se o Yap o mais possível com o YapOr. Para tal: utilizou-se um idêntico esquema para medir o tempo e para procurar todas as soluções; utilizou-se o mesmo conjunto de instruções de indexação; ambos os sistemas foram compilados com o mesmo compilador e com as mesmas opções de compilação.

Pela análise da última linha da tabela 6.1 podemos concluir que o Yap é cerca de 13% mais rápido do que o YapOr com um agente. O custo associado a este valor pode ser dividido por vários aspectos: manutenção do registo local de carga actualizado; verificação de solicitações para partilha de trabalho; verificação de mensagens de retrocesso por acção de uma operação de corte; pequenos testes para verificar se o nó mais profundo é partilhado ou privado.

A diferença de performance entre o Yap e o YapOr verificou-se ser menor (apenas 7%), quando os dois sistemas foram comparados numa outra máquina, um Pentium a 100 MHz com 8 Mbytes de memória RAM e 256 Kbytes de memória *cache*.

Verificou-se ainda que a simples deslocação de partes importantes do código do YapOr, seguido de uma nova compilação do mesmo, alterava por vezes consideravelmente o tempo de execução dos programas de teste.

A explicação para estas anomalias de comportamento é difícil de se encontrar a este nível. No entanto, supõe-se que este comportamento se deve ao facto de a memória *cache* de ambas as máquinas utilizar mapeamento directo das linhas de memória, o que por vezes pode provocar conflitos entre partes do programa constantemente utilizadas e guardadas sob a mesma linha de *cache*.

6.2.2 Execução em Paralelo

A tabela 6.2 apresenta os tempos de execução (em milisegundos) de todos os programas de teste quando executados no sistema YapOr com 1, 2, 4, 6, 7 e 8 agentes. Os tempos apresentados correspondem aos menores tempos obtidos num conjunto de 10 execuções. Entre parênteses encontra-se o ganho de velocidade[1] relativamente ao tempo de execução base (com 1 agente).

Os resultados obtidos para 8 agentes são susceptíveis de uma maior margem de erro, porque existe uma maior competição pelo CPU entre os 8 processos que representam os 8 agentes e os restantes processos do sistema.

A tabela 6.2 encontra-se dividida em três blocos de acordo com a qualidade do paralelismo existente nos programas. O primeiro bloco, constituído pelos programas *puzzle*, *5cubes*, *8-queens2* e *8-queens1*, contém paralelismo de granularidade alta. O bloco intermédio possui paralelismo de granularidade média e é constituído pelos programas *salt-mustard*, *parse5*, *parse4*, *db5*10* e *db4*10*. O último bloco é constituído pelos programas *house*10* e *farmer*100* e apresenta paralelismo de granularidade baixa. No fim de cada bloco encontra--se o somatório parcial dos tempos relativos a esse bloco e na última linha da tabela encontra-se o somatório de todos os tempos.

Os resultados que a tabela 6.2 ilustra são bastante satisfatórios. Os programas com pa-

[1] Na literatura inglesa, este ganho de velocidade é habitualmente designado por '*speedup*'.

Programas de Teste	**Número de Agentes**					
	1	**2**	**4**	**6**	**7**	**8**
puzzle	33596.4	16854.5(1.99)	8457.2(3.97)	5643.4(5.95)	4854.7(6.92)	4307.8(7.80)
5cubes	2881.6	1439.9(2.00)	722.8(3.99)	487.8(5.91)	429.9(6.70)	399.4(7.21)
8-queens2	4246.9	2140.5(1.98)	1077.7(3.94)	735.3(5.78)	633.5(6.70)	583.5(7.28)
8-queens1	1708.7	856.8(1.99)	433.4(3.94)	296.5(5.76)	259.4(6.59)	243.1(7.03)
Σ	42433.6	21291.7(1.99)	10691.1(3.97)	7163.0(5.92)	6177.5(6.87)	5533.8(7.67)
salt-mustard	126.6	64.3(1.97)	34.9(3.63)	24.3(5.21)	22.2(5.70)	20.8(6.09)
parse5	675.5	346.0(1.95)	193.3(3.49)	133.2(5.07)	123.9(5.45)	116.7(5.79)
parse4	166.5	86.8(1.92)	48.8(3.41)	36.5(4.56)	35.4(4.70)	35.2(4.73)
db5*10	440.5	234.6(1.88)	137.9(3.19)	106.5(4.14)	101.9(4.32)	94.7(4.65)
db4*10	370.1	193.8(1.91)	113.9(3.25)	96.7(3.83)	91.2(4.06)	87.5(4.23)
Σ	1779.2	925.5(1.92)	528.8(3.36)	397.2(4.48)	374.6(4.75)	354.9(5.01)
house*10	313.3	174.9(1.79)	119.5(2.62)	104.9(2.99)	113.0(2.77)	125.6(2.49)
farmer*100	515.3	335.3(1.54)	347.2(1.48)	480.6(1.07)	562.7(0.92)	664.4(0.78)
Σ	828.6	510.2(1.62)	466.7(1.78)	585.5(1.42)	675.7(1.23)	790.0(1.05)
Σ	45041.4	22727.4(1.98)	11686.6(3.85)	8145.7(5.53)	7227.8(6.23)	6678.7(6.74)

Tabela 6.2: Tempos de execução no YapOr para diferente número de agentes.

ralelismo de granularidade alta apresentam ganhos de velocidade quase lineares, enquanto que os programas com paralelismo de granularidade média apresentam ganhos bastante razoáveis. Como seria de esperar, os programas com paralelismo de granularidade baixa apresentam ganhos de velocidade muito reduzidos. Alguns dos factores que influenciam o menor desempenho dos programas da classe baixa são avaliados nas secções seguintes.

6.2.3 Comparação com o Sistema Muse

A escolha do sistema Muse para comparação com o sistema YapOr é óbvia, pois o YapOr utiliza o modelo de cópia de ambientes e baseia-se em muitos dos conceitos introduzidos no sistema Muse.

Não se pretende fazer uma comparação rigorosa entre os dois sistemas, já que o Muse é um sistema mais robusto (suporta todos os predicados *standard* do Prolog) e foi construído sob um sistema base (SICStus0.6) mais rápido do que o Yap[2]. Mesmo assim, é possível fazer uma comparação preliminar entre os dois sistemas, de forma a obter uma primeira referência da performance do YapOr.

Na tabela 6.3 encontram-se os tempos de execução (em segundos) de alguns dos programas

[2]A versão do Yap utilizada usa um algoritmo de indexação menos sofisticado. Para além disso, existem versões mais recentes (Yap 95 e Yap 96) bastante mais elaboradas e eficientes.

de teste quando executados no sistema Muse com 1, 2, 4 e 6 agentes. Os tempos apresentados foram obtidos, utilizando as ferramentas do Muse para programas de teste, na mesma máquina paralela onde foram obtidos os resultados do YapOr. A tabela 6.3 foi retirada de um estudo apresentado em [CCS94] sob a performance de vários sistemas paralelos.

Programas de Teste	**Número de Agentes** 1	2	4	6
8-queens1	0.47	0.23(2.04)	0.12(3.92)	0.09(5.22)
8-queens2	1.20	0.59(2.03)	0.32(3.75)	0.22(5.45)
salt-mustard	0.14	0.06(2.33)	0.03(4.67)	0.02(7.00)
parse4	0.41	0.21(1.95)	0.13(3.15)	0.26(1.58)
parse5	0.29	0.14(2.07)	0.09(3.22)	0.07(4.14)
db4*10	0.44	0.23(1.91)	0.13(3.38)	0.11(4.00)
db5*10	0.55	0.28(1.96)	0.17(3.24)	0.14(3.93)
house*10	0.31	0.18(1.72)	0.14(2.21)	0.19(1.63)
farmer*100	0.20	0.14(1.43)	0.20(1.00)	0.19(0.98)
Σ	4.01	2.06(1.94)	1.33(3.02)	1.29(3.11)

Tabela 6.3: Tempos de execução no Muse para diferente número de agentes.

A tabela 6.4 mostra o desempenho entre os dois sistemas para o somatório dos tempos de execução (em segundos) do conjunto de programas de teste referido na tabela 6.3. A tabela 6.5 mostra o desempenho entre os dois sistemas para os ganhos de velocidade no mesmo conjunto.

Sistema	**Número de Agentes** 1	2	4	6
YapOr	8.56	4.43	2.51	2.01
Muse	4.01	2.06	1.33	1.29
YapOr / Muse	2.13	2.15	1.89	1.56

Tabela 6.4: Desempenho entre o Muse e o YapOr para o somatório dos tempos de execução.

Os resultados das tabelas 6.4 e 6.5 são animadores. Pela análise da tabela 6.4 conclui-se que o YapOr consegue diminuir sucessivamente a diferença entre o somatório dos tempos de execução à medida que o número de agentes aumenta. Na tabela 6.5 verifica-se que o desempenho do YapOr, para os ganhos de velocidade, é superior ao do Muse para 4 e 6 agentes e idêntico para 2 agentes. Em parte, estes melhores resultados devem-se ao facto do YapOr ser 2.13 vezes mais lento, com 1 agente, do que o Muse, o que lhe garante à

Sistema	Número de Agentes 2	4	6
YapOr	1.93	3.42	4.25
Muse	1.94	3.02	3.11
YapOr / Muse	0.99	1.13	1.37

Tabela 6.5: Desempenho entre o Muse e o YapOr para os ganhos de velocidade.

partida uma maior margem para obter melhores ganhos de velocidade.

6.3 Actividades do Modelo Paralelo

O tempo de execução de um agente envolve um conjunto de actividades que contribuem para o tempo final de execução de um dado programa. Algumas dessas actividades contribuem decisivamente para uma menor performance do sistema. Nesta secção pretende--se, com base no tempo despendido nessas actividades, compreender e explicar em parte os tempos de execução verificados anteriormente.

O tempo de execução de um agente é distribuído pelas seguintes actividades:

Prolog: tempo despendido na execução de Prolog, na verificação de solicitações para partilha de trabalho e na manutenção do registo local de carga actualizado.

Procura: tempo despendido à procura de um agente ocupado com excesso de carga para uma possível operação de partilha de trabalho.

Partilha: tempo despendido nas quatro fases do processo de partilha de trabalho.

Tomar Trabalho: tempo despendido na obtenção de uma nova alternativa de um nó partilhado. Inclui o retrocesso para o nó em questão e o mecanismo de bloqueio para a obtenção dessa alternativa.

Corte: tempo despendido na execução de uma operação de corte na região partilhada da árvore, assim como no retrocesso para um dado nó por acção de uma operação de corte executada mais à esquerda.

A tabela 6.6 apresenta a percentagem do tempo de execução despendido em cada actividade, para um subconjunto dos programas de teste. Nele se incluem dois programas com paralelismo de granularidade alta, *puzzle* e *8-queens2*, dois programas com paralelismo de granularidade média, *salt-mustard* e *parse5*, e um programa com paralelismo de granularidade baixa, *farmer*100*.

Actividade	**Número de Agentes**					
	1	**2**	**4**	**6**	**7**	**8**
puzzle						
Prolog	100.00	99.95	99.56	99.20	99.02	98.68
Procura	0.00	0.02	0.16	0.32	0.41	0.60
Partilha	0.00	0.02	0.17	0.32	0.38	0.50
Tomar Trabalho	0.00	0.01	0.10	0.17	0.19	0.23
Corte	0.00	0.00	0.00	0.00	0.00	0.00
8-queens2						
Prolog	100.00	99.84	98.88	97.68	97.09	95.32
Procura	0.00	0.04	0.38	0.80	1.14	2.21
Partilha	0.00	0.06	0.46	0.95	1.19	1.63
Tomar Trabalho	0.00	0.06	0.27	0.57	0.57	0.83
Corte	0.00	0.00	0.00	0.00	0.01	0.01
salt-mustard						
Prolog	100.00	97.68	86.71	74.56	69.08	63.29
Procura	0.00	0.81	5.02	11.50	13.85	16.87
Partilha	0.00	0.86	5.64	10.17	13.14	15.76
Tomar Trabalho	0.00	0.61	2.51	3.52	3.61	3.88
Corte	0.00	0.04	0.13	0.25	0.32	0.20
parse5						
Prolog	100.00	96.96	85.76	76.54	70.16	61.37
Procura	0.00	0.26	3.68	7.31	9.55	17.92
Partilha	0.00	1.62	7.61	12.74	16.61	17.10
Tomar Trabalho	0.00	1.17	2.96	3.41	3.68	3.61
Corte	0.00	0.00	0.00	0.00	0.00	0.00
farmer*100						
Prolog	100.00	64.69	25.08	15.61	14.50	12.60
Procura	0.00	17.65	32.61	58.89	61.24	64.97
Partilha	0.00	11.59	22.94	21.23	19.81	18.45
Tomar Trabalho	0.00	5.74	19.17	4.18	4.34	3.81
Corte	0.00	0.33	0.20	0.10	0.11	0.17

Tabela 6.6: Percentagem do tempo de execução despendido nas várias actividades.

Para a obtenção dos resultados da tabela 6.6 foi necessário introduzir no sistema um mecanismo para medição do tempo gasto em cada actividade. Este mecanismo trouxe um custo adicional ao tempo de execução de cada programa. Contudo, este custo adicional não deverá introduzir alterações significativas no padrão de execução dos programas de teste.

As actividades *Procura*, *Partilha*, *Tomar Trabalho* e *Corte* são as que poderão influenciar de forma negativa a performance do sistema, já que não correspondem directamente à execução de Prolog. Na tabela 6.6 verifica-se que, no geral, o incremento do número de agentes reflecte-se no incremento do tempo gasto nestas actividades.

À medida que se aumenta o número de agentes, a percentagem de tempo despendido em *Prolog* diminui, enquanto que as de *Procura* e *Partilha* são as que mais aumentam. Como cada programa tem à partida uma quantidade limitada de paralelismo, o incremento do número de agentes provoca um decréscimo natural na quantidade de paralelismo atribuída a cada agente, e daí o aumento do tempo gasto nas actividades de procura por um agente ocupado (*Procura*) e de partilha de trabalho (*Partilha*).

A quantidade de paralelismo diminui ou por aumento do número de agentes ou por diminuição da granularidade do programa de teste. À medida que a quantidade de paralelismo diminui a actividade *Tomar Trabalho* aumenta. Uma das consequências da diminuição da quantidade de paralelismo é o aumento do número de operações de partilha de trabalho. Este aumento leva a que as alternativas disponíveis se situem cada vez mais na região partilhada da árvore, o que explica o consequente aumento da actividade *Tomar Trabalho*.

A percentagem de tempo relativa à actividade *Corte* revela-se mínima, o que de certo modo comprova o facto do suporte ao predicado de corte de alternativas não introduzir custos adicionais significativos no sistema paralelo.

Os programas de teste *puzzle* e *8-queens2* apresentam quantidades de paralelismo suficientes para se conseguir bons desempenhos para, pelo menos, o número máximo de 8 agentes. A existência de paralelismo de granularidade alta faz com que as operações associadas às actividades *Procura* e *Partilha* se efectuem menos vezes, o que se reflecte na menor degradação do desempenho do sistema.

Na tabela 6.2 verifica-se que nos programas *salt-mustard* e *parse5* o ganho de velocidade a partir de 7-8 agentes começa a estabilizar, enquanto que no programa *farmer*100* esse ganho atinge o seu máximo aos 2-4 agentes e decresce para um número maior de agentes. Estas situações devem-se essencialmente ao crescente número de agentes suspensos no sistema por falta de quantidades suficientes de paralelismo, o que origina um aumento das actividades *Procura* e *Partilha*, que não conseguem solucionar o problema dos agentes suspensos. No caso do programa *farmer*100* nota-se uma inflexão na percentagem da actividade *Tomar Trabalho* na passagem de 2 para 4 agentes. Simultaneamente, verifica-se um aumento brusco na percentagem da actividade *Procura*. Estes dois factos, indiciam a baixa granularidade existente nas tarefas executadas.

Uma possível explicação para a diminuição da quantidade de paralelismo encontra-se na tabela 6.7. Esta tabela apresenta o número médio de tarefas executadas por cada agente e o tamanho médio de cada tarefa (expresso no número médio de instruções *call*[3] executadas por tarefa). Uma tarefa é uma porção contínua de trabalho executada na região privada da árvore.

	Número de Agentes					
	1	**2**	**4**	**6**	**7**	**8**
puzzle						
Tarefas	1	213	1780	2763	3095	3813
Calls por tarefa	171024	803	96	62	55	45
8-queens2						
Tarefas	1	152	618	1223	1337	1752
Calls por tarefa	124789	821	202	102	93	71
salt-mustard						
Tarefas	1	53	188	267	308	362
Calls por tarefa	7965	150	42	30	26	22
parse5						
Tarefas	1	432	1164	1503	1675	1872
Calls por tarefa	18101	42	16	12	11	10
farmer*100						
Tarefas	1	4180	5598	6602	6623	6701
Calls por tarefa	11600	3	2	2	2	2

Tabela 6.7: Número médio de tarefas e de instruções *call* por tarefa.

O incremento do número de agentes provoca o aumento do número médio de tarefas executadas por agente. Este aumento está directamente relacionado com o aumento da actividade *Tomar Trabalho* atrás referida.

Por outro lado, o incremento do número de agentes diminui o tamanho médio de cada tarefa. Esta diminuição está directamente relacionada com o decréscimo da granularidade do paralelismo. O tamanho médio de cada tarefa diminui sucessivamente até atingir um

[3]As instruções *call* são uma boa medida para o tamanho médio de cada tarefa, pois são elas que precedem sempre uma possível criação de um ponto de escolha.

ponto de estabilidade, cerca de 10 e de 2 instruções *call* nos programas *parse5* e *farmer*100* respectivamente. Esta estabilização está em parte relacionada com o mecanismo de atraso na divulgação do excesso de carga (rever secção 4.4), que face à inexistência de paralelismo suficiente para o número de agentes disponíveis evita uma crescente degradação da performance do sistema.

Note-se ainda que no programa *farmer*100* o tamanho médio de cada tarefa é muito baixo, o que sugere que as tarefas sejam executadas rapidamente. Em consequência disso, o distribuidor de trabalho tem de intervir mais frequentemente, o que explica o peso excessivo da actividade *Procura* no tempo de execução.

6.4 Trabalho Especulativo

Nesta secção pretende-se fazer uma abordagem superficial ao problema do trabalho especulativo no contexto do YapOr. Tal como foi referido na secção 5.2.1, o YapOr apenas trata a primeira vertente do trabalho especulativo.

A tabela 6.8 apresenta os tempos de execução (em milisegundos) de três dos programas de teste para encontrar apenas a primeira solução do problema[4]. Os três programas seleccionados: *5cubes*, *8-queens2* e *puzzle* apresentam respectivamente pouco, algum e bastante trabalho especulativo da vertente não tratada pelo YapOr.

Programas de Teste	**Número de Agentes**					
	1	**2**	**4**	**6**	**7**	**8**
5cubes	911.4	456.7(1.99)	231.1(3.94)	157.4(5.79)	138.6(6.58)	122.9(7.42)
8-queens2	301.9	182.9(1.65)	125.4(2.41)	94.9(3.18)	70.0(4.31)	63.2(4.78)
puzzle	1104.9	1084.3(1.02)	586.4(1.88)	498.8(2.22)	399.4(2.77)	305.2(3.62)

Tabela 6.8: Tempos de execução para encontrar apenas a primeira solução.

Pela análise da tabela 6.8, constata-se que no programa *puzzle* existe muita computação especulativa, enquanto que no programa *8-queens2* é notória a existência de alguma dessa computação especulativa. No programa *5cubes* o sistema comporta-se eficientemente devido à pouca existência de trabalho especulativo.

[4]Encontrar apenas a primeira solução é concretizado, na prática, com a colocação do predicado de corte após o objectivo inicial do programa de teste.

Desta análise resultam duas conclusões principais. O sistema YapOr necessita de um suporte mais completo para dar uma resposta eficiente a situações de trabalho especulativo abundante. O sistema YapOr dá uma resposta eficiente ao tratamento da primeira vertente do trabalho especulativo, isto é, consegue eliminar rapidamente o conjunto de alternativas cuja exploração se tornou entretanto especulativa por acção de uma operação de corte.

Capítulo 7

Conclusões e Trabalho Futuro

A presente tese reflecte o trabalho desenvolvido no desenho, na implementação e na avaliação de performance do YapOr, um sistema paralelo de execução de Prolog que explora paralelismo-Ou implícito a partir da plataforma Yap de execução sequencial. O YapOr utiliza o modelo de cópia de ambientes, adopta muitos dos conceitos introduzidos no sistema Muse e tem como principal propósito obter altos desempenhos para a execução paralela de Prolog.

Subjacente ao objectivo principal, pretendeu-se que o novo sistema fosse capaz de executar uma ampla classe de programas escritos em Prolog. Para ampliar a classe de programas executados no YapOr foi essencialmente importante tratar o uso do predicado de corte de alternativas.

7.1 Conclusões

Para uma eficiente concretização do trabalho realizado, foi necessário dedicar bastante cuidado e atenção à resolução dos seguintes aspectos:

- O desenho de uma organização de memória capaz de responder com elevada eficácia às necessidades do processo paralelo e em particular às do mecanismo de cópia incremental.

- O desenho das estruturas de dados de suporte a todo o processo paralelo.

- A implementação do mecanismo de cópia incremental de ambientes.
- A implementação das estratégias do distribuidor de trabalho.
- O desenho de uma *interface* entre o distribuidor de trabalho e o emulador de instruções do Yap.
- A implementação do processo de partilha de trabalho.
- O tratamento do predicado de corte de alternativas.

O YapOr baseia-se no mesmo modelo de execução do sistema Muse. Contudo, difere na forma particular como foram solucionados muitos dos aspectos do modelo, dada a ausência de referências específicas e/ou detalhadas sobre a implementação de muitos desses aspectos e dado o facto da estrutura do Yap não ser completamente compatível com as descrições de referência. O conjunto desses aspectos é enumerado de seguida.

- Organização de memória.
- Mecanismos de fecho para permitir a execução atómica de conjuntos de operações.
- Diferente divisão do mecanismo normal de retrocesso pelas instruções `fail`, `retry_me` e `trust_me` entre o Yap e as descrições de referência.
- Adaptação dos procedimentos e sincronizações das fases do processo de partilha de trabalho. No Yap, ao contrário do SICStus, os pontos de escolha e os ambientes são tratados na mesma pilha de execução.
- Cálculo das áreas (das pilhas de execução) a copiar no processo de partilha de trabalho.
- Sincronizações necessárias ao processo de solicitação e partilha de trabalho.
- Mensagens entre agentes para tratar eficientemente o predicado de corte de alternativas.
- Esquema de suporte às soluções que vão sendo encontradas pelo sistema e que potencialmente podem corresponder a trabalho especulativo.
- Esquema de suporte à exploração contínua de todas as soluções de um objectivo.

Por vezes, apesar da relativa facilidade na compreensão dos conceitos abstractos do modelo paralelo, era extremamente complicado implementá-los na realidade particular do sistema Yap. Para tal contribuíram: a obscuridade de certas partes do código envolvido na implementação do Yap, pelas elaboradas e pouco perceptíveis optimizações; os conflitos entre o código herdado e partes do novo código implementado. Estas circunstâncias foram a maior dificuldade na concretização deste trabalho, não só pela sua constante ocorrência mas também pelas profundas assimilações, do código do Yap, que exigiram.

O custo de adaptar o modelo sequencial do sistema Yap ao modelo paralelo do sistema YapOr é relativamente baixo, cerca de 13% na máquina paralela de memória partilhada utilizada. O custo associado a este valor pode ser dividido por vários aspectos: manutenção do registo local de carga actualizado; verificação de solicitações para partilha de trabalho; verificação de mensagens de retrocesso por acção de uma operação de corte; pequenos testes para verificar se o nó mais profundo é partilhado ou privado.

Na execução em paralelo obtiveram-se bons resultados. Os programas com paralelismo de granularidade alta apresentaram ganhos de velocidade quase lineares, 6.87 para 7 agentes e 7.67 para 8 agentes; os programas com paralelismo de granularidade média apresentaram ganhos muito aceitáveis, 4.75 para 7 agentes e 5.01 para 8 agentes; os programas com paralelismo de granularidade baixa, como seria de esperar, apresentaram ganhos de velocidade muito baixos, 1.23 para 7 agentes e 1.05 para 8 agentes.

Na comparação com o sistema Muse, o YapOr conseguiu ganhos de velocidade superiores. No entanto, o facto do somatório dos tempos de execução para 1 agente ser 2.13 vezes superior no YapOr, garante-lhe à partida uma maior margem para obter melhores ganhos de velocidade.

A análise efectuada às actividades de cada agente durante a execução, mostrou que aquelas que mais contribuem para a degradação da eficiência do sistema (*Procura*, *Partilha* e *Tomar Trabalho*), se tornam mais importantes não só com o aumento do número de agentes, mas sobretudo com a diminuição da granularidade das tarefas. O tempo despendido na actividade *Corte* revelou-se desprezável.

Da análise efectuada à exploração de trabalho especulativo concluiu-se que o YapOr necessita de um suporte mais completo para dar uma resposta eficiente a situações de trabalho especulativo abundante. No entanto, verificou-se que consegue tratar eficiente a primeira vertente do trabalho especulativo.

O comportamento global dos programas de teste, bem como os índices de performance dos resultados por eles obtidos, foram no seu todo bastante gratificantes. O objectivo de conseguir obter altos desempenhos para a execução paralela de Prolog e de executar uma ampla classe de programas escritos em Prolog, foi assim atingido.

7.2 Perspectivas de Trabalho Futuro

Esta secção apresenta um conjunto de aspectos susceptíveis de dar continuação ao trabalho realizado. Nele incluem-se alguns dos aspectos que podem ser refinados, através de estratégias e implementações mais elaboradas, e todos aqueles que por não serem fundamentais ao funcionamento do sistema, dentro do âmbito do trabalho, não foram abordados. Estes aspectos são apresentados de seguida.

Distribuidor de trabalho. Uma mais cuidada avaliação das estratégias utilizadas pelo distribuidor de trabalho, pode conduzir a refinamentos e/ou a adaptações dessas estratégias de modo a acelerar o processo paralelo.

Trabalho especulativo. O YapOr, elimina o mais cedo possível o conjunto das alternativas cuja exploração se tornou entretanto especulativa por acção de uma operação de corte. Para tratar por completo o problema do trabalho especulativo, será também necessário introduzir novos esquemas de distribuição de trabalho que minimizem, desde logo, a possibilidade de computações especulativas. Em [AK92] é apresentada uma estratégia denominada como *actively seeking the leftmost available work strategy* que dá resposta a esta necessidade. Esta baseia-se essencialmente em concentrar o maior número possível de agentes nas ramificações com um menor carácter especulativo, de forma a simular o mais possível a execução sequencial. O distribuidor de trabalho ficaria assim com duas estratégias: uma para lidar com o trabalho especulativo e outra para lidar com os agentes suspensos à procura de trabalho.

Predicados de efeitos colaterais. A inclusão de predicados de efeitos colaterais num sistema paralelo influencia as estratégias e os mecanismos usados pelo sistema. Existem dois tipos principais de efeitos colaterais, aqueles que não afectam a configuração da árvore de procura directamente (*efeitos colaterais fracos*), e aqueles que afectam (*efeitos colaterais fortes*). Os predicados *read* e *write* pertencem à primeira categoria, enquanto que os predicados *assert* e *retract* pertencem à segunda. A inclusão deste tipo de predicados no sistema requer algum cuidado, principalmente no que diz respeito ao suporte dos predicados de efeitos colaterais fortes. No entanto, parte dos mecanismos implementados para suporte do predicado de corte podem ser facilmente adaptados para, em conjunto com novas implementações que se mostrem necessárias, dar suporte a estes predicados. Em [Cie92, AK90a, Kar92b] apresentam-se algumas ideias para uma eficiente implementação deste suporte.

Predicados *findall*, *bagof* e *setof*. O YapOr quando executa um objectivo em paralelo procura, por defeito, todas as soluções. Os predicados *findall*, *bagof* e *setof* estão relacionados com a procura de todas as soluções. Com a inclusão destes predicados no YapOr, esta procura deve realizar-se apenas quando da invocação destes predicados. Por conseguinte, uma adaptação cuidada do tratamento dado pelo YapOr à procura por todas as soluções é suficiente para suportar estes predicados.

Expansão dinâmica de memória. O YapOr despreza o facto da memória inicialmente solicitada poder vir a ser insuficiente mediante o decorrer da execução do programa. É necessário encontrar esquemas de estruturação da memória que tornem, em tempo de execução, a expansão de uma dada área dos espaços global e locais de endereçamento numa operação suficientemente eficiente.

Garbage collection. A operação de *garbage collection* estreita a informação guardada nas pilhas de execução, eliminando partes desnecessárias das cadeias de referências[1]. Num sistema paralelo a execução desta operação, além do tempo que demora, provoca inconsistências entre as pilhas de execução dos agentes durante o mecanismo de cópia incremental. O YapOr evita a execução de qualquer operação de *garbage collection*. Em [AK90b] são apresentados alguns mecanismos para tratar a operação de *garbage collection* passíveis de serem adicionados ao YapOr.

7.3 Nota de Fecho

A particular contribuição desta tese reflecte-se basicamente a dois níveis distintos. A um nível estritamente individual (do autor) e a um nível mais geral e impessoal.

[1] Para obtermos o valor de uma variável é habitualmente necessário percorrer uma cadeia de referências até atingirmos uma célula de memória onde se encontra o seu real valor. Com a operação de *garbage collection* pretende-se diminuir o mais possível o tamanho dessa cadeia.

A nível individual sobressaem os conhecimentos científicos adquiridos e o estado de maturidade alcançado, na área específica de Programação Lógica Paralela. Essa contribuição pode resumir-se a três pontos principais: (i) o estudo de implementações sequenciais de Prolog, como são o caso da WAM e do Yap; (ii) o estudo de alguns dos modelos de paralelismo-Ou existentes para a execução paralela e o estudo particular do sistema Muse; (iii) o desenho e implementação de um sistema paralelo partindo de um sistema sequencial de base.

A nível global destaca-se essencialmente o património científico consequência do trabalho realizado e alguns dos bons resultados alcançados, que poderão constituir uma fonte de motivação para uma futura prossecução do trabalho realizado.

Referências

[AK90a] Khayri A. M. Ali and Roland Karlsson. Full Prolog and Scheduling OR-Parallelism in Muse. *International Journal of Parallel Programming*, 19(6):445–475, December 1990.

[AK90b] Khayri A. M. Ali and Roland Karlsson. The Muse Approach to OR-Parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, April 1990.

[AK91] Hassan Aït-Kaci. *Warren's Abstract Machine — A Tutorial Reconstruction.* MIT Press, 1991.

[AK92] Khayri A. M. Ali and Roland Karlsson. Scheduling Speculative Work in Muse and Performance Results. *International Journal of Parallel Programming*, 21(6), December 1992.

[AKM92] Khayri A. M. Ali, Roland Karlsson, and Shyam Mudambi. Performance of Muse on Switch-Based Multiprocessor Machines. *New Generation Computing*, 11(1,4):81–103, 1992.

[Car90] Mats Carlsson. *Design and Implementation of an OR-Parallel Prolog Engine.* PhD thesis, The Royal Institute of Technology, Stockholm, March 1990.

[CCS94] Vítor Santos Costa, Manuel Eduardo Correia, and Fernando Silva. Aurora, Andorra-I and Friends on the Sun. In *Proceedings of the Post-ILPS'94 Workshop on Design and Implementation of Parallel Logic Programming Systems*, Ithaca, New York, November 1994.

[Cie92] Andrzej Ciepielewski. Scheduling in Or-parallel Prolog Systems: Survey, and Open Problems. *International Journal of Parallel Programming*, 1992.

[CW88] Mats Carlsson and Johan Widen. SICStus Prolog User's Manual. SICS Research Report R88007B, Swedish Institute of Computer Science, October 1988.

[DSCRA89] L. Damas, V. Santos Costa, R Reis, and R. Azevedo. *YAP User's Guide and Reference Manual.* Centro de Informática da Universidade do Porto, 1989.

[GACH96] G. Gupta, Khayri A. M. Ali, Mats Carlsson, and M. Hermenegildo. Parallel Execution of Prolog Programs: A Survey. March 1996. Preliminary Version.

[GJ93] G. Gupta and B. Jayaraman. Analysis of Or-parallel Execution Models. *ACM Transactions on Programming Languages*, 15(4):659–680, September 1993.

[Kar92a] Roland Karlsson. *A High Performance OR-parallel Prolog System.* PhD thesis, The Royal Institute of Technology, Stockholm, March 1992.

[Kar92b] Roland Karlsson. How to Build Your Own OR-Parallel Prolog System. SICS Research Report R92:03, Swedish Institute of Computer Science, March 1992.

[Kog91] Peter M. Kogge. *The Architecture of Symbolic Computers.* McGraw-Hill, 1991.

[Kow79] Robert A. Kowalski. *Logic for Problem Solving.* Elsevier North-Holland Inc., Amsterdam, 1979.

[LBD+88] Ewing Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, David H. D. Warren, A. Calderwood, P. Szeredi, Seif Haridi, P. Brand, M. Carlsson, A. Ciepelewski, and B. Hausman. The Aurora Or-parallel Prolog System. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 819–830. ICOT, Tokyo, November 1988.

[SC88] Vítor Santos Costa. *Implementação de Prolog.* Provas de Aptidão Pedagógica e Capacidade Científica, Universidade do Porto, Dezembro 1988.

[Ste92] W. Richard Stevens. *Advanced Programming in the UNIX Environment.* Addison–Wesley Publishing Company, 1992.

[Sze89] Péter Szeredi. Performance Analysis of the Aurora Or-Parallel Prolog System. In *Proceedings of the North American Conference on Logic Programming*, pages 713–732. MIT Press, October 1989.

[War77] David H. D. Warren. *Applied Logic—Its Use and Implementation as a Programming Tool.* PhD thesis, Edinburgh University, 1977. Available as Technical Note 290, SRI International.

[War83] David H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, September 1983.

Apêndice A

Código dos Programas de Teste

5cubes

```
c5(X) :- par, go_c5(X).

go_c5(Sol):-
   cubes(5,Qs),
   solve(Qs,[],Sol).

solve([],Rs,Rs).
solve([C|Cs],Ps,Rs):-
   set(C,P),
   check(Ps,P),
   solve(Cs,[P|Ps],Rs).

check([],_).
check([q(A1,B1,C1,D1)|Ps],P):-
   P=q(A2,B2,C2,D2),
   A1=\=A2,
   B1=\=B2,
   C1=\=C2,
   D1=\=D2,
   check(Ps,P).

set(q(P1,P2,P3),P):-
   rotate(P1,P2,P).
set(q(P1,P2,P3),P):-
   rotate(P2,P1,P).
set(q(P1,P2,P3),P):-
   rotate(P1,P3,P).
set(q(P1,P2,P3),P):-
   rotate(P3,P1,P).
set(q(P1,P2,P3),P):-
   rotate(P2,P3,P).
set(q(P1,P2,P3),P):-
   rotate(P3,P2,P).

rotate(p(C1,C2),p(C3,C4),q(C1,C2,C3,C4)).
rotate(p(C1,C2),p(C3,C4),q(C1,C2,C4,C3)).
rotate(p(C1,C2),p(C3,C4),q(C2,C1,C3,C4)).
rotate(p(C1,C2),p(C3,C4),q(C2,C1,C4,C3)).

cubes(5,[q(p(2,1),p(1,4),p(1,3)),
         q(p(3,2),p(2,0),p(3,4)),
         q(p(1,4),p(3,1),p(0,4)),
         q(p(1,0),p(2,2),p(0,4)),
         q(p(4,2),p(4,3),p(0,3))]).
```

8-queens1

```
q1(X) :- par, go_q1(8,X).

go_q1(X2,X1) :-
   poss_rows(X2,X0),
   solve(X0,0,[],X1).

solve([X7|X6],X5,X4,X3) :-
   is(X2,X5+1),
   delete([X7|X6],X1,X0),
   safe_q1(X2,X1,X4),
   solve(X0,X2,[s(X2,X1)|X4],X3).
solve([],X1,X0,X0).

delete([X3|X2],X1,[X3|X0]) :-
   delete(X2,X1,X0).
delete([X1|X0],X1,X0).

not_threaten(A,B,C,D) :-
   threaten(A,B,C,D,Ans),
   Ans==ok.

threaten(X4,X3,X2,X1,no) :-
   is(X0,X4+X3),
   is(X0,X2+X1),
   !.
threaten(X4,X3,X2,X1,no) :-
   is(X0,X4-X3),
   is(X0,X2-X1),
   !.
threaten(_,_,_,_,ok).

safe_q1(X4,X3,[s(X2,X1)|X0]) :-
   not_threaten(X4,X3,X2,X1),
   safe_q1(X4,X3,X0),
   !.
safe_q1(X1,X0,[]).

poss_rows(X1,X0) :-
   poss_rows(X1,[],X0).
poss_rows(0,X0,X0) :-
   !.
poss_rows(X3,X2,X1) :-
   is(X0,X3-1),
   poss_rows(X0,[X3|X2],X1).
```

8-queens2

```
q2(X) :- par, go_q2(8,X).
go_q2(X1,X0) :-
   solve(X1,[],X0).
snint(X2,X1) :-
   is(X0,X1-1),
   '>'(X0,0),
   snint(X2,X0).
snint(X0,X0).
solve(X2,[square(X2,X1)|X0],[square(X2,X1)|X0]) :-
   !.
solve(X3,X2,X1) :-
   newsquare(X2,X0,X3),
   solve(X3,[X0|X2],X1).
newsquare([],square(1,X1),X0) :-
   snint(X1,X0).
newsquare([square(X5,X4)|X3],square(X2,X1),X0) :-
   is(X2,X5+1),
   snint(X1,X0),
   not_threatened(X5,X4,X2,X1),
   safe_q2(X2,X1,X3).
safe_q2(X1,X0,[]).
safe_q2(X4,X3,[square(X2,X1)|X0]) :-
   not_threatened(X2,X1,X4,X3),
   safe_q2(X4,X3,X0).
not_threatened(X3,X2,X1,X0) :-
   not_thr(X3,X2,X1,X0,Ans),
   Ans==ok.
not_thr(X3,X2,X1,X0,no) :-
   threatened(X3,X2,X1,X0 ),
   !.
not_thr(X3,X2,X1,X0,ok).
threatened(X2,X1,X2,X0) :-
   !.
threatened(X2,X1,X0,X1) :-
   !.
threatened(X4,X3,X2,X1) :-
   is(X0,X4-X3),
   is(X0,X2-X1),
   !.
threatened(X4,X3,X2,X1) :-
   is(X0,X4+X3),
   is(X0,X2+X1).
```

farmer

```
fm(X) :- par, go_fm(X).
go_fm(X0) :-
   solve(fwgc(e,e,e,e),fwgc(w,w,w,w),X0).
solve(X2,X1,X0) :-
   path(X2,X1,[X2],X0).
path(X1,X1,X0,X0).
path(X4,X3,X2,X1) :-
   move(X4,X0),
   safe(X0),
   \+(already(X0,X2)),
   path(X0,X3,[X0|X2],X1).
already(X1,[X1|X0]) :-
   !.
already(X2,[X1|X0]) :-
   already(X2,X0).
move(fwgc(X4,X3,X2,X1),fwgc(X0,X3,X2,X1)) :-
   opp(X4,X0).
move(fwgc(X3,X3,X2,X1),fwgc(X0,X0,X2,X1)) :-
   opp(X3,X0).
move(fwgc(X3,X2,X3,X1),fwgc(X0,X2,X0,X1)) :-
   opp(X3,X0).
move(fwgc(X3,X2,X1,X3),fwgc(X0,X2,X1,X0)) :-
   opp(X3,X0).
opp(e,w).
opp(w,e).
safe(fwgc(X2,X1,X2,X0)) :-
   !.
safe(fwgc(X1,X1,X0,X1)).
```

house

```
hs(X) :- par, go_hs(X).
go_hs(config(X27,X26,X25,X24,X23 )) :-
   X27=[norway,X22,X21,X20,X19],
   X26=[X18,X17,X16,X15,X14],
   X25=[X13,X12,X11,X10,X9],
   X24=[X8,X7,milk,X6,X5],
   X23=[X4,X3,X2,X1,X0],
   testp(england,red,X27,X26),
   testp(japan,craven,X27,X23),
   beside(norway,blue,X27,X26),
   on_right(white,green,X26,X26),
   testp(yellow,kool,X26,X23),
   testp(spain,dog,X27,X25),
   testp(old_gold,snail,X23,X25),
   testp(gitane,wine,X23,X24),
   testp(ukr,tea,X27,X24),
   testp(green,coffe,X26,X24),
   beside(chesterfield,fox,X23,X25),
   beside(kool,horse,X23,X25),
   memb(zebra,X25),
   memb(water,X24).
memb(X2,[X1|X0]) :-
   memb(X2,X0).
memb(X1,[X1|X0]).
testp(X3,X2,[X3|X1],[X2|X0]).
testp(X5,X4,[X3|X2],[X1|X0]) :-
   testp(X5,X4,X2,X0).
on_right(X4,X3,[X4|X2],[X1,X3|X0]).
on_right(X5,X4,[X3|X2],[X1|X0]) :-
   on_right(X5,X4,X2,X0).
on_left(X4,X3,[X2,X4|X1],[X3|X0]).
on_left(X5,X4,[X3|X2],[X1|X0]) :-
   on_left(X5,X4,X2,X0).
beside(X3,X2,X1,X0) :-
   on_right(X3,X2,X1,X0).
beside(X3,X2,X1,X0) :-
   on_left(X3,X2,X1,X0).
```

puzzle

```
pz(X) :- par, go_pz(X).
go_pz([A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S]) :-
   List=[1,2,3,4,5,6,7,8,9,10,11,
         12,13,14,15,16,17,18,19],
   member(A,List,La),
   member(B,La,Lb),
   C is 38-A-B,
   member(C,Lb,Lc),
   A<C,
   member(D,Lc,Ld),
   H is 38-A-D,
   member(H,Ld,Lh),
   A<H,
   C<H,
   member(E,Lh,Le),
   member(F,Le,Lf),
   G is 38-D-E-F,
   member(G,Lf,Lg),
   L is 38-C-G,
   member(L,Lg,Ll),
   A<L,
   member(I,Ll,Li),
   M is 38-B-E-I,
   member(M,Li,Lm),
   Q is 38-H-M,
   member(Q,Lm,Lq),
   A<Q,
   member(J,Lq,Lj),
   N is 38-C-F-J-Q,
   member(N,Lj,Ln),
   K is 38-H-I-J-L,
   member(K,Ln,Lk),
   P is 38-B-F-K,
   member(P,Lk,Lp),
   S is 38-L-P,
   member(S,Lp,Ls),
   A<S,
   R is 38-Q-S,
   member(R,Ls,Lr),
   38 is D+I+N+R,
   member(O,Lr,_Lo),
   38 is M+N+O+P,
   38 is A+E+J+O+S,
   38 is G+K+O+R.
member(X,[X|Y],Y).
member(X,[X2|Y],[X2|Y2]) :-
   X\==X2,
   member(X,Y,Y2).
```

salt-mustard

```
sm(X) :- par, go_sm(X).
go_sm(X) :-
   generate_sm(X),
   test(X).
generate_sm([Barry,Cole,Dix,Lang,Mill]) :-
   possibility(Barry),
   possibility(Cole),
   possibility(Dix),
   possibility(Lang),
   possibility(Mill).
possibility(both).
possibility(neither).
possibility(salt).
possibility(mustard).
test(X) :-
   constraint1(X),
   constraint2(X),
   constraint3(X),
   constraint4(X),
   constraint5(X).
constraint1([Barry,Cole,Dix,Lang,Mill]) :-
   iff_took_salt_then_oneof_or_oneof(Barry,Cole,Lang),
   iff_took_mustard_then_eq(Barry,Dix,neither,Mill,both).
constraint2([Barry,Cole,Dix,Lang,Mill]) :-
   iff_took_salt_then_oneof_or_neither(Cole,Barry,Mill),
   iff_took_mustard_then_eq(Cole,Dix,both,Lang,both).
constraint3([Barry,Cole,Dix,Lang,Mill]) :-
   iff_took_salt_then_eq(Dix,Barry,neither,Cole,both),
   iff_took_mustard_then_eq(Dix,Lang,neither,Mill,neither).
constraint4([Barry,Cole,Dix,Lang,Mill]) :-
   iff_took_salt_then_oneof_or_oneof(Lang,Barry,Dix),
   iff_took_mustard_then_eq(Lang,Cole,neither,Mill,neither).
constraint5([Barry,Cole,Dix,Lang,Mill]) :-
   iff_took_salt_then_eq(Mill,Barry,both,Lang,both),
   iff_took_mustard_then_oneof_or_oneof(Mill,Cole,Dix).
oneof(salt).
oneof(mustard).
took_salt(salt).
took_salt(both).
took_mustard(mustard).
took_mustard(both).
iff_took_salt_then_oneof_or_oneof(A,B,C):-
   took_salt(A),
   !,
   oneof_or_oneof(B,C).
iff_took_salt_then_oneof_or_oneof(A,B,C):-
   iff_ts_ooo(A,B,C,Ans),
   Ans==ok.
iff_ts_ooo(A,B,C,no) :-
   oneof_or_oneof(B,C),
   !.
iff_ts_ooo(A,B,C,ok).
iff_took_mustard_then_oneof_or_oneof(A,B,C):-
   took_mustard(A),
   !,
   oneof_or_oneof(B,C).
iff_took_mustard_then_oneof_or_oneof(A,B,C):-
   iff_tm_ooo(A,B,C,Ans),
   Ans==ok.
iff_tm_ooo(A,B,C,no) :-
   oneof_or_oneof(B,C),
   !.
iff_tm_ooo(A,B,C,ok).
iff_took_salt_then_oneof_or_neither(A,B,C):-
   took_salt(A),
   !,
   oneof_or_neither(B,C).
iff_took_salt_then_oneof_or_neither(A,B,C):-
   iff_ts_oon(A,B,C,Ans),
   Ans==ok.
iff_ts_oon(A,B,C,no) :-
   oneof_or_neither(B,C),
   !.
iff_ts_oon(A,B,C,ok).
iff_took_salt_then_eq(A,B,BV,C,CV):-
   took_salt(A),
   !,
   eq_or_eq(B,BV,C,CV).
iff_took_salt_then_eq(A,B,BV,C,CV):-
   iff_ts_eoe(A,B,BV,C,CV,Ans),
   Ans==ok.
iff_ts_eoe(A,B,BV,C,CV,no) :-
   eq_or_eq(B,BV,C,CV),
```

```
   !.
iff_ts_eoe(A,B,BV,C,CV,ok).

iff_took_mustard_then_eq(A,B,BV,C,CV):-
   took_mustard(A),
   !,
   eq_or_eq(B,BV,C,CV).
iff_took_mustard_then_eq(A,B,BV,C,CV):-
   iff_tm_eoe(A,B,BV,C,CV,Ans),
   Ans==ok.

iff_tm_eoe(A,B,BV,C,CV,no) :-
   eq_or_eq(B,BV,C,CV),
   !.
iff_tm_eoe(A,B,BV,C,CV,ok).
```

```
oneof_or_oneof(A,B):-
   oneof(A).
oneof_or_oneof(A,B):-
   oneof(B).

oneof_or_neither(A,B):-
   oneof(A).
oneof_or_neither(A,B):-
   B==neither.

eq_or_eq(A,AV,B,BV):-
   A==AV.
eq_or_eq(A,AV,B,BV):-
   B==BV.
```

parse4

```
ps4(X) :- par, go_ps4(X).

go_ps4(X) :-
  sentence(X, [which,european,countries,bordering,a,country,in,asia,contain,a,city,
              the,population,of,which,is,more,than,nb(1),million,?],[],[],[]).

...
```

parse5

```
ps5(X) :- par, go_ps5(X).

go_ps5(X) :-
   sentence(X,[which,european,countries,that,contain,a,city,the,population,of,
              which,is,more,than,nb(1),million,and,that,border,a,country,in,
              asia,containing,a,city,the,population,of,which,is,more,than,
              nb(3),million,border,a,country,in,western_europe,containing,a,
              city,the,population,of,which,is,more,than,nb(1),million,?],[],[],[]).

...
```

db4

```
db4(X) :- par, go_db4(X).

go_db4(G) :-
   in(G,asia),
   country(G),
   borders(F,G),
   european(F),
   country(F),
   in(H,F),
   city(H),
   population(H,I),
   exceeds(I,1--million).

...
```

db5

```
db5(X) :- par, go_db5(X).

go_db5(D) :-
   in(D,asia),
   country(D),
   borders(A,D),
   european(A),
   country(A),
   borders(A,G),
   in(G,western_europe),
   country(G),
   in(H,G),
   city(H),
   population(H,I),
   exceeds(I,1--million),
   in(B,A),
   city(B),
   population(B,C),
   exceeds(C,1--million),
   in(E,D),
   city(E),
   population(E,F),
   exceeds(F,3--million).

...
```