Rolando da Silva Martins

Um Sistema baseado na Separação de Ambientes para Execução Distribuída de Prolog

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2003

Rolando da Silva Martins

Um Sistema baseado na Separação de Ambientes para Execução Distribuída de Prolog

Tese submetida à Faculdade de Ciências da Universidade do Porto para obtenção do grau de Mestre em Informática

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2003

Para Liliana com Amor.

Agradecimentos

Gostaria de agradecer a colaboração e apoio prestado por todos aqueles que a algum nível contribuíram para a elaboração deste trabalho. Em particular, desejo agradecer o trabalho incansável de orientação por parte do Prof. Doutor Fernando da Silva e do Prof. Doutor Ricardo Rocha, que demonstraram sempre total disponibilidade na prestação de ajuda.

Gostaria também de agradecer ao Prof. Doutor Vítor Santos Costa pela sua disponibilidade em vários momentos deste trabalho.

Quero carinhosamente agradecer à Liliana pelo apoio e motivação constante que me deu, que de outra maneira este trabalho não teria chegado a bom porto.

Resumo

A linguagem Prolog é um dos principais instrumentos usado pelos investigadores na área de Inteligência Artificial, em particular nas áreas de Aprendizagem Automática e Processamento de Linguagem Natural. Para tal, muito tem contribuído o excelente desempenho dos sistemas Prolog sequenciais. Um contributo adicional aos ganhos de desempenho advém do desenvolvimento de sistemas de execução paralela de Prolog. Estes sistemas exploram paralelismo implícito à linguagem, não requerendo desenvolvimento específico de programas.

Esta tese tem por objectivo desenhar e implementar um sistema de execução paralela de Prolog, capaz de explorar paralelismo-Ou em arquitecturas paralelas de memória distribuída de baixo custo, como são os *clusters beowulf*. O sistema, designado por YapDss, tem como base de desenvolvimento o sistema YapOr que explora paralelismo-Ou em arquitecturas de memória partilhada e usa o modelo de separação de ambientes distribuído para organizar a sua estratégia de divisão de trabalho pelos vários agentes do sistema paralelo.

O desenvolvimento do YapDss requereu a análise e compreensão dos sistemas YAP e YapOr, no sentido de se identificar e modificar estruturas de dados e mecanismos necessários à gestão de trabalho num ambiente distribuído. Algumas das contribuições deste trabalho são: o desenvolvimento de um mecanismo novo para partilha de código compilado do programa a executar pelos vários agentes de execução, de técnicas capazes de minimizar a quantidade de informação necessária à partilha de trabalho, de um protocolo de troca de informação relevante para gestão de trabalho e que simultaneamente mantivesse o número de mensagens num valor reduzido de modo a evitar atrasos na execução devidos à comunicação, e de um módulo de comunicação em MPI para permitir a distribuição das computações.

Por último, apresenta-se um estudo inicial da performance do YapDss num conjunto de programas Prolog habitualmente usados como programas de avaliação deste tipo de sistemas. O YapDss mostra um excelente desempenho em programas com grandes quantidades de paralelismo e de granularidade alta. Como limitação do sistema refira-se o facto de apenas suportar programas que não contêm o predicado de corte.

Abstract

Prolog is one of the major languages used by researchers in the areas of Artificial Intelligence, specially in areas such us Machine Learning and Natural Language Processing. This is explained by the excellent sequential performance of Prolog systems. The development of parallel Prolog systems have further contributed to excel the language speed. These systems exploit implicit parallelism within the language and have the advantage of not requiring extra work in program development.

The main goal of this work is to design and implement a parallel Prolog system, capable of exploiting Or-parallelism on low cost distributed memory parallel architectures, such us the beowulf clusters. The system, named YapDss, builds on the work of the YapOr system, an or-parallel system for shared memory architectures, and uses the distributed stack-splitting binding model to represent computation state and work sharing among the parallel computational agents.

The development of YapDss required the analysis and understanding of the systems YAP and YapOr, in order to identify and modify the data structures and mechanisms necessary to schedule work in a distributed environment. Some of the contributions of YapDss are: a mechanism to share program code among the computing agents; techniques to minimize the scheduling information necessary for work sharing; a protocol for workload propagation that keeps the number of messages to a minimum to avoid communication overheads; and a communication interface using MPI to support distributed computations.

Lastly, we present an initial performance study of YapDss using a set of benchmark Prolog programs commonly used in the evaluation of similar systems. The YapDss system shows excellent performance on programs with reasonably large amounts of large-grain parallelism. The main limiting factor so far is that YapDss lacks support for the cut predicate.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Com o desenvolvimento da Internet têm surgido novos paradigmas de programação que procuram tirar partido da enorme quantidade de computadores ligados por essa rede. O paradigma Peer-to-Peer (P2P) é disso um bom exemplo. Estes paradigmas são distribuídos e o seu uso crescente deve-se a factores como:

Aproveitamento computacional dos meios existentes. A *Internet* tem sido o grande motor da computação distribuída, devido ao aproveitamento do poder computacional dos milhões de nós ligados a ela.

Baixo custo. Mesmo sem usar a *Internet*, num ambiente fechado, como os *clusters*, a solução distribuída tem a enorme vantagem de custos, em relação a um sistema paralelo.

Dado o potencial de paralelismo existente na avaliação de expressões lógicas, é possível o desenvolvimento de algoritmos eficientes para a gestão transparente desse paralelismo, libertando assim o utilizador dessa árdua tarefa. Neste tipo de paralelismo (*Paralelismo implícito*), existe uma investigação mais intensa nos seguintes tipos:

Paralelismo-Ou . Este tipo de paralelismo, tem base na distribuição das várias alternativas numa árvore de execução, aparenta ser o mais fácil e eficiente, contudo na prática, uma implementação eficiente revela-se difícil, muito em parte à dificuldade de uma boa distribuição da carga pelos vários nós do sistema e também devido à difícil tarefa de representar os vários ambientes simultâneos que existem na execução de uma programa.

Paralelismos-E Ao contrário do Paralelismo-Ou, este paralelismo tem como base a distribuição dos diferentes sub-golos de um golo, tendo como principal dificuldade a representação e gestão eficiente das variáveis, visto existirem dependências inerentes a esta aproximação.

O Paralelismo-Ou tem sido alvo preferencial da investigação em sistema de execução paralela dada a maior facilidade de implementações eficientes, e foi por isso escolhido como modelo para esta tese.

A estrutura não-determinística da programação em lógica é normalmente representada na forma de uma árvore de procura (também conhecida por árvore-Ou) [VPHG01]. Cada nó representa um ponto de escolha, isto é, um ponto na execução onde múltiplas cláusulas estão disponíveis para resolver um sub-objectivo[1]. As folhas da árvore representam pontos de falha (um ponto na execução onde não conseguimos unificar um dado sub-objectivo com nenhuma claúsula existente) ou um ponto de sucesso (uma solução para o sub-objectivo inicial). Uma execução sequencial nada mais é do que a exploração da árvore de procura seguindo alguma estratégia pré-definida de procura. Enquanto a execução sequencial usa uma claúsula de cada vez para resolver um dado sub-objectivo, e eventualmente utilizando o mecanismo de retrocesso[2] para explorar as diferentes cláusulas alternativas, o Paralelismo-Ou permite o uso de várias *threads* de execução (agentes) para a exploração paralela de várias alternativas de um dado ponto de escolha. Se uma alternativa inexplorada é encontrada, o agente toma-a e começa a respectiva execução. Um agente pára quando ocorre uma falha (chega a uma folha da árvore) ou quando encontra uma solução. No caso de uma falha, o agente activa o mecanismo de retrocesso, isto é, retrocede na árvore de procura, procurando alternativas por explorar nos pontos de escolha da sua árvore de procura. Intuitivamente, podemos dizer que o Paralelismo-Ou permite a resolução concorrente de várias alternativas de um objectivo original.

A maior parte da investigação de Paralelismo-Ou é orientada para máquinas com memória partilhada. Esta tese foca o desenvolvimento de um modelo de execução distribuída de Prolog em arquitecturas com memória distribuída.

Estudos teóricos[AK90b] e experimentais[RPG99] revelam que a cópia de ambientes, em particular da cópia incremental de ambientes, como sendo uma das mais eficientes implementações para exploração de Paralelismo-Ou. A cópia de ambientes permite a partilha de trabalho através da cópia do estado computacional de um agente activo (com alternativas inexploradas) para outro agente (que está suspenso). A ideia de cópia incremental de ambientes passa pela cópia de somente da **diferença** dos estados computacionais entre os 2 agentes, tendo já sido usada com bastante sucesso em máquinas de exploração paralela (Muse [AK90b] e YapOr [Roc96]). De modo a diminuir as comunicações necessárias para a cópia de ambientes, uma técnica foi proposta, chamada de *Stack Splitting* [GP99]. A conjugação destas duas técnicas origina a técnica de *Incremental Stack Splitting*[3]. Esta técnica tem como suporte um mecanismo de etiquetas que baseado na sua comparação, permite determinar quais os segmentos de memória que precisam de ser copiados entre os 2 agentes. Nesta tese também está descrita a estratégia de partilha de trabalho usada para tirar proveito

[1] A palavra 'objectivo' traduz a palavra ' goal', habitualmente usada neste contexto na literatura inglesa.

[2] O termo retrocesso traduz a palavra 'backtracking', usada usualmente na literatura inglesa

[3] Esta designação tem como tradução a cópia incremental de ambientes com *Stack Splitting*

do *Incremental Stack Splitting*. Ambos os mecanismo foram implementados tendo por base um mecanismo de transferência de mensagens, tendo sido usada a plataforma *MPI*[4].

O objectivo desta tese passa pelo desenvolvimento de um sistema Prolog paralelo, capaz de explorar implicitamente paralelismo-Ou, em arquitecturas de memória distribuída tendo como base a técnica de cópia incremental de ambiente com Stack Splitting. Este trabalho tem por base o sistema **YapOr** [Roc96, RSC99], um sistema Prolog paralelo para arquitecturas de memória partilhada.

1.1 Objectivos e Contribuições

Tendo como base o trabalho feito, ao longo dos anos sobre computação distribuída e paralela, o objectivo desta tese é a exploração máxima do paralelismo num ambiente distribuído. O meio usado para conseguir superar as limitações impostas na partilha de trabalho por uma arquitectura distribuída, foi através do uso de cópia incremental com *Stack Splitting*, que possibilitou a limitação da partilha de informação, que de outra forma tornariam qualquer aproximação a este problema bastante difícil dado os custos associadas à manutenção deste tipo de informação. No entanto foi necessário a introdução de novos conceitos para superar as dificuldades impostas pelo ambiente distribuído. O conjunto destes conceitos é enumerado de seguida.

- Cálculo do primeiro ponto de escolha comum entre 2 agentes (*LCA*[5]).
- Criação de um mecanismo de etiquetas para mapeamento da árvore de computação de modo a possibilitar o *LCA*.
- Protocolo de comunicação para gestão e transferência de informação.
- Criação de um interface para fazer a ligação entre o YapDss e a plataforma de comunicação MPI.
- Mecanismo de sincronização do código do programa a executar pelos vários agentes do sistema.
- A introdução do conceito de 'terminação da computação', que tem como finalidade a detecção do fim da computação.
- Mecanismo para agrupamento das soluções encontradas pelos vários agentes do sistema.

[4] O *MPI* ou *Message Passing Interface* tem como tradução interface de transferência de mensagens

[5] Em toda a tese, a designação *LCA* é a abreviatura de *Least Common Ancestor* e traduz a designação de primeiro nó comum.

- Adaptação do mecanismo de cálculo das áreas (das pilhas de execução) a copiar no processo de partilha, de modo a usar a informação do *LCA* e do sistema de etiquetas.
- Adaptação do mecanismo de retrocesso pelas instruções fail, retry_me e trust_me, de modo a suportar o YapDss.
- A introdução do conceito de *offset*, que se traduziu na introdução de um novo campo nos pontos de escolha, tendo como finalidade a gestão das alternativas a tomar pelos vários agentes.
- Adaptação das instruções com ligação directa à manipulação dos pontos de escolha, de modo a possibilitar o acesso a todas as alternativas de um ponto de escolha a partir de uma posição comum.

Todos estes conceitos e mecanismo são explicados em detalhe nos capítulos seguintes.

Os resultados obtidos na execução dos programas de teste utilizados, bem como os índices de performance por eles obtidos, foram e são motivadores. O objectivo de conseguir altos desempenhos num sistema de execução distribuída de Prolog, foi assim satisfatoriamente alcançado. Os resultados alcançados comprovam que o conceito de distribuição estática de trabalho se adapta com eficiência à execução distribuída de Prolog, tendo como base a exploração do paralelismo-Ou.

1.2 Estrutura da Tese

A tese está dividida em sete capítulos, que introduzem de forma gradual o trabalho realizado. Os tópicos dos capítulos são os seguintes:

Capítulo 1: Introdução Faz uma primeira aproximação ao trabalho da tese, referenciado o conceito de **Paralelismo**. Providencia os objectivos e estrutura da tese.

Capítulo 2: Programação Lógica, Paralelismo e Stack Splitting Providencia conceitos sobre programação em lógica, programação paralela, e da máquina abstracta *WAM*. Neste capítulo também é descrito o processo de partilha de trabalho estático, o *Distributed Stack Splitting*(DSS).

Capítulo 3: Sistema YapOr Apresenta os conceitos e o desenho da implementação de execução paralela, o *YapOr*. Os conceitos abrangem os seguintes tópicos: modelo básico de execução; cópia incremental; e distribuição de trabalho.

Capítulo 4: Sistema YapDss Este capítulo ilustra os processos do modelo *YapDss*: modelo de execução básico; cópia incremental com *Stack Splitting*; distribuidor de trabalho; e terminação de computação. Os conceitos apresentados são usados no capítulo de implementação.

Capítulo 5: Extensão do Yapor para Suportar YapDss Descreve as extensões realizadas ao *YapOr* para suportar o *YapDss*

Capítulo 6: Implementação Apresenta os detalhes de implementação do *YapDss*. Os detalhes de implementação abrangem o protocolo criado, passando pelas estratégias usadas na partilha de trabalho.

Capítulo 7: Análise de Resultados Neste capítulo é verificada, a eficiência do modelo distribuído usado. Começa por apresentar o custo associado à passagem do modelo básico *Yap* para o *YapDss*, seguindo-se a comparação com o modelo paralelo *YapOr*. Por último, é comparada a performance do YapDss, mediante o número de agentes usados.

Capítulo 8: Conclusões e Trabalho Futuro Discute a pesquisa, sumaria as contribuições e sugere direcções para trabalho futuro.

O capítulo 4 é de grande importância porque introduz os conceitos e ideias por detrás das estratégias usadas, dando um contexto para um melhor entendimento da implementação descrita no capítulo 5. De referir, que no capítulo 5, certas partes de código apresentadas são desprovidas de optimizações, e certos pormenores de implementação são omitidos para uma melhor compreensão por parte do leitor.

Capítulo 2

Programação Lógica, Paralelismo e Stack Splitting

Este capítulo tem como objectivo providenciar uma descrição das áreas de investigação abrangidas por esta tese, ilustrando as ideias chaves de cada uma delas. As áreas referenciadas são: Programação Lógica; Paralelismo-Ou; e Stack Splitting.

2.1 Programação Lógica

As linguagens de programação lógica conjuntamente com as linguagens de programação funcional formam uma importante classe de linguagens, as *linguagens declarativas*. Uma característica comum a ambos os grupos é que têm uma forte base matemática. As linguagens de programação lógica são baseadas no cálculo de predicados, enquanto que as linguagens funcionais são baseadas no cálculo lambda.

As linguagens declarativas são consideradas linguagens de muito alto nível quando comparadas com as linguagens imperativas, isto porque, nas linguagens declarativas o programador pode-se concentrar mais na verdadeira essência do problema, deixando muitos dos detalhes de como resolver o problema para o computador. A forte base matemática simplifica a tarefa da programação. Um programador pode especificar os problemas a um nível mais aplicacional, tornando o raciocínio mais simples.

É consensual na literatura referir os seguintes pontos fortes das linguagens de programação lógica [Car90]:

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

Semântica procedimental simples Um programa lógico é uma colecção de procedimentos recursivos. As cláusulas são tentadas pela ordem que são escritas e os objectivos 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 popular em grande parte devido às suas implementações eficientes. O Prolog é um simples provador de teoremas baseado em lógica de primeira ordem, que dado um programa (chamado de teoria) e uma questão, tenta satisfazer a questão usando somente o programa. Em caso de sucesso, as instanciações das variáveis da questão são apresentadas como resultado final.

O esquema de execução habitual do Prolog pressupõe-se o seguinte [Kar92]:

- As variáveis são variáveis lógicas que só podem ser instanciadas uma única vez.
- As variáveis não tem 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 outra forma de satisfazer a questão. No Prolog, o retrocesso implica a investigação de diferentes alternativas (cláusulas) pela ordem 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, surgindo de um estudo da linguagem natural. Em 1977, David Warren tornou viável o Prolog como linguagem através do desenvolvimento do primeiro compilador [War77]. Os modelos actuais de Prolog tornaram-se mais eficientes com o desenvolvimento e progresso do modelo sequencial de execução apresentado em 1983 por David Warren, conhecido por WAM (*Warren Abstract Machine*) [War83].

2.1.1 Warren Abstract Machine

A WAM é uma máquina abstracta que inclui uma arquitectura de memória e um conjunto de instruções específicas para o Prolog [AK91]. Pode ser implementada eficientemente num vasto conjunto de *hardware*, e serve como alvo para os compiladores portáveis de Prolog. A WAM é aceite como o standard base para a implementação do Prolog. Hoje em dia, muito desenvolvimento em programação lógica é feito usando a tecnologia da *WAM*. A figura 2.1 apresenta a disposição das pilhas da *WAM*.

A WAM tem as seguintes pilhas de execução:

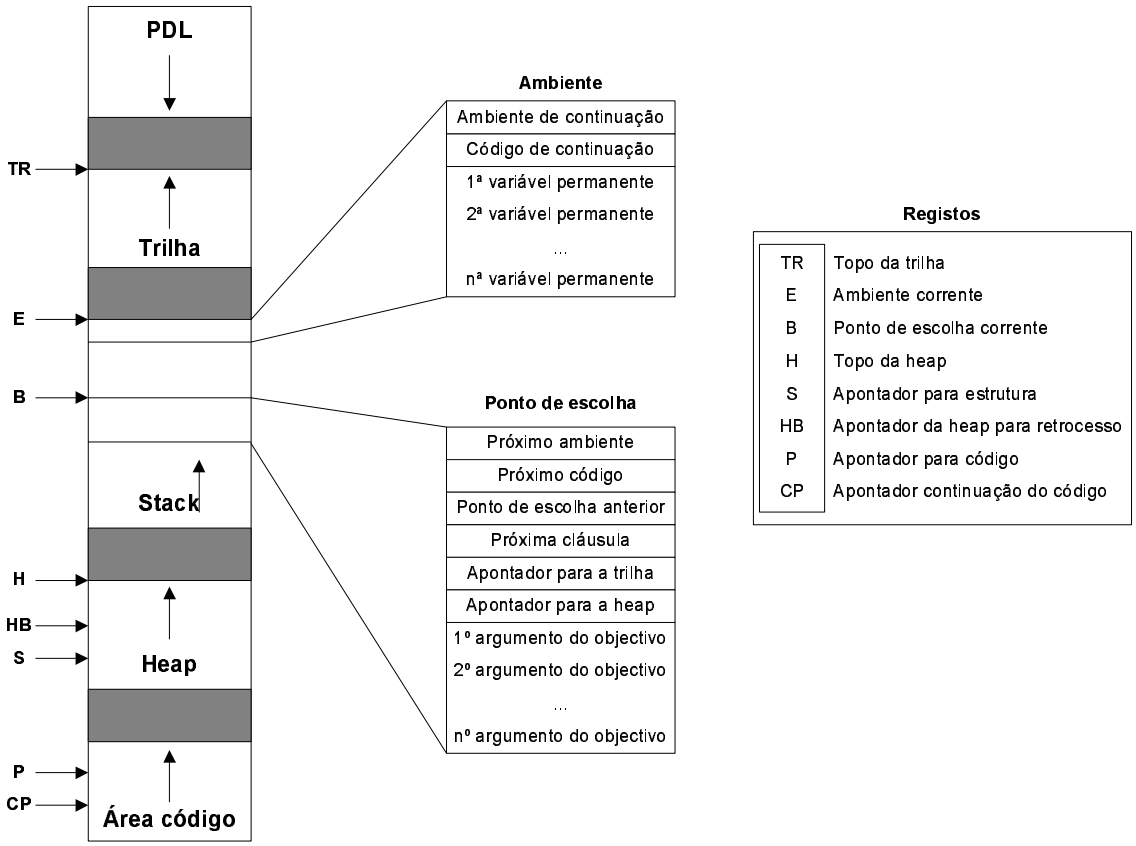

Figura 2.1: *Esquema de memória e registos da WAM.*

Área de código Esta área contém as instruções WAM compiladas dos programas carregados.

***Stack* ou Pilha Local** Esta pilha armazena as *frames* de ambiente e pontos de escolha:

- Os ambientes controlam o fluxo do programa. Um ambiente é inserido na pilha sempre que uma cláusula que contém vários objectivos é executada, e é retirado antes da execução do último objectivo. Um *frame* de ambiente consiste no endereço do ambiente precedente, para actualizar sempre que o ambiente é desalojado, o endereço do código da próxima instrução para executar caso retorne com sucesso da cláusula invocada, e uma sequência de células, tantas quanto o número de variáveis permanentes contidas no corpo da cláusula. O registo E indica o ambiente corrente.
- A função dos pontos de escolha é armazenar as alternativas por explorar de uma dada cláusula. O ponto de escolha contém informação para restaurar o estado computacional antes da execução da cláusula, e informação para a próxima alternativa a executar, caso a corrente falhe. Uma *frame* de ponto de escolha é inserida na pilha sempre que é chamada para execução uma cláusula que tem várias alternativas de execução. É removida da pilha

após a execução da última alternativa de um ponto de escolha. O registo B contém o endereço do ponto de escolha corrente.

***Heap* ou Pilha Global** É um vector de células que armazena variáveis e termos compostos que não podem ser armazenados na pilha local. O registo H aponta para o topo desta pilha.

Trilha Organizada como um vector de endereços, armazena os endereços das variáveis (de pilha local ou pilha global) que podem ser revalidadas no processo de retrocesso.

PDL O PDL é uma pilha auxiliar usada no processo de unificação.

Para além dos registos já mencionados, a *WAM* utiliza outros registos: o registo **S** é usado no processo de unificação de termos compostos; o registo **HB** é usado na determinação da atribuição condicional; o registo **P** aponta para a instrução *WAM* que está a ser executada; e o registo **CP** aponta para onde retornar após uma execução com sucesso da invocação corrente.

Na *WAM* existem quatro grupos principais de instruções:

Instruções de pontos de escolha Estas instruções tem por finalidade a manipulação dos pontos de escolha. Elas podem criar e remover pontos de escolhas, bem como recuperar o estado computacional usando a informação contida num ponto de escolha.

Instruções de controle Instruções que tem a responsabilidade da criação e remoção de ambientes e da gestão das sequências de *call* e *return* dos sub-objectivos.

Instruções de unificação Este tipo de instruções tem implementações especializadas para o processo de unificação, mediante a posição e tipo dos argumentos. Existem instruções próprias para a unificação da cabeça, unificação dos sub-argumentos e a preparação dos argumentos para os sub-objectivos. Estas três classes de instruções estão divididas em versões especializadas para tratar de primeiras ocorrências, ocorrências repetidas, constantes na cláusula, listas e outros termos compostos.

Instruções de indexação Estas instruções tem como finalidade a aceleração do processo de escolha da cláusula que unifica com uma determinada chamada de um sub-objectivo. Dependendo do primeiro argumento da *chamada*[1], elas saltam para o código especializado que tratam directamente da indexação das cláusulas a unificar.

A aparente simplicidade da WAM esconde muitos pormenores delicados da implementação. Uma introdução didáctica da WAM é feita por Ait-Kaci no seu livro sobre a WAM [AK91].

[1] A palavra 'chamada' traduz a palavra ' call', habitualmente usada neste contexto na literatura inglesa.

2.2 Paralelismo

As implementações tradicionais do Prolog foram desenhadas para computadores sequenciais. A natureza eficiente das implementações Prolog que tem como base a WAM, motivou o interesse da passagem destas implementações para um modelo paralelo. Nestes modelos, vários processadores trabalham em conjunto de modo a diminuir o tempo de execução de um programa.

Nas arquitecturas paralelas existem dois problemas fundamentais:

Reconhecimento do paralelismo num dado programa. Este problema pode ser resolvido através de compiladores com capacidade de detecção eficiente das partes paralelas do programa (**Paralelismo implícito**), ou por intervenção directa do programador através do uso de instruções explícitas de modo a explorar o paralelismo (**Paralelismo explícito**).

Distribuição eficiente do trabalho pelos nós do sistema. Este problema é resolvido através do desenvolvimento de algoritmos eficientes de distribuição de trabalho para os vários nós do sistema.

As principais formas de paralelismo existentes no Prolog são [GACH97]:

Paralelismo-Ou Quando um predicado é definido por várias cláusulas e uma chamada a esse predicado unifica com a cabeça de várias dessas cláusulas, dá lugar ao Paralelismo-Ou. O Paralelismo-Ou é um modo eficiente de procurar soluções para um questão, através da exploração de várias alternativas em paralelo.

Paralelismo-E Independente Aparece quando existe mais do que um objectivo numa questão ou no corpo de uma cláusula, e as instanciações das variáveis desses objectivos, 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 ao Paralelismo-E independente.

Paralelismo-E Dependente Aparece quando dois ou mais objectivos no corpo de uma cláusula tem variáveis comuns e são executados em paralelo. Existem duas formas de lidar com este tipo de paralelismo:

- Os dois objectivos podem ser executados independentemente, até que um deles instancie uma variável comum. Podem executar independentemente até terminarem, para aí compararem a compatibilidade das atribuições das variáveis comuns (*back unification).* Esta versão é idêntica ao Paralelismo-E independente.
- 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 continua a ser explorado.

Paralelismo na Unificação Este tipo de paralelismo tem lugar quando os argumentos de um objectivo unificam com a cabeça de uma cláusula com o mesmo nome e a mesma aridade. Os vários termos dos argumentos podem ser unificados em paralelo, bem como os diferentes subtermos num termo. Este tipo de paralelismo é de baixa granularidade e é melhor explorado usando-se máquinas com processadores especializados com múltiplas unidades de unificação [SP89]. O paralelismo na unificação não tem sido alvo de muita investigação.

2.2.1 Paralelismo-Ou

A escolha da exploração do Paralelismo-Ou deve-se ao facto de este, numa primeira fase, ser mais eficiente de explorar do que o Paralelismo-E. As principais vantagens da utilização do Paralelismo-Ou são as seguintes [LBD+88]:

Flexibilidade É relativamente simples explorar este tipo de paralelismo sem restringir o poder da programação lógica. Uma vantagem inerente a este paralelismo é a obtenção de todas as soluções de uma questão.

Simplicidade O Paralelismo-Ou não necessita da introdução de novas operações a nível de programação e não precisa de análise em tempo de compilação.

Proximidade do Prolog É possível explorar o Paralelismo-Ou num modelo de execução bastante próximo do modelo sequencial do Prolog. Isto torna mais fácil a manutenção da semântica da linguagem, e podemos tirar tudo o partido da implementação sequencial existente.

Granularidade O Paralelismo-Ou tem potencial para uma grande classe de programas, de paralelismo e granularidade alta. A granularidade esta relacionada com a quantidade de trabalho executado sem a interacção de outros pedaços de trabalho executados em paralelo.

Aplicabilidade A aplicabilidade deste tipo de paralelismo manifesta-se fundamentalmente em programas de procura como é a prova de teoremas, a análise gramatical de linguagem natural, *Data Mining* e o emprego de regras em sistemas periciais.

Dos modelo existentes de exploração do paralelismo-Ou existem dois que são tidos como referência, dado servirem de suporte aos dois sistemas mais robustos e amadurecidos de paralelismo-Ou, os sistemas Aurora [LBD+88, Car90] e Muse [Kar92]. Estes modelos são:

Binding Arrays No modelo de *binding arrays* cada agente possui uma estrutura de dados auxiliar designada por *binding array* que contém as variáveis lógicas condicionais. As variáveis condicionais são todas as variáveis que estão desreferenciadas após a criação de um ponto de escolha. Sempre que uma variável

condicional é instanciada, o seu valor é guardado no *binding array*. Todos os agentes possuem uma entrada no *binding array* para a mesma variável. Neste modelo, a operação de partilha de trabalho consiste na actualização do *binding array*. Contudo, este modelo adiciona alguma latência no computação porque todas as variáveis condicionais estão guardadas no *binding array*, e todos os acessos a uma variável condicional implicam um redirecionamento, acrescido do facto de durante a desreferenciação, o endereço e o valor das variáveis terem de ser guardados na trilha.

Cópia de Ambientes No modelo de cópia de ambientes, cada agente possui uma cópia separada do seu ambiente, no qual pode escrever sem causar conflitos de atribuições. Neste modelo nem as atribuições não condicionais são partilhadas. Quando um agente suspenso toma uma alternativa não explorada de um ponto de escolha criado por outro agente, copia todas as pilhas de execução deste. De modo a tornar este processo eficiente utiliza-se o processo de *cópia incremental*. A ideia por detrás deste processo está relacionado com o facto de se copiar somente as diferenças entre as pilhas dos agentes, evitando deste modo a cópia total destas, isto porque o agente suspenso pode já ter percorrido parte do caminho entre a raiz e o ponto de escolha mais profundo comum a ambos os agentes. Procedendo-se à cópia das pilhas a partir desse ponto de escolha comum, é igualmente garantido o mesmo estado computacional. Após o processo de cópia, cada agente pode comportar-se como um sistema de execução sequencial, requerendo somente ligeiras sincronizações com os outros agentes.

Um sistema de execução paralela que suporte todos os predicados extra-lógicos, meta-lógicos, efeitos colaterais e que produza o mesmo efeito que a execução sequencial, da esquerda para a direita, e de cima para baixo na selecção das cláusulas, é designado por sistema que suporta a *semântica sequencial do Prolog*.

O sistema Aurora utiliza o modelo de *Binding Arrays* e tem por base o sistema sequencial de Prolog SICStus Prolog 0.6 [MC89]. Dá suporte à semântica sequencial do Prolog e o seu maior custo está na mudança de contexto para a execução de uma nova alternativa. Os três distribuidores de trabalho mais usados no Aurora são:

Manchester Scheduler Distribuí o trabalho logo que possível com uma estratégia de distribuição o mais perto da raiz da árvore.

The Bristol Scheduler Este distribuidor tenta maximizar a zona partilhada de modo a minimizar os custos da partilha. Para maximizar a partilha são partilhadas sequências de nós em lugar de nós singulares e o trabalho é tomado no nó vivo mais profundo de ramificação.

The Dharma Scheduler Foi desenhado com o principal objectivo de tratar eficientemente o trabalho especulativo. A solução encontrada centra-se na descoberta do nó o mais esquerda, logo o menos especulativo pedaço de trabalho disponível, através da ligação directa entre extremidades de cada ramificação.

O sistema **Muse** utiliza o modelo de execução de cópia de ambientes e foi baseado no sistema sequencial de Prolog SICStus Prolog 0.6 , tal como o Aurora. Dá suporte à semântica sequencial do Prolog e como no sistema Aurora o maior custo reside na mudança de contexto para a execução de uma nova alternativa. O Muse é composto por agentes, que nada mais são do que WAM's estendidas. Cada um deles possui um espaço local de endereçamento, e algum espaço global que é partilhado por todos agentes. A cópia incremental é utilizada de um modo eficiente graças à organização das pilhas da WAM. Suporta com eficiência o trabalho especulativo. O distribuidor tem duas estratégias: uma encarregue de tratar 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 que tenha mais alternativas por explorar). Controla a granularidade de cada tarefa, em tempo de execução, com o objectivo de evitar a partilha de pequenos pedaços de trabalho.

No entanto, existem várias dificuldades na implementação do paralelismo-Ou num ambiente distribuído, que estão relacionadas com a inexistência de uma árvore global de computação, de onde todos os agentes seriam controlados, porque não existe um espaço global que seja partilhado por todos os agentes. Como só existe acesso às sub-árvores dos agentes (espaço local de endereçamento), foi preciso criar um mecanismo que evitasse a criação de uma representação global da árvore de execução, que implicaria um elevado custo de manutenção, que tornaria o sistema inviável. Este mecanismo, denominado como *Stack Splitting*, consiste numa distribuição estática do trabalho e tem como base a cópia incremental de ambiente.

2.3 Stack Splitting

Uma propriedade importante das linguagens de programação lógica é que estas são linguagens de atribuição única. Ao contrário das linguagens de programação convencionais, elas não permitem atribuições destrutivas e controlo explícito sobre a informação. Isto não só permite uma semântica mais simples para os programas, facilitando a sua compreensão por parte dos utilizadores, como permite a utilização de diferentes estratégias pelo avaliador lógico. Dada esta característica é possível executar diferentes operações em qualquer ordem, sem afectar o resultado final do programa, sendo deste modo possível executar estas operações em paralelo. Outra característica importante das linguagens de programação lógica é o facto do paralelismo presente no modelo de operação da linguagem poder ser explorado implicitamente. Isto pode ser feito por um avaliador em tempo de execução ou através de um compilador para o efeito. Deste modo o utilizador não precisa de preocupar em adaptar o seu código para explorar o paralelismo.

As técnicas existentes para máquinas *SMM* (*Shared Memory Multiprocessors*) são inadequadas para o ambiente distribuído, mais especificamente para as máquinas *DMM* (*Distributed Memory Multiprocessors*), porque necessitam de partilhar as pilhas de dados e de controlo. Mesmo que a necessidade para partilhar as pilhas de dados seja

retirada, continua a ser necessário a partilha das pilhas de controlo. Uma das propostas para suportar Paralelismo-Ou em máquinas *DMM* é o *Distributed Stack Splitting* (*DSS*), que é a implementação do *Stack Splitting* (*SS*) num ambiente distribuído.

Com o *DSS* não existe mais a necessidade de partilhar as pilhas de controlo, isto porque o *DSS* faz uma distribuição estática de trabalho, onde as alternativas inexploradas são dividas na partilha de trabalho. As vantagens inerentes ao *DSS* são: redução das comunicações durante a execução distribuída; maior granularidade na distribuição de trabalho para os processadores.

A nível de performance , caso os pontos de escolha estejam bem balanceados[2] é de esperar uma bom desempenho do *DSS*.

Quando os pontos de escolha apresentam poucas alternativas, pode resultar numa degradação do desempenho, por exemplo, quando o Paralelismo-Ou é usado em predicados com duas alternativas, e em que uma das alternativas não origina mais trabalho. Estes predicados ocorrem frequentemente, já que muitos programas usam predicados do tipo *member* e *select*:

```
member(X,[X|_]).                  select(X,[X|Y],Z]).
member(X,[_|Y]):-member(X,Y).     select(X,[Y|Z],[Y,R]):-select(X,Z,R).
```

Como os predicados **member** e **select** tem somente duas alternativas por ponto de escolha, com a primeira alternativa não originando mais trabalho, pode resultar na ausência de trabalho para um dos agentes. Daqui podemos ver que a estratégia de distribuição de trabalho usada assume grande importância na performance do sistema.

O *Stack Splitting* tem dois tipos principais de distribuição de trabalho: O *Vertical Splitting*; e o *Horizontal Splitting*. Em seguida, são descritas as duas aproximações distintas de *DSS*.

2.3.1 *Vertical Splitting*

Esta estratégia tem como base a divisão dos pontos de escolhas entre os agentes envolvidos na partilha de trabalho, tem como principal vantagem um melhor balanceamento em pontos de escolhas com poucas alternativas, evitando que algum dos agentes possa ficar sem trabalho no processo de partilha.

Na figura 2.2, podemos verificar como o processo decorre. O agente activo P partilha os seus pontos de escolha com o agente Q, que se encontrava suspenso à procura de trabalho (ver sub-figura 1). Após a partilha, o agente P fica com os pontos de escolha B e D (ver sub-figura 2); enquanto que o agente Q fica com os pontos de escolha A e C (ver sub-figura 3).

[2]Por bem balanceados entenda-se uma distribuição uniforme das alternativas existentes pelos vários pontos de escolha da sub-árvore de computação.

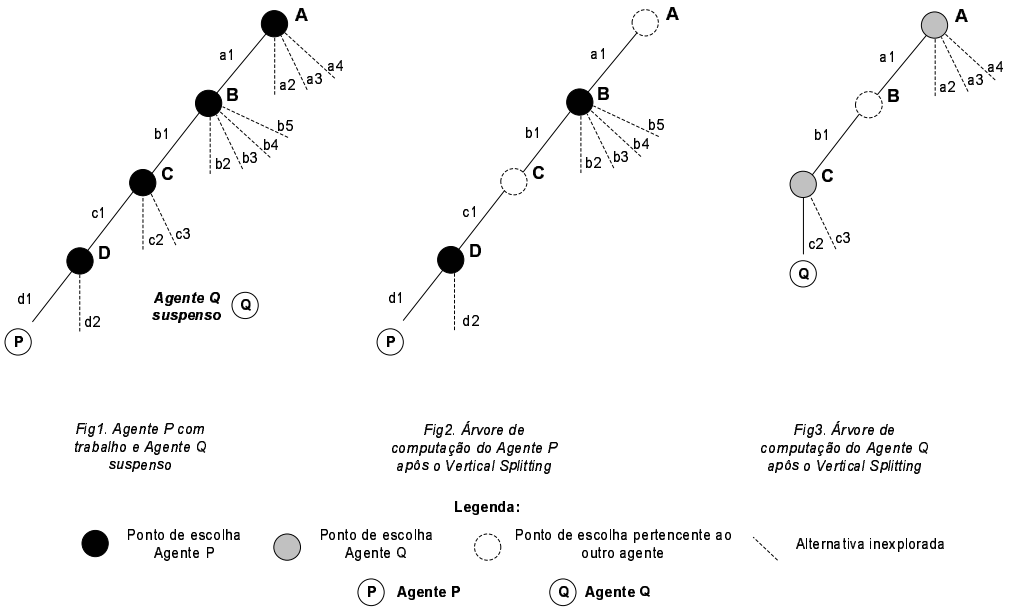

Figura 2.2: *Vertical Splitting*

2.3.2 *Horizontal Splitting*

Em alternativa à estratégia anterior, o *horizontal splitting* divide as alternativas dos pontos de escolhas pelos agentes, resultando num bom balanceamento caso os pontos de escolha tenham muitas alternativas.

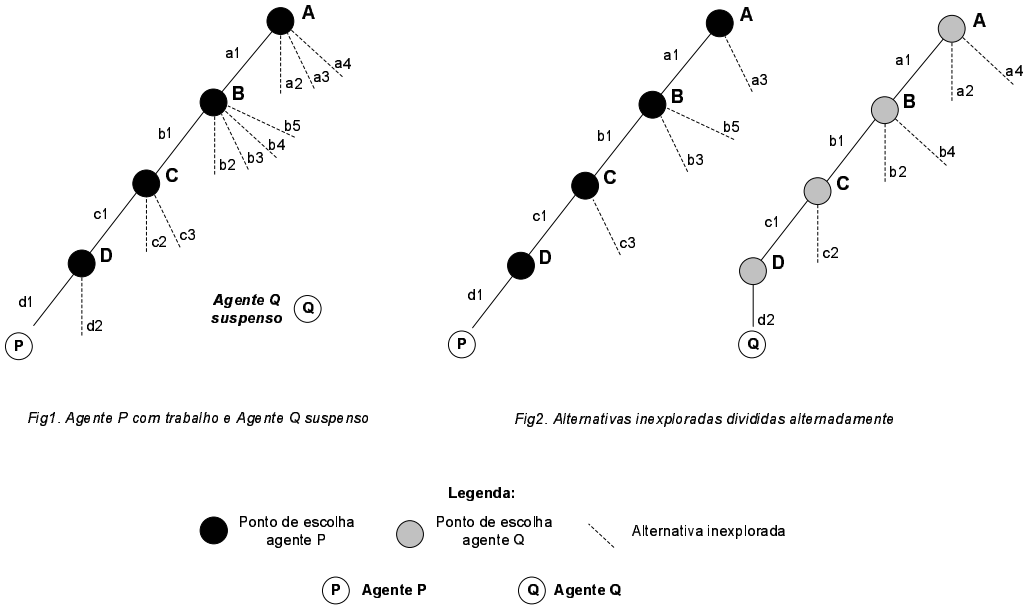

Figura 2.3: *Horizontal Splitting*

Na figura 2.3, temos as várias fases do processo de partilha através do *Horizontal*

Splitting. Na sub-figura 1, estão presentes o agente activo P e o agente passivo Q. Após a partilha o agente P e Q ficam com as alternativas alternadas (ver sub-figura 2).

2.3.3 DSS no YapDss

No *YapDss* foi utilizada uma mistura das duas aproximações de modo a retirar as vantagens de cada uma, suprimindo as desvantagens através da complementaridade de ambas.

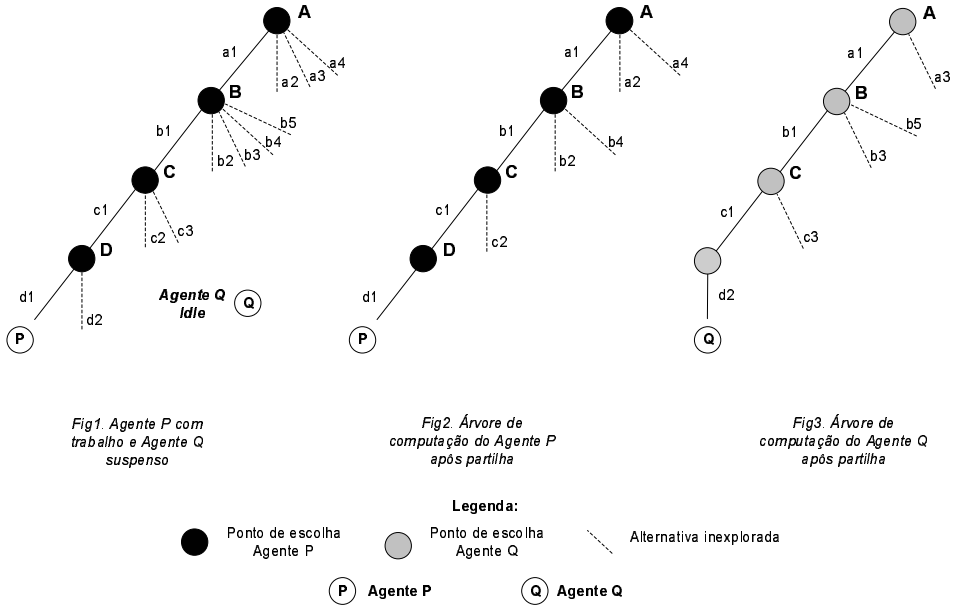

Figura 2.4: Distribuição feita pelo YapDss.

Na figura 2.4 podemos ver a distribuição dos pontos de escolha usando uma aproximação combinada. Esta nova aproximação elimina o problema da possível ausência de trabalho por parte do *Horizontal Splitting*, com a falta de precisão no balanceamento por parte do *Vertical Splitting*.

O maior problema associado ao DSS em ambientes distribuídos, está relacionado com a manutenção da informação de *scheduling* e de gestão de carga. A solução usada foi a passagem de mensagens entre os agentes.

2.4 Resumo do Capítulo

De modo a ser possível a compreensão dos conceitos relacionados com o método de partilha, o Distributed Stack Splitting, foi referenciada a máquina abstracta de

Warren, que é o alicerce na implementação do Prolog, bem como os diversos tipos de paralelismo existentes na execução do Prolog, de modo a contextualizar o processo de partilha.

No capitulo seguinte, fazemos referencia aos processos usados no *Yapor*, servindo como base para uma posterior ilustração dos processos envolvidos nesta tese.

Capítulo 3

Sistema YapOr

Este capítulo descreve o sistema *Yapor* [Roc96]. O *Yapor* é um sistema paralelo de execução do Prolog que explora o paralelismo-Ou implícito a partir da plataforma *Yap* de execução sequencial. O Yapor utiliza o modelo de cópia de ambientes, baseando-se em muitos dos conceitos introduzidos pelo sistema *Muse* [AK90a], destinado a obter altos desempenhos na execução paralela de Prolog de forma a acelerar a execução de programas sobre o sistema.

3.1 Conceitos

O modelo de execução do sistema *YapOr* está desenhado para tirar partido das arquitecturas *SMM*. Neste modelo um conjunto de unidades básicas, os agentes, cooperam para acelerar a execução de programas de Prolog. Os *agentes*, são representados a nível do sistema por um processador ou por um simples processo, tendo as seguintes características comuns:

Endereçamento Lógico Cada *agente* tem um espaço idêntico de endereçamento e partilha um certo espaço global de endereçamento.

Execução Sequencial Os *agentes* são como que entidades de execução sequencial de Prolog, o que implica que tenham as pilhas de execução da *WAM*:*local stack, heap stack,*e *trail stack*, ou pilha local, pilha de termos e trilha, respectivamente. Estas pilhas estão contidas no espaço de endereçamento local, enquanto o código da *WAM* é guardado no espaço global de endereçamento, de modo a ser acedido por todos os *agentes*.

Em Prolog, os *pontos de escolha* podem ser vistos como nós de uma árvore de computação, onde cada ramo representa uma alternativa de um dado *predicado*. O sistema *YapOr* distingue 2 tipos de nós, os privados e os partilhados:

Nós Privados São nós somente acessíveis ao *agente* que os criou.

Nós Partilhados São nós acessíveis por mais do que um *agente*, ou seja, existe mais do que um *agente* que tem esse nó na sua árvore de computação.

Sendo que cada nó está num dos referidos estados, resulta na divisão da árvore de computação em duas regiões distintas, a *região privada* e a *região partilhada*. Outro conceito a reter, é sobre o estado de computação de um dado nó.

Nó Morto Nó que não tem mais nenhuma alternativa para executar.

Nó Vivo Nó que possui alternativas por explorar.

3.2 Modelo de Execução Básico

No *Yapor* os procedimentos básicos são os seguintes:

1. Antes da execução de uma *query*, todos os *agentes* encontram-se *suspensos*. A *query* começa por ser executada somente por um *agente* **P**. Este *agente* irá criar *pontos de escolha* para computar predicados com mais do que uma alternativa.

2. Somente após tal suceder, é que outros *agentes* intervêm no processo ao enviarem pedidos de trabalho ao *agente P*, com o intuito de cooperarem na computação das alternativas pendentes nos nós criados por P.

3. *P* partilha os seus nós privados com um agente *Q* do seguinte modo:

   - Para cada nó privado, o *agente P*, cria uma *estrutura partilhada*, que reside num espaço de endereçamento global, a qual contém informação sobre o número de alternativas inexploradas e dos *agentes* envolvidos na partilha, colocando um apontador no nó privado referenciando a estrutura criada (ver figura 3.1).
   - *P* copia as suas pilhas (local, termos e trilha) para *Q*. Neste momento *P* e *Q* estão no mesmo estado computacional.

4. *P* prossegue a computação no local onde tinha sido interrompido pelo pedido de partilha, enquanto o *agente Q* simula uma falha (faz *retrocesso* sobre o último *ponto de escolha*), toma a próxima alternativa a partir da estrutura partilhada, e não a partir do *ponto de escolha* como em execução sequencial. O acesso à estrutura partilhada é sincronizado por um mecanismo de fecho que evita que 2 agentes tomem uma mesma alternativa.

5. Qualquer que seja o *agente X* suspenso, este procura computação num *agente Z* que tenha alternativas por executar, tal como o *agente Q* fez com o *agente P*.

6. Sempre que um *agente* não tenha alternativas por explorar, retorna ao estado de suspenso, procurando trabalho em *agentes ocupados*.

7. Quando todas as alternativas tiverem sido exploradas, a execução termina e todos os *agentes* regressam ao estado de suspensos.

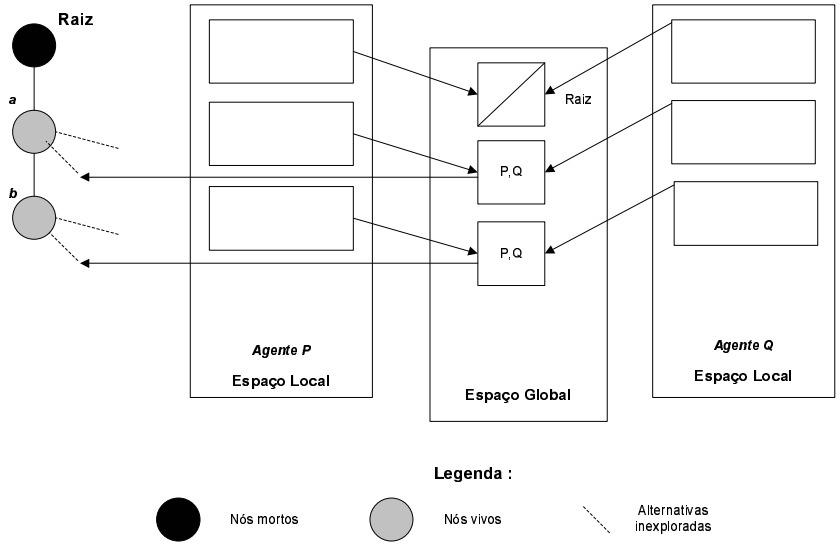

Figura 3.1: Relação entre pontos de escolha e estruturas partilhadas.

3.3 Cópia Incremental

Na operação de partilha de trabalho, a cópia do estado computacional, que basicamente é a cópia da pilha local, termos e trilha, tem um peso substancial na performance do sistema. Logo é importante a criação de mecanismos que possam reduzir o peso dessa operação. Um desses mecanismo é a **cópia incremental do ambiente**.

O mecanismo de cópia incremental consiste na cópia da diferença entre os estados dos *agentes*, sendo que um estado computacional idêntico corresponde aos *agentes* estarem no mesmo nó da árvore de computação.

Em Prolog, a criação de um *ponto de escolha* tem como função, guardar o estado corrente da computação. Para além disso, para todas as variáveis criadas antes do ponto de escolha e que venham a ser instanciadas, guarda-se uma referência na trilha. No processo de retrocesso, para além da recuperação do estado computacional, também é preciso desreferenciar as atribuições feitas às variáveis cujas as referências estão guardadas na trilha.

A cópia incremental no *YapOr* tenta optimizar ao máximo o processo de cópia. Quando um agente Q não tem mais trabalho na sua sub-árvore de computação, ou seja, não tem mais nós vivos do ponto de escolha corrente até à raiz, e existe algum agente P

que possui nós vivos (ver figura 3.2), então Q solicita trabalho a P. Antes disso, Q deve retroceder na sua árvore de computação até ficar num nó comum com P. Deste modo o estado computacional de Q é consistente com P. O agente P começa por criar as estruturas partilhadas de modo a partilhar os seus nós privados. Para o agente Q só é copiado as partes da pilha local, pilha de termos e trilha que reflectiam a diferença de estado entre ambos. Para isto é usada a informação contida no nó comum encontrado por Q e nos segmentos de topo das pilhas de P. Caso existam variáveis comuns cujas as referências estão guardadas na trilha de P, as atribuições das mesmas devem de ser actualizadas em Q, caso contrário o estado computacional é diferente (ver figura 3.2).

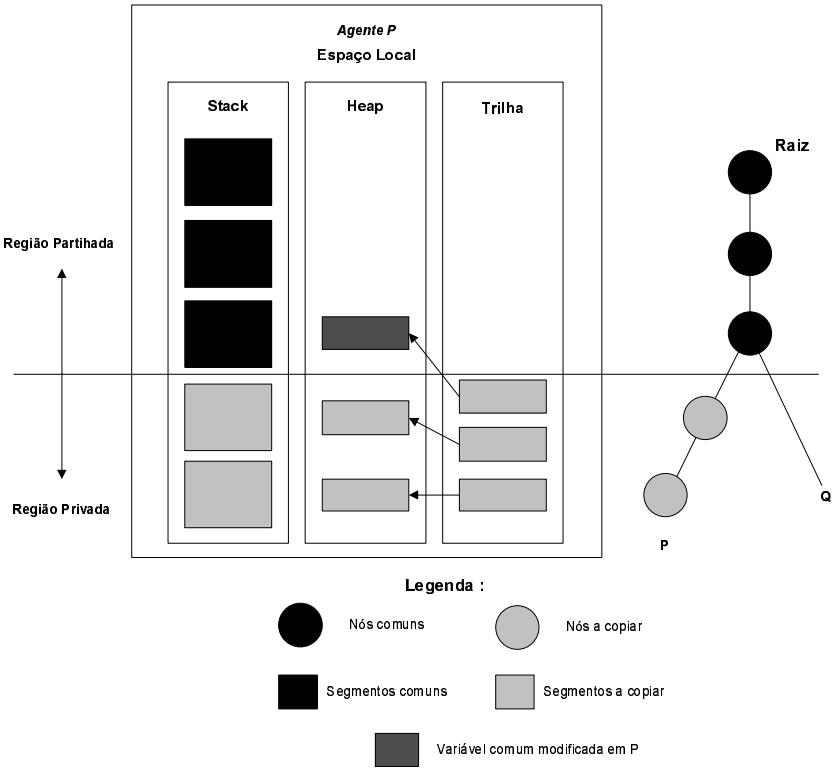

Figura 3.2: Aspectos relevantes da cópia incremental.

3.4 Distribuição de Trabalho

No *Yapor*, a execução pode ser dividida em:

Modo Procura Um *agente* está neste modo quando procura uma alternativa na região partilhada.

Modo Execução Um *agente* entra neste modo quando deixa o modo procura. Neste modo um *agente* comporta-se como se tratasse de um sistema de Prolog sequencial, mas com a capacidade de partilha de trabalho.

As principais funções do distribuidor de trabalho são: manter a correcta semântica do Prolog, e dentro do possível atribuir novas tarefas aos agentes suspensos, tentando minimizar os factores principais que contribuem para a perda de desempenho do sistema. Os factores relacionados com a perda de desempenho são: as cópias entre agentes; tornar nós privados em partilhados; e a selecção de uma nova tarefa na região partilhada.

As linhas estratégicas do distribuidor do *YapOr* para a minimização destes factores são:

- Na partilha de trabalho, o agente com trabalho deve partilhar a totalidade dos seus nós privados. Isto maximiza o trabalho partilhado contribuindo para um melhor balanceamento da carga.
- O distribuidor deve seleccionar o agente com maior carga de trabalho que se encontre mais próximo do agente suspenso, de modo a maximizar a quantidade de trabalho partilhado e a minimizar a quantidade de segmentos a copiar das várias pilhas.
- Quando o distribuidor de trabalho não encontrar trabalho para atribuir a um agente suspenso, deve posicioná-lo da melhor maneira, numa tentativa de minimizar os custos com uma futura partilha de trabalho.

A ideia geral do distribuidor pode resumir-se ao seguinte: sempre que um agente fica sem trabalho, tenta seleccionar um agente ocupado, com excesso de carga, para partilhar trabalho. Se não existe tal agente, coloca-se na melhor posição possível na árvore de procura de modo que possa dispor de trabalho logo que seja possível.

3.5 Resumo do Capítulo

Neste capitulo apresentámos sumariamente o sistema *Yapor*, fazendo particular referencia ao modelo básico de execução, à cópia incremental e à distribuição de trabalho.

As referências deste capítulo servem como base de compreensão aos processos do modelo distribuído, que sendo eminentemente diferentes tem pontos de convergência, dado a sua base no Paralelismo-Ou.

Capítulo 4

Sistema YapDss

Neste capítulo descrevem-se os conceitos e os aspectos mais importantes do modelo de execução do YapDss: o modelo básico de execução; as estratégias que o distribuidor de trabalho utiliza para melhor gerir a carga existente no sistema pelos vários agentes; o processo de cópia incremental; o processo da terminação de computação.

4.1 Modelo Básico de Execução

A figura 4.1 ilustra o modelo básico de execução, dando uma visualização do processo, desde o início até à terminação da computação.

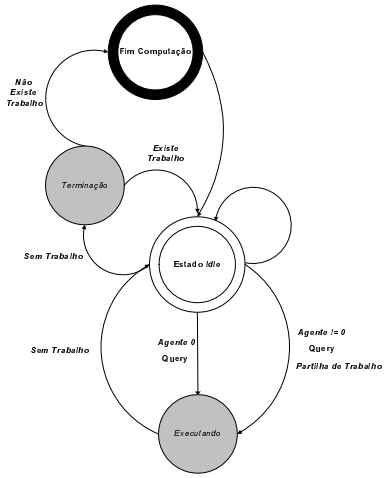

Figura 4.1: Máquina de estados do YapDss.

As acções básicas do modelo de execução do *YapDss*, resumem-se ao seguinte:

1. Anterior à execução de um programa em Prolog, todos os agentes encontram-

se *suspensos*. O agente *principal* começa por enviar, para todos os agentes, o código do programa a executar. No início da execução, somente esse agente, denominado por P, começa por explorar as alternativas disponíveis. Entretanto todos os outros agentes continuam *suspensos* até que o agente P crie na sua pilha local, um ponto de escolha correspondente a um predicado com mais do que uma alternativa de execução.

2. Após a criação do referido ponto de escolha, um dos outros agentes, denominado por Q, solicita trabalho ao agente P, de modo a cooperar na computação.

3. Em seguida, P permite que Q coopere na computação, partilhando com ele trabalho, segundo os seguintes critérios:

   - P calcula a quantidade de trabalho a partilhar, de modo a garantir um bom balanceamento de carga.
   - P copia todo o seu estado (pilha local,pilha de termos e trilha) para Q, de modo que ambos fiquem no mesmo estado computacional.

4. Após a partilha de trabalho, ambos os agentes percorrem as suas sub-árvores de computação e efectuam o Stack Splitting.

5. P prossegue com a computação, no ponto onde foi interrompido, enquanto Q simula uma falha, para deste modo activar o mecanismo de *retrocesso*. Assim Q toma a alternativa que lhe corresponde, prosseguindo como um sistema de computação sequencial se tratasse.

6. Sempre que um agente X não tenha mais nenhuma alternativa para explorar na sua árvore de computação, volta para o estado de suspenso e deve solicitar trabalho a um agente ocupado.

7. Quando não existir mais trabalho, a execução termina e o agente principal recolhe todas as soluções encontradas no sistema. Daqui os agentes voltam ao estado de suspensos.

4.2 Cópia Incremental de Ambiente Com Stack Splitting

O *YapDss* faz uso da cópia incremental de ambiente de forma a minimizar os custos na partilha de trabalho, que no caso do YapDss tem o custo acrescido da comunicação se efectuar via rede.

Para existir partilha de trabalho entre dois agentes é necessário conhecer a posição relativa entre ambos, de modo a sabermos a porção das pilhas a copiar. Um problema que desde logo surge é a necessidade de se ter uma representação eficiente da árvore de computação. Para isto não precisamos de ter acesso à árvore total da computação,

basta termos um mecanismo que nos forneça o primeiro nó comum entre os agentes envolvidos no processo de partilha. No YapDss esse mecanismo foi criado através do uso de etiquetas de marcação.

Com o objectivo de minimizar a quantidade de informação necessária à partilha de trabalho entre 2 agentes a técnica de cópia incremental de ambientes foi conjugada com a técnica de Stack Splitting, originando a cópia incremental de ambientes com Stack Splitting (*Incremental Stack Splitting*). A cópia incremental de ambientes é responsável pela correcta actualização do estado computacional do agente suspenso, enquanto que o Stack Splitting é responsável pela gestão da distribuição da carga entre os 2 agentes envolvidos no processo de partilha de trabalho.

4.2.1 Nó Comum

O cálculo do nó comum, tem um papel fulcral na cópia incremental, porque indica o início da cópia incremental, isto porque do ponto de escolha de topo (na figura 4.2 é nos dado pelo nó **Raiz**) até ao nó comum (representado na figura 4.2 pelo nó **A**) ambos os agentes tem o mesmo estado computacional, só variando a partir do ponto de escolha do nó comum. Para termos ambos os agentes no mesmo estado computacional temos que copiar as pilhas desde o nó comum até ao ponto de escolha corrente do agente que está a partilhar trabalho (neste caso é o nó **E** da figura 4.2).

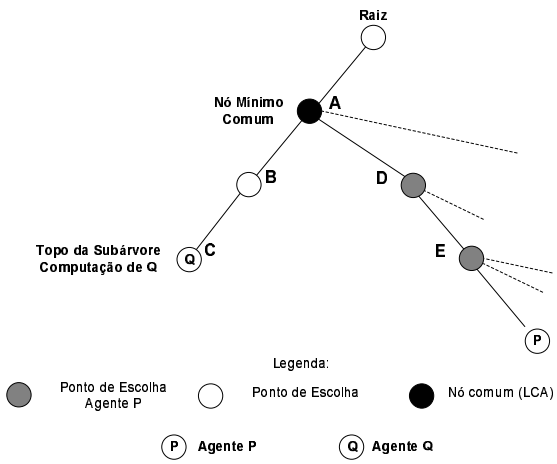

Figura 4.2: *Nó mínimo comum*

Como no YapDss não temos acesso à árvore de procura total, foi criado um sistema de etiquetas que representa a posição absoluta de um agente na árvore de computação, de modo que fosse possível calcular o nó comum.

4.2.1.1 Etiquetas nos Pontos de Escolha

A primeira aproximação a este problema passou pelo uso de etiquetas nos pontos de escolha, que se revelou demasiado lenta devido ao custo associado à criação de cada etiqueta.

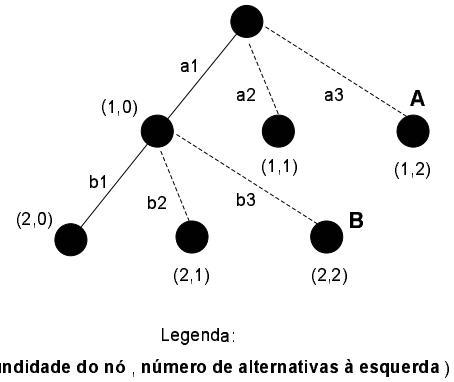

Figura 4.3: Etiquetas contidas nos pontos de escolha

Este método tem como base um número fixo de alternativas por nó. Tendo como n o número fixo de alternativas por nó, *e1* e *e2* as profundidades dos nós, *f1* e *f2* o número de blocos de n alternativas à esquerda dos nós, o cálculo do nó comum é apresentado na figura 4.4.

```
while(e1!=e2 && f1!=f2){
    if(e1>e2){
        e1--;
        f1 = f1 / n;
    }else if(e1<e2){
        e2--;
        f2 = f2 / n;
    }else{ // e1 == e2
        if(f1!=f2){
            e1--;
            e2--;
        }
        break;
    }
}
```

Figura 4.4: Cálculo do nó comum usando etiquetas nos pontos de escolha.

Usando como referências os nós **A** e **B** (ver figura 4.3), que tem respectivamente os tuplos A(e1=1,f1=2) e B(e2=2,f=2) com o número fixo de alternativas igual a 3, podemos facilmente aplicar o algoritmo para determinar o nó comum. Como B está a menor profundidade na primeira iteração fica-se com B(1,0). De seguida, como ambos estão ao mesmo nível, aplica-se a última condição, obtendo-se como nó comum o tuplo (0,0).

Os principais problemas associados a esta aproximação, são:

- Lentidão na criação das etiquetas.
- Lentidão no cálculo do *LCA*.
- A etiqueta ocupa muito espaço em memória, visto estarmos a falar em inteiros com n bytes.

Para suportar esta aproximação foi usada a biblioteca GNU MP (GNU Multi-Precision Arithmetic), que nos permite ter precisão arbitrária de inteiros. Apesar de estar fortemente optimizada, o custo desta aproximação ultrapassa qualquer benefício daí resultante.

4.2.1.2 Etiqueta de Execução

Foi então preciso criar um método que fosse suficiente rápido para que não introduzisse demasiados atrasos na computação. A solução encontrada passou pela criação de uma estado de computação que consiste em guardar a alternativa tomada em cada nível da árvore de computação com o respectivo apontador para o ponto de escolha.

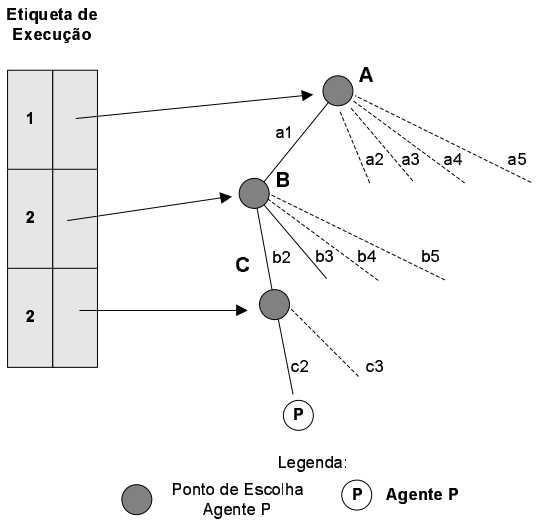

Figura 4.5: Etiqueta de execução

O estado da computação é guardado através do uso de um vector estático(de tamanho pré-definido) para garantir rapidez tanto na sua actualização, bem como para garantir maior rapidez no processo de cópia na criação de uma mensagem de solicitação de trabalho.

O algoritmo para encontrar o nó comum entre 2 agentes neste método é bastante simples consistindo apenas em encontrar a primeira posição onde os estados de computação são diferentes.

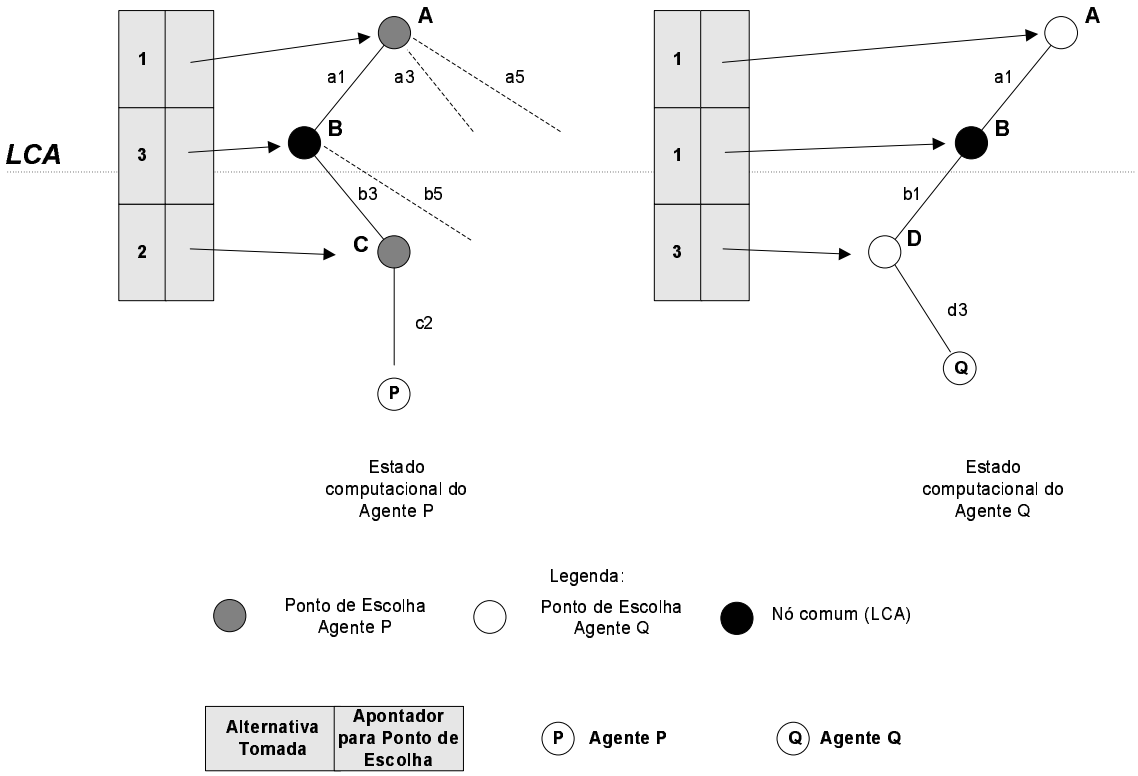

Figura 4.6: Cálculo do nó comum

Na figura 4.6, os 2 agentes P e Q, tem o mesmo estado computacional até o ponto de escolha B, a partir do ponto de escolha B, o agente P explora a alternativa *b3* enquanto que o agente Q explora a alternativa *b1*.

4.3 Distribuidor de Trabalho

Existem dois estados de execução distintos de um agente: o modo de procura e o modo de execução. Um agente encontra-se no modo de procura quando não possuí trabalho e tenta pedir trabalho aos outros agentes do sistema. No modo execução, o agente comporta-se como se de um sistema sequencial de Prolog se tratasse, exceptuando as interacções às solicitações dos outros agentes.

As duas principais funções de um distribuidor de trabalho são: manter a correcta semântica do Prolog, e gerir a distribuição equilibrada de tarefas aos agentes que vão ficando suspensos, de modo a optimizar o desempenho do sistema. Os factores que mais contribuem para a degradação do desempenho do sistema são: o processo de cópia de partes do estado de um agente para outro, manutenção das etiquetas e controlo das solicitações existentes no sistema.

4.3.1 Estratégia no Pedido de Trabalho

Um agente passivo, deve pedir trabalho ao agente activo que tem maior carga, minimizando assim os custos associados a futuras operações. A selecção do agente activo, passa pela previsão de carga, conjuntamente com outro dado importante, que é a informação do último agente com o qual partilhamos trabalho. Se for possível a partilha de trabalho com este agente, conseguimos diminuir o tamanho das pilhas que é preciso copiar, dado que o LCA deverá ser relativamente local.

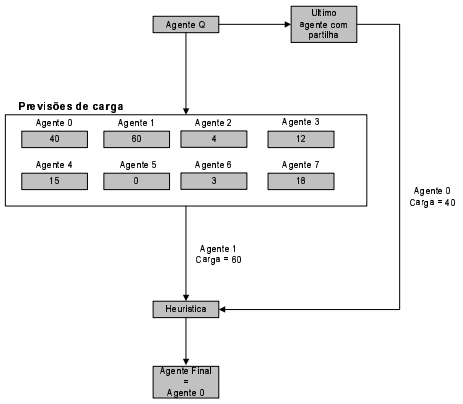

Figura 4.7: Selecção de agente activo

A figura 4.7 mostra um esquema do algoritmo que permite decidir a escolha do agente com maior carga, ao qual iremos pedir trabalho, no âmbito do processo de partilha. O agente suspenso Q, começa por verificar a carga do agente do último processo de partilha (no caso da figura 4.7 é o agente 0). De seguida, verifica nos restantes agentes qual é o que possuí maior carga (o agente 1 é o que apresenta maior carga). Na posse destes dois elementos, o agente Q decide baseado numa heurística qual o agente activo que irá pedir trabalho. Esta heurística consiste no seguinte:

- Caso o registo de carga do último agente com o qual ocorreu partilha de trabalho não seja inferior ao valor definido pela macro $LOAD\_BALANCE$ (por defeito tem o valor de 12), o agente Q escolhe esse agente (no exemplo, o agente 0).
- Caso contrário o agente Q escolhe dos restantes agentes, aquele que possuir maior registo de carga (no exemplo, o agente 1).

4.3.2 Estratégia na Resposta ao Pedido de Trabalho

Um agente só deve partilhar trabalho, caso haja trabalho suficiente para justificar o custo que se incorre no processo de partilha, de outro modo, o custo desta operação ultrapassaria o ganho que dela se obtém.

Quando um agente, está em condições de partilhar, tem de calcular quanto trabalho da sua árvore de computação deve partilhar. Começa a percorrer a árvore no ponto

de escolha corrente até encontrar o topo da sub-árvore a partilhar. Para isto, deve considerar o seguinte:

Quantidade de partilhas ocorridas O agente activo só partilha até encontrar um nó que tenha sido várias vezes (por defeito, o este valor é 3) partilhado. Desta forma preservamos o topo da árvore, evitando a partilha excessiva dos nós de topo, de forma a manter-se sempre algum trabalho disponível.

Carga relativa dos vários nós Para balancear a carga, o agente activo vai fazendo a soma relativa dos vários nós percorridos, que associada à carga total do agente, dá-nos a carga relativa até um dado nó.

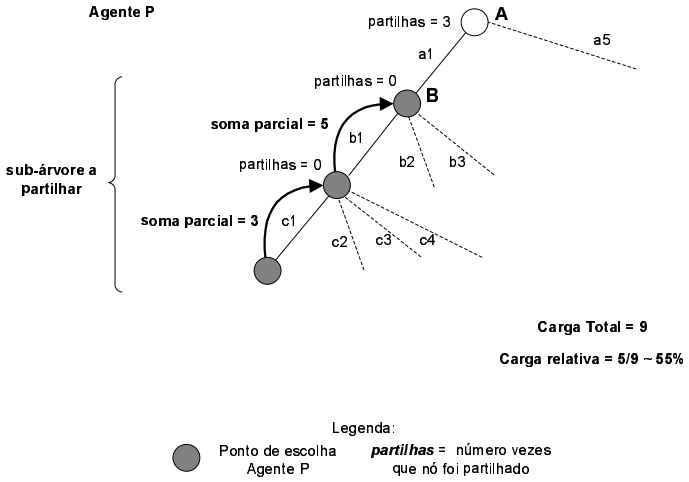

Figura 4.8: Sub-árvore de partilha

Na figura 4.8 podemos ver que o ponto de escolha A já foi partilhado três vezes, por isso ficou de fora nesta partilha de trabalho de modo a evitar o excesso de partilha do topo da árvore.

4.4 Terminação da Computação

Um aspecto importante na computação distribuída é a detecção da terminação da computação por parte dos agentes. Com o intuito de minimizar os custos associados a este processo, o YapDss tem como base o mecanismo de terminação proposto por [Mat87].

Os princípios básicos que devem regem o processo de terminação estão relacionados, com:

- Evitar falsas detecções de terminação, evitando a perda de agentes na computação.

- Evitar detecções demoradas do fim da computação.

A base para o processo de terminação consiste no uso de *tokens*. Estes *tokens* circulam de forma acíclica de agente em agente, até chegar ao agente que iniciou o processo de terminação, formando desta forma uma anel lógico. O processo de terminação acontece quando um agente inactivo emite um *token* e recebeu-o mais tarde *limpo*, caso contrário o *token* chega ao agente iniciador *sujo*. Quando um agente iniciador recebe o seu token limpo, começa o processo de terminação, enviando mensagens de terminação para todos os outros agentes envolvidos na computação. Quando um agente recebe uma mensagem de terminação, envia as sua soluções para o agente *principal*, e retoma ao estado suspenso.

Quando um agente recebe um token *sujo*, actualiza os registos de carga com a informação que vem contida no token (as cargas dos agentes activos na altura da passagem do *token*), e avança para a solicitação de trabalho para o agente que tenha maior carga.

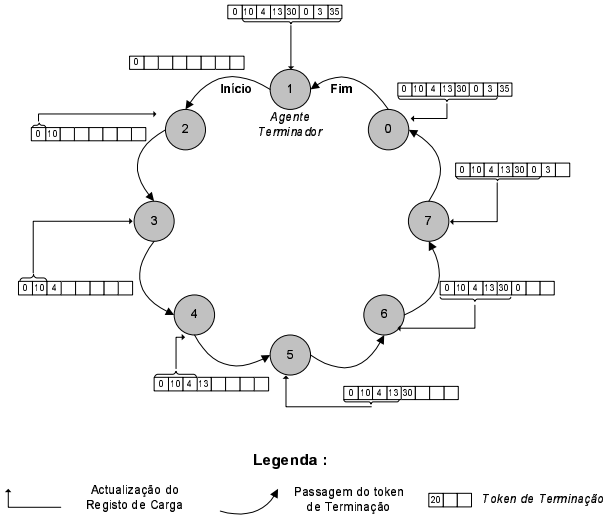

Figura 4.9: Terminação com optimização do registo de carga

Um *token* fica *sujo*, quando acontece um dos seguintes cenários:

Agente activo Quando o *token* chega a um agente activo, o *token* passa para o estado *sujo*, dado que ainda existe computação no sistema.

Mensagens pendentes Número total de mensagens recebidas no sistema é diferente do número total de mensagens enviadas, implicando a presença de mensagens ainda não processadas no sistema, logo podendo haver trabalho pendente para computar.

Caso existam vários tokens a circular, o *token* do agente com maior prioridade permanece, enquanto que os outros vão sendo descartados pelos agentes do sistema. O

agente com maior prioridade é o agente 0, sendo o agente com menor prioridade o agente *N*.

4.5 Resumo do Capítulo

Neste capitulo foram ilustrados os principais pontos do modelo do *YapDss*: o modelo básico de execução; cópia incremental de ambiente com Stack Splitting; o distribuidor de trabalho; terminação da computação. Para possibilitar a existência de cópia incremental, foram ilustrados os processos que permitem determinar o nó comum entre duas sub-árvores de procura.

O próximo capitulo pormenoriza a implementação dos processos relacionados com o modelo do *YapDss*.

Capítulo 5

Extensão do Yapor para Suportar o YapDss

Este capítulo descreve as extensões e modificações introduzidas no *YapOr* de modo a suportar o sistema implementado, o *YapDss*.

5.1 Organização de Memória

Seguindo a arquitectura habitual dos compiladores de Prolog, o sistema *Yapor* tem quatro áreas fundamentais de trabalho: **heap; trilha; global; e local**. A função de cada uma destas áreas já foi anteriormente descrita no capítulo 2.

Para além das quatro áreas acima mencionadas e à semelhança da *WAM*. Existe uma área auxiliar usada pelo ambiente de suporte, em particular para o compilador, analisador sintáctico, base de dados interna e algoritmos de indexação e unificação.

No *Yapor*, para suportar o modelo paralelo foram criados dois espaços distintos a nível organizacional:

Espaço Global serve como suporte ao paralelismo do sistema, estando aqui localizadas as estruturas de apoio. Este espaço está divido em quatro áreas fundamentais: área de código; área de informação global, que serve de suporte à sincronização dos vários agentes; área de estruturas partilhadas, que contém as estruturas de apoio ao paralelismo; e por último a área onde ficam guardadas as soluções da execução do objectivo em exploração.

Espaço Local O espaço local é um conjunto de subespaços locais, cada um deles representado um dos agentes do sistema. Para além da pilhas *WAM*, estes espaços contém também uma área de informação local usada na execução paralela. A opção de colocar as áreas de memória a crescerem em direcções opostas (ver

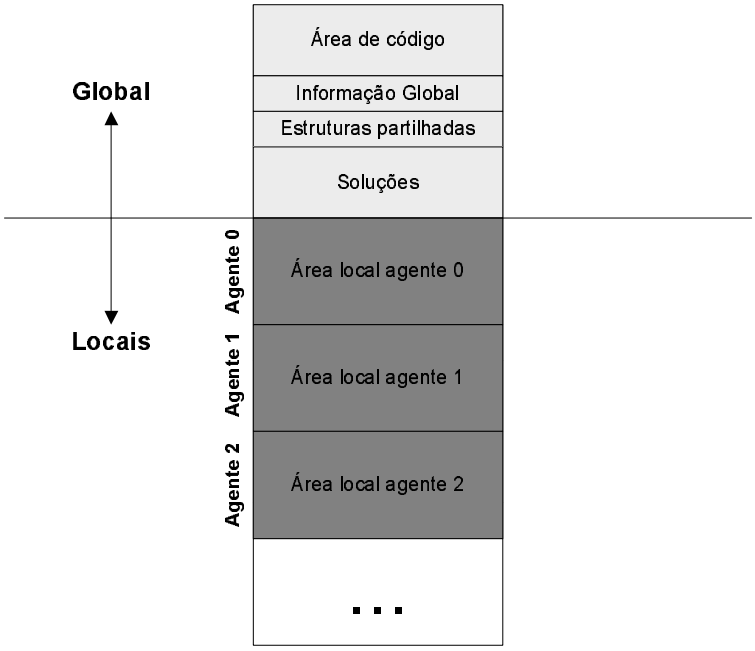

Figura 5.1: Organização de memória do YapOr.

figura 5.2) tem como objectivo um melhor aproveitamento da mesma, dado que não existem limites pré-definidos, uma pilha cresce até um ponto onde a oposta não o tenha feito. Outra razão está relacionada com o menor número de testes necessários para validar as áreas das pilhas.

Figura 5.2: Organização das pilhas *WAM* no espaço local.

O *YapDss* utilizou em parte a estruturação de memória do *YapOr*, contudo usada para diferentes objectivos, as área utilizadas foram:

Área Global Serve para armazenar soluções parciais de cada agente, estruturas de suporte e área de código.

Espaço Local O *YapDss* usa um espaço local por instância, existindo para manter o mapeamento entre os agentes do sistema.

A necessidade de manter o mesmo espaço de endereçamento[1] entre os agentes envolvidos na computação manteve-se, dai a necessidade de usar o espaço local. A área

[1]Entenda-se pelo mesmo espaço de endereçamento, o mapeamento do espaço local a partir do mesmo endereço de memória, de modo a evitar realocações de endereços durante as operações de cópia das pilhas de execução.

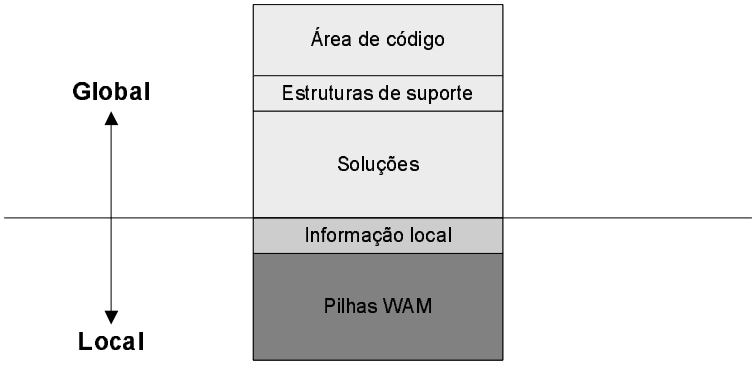

Figura 5.3: Organização da memória no YapDss.

global foi mantida de modo a utilizar os mecanismo já existentes do *Yapor*, entre os quais, armazenamento de soluções e acesso ao código do programa a executar.

5.2 Pontos de Escolha

Em Prolog, a função de um ponto de escolha é de guardar o estado computacional no momento da sua criação, para que num momento posterior de computação seja possível recuperar esse estado, de modo a podermos executar outras alternativas. Num modelo paralelo, como o *YapOr*, os pontos de escolha podem ser partilhados por vários agentes do sistema, o que desde logo requer mecanismos de sincronização para evitar que 2 agentes explorem alternativas repetidas.

As modificações feitas pelo YapOr ao sistema base Yap, foram as seguintes (ver figura 5.4):

- Apontador para estrutura partilhada (**CP_EP**)
- Total de alternativas por executar (**CP_APE**)

Para o cálculo do valor do campo CP_APE foi preciso introduzir uma modificação na formato inicial das instruções do *Yap*, porque a única maneira de se conhecer o número de alternativas que restam num dado ponto de escolha, era percorrer a lista de referências desde a alternativa *Alt* corrente até atingir a última alternativa. Dado a ineficiência desta aproximação, o Yapor modificou a estrutura existente.

As instruções que tem referências para outras alternativas estão intrinsecamente ligadas à criação e manuseamento dos pontos de escolha. Logo, todas elas tiverem de ser alteradas de modo a suportar a nova aproximação.

A figura 5.5 ilustra a finalidade e funcionalidade do campo **ape** (alternativas por explorar) introduzido pelo sistema Yapor. O código relativo a este campo é gerado

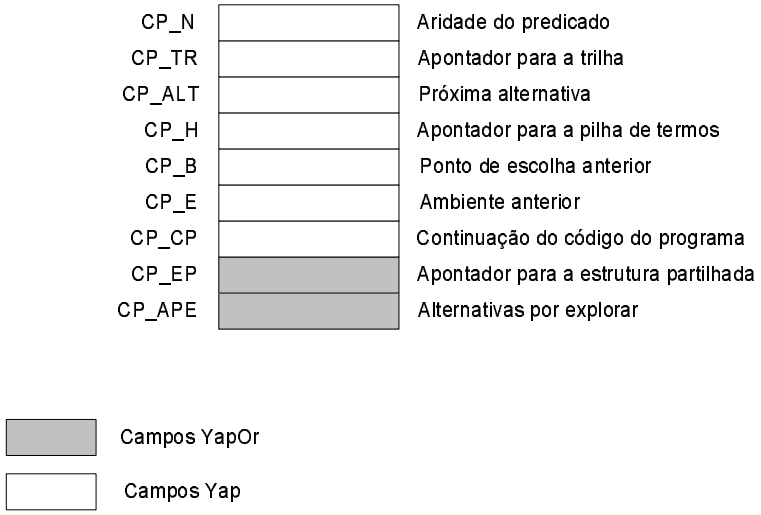

Figura 5.4: Estrutura de um ponto de escolha no YapOr.

para as instruções mencionadas, ocorrendo em tempo de compilação do programa Prolog para as instruções *WAM*.

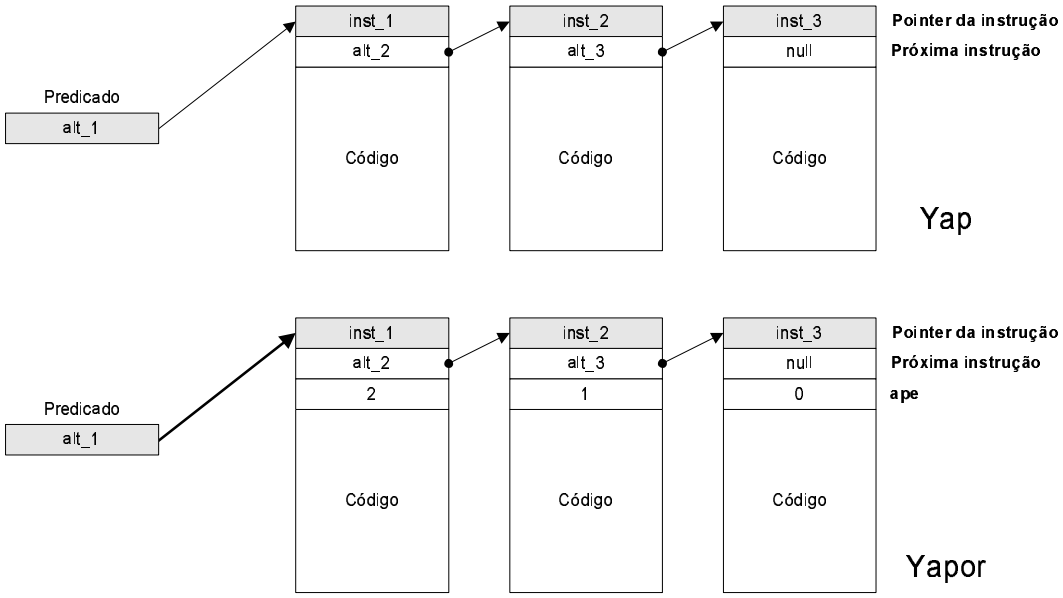

Figura 5.5: O novo campo **ape** na estrutura de dados dos predicados.

Com a introdução deste campo é possível calcular de um modo optimizado a carga do sistema num dado ponto de escolha.

Uma das adaptações importantes está relacionada com a passagem dos pontos de escolha de privados para partilhados. A figura 5.6 ilustra essa transição. Os campos **CP_ALT** (próxima alternativa a executar) e **CP_EP**(apontador para estrutura partilhada) do ponto de escolha passam a apontar respectivamente para a pseudo-instrução **procura_trabalho** e para a estrutura partilhada criada. A estrutura partilhada fica

com a alternativa que se encontrava anteriormente no campo **CP_ALT** do ponto de escolha, e com os dois agentes envolvidos na operação de partilha.

A instrução **procura_trabalho** é usada para identificar pontos de escolha partilhados. Quando um agente retrocede para um ponto de escolha que tem esta pseudo-instrução no campo **CP_ALT**, acede à estrutura partilhada e tenta ter acesso exclusivo à estrutura através do campo **fecho_de_acesso** da mesma. Isto para sincronizar os vários agentes e impedir que agentes diferentes executem a mesma alternativa.

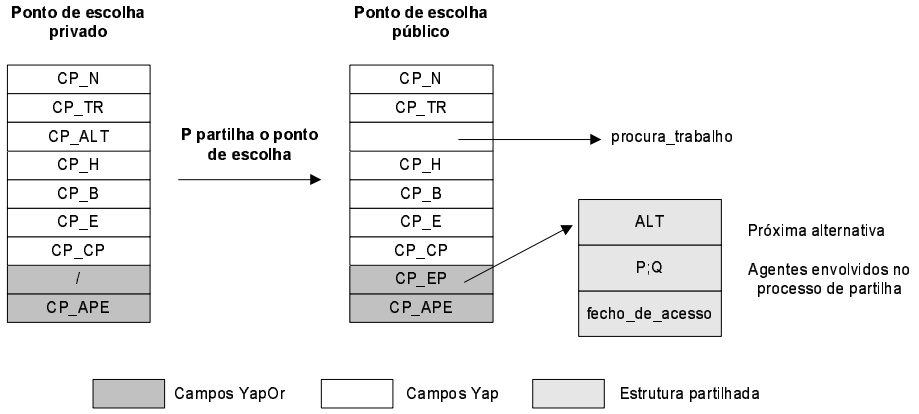

Figura 5.6: Partilha de um ponto de escolha.

Os pontos de escolhas no YapDss sofreram alterações de modo a suportar um ambiente distribuído com memória distribuída. Dada a sua natureza não é possível implementar zonas de memória partilhada entre os vários agentes do sistema, logo o uso das estruturas partilhadas foi eliminado. Contudo foi necessário criar um mecanismo capaz de controlar as alternativas a executar por cada agente envolvido no processo de computação, sendo uma peça fulcral no sistema de partilha de trabalho. Este campo tem o nome de **cp_offset** e tem como objectivo permitir ao agente saber qual a próxima alternativa a executar (ver figura 5.7). Contudo foi necessário recorrer ao campo **ape** anteriormente implementado pelo sistema Yapor que tem como função a indicação do número de instruções que ainda existem para executar num dado ponto de escolha.

5.2.1 Campo Offset

O campo *offset* foi criado de modo a que cada *agente* saiba qual a alternativa a tomar em cada *ponto de escolha*, dada a impossibilidade de se alterar a sequência das alternativas e dada a ausência de zonas de memória partilhada para se efectuar o controlo sobre a execução paralela das alternativas (ver figura 5.8).

A figura 5.9 ilustra sucessivos processos de partilha de trabalho. A figura 1 e 2 apresentam respectivamente as alternativas pertencentes a P e Q após a partilha de trabalho. Na subfigura 1.1 podemos ver as alternativas do agente P, após ter partilhado

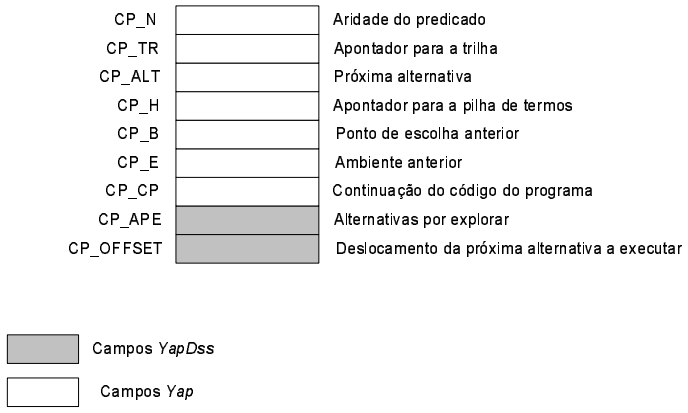

Figura 5.7: Estrutura de um ponto de escolha no YapDss.

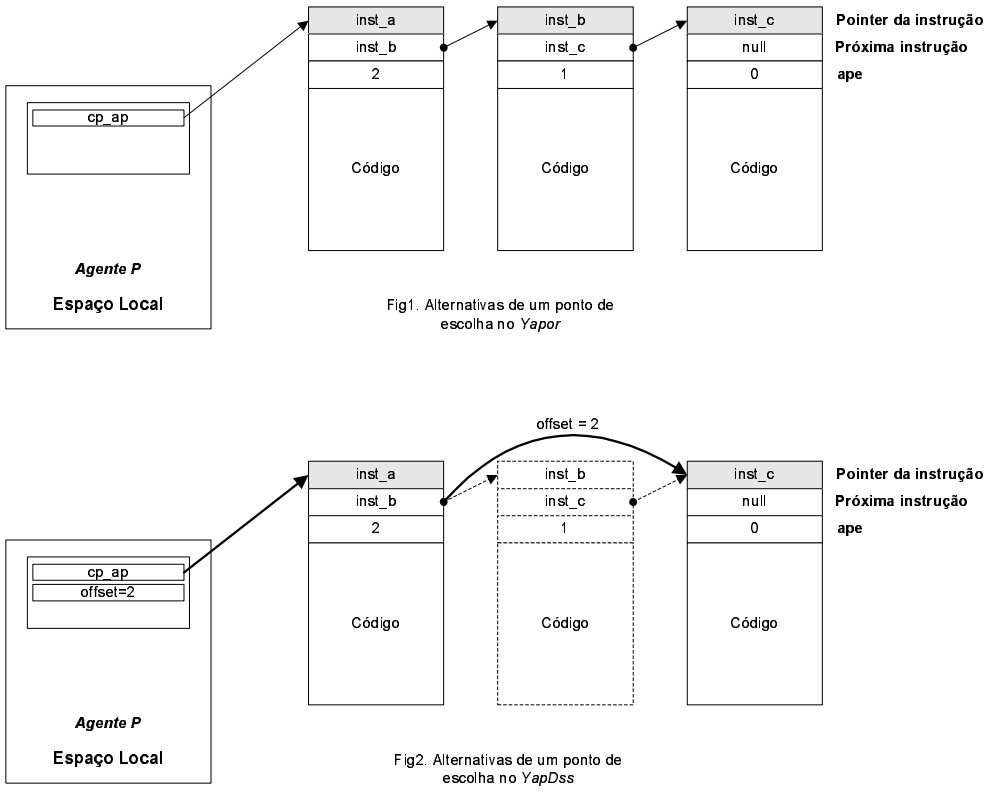

Figura 5.8: O novo campo *offset*

novamente trabalho agora com o agente Z, que por sua vez ficou com as alternativas da subfigura 1.2. Podemos ver que sempre que um ponto de escolha é partilhado o seu *offset* é actualizado pelo dobro do seu valor original, de modo a garantir que ambos os agentes não tomam as mesmas alternativa.

Para a utilização deste campo foi preciso alterar algumas instruções do *YAP* para suportar, a exploração das várias alternativas de um ponto de escolha como se de uma lista ligada se tratasse. As instruções do *Yap* já incluem uma referência para a próxima instrução. A posição do campo para a próxima alternativa (campo ld.d) foi

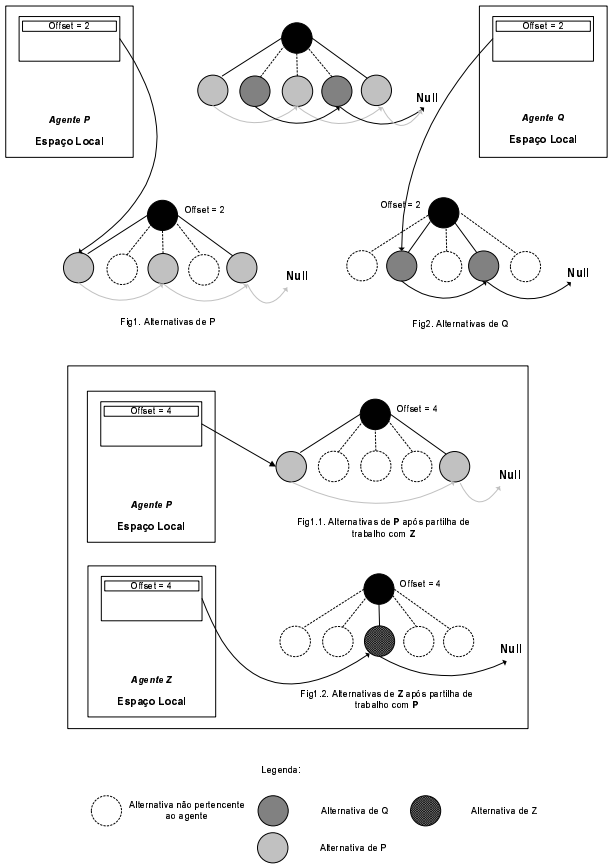

Figura 5.9: Função do *offset* na partilha e execução de pontos de escolha

uniformizado em todas as instruções, de tal modo que, tendo um offset *deslocamento* e uma instrução *inst*, podemos calcular a próxima alternativa a explorar com a seguinte função:

```
#define PROXIMA_ALT(CP)    CP->cp_ap
#define PROXIMA_INST(inst) inst->ld.d
#define YAMOP_LTT(inst)    inst->ape //Número de alternativas por explorar

 yamop* SCH_update_alternative_by_offset(yamop inst,offset deslocamento){
    yamop* current_inst;
    if(deslocamento>YAMOP_LTT(inst))
        return 0;
    current_inst = PROXIMA_ALT(inst);
    while(deslocamento>0){
        current_inst = PROXIMA_INST(current_inst);
        deslocamento--;
    }
    return current_inst;
}
```

Tendo uma posição fixa para a próxima alternativa é importante, porque evita verificar o tipo da instrução que estamos a lidar para saber como obter a próxima instrução.

Em seguida, serão analisadas as modificações feitas nas pseudo-instruções, para permitir a passagem para o modelo do YapDss.

5.3 Pseudo-Instruções

No YapOr foram introduzidas duas novas instruções para estabelecer a ligação entre o sistema sequencial do Yap e o sistema paralelo. Uma delas, que já foi anteriormente mencionada, **procura_trabalho**, e a outra é a **procura_trabalho_primeira_vez**.

As pseudo-instruções tem como origem do seu nome, o facto de não serem instruções geradas durante a compilação, mas que foram introduzidos para possibilitar o processo paralelo.

No caso da pseudo-instrução **procura_trabalho_primeira_vez** é executada no início do YapOr e no fim da exploração de um objectivo, por todos os agentes com a excepção do agente 0. Esta instrução coloca os agentes num estado suspenso à espera de uma determinada sinalização por parte do agente 0, dando deste modo início à exploração do objectivo. O agente 0 por apresentar as soluções produzidas pelo sistema, bem como a gestão da *interface* com o utilizador, até ser pedido pelo utilizador a exploração de um novo objectivo.

O YapDss usa apenas a pseudo-instrução, **procura_trabalho_primeira_vez**, do YapOr, com a mesma finalidade mas com aproximação diferente. A pseudo-instrução **procura_trabalho** deixa de ter sentido num contexto distribuído dado o facto de não existir estruturas partilhadas para controlar as alternativas a executar.

A pseudo-instrução tem funções complementares no YapDss porque é aqui que o código do programa a executar é distribuído pelo agente 0 para os restantes agentes. No YapOr como o código do programa estava em memória partilhada, era acessível por parte de todos os agentes, contudo no YapDss apenas o agente 0, que gere a interface com o utilizador, tem acesso ao código numa primeira fase. Através de um *broadcast*, todos os outros agentes recebem o código do programa, copiando-o seguidamente para a sua área de código. Quando todos os agentes tiverem copiado o código, ocorre uma sincronização. Deste ponto o processo é em tudo idêntico ao *Yapor*, com o agente 0 começando por explorar a árvore de computação e com os restantes agentes à procura de trabalho.

Quando a computação termina, o agente 0 recebe de todos os agentes as suas soluções parciais do objectivo explorada, sendo da responsabilidade do agente 0, a gestão dessa informação.

5.4 Carga de um Agente

O registo de **carga** é uma medida sobre o número de alternativas privadas que um dado agente possui. Para o cálculo desse valor em tempo de execução o Yapor tira partido do novo campo *CP_APE* introduzido nos pontos de escolha.

A medida mantida pelo YapOr, não é uma medida exacta mas sim uma quantificação aproximada do valor real, de outro modo o custo associado à manutenção deste valor seria elevado em termos computacionais. Como este valor é só usado no processo de selecção do agente na procura de trabalho, existe um compromisso entre a manutenção do valor correcto do registo de carga e a eficácia do sistema paralelo, ou seja, existe sempre um valor aproximado do registo de carga, sendo este valor exacto no momento de criação de um novo ponto de escolha.

Para possibilitar este cálculo é guardado no campo **CP_APE** de cada ponto de escolha o valor relativo aos registos de cargas dos pontos de escolha precedentes. O valor do registo de carga é igual à soma do valor do campo **CP_APE** do ponto de escolha precedente com o valor do campo **ape** da alternativa apontada pelo campo **CP_ALT** do ponto de escolha precedente mais um(ver figura 5.10). A não inclusão de informação relativa ao número de instruções dada pela instrução apontada pelo campo **CP_ALT** do ponto de escolha corrente está relacionado com a necessidade de evitar actualizações constantes sempre que ocorrer o mecanismo de retrocesso.

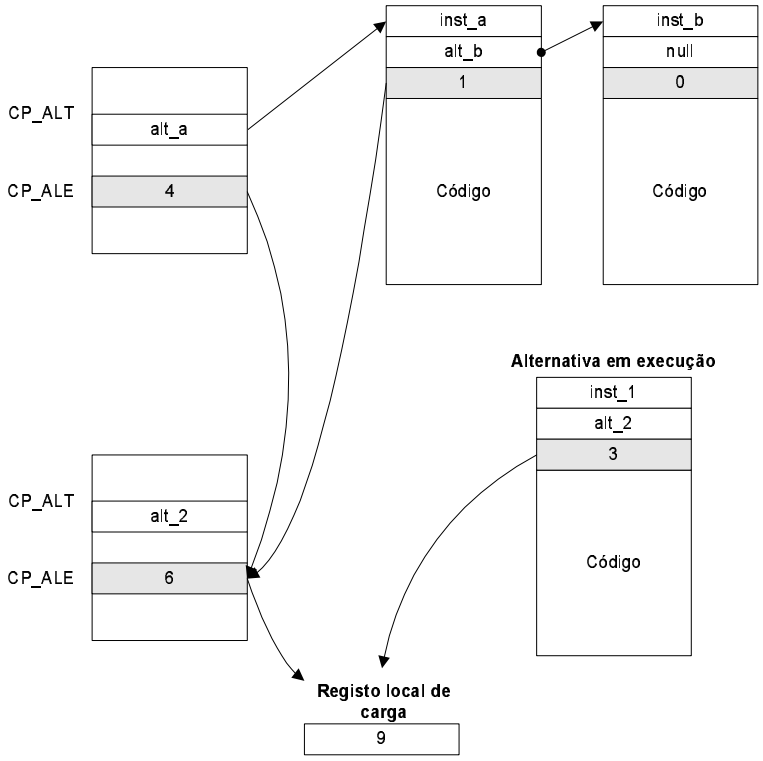

Figura 5.10: Cálculo da carga privada de um agente.

O sistema YapDss usa o mesmo mecanismo implementado no *Yapor* para ter informação sobre o número de alternativas que restam por ponto de escolha, ou seja o uso do campo **ape** nas instruções **WAM** responsáveis pelo criação e manuseamento de pontos de escolha. Contudo no YapDss não existe o conceito de nós públicos porque a distribuição de trabalho é estática, logo todos os nós são privados. Daqui resulta que a carga de um agente é a soma de todas as alternativas da árvore de computação corrente.

Num ambiente distribuído a necessidade para manter uma correcta medida da carga de todos os agente é fundamental porque o custo associado a múltiplos pedidos de trabalho e consequentes respostas devido a uma insuficiente actualização da carga podem levar a um atraso significativo no processo global de computação. Logo ao contrário do YapOr, o registo carga tem o valor real em cada momento do número de alternativas total que um agente ainda tem por explorar (ver figura 5.11).

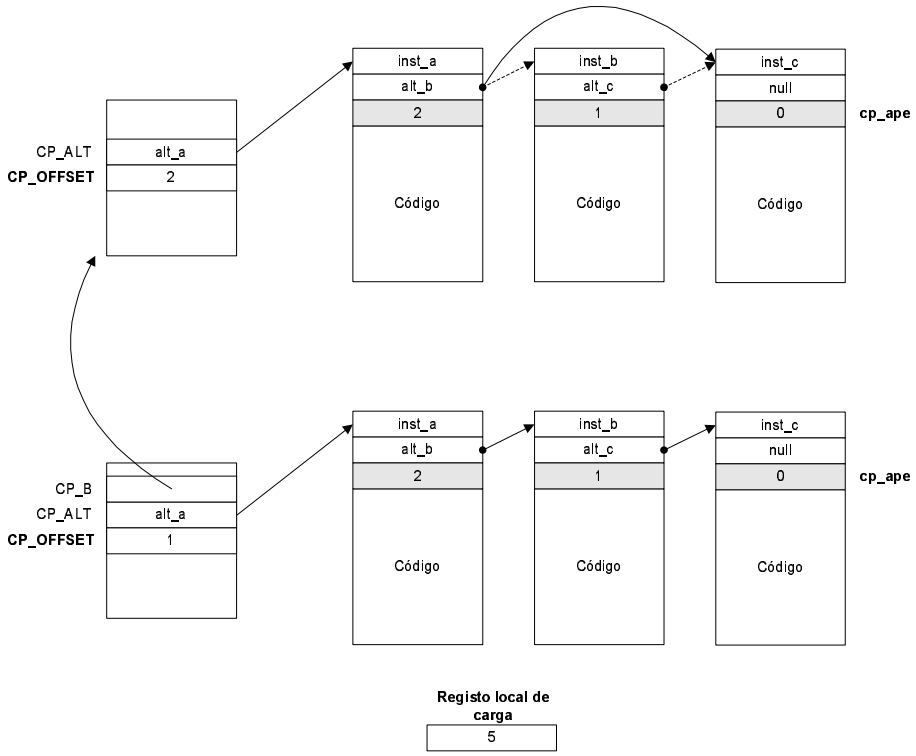

Figura 5.11: Cálculo da carga no YapDss.

Para o processo de procura e partilha de trabalho foi preciso a criação de registos adicionais sobre a carga de todos os agentes do sistema, ou seja, cada agente além do seu registo de carga, também tem registo de carga dos outros agentes. Estes registos são aproximações sendo actualizadas quando existe interacção entre os vários agentes do sistema: pedidos de trabalho e terminação da computação. Sendo estes registos aproximações, caso o registo pessoal de carga de um agente fosse também ele uma aproximação, induziria uma margem de erro bastante grande no processo de selecção dos agentes nos pedidos de trabalho.

5.5 A Área de Código e o Processo de Indexação

No YapOr, a área de código insere-se no espaço global, logo sendo acessível a todos os agentes do sistema. Qualquer alteração desta área tem de ser sincronizada, existindo um mecanismo de fecho para garantir o acesso exclusivo. O agente 0 é responsável por quase todas as alterações efectuadas nesta área, que normalmente ocorrem fora do processo paralelo. A única operação efectuada no decorrer do processo paralelo, por qualquer agente do sistema, é a indexação de um predicado. Um predicado é indexado uma única vez quando é chamado pela primeira vez.

Sempre que uma cláusula é introduzida no sistema, passa por uma série de fases. Inicialmente e após uma série de subfases, uma cláusula é compilada para uma sequência de instruções *WAM*, que são guardadas na área de código. Esta sequência além do código da cláusula, inclui um cabeçalho. Este cabeçalho é usado posteriormente para estabelecer a ligação, através de uma lista de apontadores com as cláusulas já existentes do mesmo predicado.

Antes da primeira chamada, uma cláusula não está associada ao seu código verdadeiro, mas sim a uma pseudo-instrução de indexação. Quando um predicado é chamado pela primeira vez, o sistema executa a pseudo-instrução de indexação, dando assim início ao processo de indexação. O código de indexação gerado é igualmente guardado na área de código. Futuras invocações do predicado passam a usar o código de indexação entretanto gerado. A função do mecanismo de indexação é limitar o número de cláusulas a considerar mediante o argumento usado na invocação do predicado.

No YapDss foi preciso alterar o processo de indexação dada a impossibilidade de expandir o código após o início da execução paralela sem o envio das novas alterações a todos os agentes do sistema, que tornaria o sistema ineficiente.

O processo de indexação passou a ser executado após a compilação inicial, e não quando é chamada pela primeira vez. Deste modo, quando se dá início ao processo distribuído todas as alterações estão efectuados evitando a necessidade de futuras sincronizações. Como o código de um programa pode ser alterado entre explorações implica que sempre que se inicia a exploração de um novo golo, todo o código tem que ser passado do agente 0 para os restantes agentes do sistema.

5.6 Procura de Todas as Soluções

No sistema *Yap*, após a invocação de um novo objectivo, termina a computação quando encontra a primeira solução. Poderá continuar a procurar soluções de uma forma sequencial, caso o utilizador assim o deseje. No Yapor e no YapDss, todas soluções da árvore de procura são recolhidas, ou seja, a computação termina quando não existir mais nenhuma alternativa a executar na árvore de computação e todas as soluções tiverem sido recolhidas.

Existe uma diferença fundamental entre ambos os sistemas, no *Yapor*, as soluções são armazenadas na área global, que é acessível a todos os agentes, enquanto no YapDss as soluções estão distribuídas pelo vários agentes, logo existe a necessidade de concentrar as soluções num agente, nesta caso o agente 0, dado que é este agente que tem a responsabilidade de fazer a *interface* com o utilizador.

Após as inicializações fundamentais ao sistema, o agente 0 carrega o predicado '$live', definido por defeito no ficheiro 'boot.yap'. Este ficheiro é carregado no início do sistema e tem definida toda a interface com o utilizador. De modo a ser possível efectuar toda a exploração da árvore de procura foi preciso adaptar o código em 'boot.yap'.

As principais modificações estão relacionadas com a introdução do predicado *sucesso* e *insucesso*, ambos não nativos ao Prolog, sendo implementados em C. O agente vai processando os comandos efectuados pelo utilizador através da *interface*. Se o utilizador indicar a execução de um objectivo então o agente 0 inicia a execução do predicador *yapor*(ver figura 5.12).

```
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 5.12: Código Prolog adaptado em boot.yap pelo YapOr.

No corpo do predicado *yapor* é invocado inicialmente o predicado *yapor_pergunta*, daqui pode ocorrer uma de duas situações: o objectivo não tem variáveis ou contém uma ou mais variáveis. Em ambos os casos, o agente 0 começa por executar o predicado *call*, dando início à exploração do objectivo, sinalizando os outros agentes do início do processo paralelo.

- No caso do objectivo não conter variáveis, trata-se apenas na verificação do seu insucesso ou sucesso. Isto é alcançado através da execução do predicado *yapor_sucesso*, que guarda uma referencia a indicar o sucesso ou insucesso do objectivo em questão.
- Caso existam variáveis a ser instanciadas é necessário guardar as soluções encontradas. Sempre que um agente encontre uma solução, que equivale a executar o predicado *call* com sucesso, executa o predicado *solução*. Este predicado cria uma nova estrutura na área global de soluções, colocando aí a nova solução. Daqui, o agente volta ao predicado *yapor* e executa a instrução *fail*, que obriga o agente a retomar a exploração de alternativas pendentes na sua árvore de computação. Quando toda a árvore de computação tiver sido explorada, todos os agentes com a excepção do agente 0, executam a pseudo-instrução *procura_trabalho_primeira_vez* e voltam ao estado inicial, ou seja, ficam suspensos à espera do início de uma nova exploração paralela. O agente 0 ao não encontrar

mais nenhuma alternativa por explorar, continua com a execução do código em '*boot.yap*'.

O sistema YapDss é idêntico ao Yapor, com a excepção que as soluções são armazenadas no espaço de soluções de cada agente, sendo necessária a sua recolha no final por parte do agente 0 (ver figura 5.13). Quando o agente 0 recebe as soluções de um determinando agente, insere-as no seu espaço de soluções. Quando todos os agentes tiverem enviados as suas soluções, o agente 0 fornece-as à *interface*.

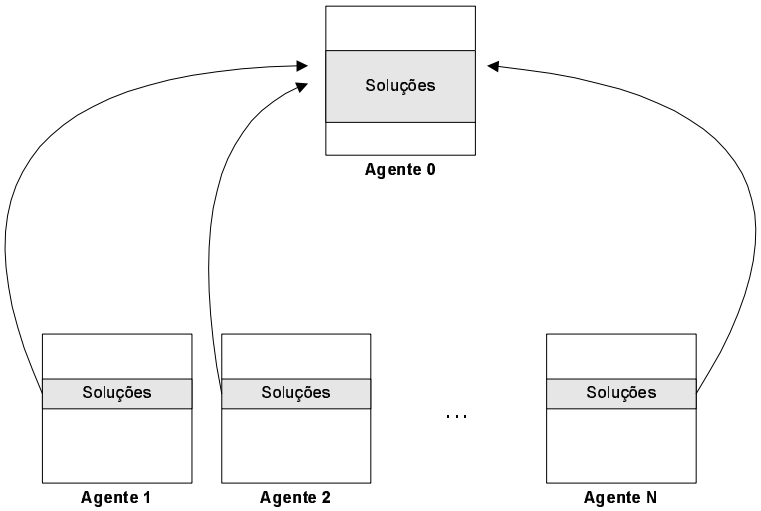

Figura 5.13: Concentração das soluções no agente 0.

Por fim, o agente 0 responde com sucesso ou insucesso ou fornece o conjunto de soluções que satisfazem o objectivo explorado pelos diversos agentes do sistema. Daqui, o agente 0 recomeça o processamento dos comandos do o utilizador.

5.7 Emulador de Instruções

No Prolog, o código das várias cláusulas é transformado numa sequência de instruções *WAM*. Estas instruções são executadas a partir de um emulador, que neste caso foi herdado do Yap em primeira instância, seguindo-se com algumas modificações o YapOr. Esta secção descreve as modificações necessárias para a implementação do YapDss.

No sistema Yap, o emulador de instruções é um ciclo infinito (ver figura 5.14), o qual contem um *switch* responsável pela selecção dos procedimentos a executar mediante a actual instrução *WAM*. A instrução *WAM* a executar é nos dada pelo registo de *program_counter*, sendo este actualizado no final da execução de cada instrução, seguido da execução da instrução 'goto ciclo_wam', que nos leva ao início do ciclo do emulador.

```
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;
   case try_me:
            alt = próxima alternativa a executar;
            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 executar;
            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 ponto de escolha corrente;
            program_counter = próxima instrução;
            goto ciclo_wam;
   ...
falha:
   case fail:
            usar a trilha para desreferenciar as atribuições
                às variáveis  aí guardadas;
            program_counter = próxima instrução;
            goto ciclo_wam;
}
```

Figura 5.14: Instruções da WAM do Yap.

A instrução *fail* pode ser executada a partir do *switch* ou através de um salto incondicional com o uso da instrução 'goto fail;'. Esta instrução de salto encontra-se em algumas instruções *WAM*, por exemplo, nas instruções ligadas ao processo de unificação.

Na execução de um predicado *pred*, ou seja, quando é invocada a instrução *call* com o argumento *pred*, mediante o número de cláusulas, pode acontecer o seguinte:

- Caso o predicado tenha mais do que uma cláusula, o *program_counter* é associado com a primeira alternativa do predicado e é executado de imediato uma instrução do tipo *try_me*, seguindo-se o verdadeiro código da alternativa.
- Caso *pred* tenha somente uma alternativa, o *program_counter* é instanciado com esse código.

Na instrução *fail* tem lugar a recuperação das variáveis de topo da trilha, seguindo-se a execução da próxima alternativa guardada no ponto de escolha corrente. A instrução

de topo da sequência de instruções da nova alternativa contém uma referência para a próxima alternativa ou o valor *null* caso não exista mais nenhuma alternativa. As instruções *retry_me* e *trust_me* são as instruções de topo dessas sequências. Em ambas as instruções o estado computacional é recuperado do ponto de escolha corrente, enquanto que na instrução *retry_me*, é guardada a próxima alternativa, na instrução *trust_me* o ponto de escolha é removido.

Na figura 5.15, ilustra as principais alterações feitas ao emulador do *Yap* por parte do *YapOr*. Dessas instruções destacam-se os seguintes pontos:

- O registo **B** tem a referência para o ponto de escolha corrente, enquanto que o *nó_partilhado_corrente* contém a referência para o ponto de escolha partilhado mais profundo.
- A função *incializações_YapOr()* é responsável pelas inicializações necessárias ao processo paralelo.
- Na instrução *call* foi introduzida a função *verificar_solicitações()*. Esta função é responsável pela verificação de pedidos de trabalho pelos outros agentes do sistema.
- A instrução *fail* permanece igual.
- Na instrução *try_me* foi introduzida a função *actualizar_carga()*, que actualiza a carga do agente no momento da criação do novo ponto de escolha.
- Nas instruções *retry_me* e *trust_me* foram introduzidas alterações, de modo a suportar o processo paralelo, entre as quais a manipulação e sincronização das estruturas partilhadas.
- Nas pseudo-instruções, existe a função *em_espera()*, responsável pelo mecanismo de sincronização entre agentes; a função *distribuidor_trabalho()*, é responsável por encontrar novo trabalho para exploração. Caso exista trabalho é simulada uma falha através de 'goto falha_partilhada', que actualiza o ponto de escolha corrente, seguindo-se a execução da instrução *fail*. Caso contrário, não existe mais trabalho e dá-se o fim da computação.

No YapDss o uso das pseudo-instruções e a sua ligação ao YAP é ligeiramente diferente (ver figura 5.16). As principais diferenças são:

- Ausência das estruturas partilhadas, dado que deixam de fazer sentido no ambiente distribuído.
- A função *incializações_Dss()* é responsável pelas inicializações necessárias ao processo distribuído.
- Os mecanismos de sincronização são agora realizados por troca de mensagens.

```
inicializações_Yapor();
program_counter = primeira instrução;
ciclo_wam:
switch(program counter){
   case call :
           pred = predicado relativo ao call;
           verificar_solicitações();
           ...
   case try_me:
            alt = próxima alternativa a executar;
            criar novo ponto de escolha;
            actualizar_carga(try_me);
            ...
   case retry_me:
            ...
   case trust_me:
            ...
   case procura_trabalho:
           estrutura_partilhada_corrente = nó_partilhado_corrente->CP_EP;
           fecho(estrutura_partilhada_corrente);
           if (nó morto){
              libertar_acesso (estrutura_partilhada_corrente);
              if (distribuidor_trabalho()){
                 goto falha_partilhada;
              }
              finalizações;
              if (agente 0) return;
              program_counter = procura_trabalho_primeira_vez;
              goto ciclo_wam;
           }else{
              program_counter = próxima alternativa;
              goto ciclo_wam;
           }
   case procura_trabalho_primeira_vez:
           em_espera (sinal de início por parte do agente 0);
           if (distribuidor_trabalho()){
                 goto falha_partilhada;
           }
           finalizações;
           program_counter = procura_trabalho_primeira_vez;
           goto ciclo_wam;
   ...

falha_partilhada:
            B = nó_partilhado_corrente;
falha:
   case fail:
            ...
}
```

Figura 5.15: Instruções relacionados com o processo paralelo no YapOr.

- Na inicialização do sistema distribuído, o agente 0 envia a área de código para os restantes agentes do sistema, antes de iniciar a execução.

- O pedido de trabalho acontece quando não existir mais nenhuma alternativa na sub-árvore de procura do agente por explorar. Para isso, foi preciso alterar a instrução *fail*, para que quando um agente não tenha mais nenhuma alternativa para executar no ponto de escolha corrente **B**, o agente suba na sua sub-árvore através da referência guardada em B->CP_B, que nos dá o ponto de escolha precedente. Este processo repete-se até encontrar-mos um ponto de escolha que tenha uma alternativa por explorar ou até alcançarmos o topo da sub-árvore de procura. Caso exista uma alternativa por executar, prossegue-se com o a instrução de *fail*, ou seja, com o mecanismo de retrocesso. Caso não haja mais nenhuma alternativa, o agente tenta pedir trabalho a outros agentes do sistema.

- Na instrução *call*, não se procede sempre à verificação da recepção de mensagens e consequente processamento, porque sendo este um mecanismo lento levaria a atrasos significativos caso não existisse uma contenção no processamento das mesmas. Para evitar esse atraso, o *YapDss* usa um contador que controla o mecanismo de verificação da recepção de mensagens.

- Quando o processamento distribuído acaba, as soluções são reunidas no agente 0.

- Ao contrário do YapOr, a actualização da carga do agente é realizada em vários pontos do emulador: na instrução fail, o agente diminuí a sua carga em um quando toma uma nova alternativa; na instrução *try_me*, a carga do agente é actualizada com o número de alternativas que o agente tem nesse novo ponto de escolha.

```
inicializações_Dss();
program_counter = primeira instrução;
ciclo_wam:
switch(program counter){
   case call :
           pred = predicado relativo ao call;
           if (Contador solicitações){
              verificar_solicitações();
           }

   case try_me:
            alt = próxima alternativa a executar;
            criar novo ponto de escolha;
            ...
   case retry_me:
            ...
   case trust_me:
            ...
shared_end:
             finalizações();
             if (agente == 0)
                goto ciclo_wam;
           }
   case procura_trabalho_primeira_vez:
           em_espera (sinal de início por parte do agente 0);
           if (distribuidor_trabalho()){
                 goto falha;
           }
           finalizações;
           program_counter = procura_trabalho_primeira_vez;
           goto ciclo_wam;
   ...
falha:
   case fail:
            alternativa na etiqueta de execução += B->offset;
            if( alternativa_corrente == 0){
               while (B != ponto de escolha de topo){
                   B = B->CP_B;
                   if (B->CP_AP != 0){
                      alternativa_corrente = B->CP_AP;
                      break;
                   }
               }
               alternativa na etiqueta de execução += B->offset;
               if (alternativa_corrente == 0){
                  if (distribuidor_trabalho()){
                     goto falha;
                  }else
                     goto shared_end;
               }else
                  carga_agente--;

            }else
                 carga_agente--;

    ...
}
```

Figura 5.16: Instruções relacionados com o processo distribuído no YapDss.

Capítulo 6

Implementação

Este capítulo apresenta os detalhes de implementação dos processos referidos anteriormente.

6.1 LAM/MPI

O *LAM* (*Local Area Multicomputer*) é um ambiente de programação e desenvolvimento para redes heterogéneas de computadores, que implementa por completo o *standard* ***MPI*** (*Message Passing Interface*). Com o *LAM*, qualquer rede ou *cluster* pode ser usada como se de um computador paralelo se tratasse. A utilização do **LAM/MPI** no sistema YapDss permite abstrair a camada de rede, simplificando a comunicação e troca de mensagens entre os agentes.

O *LAM/MPI* tem as seguintes características:

- Completa implementação do *standard* MPI
- Ferramentas para monitorização e *debugging*
- Suporte para redes heterogéneas de computadores
- A inclusão ou exclusão de nós
- Detecção de erros e recuperação
- Extensões *MPI* e suplementos de programação
- Comunicação directa entre processos de aplicações
- Gestão robusta de recursos
- Processos dinâmicos de *MPI-2*

- Vários protocolos de comunicação (memória partilhada e rede)

O LAM corre em cada computador como um único *daemon*, unicamente estruturado como um nano-*kernel*. O componente nano-*kernel* providencia um mecanismo de passagem de mensagens e sincronização para processos locais. Alguns dos processos do *daemon* formam uma rede de comunicações, de modo a ser possível a transferência de mensagens de um nó para outro. O subsistema de rede adiciona o empacotamento e *buffering* à sincronização base.

Figura 6.1: *Camadas do LAM/MPI*

Através do *Message Passing Interface*, uma aplicação vê o seu ambiente paralelo como se de uma grupo estático de processos de tratasse. Um processo *MPI* é criado com zero ou mais processos filhos. Ao grupo inicial de processos dá-se o nome de *grupo global*. Um número único, chamado *posto*, identifica cada processo, da sequência de zero até *N-1*, onde *N* é o numero total de processos no grupo global. Cada membro, pode questionar sobre o seu posto e o tamanho do grupo global. Um grupo global pode ser subdividido, criando subgrupos que podem ter potencialmente postos diferentes.

Um processo envia uma mensagem para o posto de destino no grupo desejado. O processo pode ou não especificar um posto de origem quando recebe uma mensagem. As mensagens também podem ser filtradas por um inteiro de sincronização chamado *etiqueta*, que o receptor também pode ignorar.

Um característica importante do MPI é a sua abilidade em garantir que a etiqueta usada por um dado programador numa biblioteca não entra em conflito com outra qualquer biblioteca existente, logo garantido independência entre o *software*. As quatro variáveis principais de comunicação são:

- Posto de origem
- Posto de destino
- Etiqueta
- Comunicador

No YapDss as etiquetas desempenham um papel importante, porque é possível num dado instante tratar mensagens de um certo tipo, descartando todos os outros tipos.

Isto é particularmente útil porque num ambiente distribuído é frequente o atraso de mensagens que chegam foram de contexto e que se tratadas só trariam atraso para o processo global de computação, por exemplo, poderíamos ter chegado ao fim da computação e um dado agente do sistema, ainda estar a receber mensagens anteriores ao fim da computação, como pedidos de partilha de trabalho atrasados. Estas mensagens deixaram de ter qualquer sentido, no novo contexto provocando atrasos no sistema caso sejam processadas.

Um comunicador é uma estrutura opaca do MPI que abstrai o conceito de sobre um grupo. O comunicador é argumento em todas as funções de comunicação do MPI. Quando um processo inicializa o MPI, estes três comunicadores pré-definidos estão disponíveis:

MPI_COMM_WORLD O grupo global

MPI_COMM_SELF Grupo com um membro, o próprio processo

MPI_COMM_PARENT Um intercomunicador entre dois grupos, o meu grupo e o grupo do meu grupo pai

As principais funções de comunicação usadas, foram:

MPI_Send Permite o envio de uma mensagem para um agente do sistema.

MPI_Recv Permite a recepção de uma mensagem de um dado agente.

MPI_Barrier Permite a sincronização entre todos agentes.

MPI_Broadcast Permite o envio de uma mensagem para vários agentes em simultâneo.

MPI_Gather Permite a recolha das soluções do sistema, isto é, todas as soluções de todos os agentes do sistema.

6.2 Protocolo de Comunicação

O protocolo de comunicação implementado no YapDss teve como objectivo principal a minimização do envio de mensagens de modo a evitar o excesso de processamento de mensagens por parte dos agentes.
O envio de mensagens é síncrono, no sentido, que cada agente só pode enviar uma mensagem quando receber a resposta da mensagem anterior. No entanto, responde a todas as mensagens que receber. As mensagens dividem-se em dois tipos:

Mensagens de Controlo Estas mensagens tem como finalidade a manipulação dos estados dos agentes. Existem 2 tipos mensagens de controlo:

- **Terminação** Inicia o processo de terminação da computação.
- **Pedido de Trabalho** Solicita trabalho a um agente.

Mensagens de Transferência Este tipo de mensagens serve para transferir informação entre agentes. Existem igualmente 2 tipos de mensagens de transferência:

- **Resposta ao pedido de Trabalho** Dada a carga actual do agente que recebeu o pedido de trabalho, a resposta pode ser afirmativa, contendo deste modo a informação relativa às diferentes pilhas, ou pode ser uma resposta negativa contendo somente a informação da carga actual do agente activo (agente com trabalho).
- **Envio de soluções** Usada no envio das soluções do sistema para o agente principal.

As mensagens utilizadas no *YapDss* podem dividir-se em 2 partes: o cabeçalho e o corpo da mensagem (ver figura 6.2). O cabeçalho serve para identificar o agente que enviou a mensagem, o comprimento da informação contida no corpo e o tipo. As mensagens de controlo tem tamanho do corpo fixo, enquanto que as mensagens de transferência tem tamanho do corpo variável dado que transportam informação sobre as pilhas a partilhar, e estas têm obviamente tamanho variável.

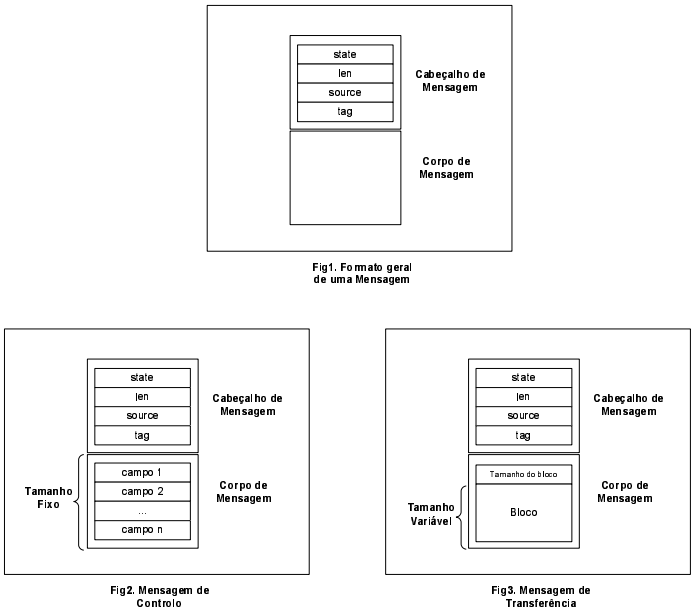

Figura 6.2: Estrutura dos diferentes tipos de mensagens

6.3 Partilha de Trabalho

O processo de partilha de trabalho, tem como objectivo a transferência de trabalho de um agente activo para um agente passivo.

6.3.1 Solicitação para a partilha de trabalho

O processo de partilha de trabalho começa com a solicitação para a partilha de trabalho (ver figura 6.4). Um agente Q, em estado suspenso, procura nos seus registos de carga o agente que possua maior carga. Seja esse agente P, o agente Q envia uma mensagem de solicitação de trabalho contendo os seguintes elementos:

Topo da sub-árvore de execução É o nó raiz da árvore de execução do agente.

Tamanho da etiqueta de execução O tamanho do array da etiqueta de execução.

Etiqueta de execução A etiqueta de execução, referente ao ponto que estamos na árvore global de execução.

O agente Q espera pela resposta do agente P, antes de tentar outro agente. Isto serve para evitar excessos de mensagens no sistema. No entanto, o agente Q responde aos pedidos de solicitação de trabalho e terminação da computação por parte de outros agentes, contudo como o agente está sem trabalho recusa sempre as solicitações de trabalho.

6.3.2 Resposta à solicitação de partilha de trabalho

Quando um agente P, em estado activo, recebe uma solicitação de trabalho, começa por analisar a sua carga actual, bem como a profundidade em que se encontra na árvore de execução. Caso esteja com carga e profundidade suficiente para partilhar trabalho, então o agente calcula a sub-árvore a partilhar com o agente Q.

O cálculo do nó de topo da sub-árvore a partilhar começa no nó actual, subindo na árvore de execução até encontrar um nó que não obedeça a um dos seguintes critérios:

Número de partilhas do nó O número de partilhas do nó (é nos dada pelo valor do offset) não pode ser elevado (por defeito 2).

Carga total da sub-árvore A carga total da sub-árvore a partilhar não pode passar de uma certa percentagem pré-definida (por defeito 55%, sendo configurável pela macro *DSS_IC_RATIO*).

Depois do cálculo da sub-árvore (ver figura 6.3) o agente P envia para o agente Q uma mensagem de resposta à solicitação prévia, contendo o seguinte (ver figura 6.5):

Aceitação do pedido Campo que informa da aceitação ou recusa do pedido de trabalho por parte de P.

Carga Carga actual do agente P.

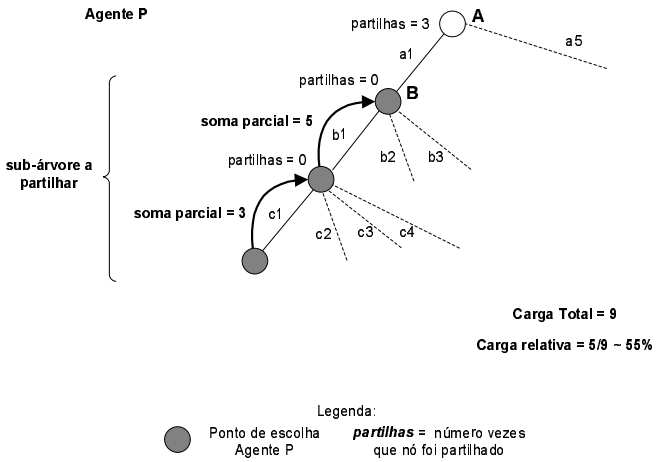

Figura 6.3: Cálculo da sub-árvore a copiar

Pilhas Áreas das pilhas a actualizar.

Caso não seja possível partilhar trabalho a resposta é negativa e a mensagem só contém a negação da pedido anterior e a carga actual de P.

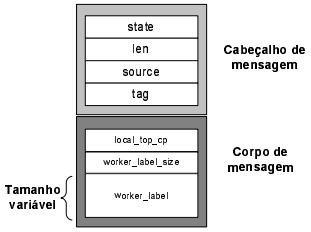

Figura 6.4: Partilha de Trabalho

De seguida, encontra-se a descrição detalhada da função de cada um dos campos ilustradas na figura 6.5:

reply Indica se a partilha foi aceite ou não. Caso a resposta seja negativa, somente os campos ***reply*** e ***load*** são enviados dado que a partilha não se concretizou.

load Este campo tem a informação sobre a carga do agente activo que respondeu ao pedido de trabalho.

LCA->cp_h Indica o endereço para o cálculo da área a copiar da pilha global.

LCA->cp_tr Indica o endereço para o cálculo da área a copiar da trilha.

tamanho segmento pilha global Indica o tamanho do segmento a copiar da pilha global.

segmento pilha global O segmento a copiar para a pilha global.

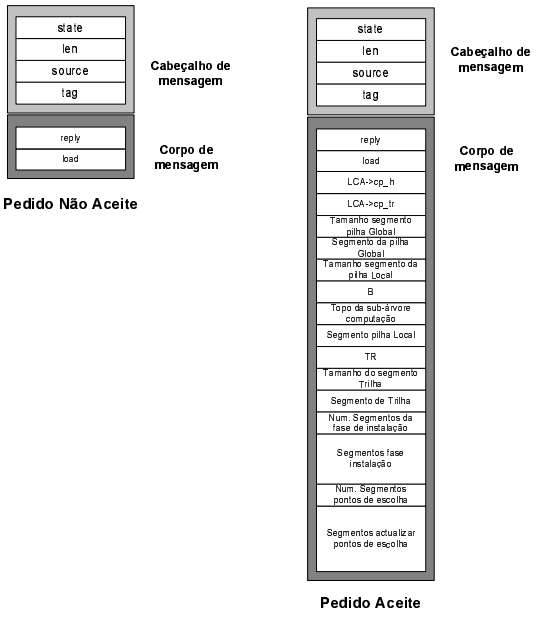

Figura 6.5: Mensagem de resposta à solicitação de trabalho

tamanho segmento pilha local Este indica o tamanho do segmento a copiar da pilha local.

B Este campo serve como optimização, para o agente suspenso não ter que proceder ao cálculo do novo **B**.

topo da sub-árvore O topo da sub-árvore é o ponto de escolha menos profundo da sub-árvore partilhada.

segmento pilha local O segmento a copiar para a pilha local.

tamanho segmento pilha trilha Indica o tamanho do segmento a copiar da trilha.

segmento pilha trilha O segmento a copiar para a trilha.

segmentos fase instalação Usado para actualizar referências que possam estar desactualizadas.

segmentos actualizar pontos escolha A função deste campo está relacionado com a introdução do campo *offset*. É necessário actualizar o campo offset que vai desde o *LCA* até o novo topo da sub-árvore, isto porque o agente activo pode ter partilhado com outros agentes essa parte da árvore, levando a incongruências na selecção da alternativa a executar.

6.3.3 Fases do Processo de Partilha

O processo de partilha divide-se em quatro fases, a saber:

Fase de preparação inicia-se o cálculo dos segmentos a copiar, começando no cálculo do *LCA*.

Fase de partilha o agente P, calcula os segmentos a partilhar baseado na carga.

Fase de cópia envolve o envio da mensagem com o trabalho e respectivas pilhas por parte do agente P, bem como a cópia dessa informação por parte do agente Q para o seu espaço de endereçamento.

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

Ao contrário do YapOr, o YapDss não permite ajudas no processo de cópia das pilhas dada a sua natureza distribuída. Existem fases distintas da intervenção do agente activo P e do agente passivo Q.

O agente Q começa por enviar no sua solicitação de trabalho, a sua etiqueta de execução, de modo a que o agente P possa calcular o nó comum. O agente P é o único responsável pelo cálculo das áreas a copiar, bem como os segmentos necessários para a fase da instalação, dado que seria necessário ao agente Q aceder às pilhas de P, como não é possível, isto levaria ao envio de outra mensagem e respectiva resposta para a obtenção dos referidos segmentos.

Depois de calcular as áreas, o agente P, envia para o agente Q, uma mensagem com as referidas áreas. Quando Q recebe a mensagem, somente tem de copiar a informação recebida para as suas pilhas (ver figura 6.6).

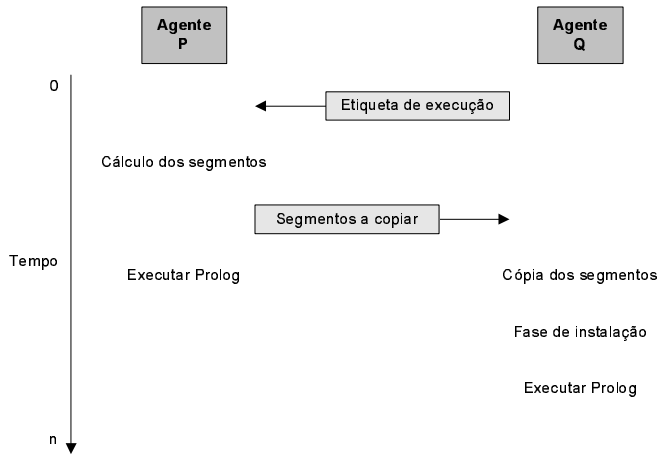

Figura 6.6: Processo de sincronização das pilhas

Existem os seguintes pontos a considerar:

- O agente P deve retomar a computação o mais rápido possível.
- A fase de preparação e partilha são feitas exclusivamente por P.

- Na fase de cópia existe interacção entre P e Q, através do envio de uma mensagem com o trabalho partilhado, seguindo-se a actualização das pilhas por Q, bem como a fase de instalação.
- O agente P pode retomar a computação assim que acabe de enviar a mensagem de partilha de trabalho.
- O agente Q só retoma o trabalho após a fase de instalação.
- P pode proceder ao retrocesso sobre o nó corrente, visto as pilhas serem independentes.

O processo centra-se no seguinte: nas fases de preparação e partilha, o agente P calcula a quantidade de trabalho a partilhar bem como todos os segmentos necessários para a cópia incremental, inclusive as atribuições condicionais feitas a variáveis nos segmentos a partilhar, visto ser impossível o agente Q aceder posteriormente ao espaço de endereçamento de P na fase de instalação. A fase de cópia e instalação, não passa da simples operação de cópia de memória do buffer da mensagem para as respectivas pilhas.

Após a conclusão do processo de partilha por parte de ambos os agentes envolvidos no processo, existe uma actualização do campo *offset* e próxima_alternativa nos pontos de escolha da sub-árvore partilhada. De outro modo, ambos os agentes estariam a executar as mesmas alternativas. No caso do agente P este processo de actualização ocorre quando este acaba o envio da mensagem de partilha de trabalho, enquanto no caso do agente Q só acontece depois da fase de instalação, ou seja, quando todas as pilhas foram actualizadas (ver figura 6.7).

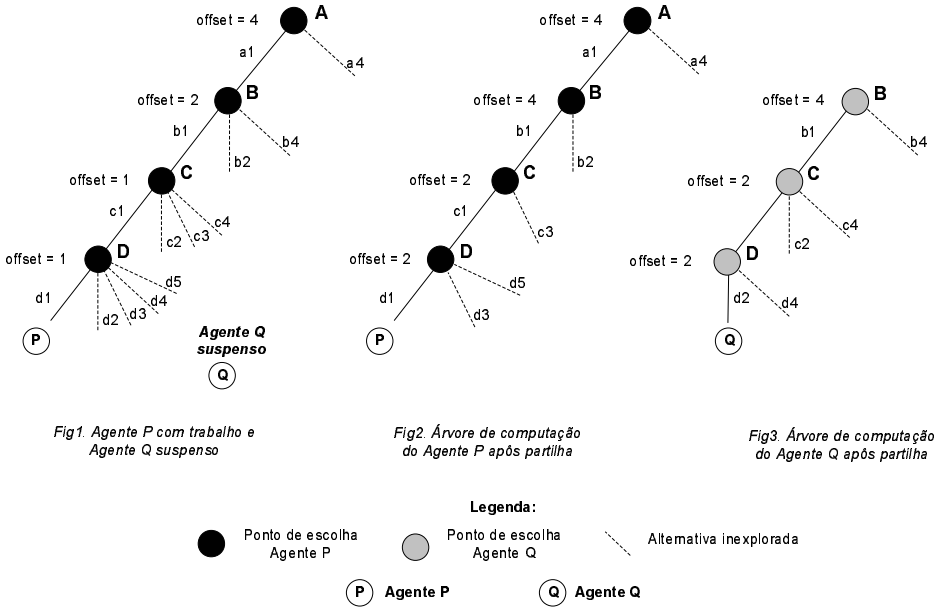

Figura 6.7: Actualização das alternativas e *offset* após partilha de trabalho.

6.3.4 Cálculo das Áreas a Partilhar

O cálculo do **nó comum** é a peça fundamental para se calcular as áreas das pilhas a copiar. Sem o nó comum não seria possível saber onde um dado agente se situa na árvore de computação global, logo seria impossível saber qual a relação posicional entre os vários agentes .

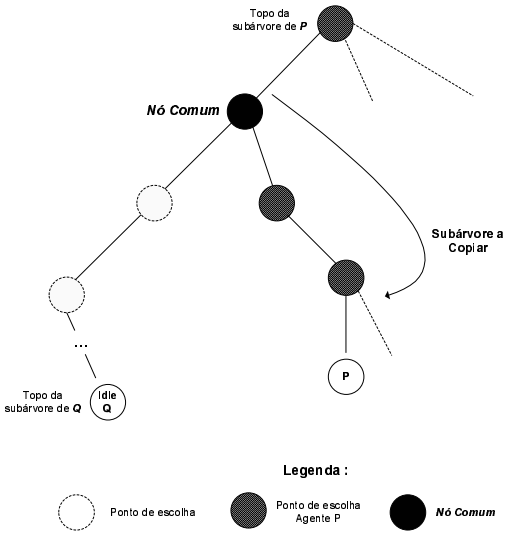

Figura 6.8: Visualização das sub-árvores, com o cálculo do nó comum.

No YapDss e baseando-se no mecanismo de cópia incremental, P é responsável pelo cálculo das áreas a copiar, o qual se apresenta esquematizado na figura 6.9. As áreas a copiar situam-se desde o primeiro nó comum até o ponto de escolha corrente (dado pelo registo B) de P.

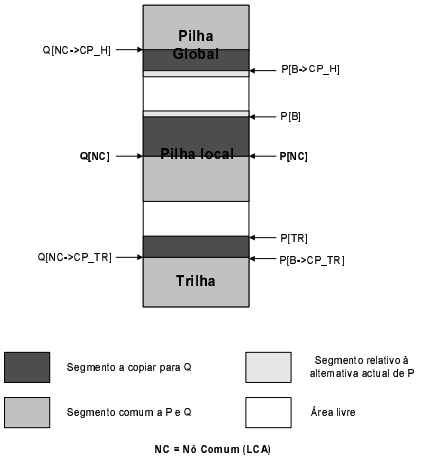

Figura 6.9: Visualização dos segmentos envolvidos no processo de partilha de trabalho

O apontador guardado em Q[NC->cp_h] indica o topo da pilha de termos , o apontador Q[NC->cp_tr] é o topo da trilha, em ambos referentes ao nó Q[NC], ou seja, eram

estes topos no momento da criação do ponto de escolha Q[NC] (que representa o ponto de escolha comum entre os 2 agente). No lado do agente P, o apontador P[TR] representa 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. O apontadores P[B] e P[B->CP_H] representam respectivamente o ponto de escolha corrente do agente P e o topo da pilha global no momento da criação de P[B].

6.3.5 Fase de Instalação

Neste fase o agente Q actualiza os segmentos da pilha local e pilha de termos, com o conjunto das atribuições condicionais efectuadas pelo agente P. A figura 6.10 apresenta o código responsável pela localização e cópia para a mensagem de resposta das atribuições condicionais, este código é executado pelo agente P.

```
char* buffer;   //buffer da mensagem
int num_tr = 0; //num. de segmentos da fase de instalação
aux_tr = B->cp_tr;
while (lca->cp_tr != aux_tr) {
    aux_cell = TrailTerm(--aux_tr);
    if (aux_cell < lca->cp_h ||
        lca <= aux_cell) {
        Copiar(buffer++,&aux_cell);  //endereço a copiar
        Copiar(buffer++,aux_cell); //valor do endereço a copiar
        num_tr++;
    }
}
```

Figura 6.10: Fase da instalação de P

A figura 6.11 apresenta o complementar da fase de instalação a ser executado pelo agente. O agente Q, somente tem de utilizar os endereços e respectivos valores de actualização e copiar para o seu espaço de endereçamento.

```
Cell aux_cell;
int num_tr;
num_tr = *buffer++);
while(num_tr > 0){
    Copiar(&aux_cell,buffer++);
    Copiar(aux_cell,buffer++);
    num_tr--;
}
```

Figura 6.11: Fase da instalação de Q

6.4 Actualização dos Registos de Cargas

Os registos de carga tem um papel fundamental no bom balanceamento da carga no YapDss. É através deles que a selecção do agente activo ao qual se irá pedir trabalho

é feita, logo uma má manutenção destes registo resulta numa perda de *performance* do sistema. Também é mediante a carga do agente que se aceita ou rejeita solicitações de trabalho, logo uma má actualização deste valor resultará na partilha indevida ou na falta dela.

6.4.1 Carga do agente

A carga de um agente é o soma das alternativas por explorar nos vários nós da árvore de execução. Todos os agentes tem uma estimativa da cargas dos outros agentes no sistema, obviamente dada a natureza distribuída da computação, este valor nunca passará de uma estimativa, estas cargas são denominadas como previsão de cargas.

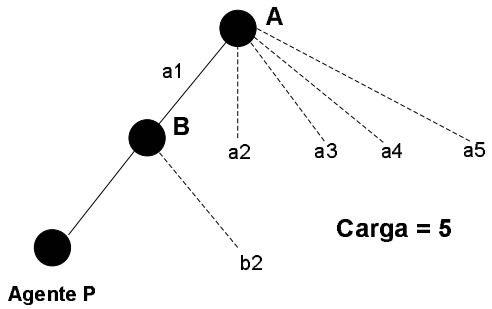

Figura 6.12: Carga de um agente

Existem três pontos onde a carga de um agente é actualizada:

Criação de um ponto de escolha A carga de uma agente é incrementada pelo número de alternativas do ponto de escolha que acabou de ser criado.

Retrocesso Quando acontece um retrocesso na computação, a carga de um agente é decrementada em um.

Partilha de trabalho Na partilha de carga os registos de cargas dos dois agentes envolvidos são actualizados mediante a repartição que for feita pelo distribuidor de trabalho.

6.4.2 Previsão de Carga

Existem dois pólos distintos de intervenção do registo de carga. O primeiro está relacionado com a quantidade de trabalho que um agente activo deve partilhar num dado instante, o outro, com a influencia na escolha do agente activo no processo de pedido de trabalho por parte de uma agente passivo.

As previsões de carga sofrem alterações nos seguintes cenários:

Pedido de trabalho Quando um agente passivo pede trabalho, recebe sempre o carga do agente activo ao qual pediu trabalho, actualizando esse valor no registo de previsões de carga.

Terminação Quando um agente inicia o processo de terminação da computação, os registos de carga vão sendo recolhidos conforme o *token* é passado de agente em agente(ver figura 6.13).

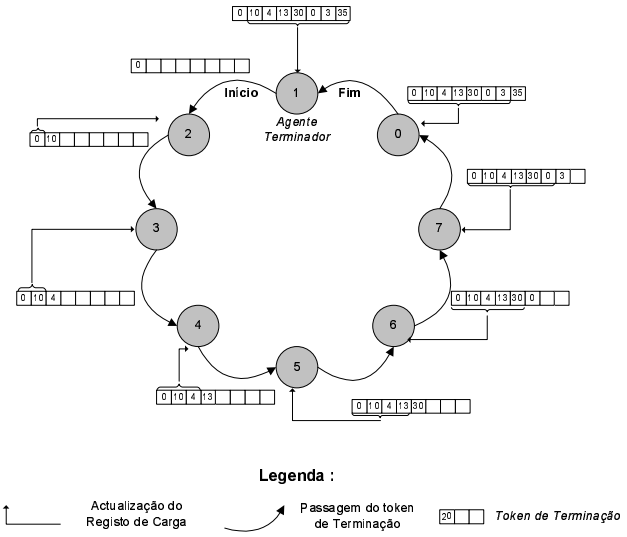

Figura 6.13: Terminação com optimização do registo de carga

6.5 Resumo do Capítulo

Neste capítulo descreveram-se em detalhe os pormenores de implementação dos mecanismos inerentes ao YapDss para suporte de Paralelismo-Ou num ambiente distribuído com memória distribuída.

Em seguida, é descrito os resultados obtidos da execução de um conjunto de programas de teste, bem como a análise a esses resultados, que incluem análises aos processos de partilha de trabalho e terminação.

Capítulo 7

Análise de Resultados

Este capítulo é dedicado à análise da performance e custos do sistema YapDss. Para isso são analisados resultados de um conjunto de programas de teste. Todos os testes apresentados foram obtidos no cluster do NCC (Núcleo de Ciência de Computadores), composto por um conjunto de 4 máquinas com duplo processador Pentium-II com uma velocidade de relógio de 350 Mhz e256 Mbytes de memória RAM cada.

7.1 Programas de Teste

O conjunto de programas usadas no estudo da análise da performance do sistema YapDss são os habitualmente utilizados para o efeito. O conjunto escolhido não tem predicados de corte, dado que o YapDss não tem suporte. Os programas são os seguintes:

magic um programa para resolver o cubo mágico de Rubik.

nsort algoritmo para o ordenamento de uma lista com 11 elementos através do uso de força bruta começando com o caso da ordem inversa.

queens12 um programa que usa a geração e teste como forma de resolução do problema da colocação de 12 rainhas num tabuleiro de xadrez.

puzzle4x4 algoritmo que resolve um puzzle da disposição de 16 quadrados, onde 1 dos quadrados é um espaço vazio.

cubes7 um programa para empilhar 7 cubos coloridos de forma a que num dado lado não aparecem cores repetidas.

ham algoritmo para a descoberta de todos os ciclos hamiltonianos num grafo com 26 nós, com cada nó ligado a 3 outros nós.

nrevwarren algoritmo para obtenção da lista inversa através de uma aproximação *naive*.

Todos os programas de teste descobrem todas as soluções possíveis. As várias soluções são conseguidas através da simulação automática de uma falha sempre que se consegue uma resposta válida. O controlo do tempo está embebido no YapDss, tendo como início de contagem o momento imediatamente após a distribuição do código do programa pelos vários agentes do sistema, e o fim da contagem dá-se logo após o fim da terminação da computação, ou seja, antes da recolha das soluções dos vários agentes.

7.2 Análise de Performance

A análise da performance começou por avaliar o custo que acarreta o ambiente distribuído do YapDss em relação ao sistema sequencial Yap. De seguida, comparamos o sistema YapDss com o sistema YapOr, que tendo aproximações completamente diferentes ao problema da execução paralela do Prolog, tem pontos de convergência importantes como a cópia incremental.

7.2.1 Custo do Modelo Distribuído

A tabela 7.1 apresenta os tempos de execução (em segundos) de todos os programas de teste, quando executados no sistema Yap e no sistema YapDss com 1 agente. Os resultados apresentados correspondem aos menores tempos obtidos num conjunto de 10 execuções por programa.

Programas de teste	**Sistema**		**YapDss/Yap**
	Yap	**YapDss**	
magic	29.310	30.904	1.05
nsort	188.500	218.688	1.16
queens12	65.031	72.802	1.12
puzzle4x4	54.610	67.914	1.24
cubes7	1.260	1.318	1.05
ham	0.230	0.305	1.32
nrevwarren	3.070	3.143	1.02
$\sum$	342.011	395.074	1.16
Média			1.14

Tabela 7.1: Comparação de performance entre o sistema YapDss e o sistema Yap.

A última coluna da tabela 7.1 mostra a proporção dos tempos obtidos nos dois sistemas, a que corresponde o custo do modelo distribuído, isto é, o custo da adaptação do

modelo sequencial Yap para o modelo distribuído YapDss. 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 YapDss. Para tal, utilizou-se um esquema idêntico de contagem do tempo de execução e para procura de todas as soluções possíveis. Ambos os sistemas foram compilados usando o mesmo compilador e com as mesmas opções de compilação.

Pela análise da tabela 7.1 podemos concluir que o sistema YapDss é em média 14% mais lento do que o sistema sequencial Yap com um agente. O atraso do modelo YapDss em relação ao modelo sequencial Yap, tem as suas razões na complexidade do controlo numa arquitectura distribuída e na consequente necessidade de criação de mecanismos de controlo para garantir a correcta funcionalidade do modelo distribuído. O custo associado a este valor pode ser dividido em vários aspectos:

Actualização do valor real de carga A manutenção do valor real da carga do sistema tem bastante peso computacional, sendo um dos factores que mais contribui para o atraso do YapDss em relação ao modelo sequencial. Contudo esta correcta manutenção da carga é necessária para assegurar um bom balanceamento do sistema, que de outra forma tornaria o processo de partilha de trabalho bastante ineficaz, resultando numa baixa geral da performance do sistema.

Actualização da etiqueta de execução A actualização da etiqueta da execução é outra peça fundamental do sistema, que garante a correcta semântica do Prolog. Sem a correcta actualização da etiqueta, não seria possível existir a cópia de ambiente incremental, dada a impossibilidade de saber com precisão a relação hierárquica entre os agentes envolvidos no processo de partilha.

Verificação da recepção de mensagens O mecanismo de verificação de mensagens também introduz atraso na computação do sistema, mas é um processo fundamental para iniciar o processamento das mensagens enviadas pelos agentes do sistema.

Em seguida, vamos fazer uma análise ao modelo YapDss, desde os tempos de execução com vários agentes até as estatísticas gerais do modelo.

7.2.2 Execução Distribuída

A tabela 7.2 apresenta os tempos de execução dos programas de teste para 2,4,6 e 8 agentes.

As colunas apresentam os ganhos de velocidade[1] entre parêntesis em relação ao tempo de execução base (com 1 agente). Os tempos apresentado e respectivos ganhos de

[1]Na literatura inglesa, este ganho de velocidade tem como tradução habitual *'speedup'*

velocidade correspondem aos menores tempos obtidos num conjunto de 10 execuções.

Existem três níveis de paralelismo patente nos resultados, divididos em três blocos de programas de teste de acordo com o grau de granularidade. Existe um subconjunto de programas de teste com um *alto nível* de paralelismo, este primeiro bloco tem como constituintes os seguintes programas: 'queens12'; 'magic'; 'nsort'; 'puzzle4x4'. De seguida podemos ver a presença de um subconjunto com uma *granularidade média*, fazendo parte deste segundo bloco, o 'cubes' e o 'ham'. Por último, temos o subconjunto de *baixa granularidade* que forma o terceiro e último bloco da tabela 7.2, contendo o 'qsort'(programa com bastante paralelismo-E e pouco paralelismo-Ou) e o 'nrevwarren'. No fim de cada bloco e na última linha da tabela encontra-se o somatório de cada bloco e o somatório de todos os tempos.

Programas de teste	**Número de Agentes**			
	2	**4**	**6**	**8**
magic	15.500(1.99)	7.885(3.92)	5.588(5.53)	4.380(7.05)
nsort	124.244(1.98)	63.149(3.90)	42.445(5.80)	33.067(7.45)
queens12	38.936(1.99)	19.638(3.94)	13.364(5.80)	10.123(7.66)
puzzle4x4	34.005(1.99)	17.345(3.91)	11.830(5.73)	9.417(7.20)
$\sum$	212,685(1.98)	107,987(3.91)	73,227(5.77)	56,987(7.41)
cubes7	0.671(1.96)	0.404(3.26)	0.338(3.90)	0.237(4.80)
ham	0.174(1.75)	0.108(2.81)	0.097(3.13)	0.103(2.95)
$\sum$	0,845(1.91)	0,512(3,17)	0,435(3,73)	0,376(4.31)
nrevwarren	3.820(0.95)	4.118(0.88)	4.141(0.88)	4.315(0.85)

Tabela 7.2: Performance mediante o número de agentes no YapDss.

Quanto maior for o número de agentes no sistema, maior é a latência do sistema, principalmente durante o processo de terminação da computação, porque é necessário haver um maior sincronismo entre todos os agentes do sistema.

Em seguida, vamos fazer um breve análise entre o modelo paralelo Yapor e o YapDss, de modo a fazer um comparativo entre ambas as aproximações.

7.2.3 Comparação entre o Modelo Distribuído e o Modelo Paralelo

A tabela 7.3 apresentas os tempos de execução (em segundos). De todos os programas no sistema YapDss e YapOr, usando 2 agentes para garantir *justiça* na comparação, porque cada nó do cluster tem somente 2 processadores, logo ficamos restringidos a testes com 2 agentes em ambos os sistemas.

A última coluna mostra a proporção entre os tempos obtidos nos dois sistemas, reflectindo o custo do processo distribuído em relação ao processo paralelo.

Programas de teste	**Sistema**		**YapDss/Yapor**
	YapDss	**YapOr**	
queens12	38.936	34.457	1.13
magic	15.500	15.272	1.02
nsort	124.244	107.704	1.15
puzzle4x4	34.005	28.722	1.18
cubes7	0.671	0.667	1.01
ham	0.174	0.127	1.37
nrevwarren	3.630	3.090	1.17
$\sum$	217.16	190.039	1.14
Média			1.15

Tabela 7.3: Comparação de performance entre YapDss e YapOr.

O esquema utilizado no YapDss para medir tempo, é o mesmo mecanismo usado no YapOr. As únicas diferenças estão relacionados com o inicio de contagem do tempo de execução. No caso do YapDss essa contagem começa após ter sido enviado (por parte do agente 0) e copiado o código do programa a executar pelos agentes do sistema. A outra diferença prende-se com o fim da contagem do tempo de execução, no YapDss esta contagem termina quando acontece a terminação da computação, ficando deste modo de fora o tempo necessário para a concentração das soluções do sistema no agente 0. De outra forma não seria possível fazer uma comparação justa com o YapOr, dada a sua natureza paralela não implicar a gestão das soluções.

Podemos concluir que o YapDss é em **média 15% mais lento** que o YapOr, no mesmo cenário. No cenário de teste foi utilizado apenas uma máquina do *cluster*. O factor fundamental para a penalização da performance do YapDss está intrinsecamente ligado à sua natureza distribuída, que ao invés da comunicação se efectuar por um mecanismo rápido como a memória partilhada do YapOr, é realizado usando mensagens de rede com as latências a ela associada. Logo a performance do YapDss está directamente associada à performance da rede que o sustenta. Dado que o Yapor é mais rápido que o YapDss com um agente, logo também o é para 2 agentes.

O grau de granularidade de qualquer programa de teste tem um impacto profundo na performance do YapDss, isto porque se um programa tem baixa granularidade, o número de mensagens no sistema sobe, dado a procura dos agentes por trabalho, implicando um aumento significativo do tempo de computação necessário para o processamento destas mensagens. Este tempo é por consequente desviado do processamento da máquina abstracta para o processamento das mensagens de controlo. Outro factor preponderante está relacionado com o tempo de execução de um dado

programa de teste, que caso seja baixo leva à insuficiência de trabalho no sistema para cobrir os custos associados ao modelo distribuído, entre eles estão: latência no envio das mensagens, actualização das estruturas de apoio (como a etiqueta de execução); e processamento de mensagens. Este facto leva a uma degradação da performance do sistema.

Para termos uma noção mais exacta da problemática inerente à dialéctica entre controlo do sistema versus quantidade de trabalho partilhado, vamos na próxima secção fazer uma dissecação das tarefas com maior peso no YapDss.

7.3 Actividades do Modelo Distribuído

O tempo de execução de um programa corresponde à soma dos tempos parciais das várias actividades do modelo. Estas actividades influenciam com diferentes pesos a performance final do sistema. Nesta secção pretende-se analisar e explicar os tempos obtidos em cada uma das actividades.

O tempo de execução de um programa está dividido em distintas actividades, cada uma delas reflectindo os vários processos do modelo distribuído, desde a execução da máquina abstracta WAM, passando pelo tempo despendido na procura e partilha de trabalho. Segue-se uma descrição das actividades no modelo distribuído que iremos considerar:

Prolog tempo despendido na execução de Prolog, na actualização do registo de carga e da etiqueta.

Procura tempo despendido na procura de um agente ocupado com excesso de carga para uma possível partilha de trabalho, bem como o tempo gasto nas partilhas de trabalhos (do ponto de vista do agente Q no esquema de partilha de trabalho, ou seja do agente que recebe trabalho) e no processo de terminação.

Partilha tempo despendido na partilha activa de trabalho, que engloba as quatro fases do processo de partilha.

Com vista a uma melhor enquadramento dos resultados obtidos, procedeu-se ainda a um segundo nível de análise de performance, que consiste na verificação do número de ocorrências das tarefas mais importantes do modelo distribuído. Estas tarefas são as seguintes:

Procura Trabalho(A) Quantidade de pedidos de trabalho **enviados** e aceites

Procura Trabalho(R) Quantidade de pedidos de trabalho **enviados** e rejeitados

Carga Recebida Número de alternativas recebidas nos vários pedidos de trabalho enviados

Tokens Emitidos Número de Tokens enviados, de modo a iniciar o processo de terminação da computação

Token Recebidos Número de Tokens processados, ou seja, que foram enviados por outros agentes para iniciar a terminação da computação

Soluções Número de soluções encontradas

Todas as tarefas acima mencionadas tem um papel fulcral no YapDss. Contudo a sua análise carece de uma especial atenção dada a estrita ligação entre todas elas. Os pedidos e procura de trabalho reflectem a intensidade na procura e partilha de trabalho entre os vários agentes. A carga recebida quantifica nas sucessivas partilhas de trabalho, o número de alternativas inexploradas recebidas nos vários pontos de escolha das várias sub-árvores de procura partilhadas. O número de tokens, tanto emitidos como recebidos, demonstram o maior ou menor grau de granularidade do programa em teste, sendo uma indicação para o número de agentes que estiveram sem trabalho durante bastante tempo. O aspecto chave para a avaliação do balanceamento do sistema está com o tempo passado na execução da máquina abstracta. O número de soluções tem um valor meramente informativo.

Nas tabelas 7.4, 7.5 e 7.6 temos um exemplo de um programa de alta granularidade, o *queens12*, seguido pelo *ham*, um programa de granularidade média, e por fim temos o *nrevwarren* que é um programa de granularidade baixa. Estas tabelas mostram o efeito do consequente aumento do número de agentes no sistema, começando com 2 agentes na tabela 7.4, passando para a tabela 7.5 que já apresenta os resultados da execução com 4 agentes, finalizando-se com apresentação da tabela 7.6 com o total de 8 agentes.

A introdução dos mecanismos que possibilitam este nível de análise introduziram alguma latência na execução dos programas, mas como o foco da análise é a quantificação das tarefas do sistema, este custo adicional não deverá introduzir alterações significativas no padrão de execução dos programas de teste.

Na tabela 7.4 podemos observar o comportamento do YapDss com dois agentes em execução. No programa de teste 'queens12' podemos constatar um bom balanceamento da carga, apoiado pelo facto dos tempos de execução (*Prolog*) serem muito aproximados, revelando que ambos os agentes despenderam igual tempo na computação da máquina abstracta WAM. O distribuidor de trabalho revela um bom comportamento como podemos constatar pelo pouco tempo despendido na procura e partilha de trabalho. No programa de teste 'ham', que apesar de ser de granularidade média, apresenta um bom comportamento com os tempos de computação da máquina abstracta bastante aproximados. No último programa de teste, o nrevwarren apresenta os diversos sinais de um programa com pouco tempo de execução e de pouca granularidade, isto porque o número de pedidos de trabalho por parte de ambos agentes são bastante altos, revelando sucessivas tentativas de conseguir trabalho, que quando bem sucedidas contêm pouca granularidade, tendo como implicação directa um aumento do número de *tokens* de terminação no sistema.

Tarefas	Agente 0	1	∑	Média
queens12				
Prolog	39.468	39.507	78.975	39.5
Procura	0.046	0.007	0.053	0.1
Partilha	0.106	0.106	0.212	0.1
Procura Trabalho(A)	7	1	8	4.0
Procura Trabalho(R)	2	1	3	1.5
Carga Recebida	264	32	296	148.0
Tokens Recebidos	5	10	15	7.5
Tokens Emitidos	9	1	10	5.0
Soluções	7098	7102	14200	7100.0
ham				
Prolog	0.156	0.147	0.303	0.2
Procura	0.023	0.033	0.263	0.1
Partilha	0.002	0.001	0.003	0.1
Procura Trabalho(A)	2	4	6	3.0
Procura Trabalho(R)	1	2	3	1.5
Carga Recebida	16	48	64	32.0
Tokens Recebidos	7	4	11	5.5
Tokens Emitidos	1	5	6	3.0
Soluções	27	33	60	30.0
nrevwarren				
Prolog	0.056	3.165	3.221	1.6
Procura	3.513	0.210	3.723	1.9
Partilha	0.126	0.315	0.441	0.2
Procura Trabalho(A)	41	1	42	21.0
Procura Trabalho(R)	126	5	131	65.5
Carga Recebida	1	1	2	1.0
Tokens Recebidos	171	172	343	171.5
Tokens Emitidos	167	5	172	86.0
Soluções	1	0	1	0.5

Tabela 7.4: Tempo despendido nas várias tarefas do modelo distribuído com 2 agentes.

Na tabela 7.5 apresenta a quantificação das várias tarefas com 4 agentes em execução. Seguindo na mesma linha que na tabela anterior, o programa de teste 'queens12' revela um bom balanceamento de carga. As diferenças presentes na passagem de um sistema com 2 agentes para 4 agentes, estão ligadas com a estratégia utilizada no distribuidor de trabalho, visto que a orientação principal é de tentar preservar o topo da árvore de procura, levando a que os agente apresentem um crescimento linear no pedido de trabalho rejeitado (*Procura Trabalho(R)*), dado que cada agente tenta preservar o topo da sua respectiva árvore de procura implicando que certos agentes operam num nível mais profundo da árvore de procura, onde a granularidade decresce. O programa de teste 'ham' revela os efeitos do seu nível de granularidade médio, onde é visível pelo registo de Procura de Trabalho(R), um nível pouco elevado de granularidade, logo torna-se difícil aproveitar todo o potencial computacional dos vários agentes do sistema. O nível de balanceamento é bom, contudo já não sendo tão uniforme como na tabela 7.4, dado o maior custo associado ao crescente número de agentes e da estratégia usada pelo distribuidor de trabalho. Por fim, temos o 'nrevwarren' que revela os mesmos problemas que na tabela anterior, agravados pelo aumento provocado pela introdução de mais 2 agentes no sistema.

Como podemos constatar, o aumento sucessivo do número de agentes permite compreender quais os aspectos que são mais determinantes na performance do sistema.

Tarefas	Agente 0	1	2	3	∑	Média
queens12						
Prolog	19.858	19.786	19.708	19.629	78.981	19.7
Procura	0.096	0.170	0.251	0.330	0.847	0.2
Partilha	0.089	0.086	0.085	0.1	0.344	0.1
Procura Trabalho(A)	10	18	34	37	99	24.8
Procura Trabalho(R)	60	126	16	6	208	52.0
Carga Recebida	361	622	1261	1324	3568	892.0
Tokens Recebidos	44	19	37	62	162	40.5
Tokens Emitidos	11	15	33	34	93	23.3
Soluções	2880	4192	3436	3692	14200	3550.0
ham						
Prolog	0.087	0.111	0.060	0.052	0.310	0.1
Procura	0.055	0.030	0.083	0.090	0.258	0.1
Partilha	0.004	0.003	0.001	0.002	0.010	0.1
Procura Trabalho(A)	5	2	5	3	15	3.8
Procura Trabalho(R)	30	11	41	5	87	21.8
Carga Recebida	39	16	37	43	135	33.8
Tokens Recebidos	12	7	10	14	43	10.8
Tokens Emitidos	4	2	2	4	12	3.0
Soluções	17	23	11	9	60	15.0
nrevwarren						
Prolog	3.258	0.000	0.000	0.000	3.258	0.8
Procura	0.017	3.911	3.907	3.908	11.743	2.9
Partilha	0.638	0.000	0.000	0.000	0.639	0.2
Procura Trabalho(A)	0	27	27	10	64	16.0
Procura Trabalho(R)	3	846	84	35	968	242.0
Carga Recebida	0	0	1	0	1	0.3
Tokens Recebidos	83	90	82	144	399	99.8
Tokens Emitidos	2	27	44	11	84	21.0
Soluções	0	0	0	1	1	0.3

Tabela 7.5: Tempo despendido nas várias tarefas do modelo distribuído com 4 agentes.

Com um conjunto de 8 agentes, a tabela 7.6 mostra todas as tendências anteriormente apontadas nos diversos programas de teste. No caso do 'queens12', podemos verificar um bom balanceamento, onde se continua a verificar tempos de execução da máquina abstracta muito aproximados. Na passagem para o nível de granularidade média, o 'ham' continua a denotar a sua granularidade com os custos do modelo distribuído a aumentar, podemos verificar este facto pelos tempos de procura e partilha de trabalho, que já apresentam um maior peso no sistema. Por último, no programa de teste 'nrevwarren' verifica-se a insuficiência de trabalho no sistema, para ocupar mais do que 2 agentes. Esta situação provoca que os restantes agentes do sistema estejam a criar latência na performance do sistema, visto que apesar de não estarem a contribuir para a execução do programa de teste continuam a enviar mensagens numa tentativa de conseguir trabalho provocando ainda um maior atraso.

Nos vários cenários apresentados existem múltiplas relações entre várias das actividades consideradas. O registo dado pelo *Workload* não tem uma relação directa com o número de soluções finais de cada agente porque o *Workload* reflecte o número de alternativas inexploradas recebidas por um dado agente, que podem ou não conter soluções. As actividades de *Procura* e *Partilha* tem um forte elo de ligação, porque reflectem o esforço despendido na partilha de trabalho, estando directamente relacionado com o grau de granularidade do programa de teste.

Os programas de alta granularidade apresentam um bom balanceamento de carga

Tarefas	Agente 0	1	2	3	4	5	6	7	∑	Média
queens12										
Prolog	9.736	9.770	9.528	9.559	10.070	9.892	10.295	10.307	79.157	9.9
Procura	0.800	0.773	1.018	0.990	0.521	0.654	0.234	0.232	5.222	0.6
Partilha	0.074	0.063	0.060	0.057	0.068	0.061	0.077	0.067	0.527	0.1
Procura Trabalho(A)	9	14	39	37	3	25	17	17	160	20.0
Procura Trabalho(R)	1556	1181	1042	957	1351	532	3	8	6630	828.8
Carga Recebida	253	484	1341	1296	121	881	618	598	5592	699.0
Tokens Recebidos	90	120	141	156	138	107	71	38	861	107.6
Tokens Emitidos	62	9	24	26	3	17	14	14	169	21.2
Soluções	1631	1747	1876	1773	1629	1960	1820	1764	14200	1775.0
ham										
Prolog	0.085	0.123	0.045	0.003	0.0179	0.004	0.001	0.039	0.318	0.1
Procura	0.069	0.028	0.110	0.116	0.138	0.152	0.155	0.153	0.921	0.1
Partilha	0.004	0.005	0.001	0.001	0.001	0.000	0.000	0.000	0.011	0.1
Procura Trabalho(A)	3	4	2	3	2	1	1	2	18	2.3
Procura Trabalho(R)	18	9	7	36	68	69	65	74	346	43.4
Carga Recebida	36	41	19	38	14	11	14	22	195	24.4
Tokens Recebidos	22	8	7	13	21	27	31	33	162	20.3
Tokens Emitidos	4	1	3	4	3	2	2	2	21	2.6
Soluções	16	26	6	3	1	0	2	6	60	7.5
nrevwarren										
Prolog	3.294	0.003	0.001	0.001	0.001	0.001	0.001	31.414	34.716	4.3
Procura	0.053	4.410	4.401	4.401	4.401	4.403	4.401	0.888	27.358	3.4
Partilha	1.061	0.000	0.000	0.000	0.000	0.000	0.000	0.164	1.223	0.2
Procura Trabalho(A)	0	0	11	11	11	11	11	1	56	7.0
Procura Trabalho(R)	2	2	41	41	41	41	39	7	214	26.8
Carga Recebida	0	1	0	0	0	0	0	0	1	0.1
Tokens Recebidos	0	50	18	34	50	66	82	98	398	49.8
Tokens Emitidos	54	10	8	8	1	8	8	8	105	13.1
Soluções	1	0	0	0	0	0	0	0	1	0.1

Tabela 7.6: Tempo despendido nas várias tarefas do modelo distribuído com 8 agentes.

que está patente pela uniformidade dos tempos de computação da máquina abstracta (dado pelo actividade *Prolog*). O distribuidor de trabalho apresenta um comportamento bastante eficiente, dado o baixo tempo despendido na actividade de *Procura* apresentado pelos vários agentes do sistema.

No subconjunto dos programas de granularidade média já nos deparamos com diversos indícios de problemas no bom balanceamento da carga: o tempo despendido no processamento da máquina abstracta (actividade *Prolog)* já não é tão uniforme como nos programas de alta granularidade dado o menor nível de trabalho disponível a cada momento, os agentes tentam evitar o excesso de partilha. Caso contrário, os benefícios resultantes da partilha de trabalho seriam rapidamente suplantados pelo crescente controlo necessário; o tempo despendido na actividade de *Procura* já tem um peso muito superior, suplantando em alguns agentes o tempo usado para a computação da máquina abstracta, dada a impossibilidade de garantir trabalho para todos os agentes em cada instante.

O comportamento do programa de teste *nrevwarren* reflecte o baixo nível de granularidade e o baixo tempo de execução. O problema central à baixa performance neste programa de teste está relacionado com o baixo nível de alternativas por ponto de escolha conjugado com o pouco tempo de execução. Este facto tem implicações profundas no processo de partilha de trabalho nos seguintes pontos:

- A quantidade de trabalho partilhada no processo de partilha é muito baixa, originando que a maior parte dos agentes esteja muito tempo inactivo. A inactividade dos agentes provoca um aumento do número de mensagens de controlo numa tentativa de balancear a carga.

- Quando um agente tem pouco trabalho tenta preservar os pontos de escolha de topo da sua árvore de computação, de modo a maximizar os futuros processos de partilha, mas num sistema de baixa granularidade estamos postos perante o dilema de dar prioridade à computação da *WAM* ou de realizar processos de partilha com pouca quantidade de trabalho, não havendo solução perfeita.

- Devido ao pouco tempo de execução, a maior parte dos agentes do sistema nem chegam a ter trabalho dado o pouco trabalho existente no sistema, contudo devido às sucessivas tentativas de requisitarem trabalho leva a uma deterioração da performance do sistema.

Após a análise das tabelas apresentadas, podemos concluir que o YapDss tem um elevado desempenho no maior parte dos programas de teste, estando dependente como todos os sistema de execução paralela, do grau de granularidade dos programas executados. Em programas de alta granularidade podemos constatar ganhos quase lineares, mesmo com a introdução de um número crescente de agentes no sistema.

Capítulo 8

Conclusões e Trabalho Futuro

A presente tese contém o trabalho desenvolvido no estudo, no desenho, na implementação e na avaliação de desempenho do YapDss, um sistema de execução distribuída de Prolog que explora Paralelismo-Ou implícito a partir da plataforma Yap de execução sequencial. O YapDss utiliza o modelo de cópia de ambientes, e a distribuição estática de trabalho através do uso do modelo de *Distributed Stack Splitting*, utilizando também conceitos introduzidos pelos sistemas Yapor e Muse e tem como principal objectivo a obtenção de elevado desempenho na execução de Prolog num ambiente distribuído.

8.1 Conclusões

Para a realização desta tese foi necessário a dedicação de bastante cuidado e atenção aos seguintes aspectos:

- A reorganização da memória para melhor responder às necessidades do YapDss.
- O desenho das estruturas de suporte ao processo distribuído.
- A implementação da *Incremental Stack Splitting* usando etiquetas de execução para a descoberta do primeiro nó comum entre 2 agentes.
- A implementação do protocolo de comunicação.
- A implementação do processo de terminação.
- A implementação das estratégias do distribuidor de trabalho.
- A implementação do *interface* entre o distribuidor e o emulador de instruções do YAP.
- A implementação do processo de partilha de trabalho.

- A adaptação de algumas instruções WAM para permitir o acesso sequencial entre alternativas de um ponto de escolha sem a necessidade de interpretação do tipo de instrução
- A modificação do processo de indexação do Yap para ocorrer antes do início do processo distribuído.

O YapDss usa alguns mecanismos do Yapor. Contudo, difere na utilização dos mesmos, servindo apenas como suporte ao modelo distribuído. O conjunto desses aspectos é enumerado de seguida.

- Organização de memória
- Diferente implementação do processo normal de retrocesso pelas instruções **fail**, **retry_me** e **try_me** entre o YAP e as descrições de referência.
- Cálculo das áreas (das pilhas de execução) a copiar no processo de partilha.
- Esquema de suporte à exploração contínua de todas as soluções de um objectivo.

Uma das maiores dificuldades encontradas está relacionada com a percepção do código envolvido na implementação do Yap, devido em muito à obscuridade de certas partes de código relacionado com as optimizações complexas e pouco perceptíveis, que exigiram um esforço redobrado na elaboração deste trabalho.

O custo de manutenção do modelo distribuído do YapDss é relativamente baixo, cerca de 14% na máquina do cluster utilizada. O custo associado a este valor pode ser dissecado em vários componentes: manutenção do registo actualizado de carga; manutenção da etiqueta de manutenção; e latência no envio das mensagens por rede.

8.2 Trabalho Futuro

Existem dois pontos principais de trabalho futuro para o sistema YapDss. O primeiro referente à construção de um mecanismo de terminação da computação mais escalável, capaz de suportar de uma maneira eficaz um número crescente de agentes no sistema. Uma maneira de alcançar isso é através da hierarquização dos agentes em grupos. O outro ponto é referente à implementação do *cut*, permitindo deste modo o alargamento do conjunto de programas, que podem ser corridos no sistema YapDss.

Para aumentar a escalabilidade do sistema, é necessário a modificação da estrutura actual de anel lógico para uma organização hierarquizada, caso contrário com um número elevado de agentes no sistema, o processo de terminação da computação é muito ineficaz.

Um possível aproximação seria através da utilização de uma árvore bem balanceada em que cada nó controla a terminação dos seus filhos. A ideia geral seria:

- Um agente inicia o processo de computação enviando um token para cada um dos seus filhos, caso existam, de outra forma envia um token para o seu pai.
- Quando um agente recebe um token de um filho, envia tokens para os seus restantes filhos. Caso todos os filhos estejam suspensos, o agente envia o token para o seu pai (ver subfigura 2 da figura 8.1), se algum dos filhos estiver activo, o agente retorna o token com o estado 'sujo' para o filho que tinha enviado o token em primeiro lugar, abortando desta forma o processo (ver subfigura 3 da figura 8.1).
- Sempre que um agente recebe um token do pai, envia o token para todos os restantes filhos. Quando um agente recebe os tokens de todos os seus filhos, envia para o seu pai o estado da computação dos seus descendentes, ou seja um token 'sujo' ou 'limpo', caso exista ou não computação.

O principal objectivo desta estratégia é evitar congestionamentos no sistema, para isso a verificação começa por ser feita na sub-árvore do agente que inicia o processo de terminação , caso todos os agentes da sub-árvore estejam suspensos, o processo estende-se para o resto da árvore.

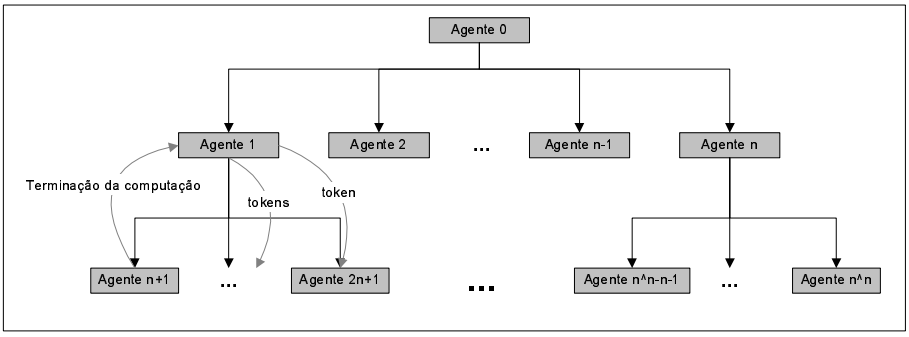

Fig1. Início da terminação

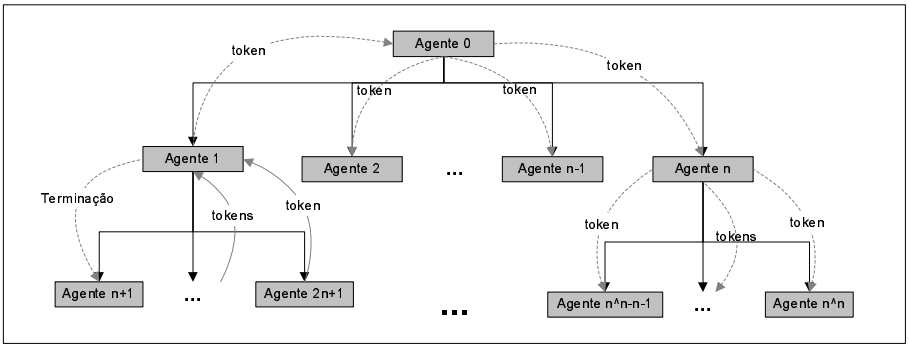

Fig2. Verificação terminação no resto do sistema

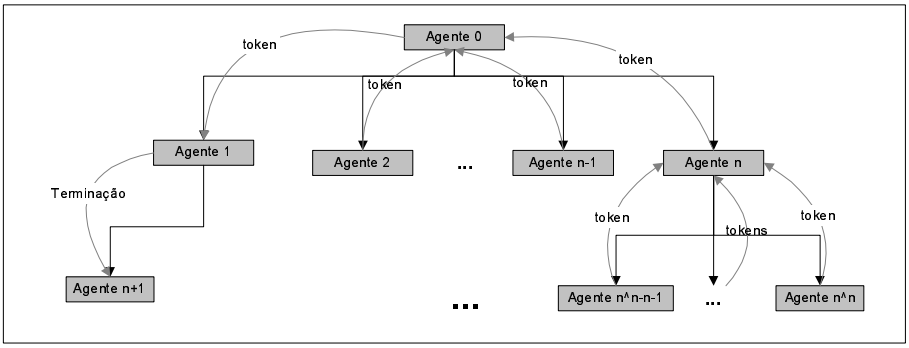

Fig3. Conclusão da terminação

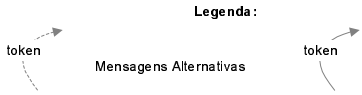

Figura 8.1: Processo proposto para a terminação da computação.

Apêndice A

A.1 Código dos Programas de Teste

Queens12

```
queens12(S) :- queens12_get_solutions(12,S).

queens12_get_solutions(Board_size, Soln) :-
  queens12_solve(Boars_size, [], Soln).

queens12_newsquare([square(I,J)|Rest],square(X,Y)) :-
  X is I+1,
  queens12_snint(Y),
  queens12_not_threatened(I,J,X,Y),
  queens12_safe(X,Y,Rest).
queens12_newsquare([],square(1,X)) :-
  queens12_snint(X).
queens12_safe(X,Y,[square(I,J)|L]) :-
  queens12_not_threatened(I,J,X,Y),
  queens12_safe(X,Y,L).
queens12_safe(X,Y,[]).

queens12_not_threatened(I,J,X,Y) :-
  I =\= X,
  J =\= Y,
  I-J =\= X-Y,
  I+J =\= X+Y.
```

```
queens12_solve(Board_size,Initial,Final) :-
  queens12_newsquare(Initial,Next),
  queens12_solve(Board_size,[Next|Initial],Final).

queens12_solve(Bs,[square(Bs,Y)|L],[square(Bs,Y)|L]) :-
  queens12_size(Bs).

queens12_size(12).
queens12_snint(1).
queens12_snint(2).
queens12_snint(3).
queens12_snint(4).
queens12_snint(5).
queens12_snint(6).
queens12_snint(7).
queens12_snint(8).
queens12_snint(9).
queens12_snint(10).
queens12_snint(11).
queens12_snint(12).
```

Puzzle4x4

```
puzzle4x4(X) :- pz4x4_go8(X).

pz4x4_go8(S):-
    pz4x4_problem8(Y), pz4x4_solve(8,Y,S).
pz4x4_go9(S):-
    pz4x4_problem9(Y), pz4x4_solve(9,Y,S).
pz4x4_go10(S):-
    pz4x4_problem10(Y), pz4x4_solve(10,Y,S).
pz4x4_go11(S):-
    pz4x4_problem11(Y), pz4x4_solve(11,Y,S).
pz4x4_go12(S):-
    pz4x4_problem12(Y), pz4x4_solve(12,Y,S).

pz4x4_problem8([ 1, 2, 3, 8,
          5, 6,11, 7,
          9, o,10,16,
         13,14,12,15]).

pz4x4_problem9([ 1, 2, 3, 8,
          5, 6,11, 7,
          o, 9,10,16,
```

```
pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,  o,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m24|L]):-
     pz4x4_movimento(N,N1),
     pz4x4_solve(N1,[P11,P12,P13,P14,
                     P21,P22,P24,  o,
                     P31,P32,P33,P34,
                     P41,P42,P43,P44],L).

pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,  o,
                P41,P42,P43,P44],[m24|L]):-
     pz4x4_movimento(N,N1),
     pz4x4_solve(N1,[P11,P12,P13,P14,
                     P21,P22,P23,  o,
                     P31,P32,P33,P24,
                     P41,P42,P43,P44],L).

pz4x4_solve(N, [P11,P12,P13,P14,
```

```
         13,14,12,15]).

pz4x4_problem10([ 1, 2, 3, 8,
           5, 6,11, 7,
          13, 9,10,16,
           o,14,12,15]).

pz4x4_problem11([ o, 2, 3, 8,
           1, 6,11, 7,
           5, 9,10,16,
          13,14,12,15]).

pz4x4_problem12([ 2, o, 3, 8,
           1, 6,11, 7,
           5, 9,10,16,
          13,14,12,15]).

pz4x4_movimento(1,0).
pz4x4_movimento(2,1).
pz4x4_movimento(3,2).
pz4x4_movimento(4,3).
pz4x4_movimento(5,4).
pz4x4_movimento(6,5).
pz4x4_movimento(7,6).
pz4x4_movimento(8,7).
pz4x4_movimento(9,8).
pz4x4_movimento(10,9).
pz4x4_movimento(11,10).
pz4x4_movimento(12,11).

pz4x4_solve(_,[ 1, 2, 3, o,
          5, 6, 7, 8,
          9,10,11,12,
         13,14,15,16],[]).

pz4x4_solve(N, [P11,  o,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m11|L]):-
    pz4x4_movimento(N,N1),

pz4x4_solve(N1,[  o,P11,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).


pz4x4_solve(N, [P11,P12,P13,P14,
                  o,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m11|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[  o,P12,P13,P14,
                P11,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).


pz4x4_solve(N, [  o,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m12|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P12,  o,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).


pz4x4_solve(N, [P11,P12,  o,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m12|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,  o,P12,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).
                  o,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m31|L]):-

      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                      P31,P22,P23,P24,
                        o,P32,P33,P34,
                      P41,P42,P43,P44],L).

pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,  o,P33,P34,
                P41,P42,P43,P44],[m31|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                      P21,P22,P23,P24,
                        o,P31,P33,P34,
                      P41,P42,P43,P44],L).

pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                  o,P42,P43,P44],[m31|L]):-
       pz4x4_movimento(N,N1),
       pz4x4_solve(N1,[P11,P12,P13,P14,
                       P21,P22,P23,P24,
                         o,P32,P33,P34,
                       P31,P42,P43,P44],L).

pz4x4_solve(N, [P11,P12,P13,P14,
                P21,  o,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m32|L]):-
       pz4x4_movimento(N,N1),
       pz4x4_solve(N1,[P11,P12,P13,P14,
                       P21,P32,P23,P24,
                       P31,  o,P33,P34,
                       P41,P42,P43,P44],L).


pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,P23,P24,
                  o,P32,P33,P34,
                P41,P42,P43,P44],[m32|L]):-
       pz4x4_movimento(N,N1),
       pz4x4_solve(N1,[P11,P12,P13,P14,
                       P21,P22,P23,P24,
                       P32,  o,P33,P34,
                       P41,P42,P43,P44],L).

pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,  o,P34,
                P41,P42,P43,P44],[m32|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                      P21,P22,P23,P24,
                      P31,  o,P32,P34,
                      P41,P42,P43,P44],L).

pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,  o,P43,P44],[m32|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                      P21,P22,P23,P24,
                      P31,  o,P33,P34,
                      P41,P32,P43,P44],L).

pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,  o,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m33|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                      P21,P22,P33,P24,
                      P31,P32,  o,P34,
                      P41,P42,P43,P44],L).
```

```
pz4x4_solve(N, [P11,P12,P13,P14,
                P21,  o,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m12|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,  o,P13,P14,
                P21,P12,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).


pz4x4_solve(N, [P11,  o,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m13|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P13,  o,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).


pz4x4_solve(N, [P11,P12,P13,  o,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m13|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,  o,P13,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).


pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,  o,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m13|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,  o,P14,
                P21,P22,P13,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).


pz4x4_solve(N, [P11,P12,  o,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m14|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P14,  o,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).


pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,P23,  o,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m14|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,  o,
                P21,P22,P23,P14,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).


pz4x4_solve(N, [  o,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m21|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P21,P12,P13,P14,
                  o,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).


pz4x4_solve(N, [P11,P12,P13,P14,
                P21,  o,P23,P24,
                P31,P32,P33,P34,
```

```
pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,  o,P33,P34,
                P41,P42,P43,P44],[m33|L]):-
       pz4x4_movimento(N,N1),
       pz4x4_solve(N1,[P11,P12,P13,P14,
                       P21,P22,P23,P24,
                       P31,P33,  o,P34,
                       P41,P42,P43,P44],L).

pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,  o,
                P41,P42,P43,P44],[m33|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                      P21,P22,P23,P24,
                      P31,P32,  o,P33,
                      P41,P42,P43,P44],L).

pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,  o,P44],[m33|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                      P21,P22,P23,P24,
                      P31,P32, o,P34,
                      P41,P42,P33,P44],L).

pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,P23,  o,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m34|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                      P21,P22,P23,P34,
                      P31,P32,P33,  o,
                      P41,P42,P43,P44],L).

pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,  o,P34,
                P41,P42,P43,P44],[m34|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                      P21,P22,P23,P24,
                      P31,P32,P34,  o,
                      P41,P42,P43,P44],L).

pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,  o],[m34|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                      P21,P22,P23,P24,
                      P31,P32,P33,  o,
                      P41,P42,P43,P34],L).

pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,P23,P24,
                  o,P32,P33,P34,
                P41,P42,P43,P44],[m41|L]):-
       pz4x4_movimento(N,N1),
       pz4x4_solve(N1,[P11,P12,P13,P14,
                       P21,P22,P23,P24,
                       P41,P32,P33,P34,
                         o,P42,P43,P44],L).

pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,  o,P43,P44],[m41|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                      P21,P22,P23,P24,
                      P31,P32,P33,P34,
                        o,P41,P43,P44],L).
```

```
              P41,P42,P43,P44],[m21|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                 o,P21,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).


pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,P23,P24,
                  o,P32,P33,P34,
                P41,P42,P43,P44],[m21|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                  o,P22,P23,P24,
                P21,P32,P33,P34,
                P41,P42,P43,P44],L).


pz4x4_solve(N, [P11,  o,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m22|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P22,P13,P14,
                P21,  o,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).


pz4x4_solve(N, [P11,P12,P13,P14,
                  o,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m22|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P22,  o,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).


pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,  o,P33,P34,
                P41,P42,P43,P44],[m22|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,  o,P23,P24,
                P31,P22,P33,P34,
                P41,P42,P43,P44],L).


pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,  o,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m22|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                P21,  o,P22,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).


pz4x4_solve(N, [P11,P12,  o,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m23|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P23,P14,
                P21,P22,  o,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],L).


pz4x4_solve(N, [P11,P12,P13,P14,
                P21,  o,P23,P24,
                P31,P32,P33,P34,
                P41,P42,P43,P44],[m23|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
```

```
pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,  o,P33,P34,
                P41,P42,P43,P44],[m42|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                  P21,P22,P23,P24,
                  P31,P42,P33,P34,
                  P41,  o,P43,P44],L).

pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                  o,P42,P43,P44],[m42|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                  P21,P22,P23,P24,
                  P31,P32,P33,P34,
                  P42,  o,P43,P44],L).

pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,  o,P44],[m42|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                  P21,P22,P23,P24,
                  P31,P32,P33,P34,
                  P41,  o,P42,P44],L).

pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,  o,P34,
                P41,P42,P43,P44],[m43|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                  P21,P22,P23,P24,
                  P31,P32,P43,P34,
                  P41,P42,  o,P44],L).

pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,  o,P43,P44],[m43|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                  P21,P22,P23,P24,
                  P31,P32,P33,P34,
                  P41,P43,  o,P44],L).

pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34
                P41,P42,P43,  o],[m43|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                  P21,P22,P23,P24,
                  P31,P32,P33,P34,
                  P41,P42,  o,P43],L).

pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,  o,
                P41,P42,P43,P44],[m44|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                  P21,P22,P23,P24,
                  P31,P32,P33,P44,
                  P41,P42,P43,  o],L).

pz4x4_solve(N, [P11,P12,P13,P14,
                P21,P22,P23,P24,
                P31,P32,P33,P34,
                P41,P42,  o,P44],[m44|L]):-
      pz4x4_movimento(N,N1),
      pz4x4_solve(N1,[P11,P12,P13,P14,
                  P21,P22,P23,P24,
                  P31,P32,P33,P34,
                  P41,P42,P44,  o],L).
```

```
               P21,P23,  o,P24,
               P31,P32,P33,P34,
               P41,P42,P43,P44],L).


pz4x4_solve(N, [P11,P12,P13,P14,
               P21,P22,P23,P24,
               P31,P32,  o,P34,
               P41,P42,P43,P44],[m23|L]):-
     pz4x4_movimento(N,N1),
     pz4x4_solve(N1,[P11,P12,P13,P14,
               P21,P22,  o,P24,
               P31,P32,P23,P34,
               P41,P42,P43,P44],L).


pz4x4_solve(N, [P11,P12,P13,P14,
               P21,P22,P23,  o,
               P31,P32,P33,P34,
               P41,P42,P43,P44],[m23|L]):-
     pz4x4_movimento(N,N1),
     pz4x4_solve(N1,[P11,P12,P13,P14,
               P21,P22,  o,P23,
               P31,P32,P33,P34,
               P41,P42,P43,P44],L).
```

```
pz4x4_solve(N, [P11,P12,P13,  o,
               P21,P22,P23,P24,
               P31,P32,P33,P34,
               P41,P42,P43,P44],[m24|L]):-
     pz4x4_movimento(N,N1),
     pz4x4_solve(N1,[P11,P12,P13,P24,
                     P21,P22,P23,  o,
                     P31,P32,P33,P34,
                     P41,P42,P43,P44],L).
```

Magic

```
magic_go6(S):- magic_problem6(Y), magic_solve(6,Y,S).

magic_problem6([ [2,5,2,2,5,2,2,2,2],
                 [1,3,6,1,2,1,1,3,6],
                 [4,6,4,4,6,6,4,3,4],
                 [5,1,3,4,4,3,5,1,3],
                 [3,2,3,5,3,5,6,4,6],
                 [1,4,1,6,1,6,5,5,5] ]).

magic_movimento(1,0).
magic_movimento(2,1).
magic_movimento(3,2).
magic_movimento(4,3).
magic_movimento(5,4).
magic_movimento(6,5).
magic_movimento(7,6).

magic_solve(_,[ [1,1,1,1,1,1,1,1,1],
                [2,2,2,2,2,2,2,2,2],
                [3,3,3,3,3,3,3,3,3],
                [4,4,4,4,4,4,4,4,4],
                [5,5,5,5,5,5,5,5,5],
                [6,6,6,6,6,6,6,6,6] ],[]).

magic_solve(N,[ [L11,L12,L13,_,_,_,_,_,_],
                [L21,L22,L23,_,_,_,_,_,_],
                [L31,L32,L33,_,_,_,_,_,_],
                [L41,L42,L43,_,_,_,_,_,_],
                [L51,L52,L53,L54,_,L56,L57,L58,L59],
                [_,_,_,_,_,_,_,_,_] ], [dir1|L]):-
                magic_movimento(N,N1),
                magic_solve(N1,[ [L41,L42,L43,_,_,_,_,_,_],
                                 [L11,L12,L13,_,_,_,_,_,_],
                                 [L21,L22,L23,_,_,_,_,_,_],
                                 [L31,L32,L33,_,_,_,_,_,_],
                                 [L53,L56,L59,L52,_,L58,L51,L54,L57],
                                 [_,_,_,_,_,_,_,_,_] ],L).

magic_solve(N,[ [L11,L12,L13,L14,L15,L16,L17,L18,L19],
          [L21,L22,L23,L24,L25,L26,L27,L28,L29],
          [L31,L32,L33,L34,L35,L36,L37,L38,L39],
          [L41,L42,L43,L44,L45,L46,L47,L48,L49],
          [L51,L52,L53,L54,L55,L56,L57,L58,L59],
          [L61,L62,L63,L64,L65,L66,L67,L68,L69] ], [dir2|L]):-
     magic_movimento(N,N1),
     magic_solve(N1,[ [L11,L12,L13,L44,L45,L46,L17,L18,L19],
                [L21,L22,L23,L14,L15,L16,L27,L28,L29],
                [L31,L32,L33,L24,L25,L26,L37,L38,L39],
```

```
                [L41,L42,L43,L34,L35,L36,L47,L48,L49],
                [L53,L56,L59,L52,L55,L58,L51,L54,L57],
                [L61,L62,L63,L64,L65,L66,L67,L68,L69] ],L).

magic_solve(N,[ [L11,L12,L13,L14,L15,L16,L17,L18,L19],
          [L21,L22,L23,L24,L25,L26,L27,L28,L29],
          [L31,L32,L33,L34,L35,L36,L37,L38,L39],
          [L41,L42,L43,L44,L45,L46,L47,L48,L49],
          [L51,L52,L53,L54,L55,L56,L57,L58,L59],
          [L61,L62,L63,L64,L65,L66,L67,L68,L69] ], [dir3|L]):-
      magic_movimento(N,N1),
      magic_solve(N1,[ [L11,L12,L13,L14,L15,L16,L47,L48,L49],
        [L21,L22,L23,L24,L25,L26,L17,L18,L19],
                [L31,L32,L33,L34,L35,L36,L27,L28,L29],
                [L41,L42,L43,L44,L45,L46,L37,L38,L39],
                [L51,L52,L53,L54,L55,L56,L57,L58,L59],
                [L63,L66,L69,L62,L65,L68,L61,L64,L67] ],L).

magic_solve(N,[ [L11,L12,L13,L14,L15,L16,L17,L18,L19],
          [L21,L22,L23,L24,L25,L26,L27,L28,L29],
          [L31,L32,L33,L34,L35,L36,L37,L38,L39],
          [L41,L42,L43,L44,L45,L46,L47,L48,L49],
          [L51,L52,L53,L54,L55,L56,L57,L58,L59],
          [L61,L62,L63,L64,L65,L66,L67,L68,L69] ], [lat1|L]):-
      magic_movimento(N,N1),
      magic_solve(N1,[ [L67,L12,L13,L64,L15,L16,L61,L18,L19],
                [L21,L22,L23,L24,L25,L26,L27,L28,L29],
                [L31,L32,L57,L34,L35,L54,L37,L38,L51],
                [L43,L46,L49,L42,L45,L48,L41,L44,L47],
                [L11,L52,L53,L14,L55,L56,L17,L58,L59],
                [L33,L62,L63,L36,L65,L66,L39,L68,L69] ],L).

magic_solve(N,[ [L11,L12,L13,L14,L15,L16,L17,L18,L19],
          [L21,L22,L23,L24,L25,L26,L27,L28,L29],
          [L31,L32,L33,L34,L35,L36,L37,L38,L39],
          [L41,L42,L43,L44,L45,L46,L47,L48,L49],
          [L51,L52,L53,L54,L55,L56,L57,L58,L59],
          [L61,L62,L63,L64,L65,L66,L67,L68,L69] ], [lat2|L]):-
      magic_movimento(N,N1),
      magic_solve(N1,[ [L11,L68,L13,L14,L65,L16,L17,L62,L19],
                [L21,L22,L23,L24,L25,L26,L27,L28,L29],
                [L31,L58,L33,L34,L55,L36,L37,L52,L39],
                [L41,L42,L43,L44,L45,L46,L47,L48,L49],
                [L51,L12,L53,L54,L15,L56,L57,L18,L59],
                [L61,L32,L63,L64,L35,L66,L67,L38,L69] ],L).

magic_solve(N,[ [L11,L12,L13,L14,L15,L16,L17,L18,L19],
          [L21,L22,L23,L24,L25,L26,L27,L28,L29],
          [L31,L32,L33,L34,L35,L36,L37,L38,L39],
          [L41,L42,L43,L44,L45,L46,L47,L48,L49],
          [L51,L52,L53,L54,L55,L56,L57,L58,L59],
          [L61,L62,L63,L64,L65,L66,L67,L68,L69] ], [lat3|L]):-
      magic_movimento(N,N1),
      magic_solve(N1,[ [L11,L12,L69,L14,L15,L66,L17,L18,L63],
                [L27,L24,L21,L28,L25,L22,L29,L26,L23],
                [L59,L32,L33,L56,L35,L36,L53,L38,L39],
                [L41,L42,L43,L44,L45,L46,L47,L48,L49],
                [L51,L52,L13,L54,L55,L16,L57,L58,L19],
                [L61,L62,L31,L64,L65,L34,L67,L68,L37] ],L).

magic_solve(N,[ [L11,L12,L13,L14,L15,L16,L17,L18,L19],
          [L21,L22,L23,L24,L25,L26,L27,L28,L29],
          [L31,L32,L33,L34,L35,L36,L37,L38,L39],
          [L41,L42,L43,L44,L45,L46,L47,L48,L49],
          [L51,L52,L53,L54,L55,L56,L57,L58,L59],
          [L61,L62,L63,L64,L65,L66,L67,L68,L69] ], [frn1|L]):-
      magic_movimento(N,N1),
      magic_solve(N1,[ [L17,L14,L11,L18,L15,L12,L19,L16,L13],
                [L57,L22,L23,L58,L25,L26,L59,L28,L29],
                [L31,L32,L33,L34,L35,L36,L37,L38,L39],
                [L41,L42,L67,L44,L45,L68,L47,L48,L69],
                [L51,L52,L53,L54,L55,L56,L49,L46,L43],
                [L61,L62,L63,L64,L65,L66,L27,L24,L21] ],L).

magic_solve(N,[ [L11,L12,L13,L14,L15,L16,L17,L18,L19],
          [L21,L22,L23,L24,L25,L26,L27,L28,L29],
          [L31,L32,L33,L34,L35,L36,L37,L38,L39],
          [L41,L42,L43,L44,L45,L46,L47,L48,L49],
          [L51,L52,L53,L54,L55,L56,L57,L58,L59],
          [L61,L62,L63,L64,L65,L66,L67,L68,L69] ], [frn2|L]):-
```

```
      magic_movimento(N,N1),
      magic_solve(N1,[ [L11,L12,L13,L14,L15,L16,L17,L18,L19],
                 [L21,L54,L23,L24,L55,L26,L27,L56,L29],
                 [L31,L32,L33,L34,L35,L36,L37,L38,L39],
                 [L41,L64,L43,L44,L65,L46,L47,L66,L49],
                 [L51,L52,L53,L48,L45,L42,L57,L58,L59],
                 [L61,L62,L63,L28,L25,L22,L67,L68,L69] ],L).

magic_solve(N,[ [L11,L12,L13,L14,L15,L16,L17,L18,L19],
          [L21,L22,L23,L24,L25,L26,L27,L28,L29],
          [L31,L32,L33,L34,L35,L36,L37,L38,L39],
          [L41,L42,L43,L44,L45,L46,L47,L48,L49],
          [L51,L52,L53,L54,L55,L56,L57,L58,L59],
          [L61,L62,L63,L64,L65,L66,L67,L68,L69] ], [frn3|L]):-
      magic_movimento(N,N1),
      magic_solve(N1,[ [L11,L12,L13,L14,L15,L16,L17,L18,L19],
                 [L21,L22,L51,L24,L25,L52,L27,L28,L53],
                 [L33,L36,L39,L32,L35,L38,L31,L34,L37],
                 [L61,L42,L43,L62,L45,L46,L63,L48,L49],
                 [L47,L44,L41,L54,L55,L56,L57,L58,L59],
                 [L29,L26,L23,L64,L65,L66,L67,L68,L69] ],L).
```

Nsort

```
nsort(L) :-
    go_nsort([11,10,9,8,7,6,5,4,3,2,1],L).

go_nsort(L1,L2) :-
    nsort_permutation(L1,L2),
    nsort_sorted(L2).

nsort_sorted([X,Y|Z]) :-
    X =< Y,
    nsort_sorted([Y|Z]).
nsort_sorted([_]).

nsort_permutation([],[]).
nsort_permutation(L,[H|T]):-
    nsort_delete(H,L,R),
    nsort_permutation(R,T).

nsort_delete(X,[X|T],T).
nsort_delete(X,[Y|T],[Y|T1]) :-
    nsort_delete(X,T,T1).
```

Cubes7

```
cubes7(Sol):-
   cubes(7,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(7,[q(p(5,1),p(0,5),p(3,1)),
         q(p(2,3),p(1,4),p(4,0)),
         q(p(3,6),p(0,0),p(2,4)),
         q(p(6,4),p(6,1),p(0,1)),
         q(p(1,5),p(3,2),p(5,2)),
         q(p(5,0),p(2,3),p(4,5)),
         q(p(4,2),p(2,6),p(0,3))]).
```

Ham

```
ham(H):-cycle_ham([a,b,c,d,e,f,g,h,i,j,
    k,l,m,n,o,p,q,r,s,t],H).

cycle_ham([X|Y],[X,T|L]) :-
    chain_ham([X|Y],[],[T|L]),
    ham_edge(T,X).

chain_ham([X],L,[X|L]).
chain_ham([X|Y],K,L) :-
    ham_del(Z,Y,T),
    ham_edge(X,Z),
    chain_ham([Z|T],[X|K],L).

ham_del(X,[X|Y],Y).
ham_del(X,[U|Y],[U|Z]) :- ham_del(X,Y,Z).

ham_edge(X,Y) :-
    ham_connect(X,L),
    ham_el(Y,L).

ham_el(X,[X|_]).
ham_el(X,[_|L]) :- ham_el(X,L).
```

```
ham_connect(a,[b,j,k]).
ham_connect(b,[a,c,p]).
ham_connect(c,[b,d,l]).
ham_connect(d,[c,e,q]).
ham_connect(e,[d,f,m]).
ham_connect(f,[e,g,r]).
ham_connect(g,[f,h,n]).
ham_connect(h,[i,g,s]).
ham_connect(i,[j,h,o]).
ham_connect(j,[a,i,t]).
ham_connect(k,[o,l,a]).
ham_connect(l,[k,m,c]).
ham_connect(m,[l,n,e]).
ham_connect(n,[m,o,g]).
ham_connect(o,[n,k,i]).
ham_connect(p,[b,q,t]).
ham_connect(q,[p,r,d]).
ham_connect(r,[q,s,f]).
ham_connect(s,[r,t,h]).
ham_connect(t,[p,s,j]).
```

Nrevwarren

```
nrev([],[]).

nrev([X|Rest],Ans):-
    nrev(Rest,L), app(L,[X],Ans).

app([],L,L).

app([X|L1],L2,[X|L3]):-
    app(L1,L2,L3).
```

Referências

[AK90a] K. Ali and R. Karlsson. The Muse Approach to OR-Parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, 1990.

[AK90b] K.A.M. Ali and R. Karlsson. The Muse Or-parallel Prolog Model and its Performance. In *In N. American Conf. on Logic Prog.*, page 757776. The MIT Press, 1990.

[AK91] Hassan Ait-Kaci. *Warren's Abstract Machine*. The MIT Press, 1991.

[AK92] K. Ali and R. Karlsson. Scheduling Speculative Work in MUSE and Performance Results. *International Journal of Parallel Programming*, 21(6):449–476, 1992.

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

[GACH97] Gopal Gupta, Khayri Ali, Mats Carlsson, and M. Hermenegildo. Parallel execution of prolog programs: A survey. *ACM Transactions on Computational Logic*, 1997.

[GP99] G. Gupta and E. Pontelli. Stack-splitting: A simple technique for implementing or-parallelism on distributed machines. *In ICLP. MIT Press*, page 290304, 1999.

[Kar92] Roland Karlsson. *A High Performance OR-parallel Prolog System*. PhD thesis, SICS, The Royal Institute of Technology, Sweden, March 1992.

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

[LBD+88] E. 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 *International Conference on Fifth Generation Computer Systems*, pages 819–830. ICOT, Japan, November 1988.

[Mat87] Fridemann Mattern. Algorithms for distributed termination detection. *Distributed Computing, Volume 2, Number 3*, pages 161–175, 1987.

[MC89] Johan Widen Mats Carlsson. SICS Research R88007B,SICStus Prolog User's Manual. Technical report, Swedish Instituite of Computer Science, October 1989.

[Roc96] Ricardo Rocha. Um Sistema Baseado na Cópia de Ambientes para a Execução de Prolog em Paralelo. Master's thesis, Universidade do Minho, 1996.

[RPG99] D. Ranjan, E. Pontelli, , and G. Gupta. On the complexity of or-parallelism. *NGC*, page 17(3):285308, 1999.

[RSC99] Ricardo Rocha, Fernando Silva, and Vítor Santos Costa. YapOr: an Or-Parallel Prolog System Based on Environment Copying. In *9th Portuguese Conference on Artificial Intelligence (EPIA'99), Évora, Portugal*, pages 178–192. Springer-Verlag, LNAI 1695, September 1999.

[SP89] A. Singhal and Y. Patt. Unification Parallelism: How much can be Exploited? In *The 1989 North American Conference on Logic Programming*, pages 1135–1148. The MIT Press, 1989.

[VPHG01] K. Villverde, E. Pontelli, H.Guo, and G.Gupta. PALS: An Or-Parallel Implementation of Prolog on Beowulf Architectures. In *ICLP 2001*, pages 27–42, 2001.

[War77] David H. D. Warren. *Applied Logic – Its Use and Implementation as a Programming Tool*. PhD thesis, 1977.

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