Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Mecanismos de Suporte à Execução Concorrente de Programas em Lógica

João Manuel dos Santos Lourenço

Dissertação apresentada para a obtenção do Grau de Mestre em Engenharia Informática pela Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia.

Lisboa
(1994)

à Teresa
e aos meus Pais

Sumário

A evolução do *hardware* dos computadores para arquitecturas paralelas, incentivou a concepção de novos modelos de programação e o desenvolvimento dos sistemas de suporte à execução correspondentes, de forma a conseguir uma melhor exploração do paralelismo. A linguagem de programação Prolog, pelas suas características declarativas e operacionais, tem vindo a ser objecto de estudo nesta área, através de adaptações da linguagem e/ou da sua máquina de inferência.

Este trabalho incide sobre os aspectos de concepção e implementação de um modelo de um sistema de suporte à execução de programas em Prolog, em arquitecturas de múltiplos processadores, com unidades de memória fisicamente distribuídas.

O modelo propõe extensões a um executor de Prolog convencional, de forma a disponibilizar funcionalidades que permitam o controlo do paralelismo e da distribuição. Estas funcionalidades podem ser utilizadas para a implementação de modelos de linguagens lógicas concorrentes de mais alto nível, ou então serem utilizadas directamente para a programação de sistemas distribuídos, em que múltiplos executores Prolog cooperam na resolução de um golo, comunicando com base em mensagens. Para avaliar a funcionalidade do modelo proposto, concebeu-se e implementou-se um sistema de distribuição de golos Prolog, que permite recorrer a diversas estratégias para composição sequencial e paralela de golos, escondendo os aspectos de gestão explícita dos recursos efectivos.

A dissertação inclui uma discussão dos aspectos mais relevantes da realização do protótipo do modelo proposto sobre uma arquitectura baseada em *Transputers*.

Abstract

The hardware evolution towards parallel computer architectures triggered the design of new programming languages models and the corresponding development of their runtime support in order to allow a better exploitation of parallelism. The Prolog programming language, due to its declarative and operational characteristics has been the subject of intensive research in this direction, through extensions made to the language and/or its inference engine.

This dissertation discusses the design and implementation of a model of a runtime support system for the Prolog language, suited to parallel execution on multiple processors architectures with physically distributed memory.

The model defines a set of extensions to a conventional Prolog executor towards the control of parallelism and distribution. Such functionalities may be used in the implementation of higher-level concurrency models in logic languages. Alternatively they may directly be used in the programming of distributed Prolog systems, where multiple agents cooperate towards the resolution of a common goal, using message-passing based communication.

The suitability of the proposed model was tested in the implementation of a parallel programming toolkit which provides a transparent scheme for goal distribution and resolution, according to sequential and parallel execution strategies.

The dissertation also includes a discussion on the most relevant aspects of the developed prototype of the proposed model, on a *Transputer*-based parallel architecture.

Simbologia e notações

Na elaboração desta dissertação foram usados, sempre que possível, os termos portugueses correntes para os temas em causa (que algumas vezes não correspondem às melhores traduções). Por exemplo, o termo português usado como equivalente do termo anglo-saxónico *goal* foi **golo** (e não **objectivo**).

Para os termos anglo-saxónicos para os quais não existe uma tradução vulgarizada, optou-se pela sua utilização na forma original, em itálico. Por exemplo, durante toda a dissertação são sempre usados os termos *byte* e *buffer*, que não têm equivalente corrente em português.

Nos capítulos 4 e 5, todos os nomes de argumentos dos predicados apresentados têm o prefixo "+" ou "−", indicando que são argumentos de entrada ou saída, respectivamente. Por exemplo, em sys_at_halt(+Goal, −Status) o argumento `Goal` é de entrada, tendo que estar instanciado na invocação do predicado, e o argumento `Status` é de saída, tendo de corresponder a uma variável livre na invocação do predicado.

Conteúdo

Lista de Figuras

Capítulo 1

Apresentação

1.1 Introdução

O Prolog é uma linguagem de programação que, na sua forma mais pura, permite a representação do conhecimento sob a forma de cláusulas em *lógica de cláusulas de Horn* e que, associado a um mecanismo computacional, permite a geração automática de conclusões lógicas no universo de conhecimento representado pelo programa.

A evolução do *hardware* dos computadores, a partir das arquitecturas convencionais de Von Neumann para arquitecturas alternativas paralelas, incentivou a exploração dessas novas arquitecturas, através da concepção de novas linguagens de programação, da adaptação de linguagens já existentes e o desenvolvimento de novos sistemas de operação e de suporte à execução de programas. A linguagem de programação em lógica Prolog, pelas suas características declarativas e operacionais, tem vindo a ser objecto de estudo nesta área, através de adaptações da linguagem e/ou a sua máquina de inferência, de forma a que permita explorar convenientemente as novas arquitecturas paralelas.

A disponibilização de plataformas de suporte à execução paralela de programas, a um nível intermédio de um sistema de computação, tema central desta tese, é um aspecto essencial para permitir uma adequada experimentação com vista à mencionada exploração do paralelismo.

Em particular, este trabalho incide sobre os aspectos de concepção e implementação de um modelo de um sistema de suporte à execução de programas em lógica em arquitecturas de múltiplos processadores, com unidades de memória fisicamente distribuídas. Um aspecto interessante do modelo é o de ser concebido sob a forma de extensões a um executor de Prolog convencional (seja ele baseado num interpretador ou num compilador), cuja sintaxe e semântica não são modificadas. Outro aspecto interessante, é o de o modelo disponibilizar funcionalidades de nível intermédio para o controlo de paralelismo e da distribuição, que podem ser utilizadas para a implementação de modelos de linguagens lógicas concorrentes de mais alto nível — desde que orientadas para processos — ou então, serem utilizadas directamente para a programação de sistemas distribuídos, em que múltiplos agentes Prolog comunicam por mensagens. Por outro lado, embora o trabalho descreva um protótipo de um modelo implementado sobre um sistema de operação e arquitectura específicos, a definição das primitivas do modelo é independente das camadas de *software*/*hardware* subjacentes, permitindo outras implementações sobre ambientes de execução diferentes.

Neste capítulo faz-se um enquadramento do trabalho que esta dissertação apresenta, são indicados os objectivos que se propõem atingir e apresenta-se a organização dos restantes capítulos desta dissertação.

1.2 Enquadramento do trabalho

Nesta secção é feito um enquadramento do trabalho descrito nesta dissertação, através duma apresentação dos factores que o motivaram e do relato dos seus antecedentes históricos.

1.2.1 Motivação

A implementação de um modelo de execução concorrente ou paralela de programas em lógica apresenta, geralmente, algumas dificuldades devido às distâncias, formal e operacional, entre o nível do modelo a implementar e o nível do modelo disponibilizado pelo sistema de operação e arquitectura dos computadores subjacente. As dificuldades de realização dos conceitos e abstracções do modelo a implementar, são tanto maiores quanto maior for este denominado *abismo semântico* (ver figura 1.1).

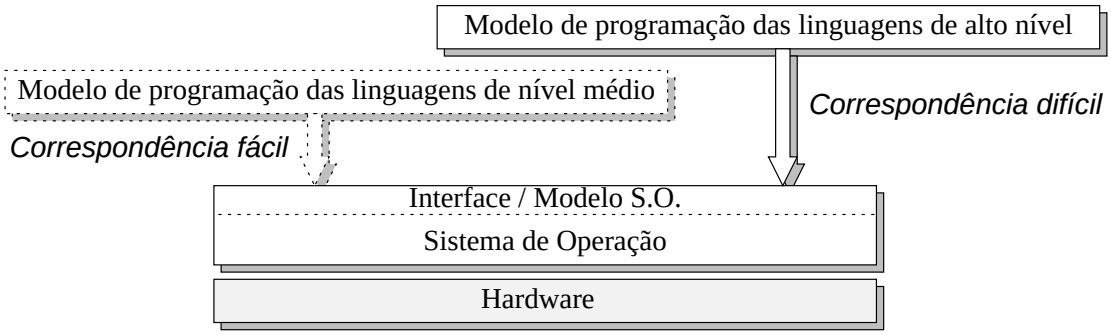

Figura 1.1: Distância entre os modelos de linguagem e de sistema de operação

As dificuldades que surgem durante a fase de implementação de um modelo de programação de alto nível (como é o caso de um modelo que suporte a concorrência numa linguagem de programação em lógica) levam, muitas vezes, a restrições do modelo operacional idealizado para a linguagem, forçando-o a contemplar apenas um subconjunto do modelo formal ou a restringir algumas das funcionalidades que este propõe.

Por outro lado, devido à grande diversidade e especificidade dos sistemas de operação para as arquitecturas paralelas (grande parte dos fabricantes usam sistemas de operação específicos e de que são proprietários), cada processo de transporte de um programa de uma arquitectura para outra envolve o estudo específico das características do novo sistema de operação e de como realizar todas as abstracções e funcionalidades desejadas sobre o novo sistema.

Justifica-se, assim, a conveniência de um sistema que suporte um nível intermédio (ver figura 1.2) entre o modelo de programação de alto nível a implementar, *e.g.* modelo de programação Prolog, e o modelo disponibilizado pelo sistema de operação da máquina real. O nível intermédio virtualiza as abstracções e funcionalidades do ambiente real de operação, tal como este virtualiza os dispositivos físicos da máquina em termos de dispositivos lógicos.

Um sistema com estas características contribui significativamente para a obtenção, num curto espaço de tempo, de uma realização final do modelo de alto nível (que esteja correcta do ponto de vista formal e operacional), através de um protótipo, com uma forma independente da arquitectura que suporta a sua implementação[1]. A implementação do modelo de alto nível recorre aos serviços do sistema de nível intermédio para satisfação dos serviços que não são genéricos, que por sua vez os realiza com base nos serviços do sistema de operação.

[1] O transporte do modelo para uma nova arquitectura consiste, essencialmente, no transporte do sistema intermédio.

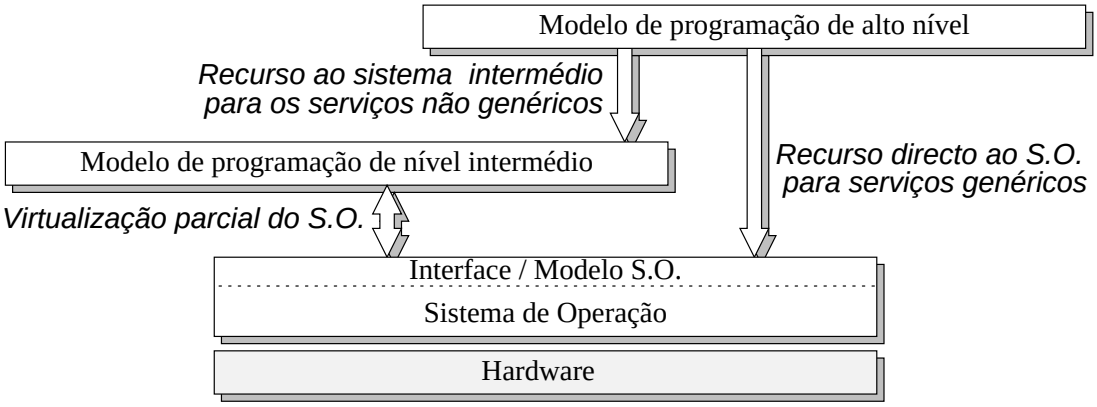

Figura 1.2: A integração de um sistema de nível intermédio

Entende-se por *serviços não genéricos*, aqueles que correspondem a funcionalidades específicas do modelo de alto nível a implementar, *i.e.* que satisfazem as definições específicas da semântica operacional desse modelo; por oposição aos *serviços genéricos*, que são oferecidos pelo sistema de operação e que têm definições mais "universais" e menos comprometidas com um determinado modelo de alto nível (seja relativamente a controlo de processos, a comunicação entre processos ou a acesso a ficheiros).

Depois de testada a implementação do modelo e, se o desempenho é um dos factores relevantes, é sempre possível rever a implementação do sistema, fazendo acesso directo aos recursos da máquina, a fim de se obter uma implementação tão eficiente quanto possível, embora com mais dependências da arquitectura e/ou sistema de operação usados.

1.2.2 Enquadramento histórico

Este trabalho insere-se no contexto de uma linha de investigação que tem sido desenvolvida no Departamento de Informática da Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa (DI/UNL) desde 1983, com origem na apresentação do modelo teórico da Lógica Distribuída [Mon83, Mon86] e com a proposta do *Delta Prolog* [PN84, PMCA86], uma linguagem de programação concorrente em lógica, fundamentada naquele modelo teórico.

Desde 1984 que, no Departamento de Informática da Universidade Nova de Lisboa, se estudam aspectos de concepção e implementação do *Delta Prolog* [CA84, CMC87, Cun88, Car91]. A experiência revelou que uma abordagem da implementação "por camadas" — começando por estender um interpretador/compilador de Prolog com instruções de controlo da concorrência e da comunicação, sobre o qual são depois implementadas as contruções próprias do *Delta Prolog*, facilita o processo de implementação da linguagem, e aumenta consideravelmente a facilidade de transporte do código para novas arquitecturas. A independência entre as "camadas" permite a reutilização dos estratos inferiores, como suporte para a implementação de novos modelos de linguagens concorrentes em lógica, para além do *Delta Prolog*.

Em 1987, tal como descrito em [CMC87], foi aumentada a funcionalidade de um interpretador de Prolog com predicados de comunicação (baseados num modelo de comunicação orientada para portas), designado internamente ao DI/UNL por *Port Prolog*. Esta implementação foi feita originalmente sobre VMS e UNIX e posteriormente influenciou, de forma decisiva, a concepção de um núcleo de sistema de operação distribuído — o T-NET — integralmente desenvolvido para uma arquitectura paralela baseada em *Transputers* [Lim88b] (com mecanismos de comunicação optimizados e parti-

cularmente eficientes), com objectivos mais vastos, para além do suporte de programação em lógica [SC90].

A partir de 1990 e na sequência da aquisição de uma máquina paralela baseada em *Transputers*, no âmbito de um projecto ESPRIT (*Parallel Computing Action*) [CMP88], iniciou-se a exploração sistemática de uma linha de investigação em Processamento Paralelo e Distribuído [Cun94], que actualmente decorre sob a coordenação do orientador desta dissertação.

No ano de 1991, as funcionalidades e o desempenho do protótipo do modelo do *Port Prolog* foram substancialmente melhorados, ainda no contexto de uma implementação para uma arquitectura baseada em *Transputers* [LB91], na qual o autor desta dissertação trabalhou, no contexto da referida linha de investigação. As melhorias derivaram duma maior flexibilidade nos predicados de sistema suportados — para o controlo de instâncias concorrentes de processos Prolog e para a geração e tratamento de sinais/interrupções no contexto do executor de Prolog — e dum melhor desempenho, que se ficou a dever ao uso de um sistema de Prolog baseado num compilador e máquina abstracta do tipo WAM [War83, War88] (uma versão desenvolvida no DI/UNL e denominada *Nanoprolog* [Dia90]).

Os desenvolvimentos que se relatam nesta dissertação enquadram-se ainda nos objectivos de dois projectos de investigação, recentemente iniciados, visando a exploração do paralelismo em lógica [P$^+$94, eJCC94].

1.3 Objectivos

O principal objectivo deste trabalho é o de estudar formas de melhorar a funcionalidade dos sistemas de suporte à execução paralela de programas em lógica, a um nível intermédio (ver figura 1.1) de um sistema computacional. Ao nível do modelo, as funcionalidades que se acrescentam ao sistema de execução sequencial de Prolog, surgem na forma de predicados extra-lógicos ou de controlo, que se podem integrar em qualquer sistema Prolog, seja ele baseado em interpretadores seja em compiladores. Assim, a proposta dessas funcionalidades fica condicionada aos requisitos postos pelas camadas superiores, em particular os modelos específicos que suportam concorrência em linguagens lógicas baseadas em Prolog.

O modelo intermédio propõe linguagens lógicas orientadas para processos distribuídos, *i.e.* nas quais se especificam múltiplas instâncias de executores de Prolog, cada uma das quais se envolve num processo de resolução de golos, com espaço de cláusulas e de trabalho privados. O controlo das múltiplas instâncias de programas, a gestão da comunicação e a afectação de processos a processadores reais, são os aspectos contemplados pelo modelo intermédio.

Um objectivo adicional deste trabalho é o de estudar a viabilidade da implementação deste tipo de modelos sobre máquinas paralelas baseadas em *Transputers*, explorando as funcionalidades específicas do sistema de operação, mas preservando um nível razoável de independência da máquina e do seu sistema de operação, compatível, no entanto, com as exigências de desempenho. Quanto ao sistema Prolog, pressupõe-se, na implementação descrita, o uso de sistemas baseados num compilador e máquina abstracta do tipo WAM[AK90], nos quais se integra o suporte para os referidos predicados extra-lógicos, embora o mesmo modelo se possa aplicar a sistemas baseados em interpretadores.

Este trabalho propõe-se completar o modelo do *Port Prolog* descrito em [CMC87, LB91] com a clarificação e sistematização do seu modelo operacional, através da adição de novas funcionalidades, compatíveis com a evolução entretanto verificada nos modelos de concorrência e nas arquitecturas paralelas, em particular, as de memória distribuída. Para demonstração da funcionalidade do sistema proposto, apresenta-se um módulo de suporte ao processamento paralelo de golos de um programa

em Prolog. Este módulo pode, por sua vez, ser usado na implementação de mecanismos de mais alto nível, como é o caso do operador "//" do *Delta Prolog* (ilustrado na secção 5.3).

Como todo o planeamento e concepção dos modelos atrás referidos foi acompanhado por um trabalho experimental, que resultou na implementação completa destes, apresenta-se ainda a descrição dos pontos mais relevantes dessa implementação sobre um multicomputador baseado em *Transputers* (*Meiko CS/1*), sobre a qual existe, actualmente, um protótipo completamente operacional.

1.4 Organização da dissertação

Neste capítulo foram apresentados os objectivos e motivação deste trabalho, e feito o seu enquadramento histórico. No capítulo 2, são discutidas as dimensões dos sistemas de suporte à execução paralela e distribuída de programas; no capítulo 3, são discutidas algumas das dimensões dos mecanismos de controlo de paralelismo em programas em lógica; no capítulo 4, é apresentado um modelo de suporte à execução de programas paralelos e/ou distribuídos em Prolog, tema central desta dissertação; no capítulo 5, é avaliada a funcionalidade do modelo proposto, através da apresentação de uma camada de paralelização/distribuição de golos Prolog; no capítulo 6, são focados os pontos mais relevantes da implementação do protótipo realizado num multicomputador; finalmente, no capítulo 7, tiram-se conclusões do trabalho realizado e apresentam-se sugestões para trabalho futuro.

Capítulo 2

Ambientes de execução paralela e distribuída

2.1 Introdução

A evolução das arquitecturas alternativas à da máquina de Von Neumann, levou os investigadores a repensarem os algoritmos e linguagens de programação, de forma a conseguirem explorar os múltiplos modelos de organização física que estas máquinas disponibilizam. Este capítulo estuda as grandes classes de arquitecturas paralelas, do ponto de vista da sua organização física e do seu modelo de programação.

Na próxima secção é apresentada uma classificação, que permite distinguir as máquinas em termos da sua arquitectura física; na secção seguinte apresentam-se os principais modelos de programação para estas arquitecturas aternativas.

2.2 Modelos de arquitecturas paralelas e distribuídas

A existência de múltiplas variantes de arquitecturas sequenciais e paralelas, gerou a necessidade de as classificar segundo o seu tipo de processamento. Em [Fly72], Flynn classificou as diversas arquitecturas existentes segundo o número de instruções e de dados que estão activos em cada instante. Young, em [Y+87], e depois Raina, em [Rai93], completaram sucessivamente esta classificação tendo em conta a organização da memória, o espaço de endereçamento e o tipo de acesso à memória das várias arquitecturas. Em [Ass94] faz-se um estudo mais aprofundado sobre este tema.

2.2.1 As principais abordagens

Segundo a classificação de Flynn, distinguem-se 4 modelos de arquitecturas paralelas e distribuídas: SISD, SIMD, MISD, MIMD. Resumem-se, a seguir, as características essenciais de cada modelo, com excepção do modelo MISD (*Multiple Intruction Single Data*), por ser vulgarmente considerada bastante discutível a inclusão de qualquer arquitectura real neste modelo.

Arquitecturas “Single Intruction Single Data” (SISD)

As arquitecturas SISD correspondem, na realidade, às arquitecturas baseadas na máquina de Von Neumann, *i.e.* os computadores sequenciais “convencionais”, com o programa armazenado em memória.

Nesta classe, estão incluídas todas as arquitecturas que, em cada instante, estão a executar apenas uma instrução sobre um único elemento de dados (sendo máquinas com apenas uma unidade de controlo de execução das instruções).

Arquitecturas "Single Intruction Multiple Data" (SIMD)

Nas arquitecturas SIMD, uma unidade de controlo supervisiona a distribuição da instrução corrente por múltiplos elementos processadores, em que cada um destes é dotado de uma unidade aritmética e lógica simplificada e de algumas células de memória locais, nas quais guarda os elementos de dados sobre os quais opera, em completo sincronismo (*lock step mode*) com os outros elementos processadores. Findo um ciclo, no qual todas (ou apenas algumas) unidades aplicaram a mesma operação sobre os seus dados locais, inicia-se um novo ciclo com a difusão da próxima instrução, pela unidade de controlo, para os múltiplos processadores.

Este tipo de arquitectura apenas é eficiente para aplicações específicas que possam tirar partido do correspondente modelo de execução, *e.g.* a soma e a multiplicação de matrizes, não permitindo um uso genérico, para uma larga gama de aplicações o que, acompanhado em geral por um elevado custo (dada a especificidade do *hardware* e a necessidade de dispor de um número apreciável de elementos processadores), as torna menos divulgadas do que as arquitecturas do modelo MIMD, que se discute a seguir.

Arquitecturas "Multiple Intruction Multiple Data" (MIMD)

Nesta classe estão incluídas todas as máquinas que, tendo várias unidades processadoras, permitem, em cada instante, a execução de múltiplas instruções diferentes sobre outros tantos elementos de dados. Dos dois tipos de arquitecturas paralelas, SIMD e MIMD, estas últimas são, sem dúvida, mais genéricas e polivalentes, sendo também as de maior divulgação. Esta classe poderá ainda ser subdividida em três variantes, onde se classificam as arquitecturas de acordo com a forma como é gerido o espaço de endereçamento e o acesso às unidades de memória.

Variante "Shared Address space Shared Memory" (SASM) Nesta classe incluem-se todas as máquinas que são detentoras de múltiplas unidades processadoras e que partilham um espaço de endereços comum, através do acesso a uma única unidade de memória física, em geral, através de um *bus* comum.

Os conflitos de acesso à memória comum penalizam o desempenho e impõem um limite superior ao número de unidades processadoras que se admitem neste tipo de arquitecturas. São, portanto, arquitecturas dificilmente expansíveis[1], quanto ao aspecto do desempenho (ainda que a adição física de novos elementos seja fácil, dada a uniforme interface com o *bus*).

Estas arquitecturas exibem, contudo, uma propriedade interessante: a uniformidade no tempo de acesso a qualquer célula de memória, a partir de qualquer elemento processador, designada normalmente pela sigla UMA (*Uniform Memory Access*).

Variante "Disjoint Address space Distributed Memory" (DADM) Nesta classe incluem-se as máquinas formadas por várias unidades processadoras, cada uma com a sua memória privada (a que chamamos *nó*). Neste caso não existe qualquer tipo de memória física comum, pelo que a comunicação entre os diferentes nós é feita através de uma rede de interconexão, de topologia distinta consoante o modelo específico de arquitectura.

[1] Propriedade designada na notação anglo-saxónica por "*scalability*".

Os aspectos críticos deste tipo de arquitectura são a estrutura e as características de desempenho deste tipo de interconexão entre os elementos processadores. Cada unidade processadora acede directamente apenas à sua unidade física de memória privada, como se fosse um computador sequencial de Von Neumann. Em alguns modelos (ainda que pouco frequentes), este tipo de arquitectura disponibiliza mesmo interfaces físicas de entrada/saída dedicadas a cada processador (ao invés das arquitecturas SASM, ditas vulgarmente multiprocessadores de memória partilhada, em que também o acesso aos periféricos é partilhado através do *bus* comum). As características mencionadas justificam a designação de *multicomputador*, que é vulgarmente atribuída a este tipo de arquitecturas.

Quanto à uniformidade no acesso às múltiplas unidades de memória, estas arquitecturas não permitem, a um dado processador, o acesso directo a unidades *remotas* de memória, *i.e.* associadas a outros nós, cabendo-lhes a designação vulgar de NORMA (*No Remote Memory Access*). Esse acesso faz-se indirectamente, através do processador remoto associado à respectiva unidade de memória, sendo a informação encaminhada por esse processador, através da rede de interconexão e sob a forma de mensagens.

Variante "Shared Address space Distributed Memory" (SADM) Esta categoria de arquitecturas pretende combinar as vantagens de ambas as classes anteriores: tem memória física distribuída mas disponibiliza por *hardware* um espaço de endereçamento comum, visando permitir a partilha eficiente de informação por uma abstracção do modelo de *memória partilhada* (ver secção 2.3.1).

2.2.2 Análise da arquitectura usada nesta dissertação

No contexto do trabalho relatado nesta dissertação, a classe de arquitecturas mais interessante é a dos multicomputadores, sejam eles realizados com base num conjunto de computadores convencionais (do tipo de "estações de trabalho" com sistemas de operação convencionais, *e.g.* UNIX), interligados através de uma rede local; ou sejam realizados com base em elementos processadores homogéneos, com redes de interconexão dedicadas e integradas em unidades físicas também dedicadas (em contraste com a distribuição espacial e interligação física das redes locais).

Nestas unidades dedicadas, que contêm múltiplos elementos processadores (pares CPU/memória), é possível conceber programas de sistema de operação próprios e mais adequados à gestão eficiente dos recursos da máquina paralela. Por outro lado, a própria proximidade física dos componentes deste tipo de multicomputadores e a menor relevância de considerações relativas ao funcionamento autónomo na presença a falhas (pelo menos enquanto a dimensão do sistema não atingir um número muito elevado de processadores), bem como a homogeneidade dos processadores e da representação de dados em memória, permitem optimizar o *software* de suporte de comunicação entre processadores. Isto contrasta com a situação verificada a nível das redes locais de computadores, em que a comunicação é sobrecarregada com múltiplos níveis, para acomodar diferentes protocolos de transmissão de dados, com garantia de robustez face a falhas.

Estas considerações justificam uma classificação mais fina da classe dos multicomputadores, baseada no grau de ligação entre elementos, associado ao tempo de latência da comunicação (que pode variar uma ordem de grandeza de $ms \rightarrow \mu s$ entre a rede local e a máquina dedicada) e ao grau de exigência quanto ao comportamento na presença de falhas (que pode chegar, no caso da máquina dedicada, a sistemas que não contemplam este aspecto).

Tal classificação permite uma análise das aplicações cuja paralelização admite desempenhos aceitáveis, em função da granularidade das computações *versus* a latência da comunicação na arquitectura real.

Daqui decorrem abordagens distintas para o desenvolvimento de plataformas de suporte ao processamento paralelo:

i) uma abordagem que vise a máxima flexibilidade e uma boa exploração dos recursos computacionais terá, certamente, que considerar plataformas *hardware* heterogéneas, com infraestrutura de tipo de rede local, possivelmente incluindo nós convencionais e nós especializados (máquinas dedicadas, *e.g.* multicomputadores);

ii) uma abordagem que vise o máximo desempenho concentra-se, quase exclusivamente, numa abordagem que considera apenas multicomputadores dedicados, homogéneos e com *software* de sistema de operação optimizado.

Como se reconhecem vantagens em ambas as abordagens, o desafio que tem sido colocado aos investigadores, nos últimos anos, tem sido a tentativa de conciliar i) e ii). Para a compreensão das acções envolvidas nesta problemática[2] há que considerar os aspectos dos modelos de programação e dos ambientes de suporte à execução, que serão brevemente revistos nas secções seguintes.

2.3 Modelos de programação paralela e distribuída

A ideia básica do cálculo paralelo é simples: diferentes partes independentes de um algoritmo podem ser executadas em simultâneo, desde que disponham dos dados necessários. Contudo, diversas situações práticas oferecem diferentes motivações para contemplar formas de execução paralela a nível das aplicações:

i) aplicações que exigem computações muito complexas, exibindo tempos de computação consideravelmente longos, quando executadas por computadores sequenciais: a motivação é o aumento do desempenho;

ii) aplicações que, devido à sua natureza distribuída intrínseca, beneficiam de soluções em que se faz a decomposição funcional do programa em processos cooperantes, cada um executável por um processador (virtual ou real) distinto: a motivação é a adaptação da solução à natureza do problema a resolver;

iii) aplicações que, devido à repartição geográfica das entidades processadoras e/ou do arquivo de informação, exigem uma distribuição espacial das componentes da aplicação: a motivação é idêntica à de ii).

iv) aplicações que visam elevados graus de robustez às falhas e garantia de disponibilidade dos serviços que oferecem, recorrendo, para esse efeito, à replicação dos processos e dos dados e à sua afectação a unidades processadoras e de memória distintas: a motivação é a robustez às falhas.

Embora tradicionalmente se caracterizem as situações iii) e iv) no domínio dos denominados *sistemas distribuídos* e as do tipo i) no domínio dos *sistemas paralelos*, a tendência é para a disponibilização de plataformas que suportam mecanismos básicos, sobre os quais os serviços necessários para quaisquer daqueles tipos de aplicações possam ser realizados.

Existem várias alternativas na forma como os processos duma aplicação, executados por uma arquitectura paralela ou distribuída, cooperam entre si. Estas formas de cooperação exigem modelos de execução subjacentes que, ou são suportados directamente por um tipo específico de arquitectura paralela, ou só se conseguem obter por emulação, por *software*, de recursos que a máquina não dispõe, revelando-se normalmente, pouco eficientes.

[2] Estas opções estiveram claramente presentes no desenvolvimento do sistema usado no trabalho experimental: o sistema de operação *Trollius*.

Para obter sucesso na paralelização de uma aplicação há que combinar, da melhor forma, a estrutura da aplicação, a estratégia e o algoritmo de paralelização a usar, e o tipo de arquitectura onde a aplicação vai correr.

2.3.1 As principais abordagens

Na perspectiva de um programador, há duas atitudes em relação ao problema de paralelização de aplicações: ignorá-la completamente ou procurar explorá-la explicitamente. A primeira atitude determina, eventualmente, o recurso a utensílios — compiladores — que procuram identificar algumas formas de paralelismo no programa — em geral expresso num linguagem convencional, *e.g.* Fortran [For94a] — e gerar código para o controlo do paralelismo. A segunda atitude exige a disponibilidade, ao nível de uma interface de programação, de construções para a especificação do paralelismo, que o programador possa indicar expressamente. Dir-se-á que a primeira atitute é mais atraente para o utilizador (programador), dado o maior grau de transparência que oferece, mas tem fortes limitações quanto ao grau de paralelismo, *i.e.* número de entidades independentes e paralelizáveis, que o compilador pode detectar automaticamente. Por outro lado, não é adequada para as situações ii), iii) e iv) acima mencionadas.

Assim, na perspectiva deste trabalho, bem como na da linha de investigação na qual ele se insere [Cun94], a segunda atitude permite maior flexibilidade. Quanto à aparente menor transparência que oferece, tudo depende do nível a que nos consideramos. Por exemplo, é possível definir um modelo de programação paralela, numa linguagem de alto nível — seja imperativa, funcional, lógica e/ou orientada para objetos — com base em abstracções próximas do domínio do problema, escondendo aspectos dependentes dos níveis de sistema de operação e da arquitectura real. É esta a perspectiva em que o trabalho aqui relatado se insere (como se detalha no capítulo 4): a de disponibilizar uma camada de nível intermédio, para o controlo explícito de paralelismo, sobre a qual modelos de mais alto nível se possam implementar.

De uma forma independente do paradigma básico de uma linguagem — imperativa ou declarativa — os modelos de programação paralela consideram dois aspectos essenciais[3]:

i) quais as unidades de concorrência, *i.e.* as entidades activas que são paralelizáveis: processos, instruções, expressões ou objectos;

ii) quais os modelos de comunicação entre aquelas entidades.

Conforme a semântica de cada linguagem, haverá uma ou mais interpretações apropriadas para cada aspecto referido em i) e ii) [Bal90]. Relativamente a estes aspectos, faz-se uma discussão breve dos problemas envolvidos, no contexto das linguagens lógicas, no capítulo 4.

Nesta secção, contudo, revêem-se os três modelos básicos de comunicação que são vulgarmente disponibilizados, seja a nível de linguagem, seja a nível de sistema de operação, ou seja ainda a nível de uma interface de programação paralela, *e.g.* suportada por uma biblioteca de apoio.

Memória partilhada

A memória partilhada proporciona um modelo de comunicação entre processos baseado em estruturas partilhadas (que podem ser variáveis, objectos passivos, elementos de um espaço de tuplos, ou simplesmente regiões, *e.g.* páginas, de um espaço de endereços global). Este tipo de modelo é claro,

[3] Não se inclui o comportamente face às falhas, aspecto essencial nos sistemas do tipo iii) e iv) atrás referidos, visto que este aspecto não foi contemplado neste trabalho.

transparente e intuititvo, pelo que representa uma aproximação muitas vezes seguida para exprimir a comunicação em aplicações e/ou algoritmos. A automatização da paralelização de aplicações, através da concepção de compiladores que paralelizam o código com intervenção reduzida do programador, como o HPF (*High Performance Fortran*) [For94a], é também mais fácil de realizar sobre este tipo de modelo.

Nas máquinas com uma arquitectura do tipo SASM, a implementação deste modelo é trivial e particularmente eficiente, pois todos os processadores acedem à mesma memória comum, embora possa sofrer dos problemas de conflito nos acessos simultâneos à memória. Nas máquinas com uma arquitectura do tipo DADM, devido à ausência de memória comum, este modelo tem de ser suportado por *software* e (eventualmente) *hardware* adicionais, sendo a sua eficiência altamente dependente do padrão de localidade e tipo (leitura/escrita) das referências, e da quantidade e granularidade das estruturas partilhadas.

Nas máquinas com uma arquitectura do tipo SADM, o uso deste modelo pode não ser tão eficiente quando comparado com as arquitecturas SASM (dependendo do número de unidades envolvidas), mas é mais eficiente que nas arquitecturas DADM, pois todo o *hardware* da máquina já foi pensado de forma a suportar este modelo, estando por isso optimizado.

Troca de mensagens

O modelo de comunicação por *troca de mensagens* é, geralmente, considerado uma generalização do modelo de entradas/saídas, *i.e.* para todos os efeitos é equivalente à transferência de valores entre espaços de endereços de processos distintos. Neste sentido é um modelo mais primitivo do que o baseado em memória partilhada, apesar de estes serem duais, *i.e.* é possível emular um sistema de troca de mensagens com base num de memória partilhada e vice-versa. A sua utilização tem sido considerada interessante a dois níveis distintos de abstracção:

i) ao nível dos modelos de programação, como o *Communicating Sequencial Processes* (*CSP*) [Hoa78] e o *Occam* [Lim88a], a filosofia é disponibilizar operadores básicos com os quais o programador possa especificar a decomposição do problema em termos de processos e a sua comunicação em termos de operadores de entrada e saída de mensagens (seja directamente entre processos, *e.g. CSP*, seja indirectamente, *e.g.* através de canais, em *Occam*); uma teoria de processos concorrentes formaliza a semântica dos programas destas linguagens [Hoa78], caracterizada por modelos de comunicação síncrona, com *rendez-vous*; outros modelos de programação, contudo, têm proposto linguagens e definido as respectivas semânticas, para o caso de modelos de comunicação assíncrona, *e.g.* o modelo dos actores [Agh86].

ii) ao nível das camadas intermédias e/ou dos sistema de operação, este modelo é de fácil a implementação nas arquitecturas MIMD, sejam de memória partilhada ou sejam, em particular, de memória distribuída.

Um modelo de comunicação baseado em mensagens é caracterizado pelas seguintes dimensões fundamentais:

i) tipo de designação das entidades comunicantes: directa ou indirecta;

ii) sincronização de processos nas operações de comunicação: modos de operação síncronos ou assíncronos, tanto no envio como na recepção de mensagens;

iii) formas de recepção de mensagens explícita ou implícita, neste caso através de um mecanismo assíncrono de interrupções, com ou sem invocação de *threads* de tratamento;

iv) formas de interacção simétrica ou assimétrica, entre os parceiros comunicantes; e o sentido do fluxo de informação (unidireccional/bidireccional);

v) formas de controlo do não-determinismo global ou externo a um processo;

vi) modelos de comunicação ponto-a-ponto ou de um processo para muitos e, neste segundo caso, formas de difusão de mensagens para todos (*broadcast*) ou para grupos de processos (*multi-cast*);

vii) propriedades de ordenação na entrega das mensagens, seja em modelos ponto-a-ponto ou de difusão, relativos à preservação da uniformidade e da causalidade;

viii) comportamentos na presença de falhas e propriedade de atomicidade das primitivas de comunicação.

Dada a diversidade de dimensões dos modelos de comunicação baseados em mensagens, optou-se por não incluir nesta dissertação uma discussão de pormenor das mesmas. Para um estudo mais profundo ver [Cun91, AS83, Med89].

O capítulo 4 ilustra — na proposta de um modelo concreto — as opções particulares tomadas em relação às dimensões acima referidas.

Invocação de Procedimentos Remotos

O modelo de *Invocação de Procedimentos Remotos* (*Remote Procedure Call — RPC*) [Han78, BL84, Nel81] propõe um mecanismo de comunicação que permite o desenvolvimento de aplicações distribuídas que comunicam através de um mecanismo especial de invocação de procedimentos, concebido de forma a esconder os mecanismos de comunicação suportados pelo ambiente de execução subjacente.

Do ponto de vista do processo cliente, uma invocação de um procedimentos remoto é semelhante à invocação de um procedimento local. No caso de uma invocação local, o procedimento invocador coloca os argumentos para o procedimento a invocar num sítio bem definido, *e.g.* os registos do processador, e depois transfere o fluxo de execução para esse procedimento. Quando o processamento do procedimento invocado terminar, o procedimento invocador recupera o controlo do fluxo de execução. É de notar que, neste modelo de comunicação, apenas um dos dois processos intervenientes está activo em cada instante.

No caso de uma invocação de um procedimento remoto, o fluxo de execução é, logicamente, transferido entre dois processos — um é o processo cliente, o outro o processo servidor. Os argumentos do procedimento local são colocados numa mensagem e enviados ao servidor, que os extrai da mensagem e os processa, devolvendo ao cliente, também numa mensagem, os resultados obtidos. Estes são, depois, extraídos da mensagem pelo cliente e afectados aos resultados do procedimento local.

2.3.2 Comparação dos modelos

Dos dois modelos analisados, verifica-se que oferecem distintos níveis de abstracção. O modelo de memória partilhada é o que oferece um maior potencial de transparência face à arquitectura subjacente, embora a viabilidade do seu uso e implementação, em arquitecturas de memória física distribuída, esteja limitada a certos condicionalismos da aplicação.

O modelo de troca de mensagens é talvez o mais primitivo e flexível, dado que se adapta bem a qualquer tipo de implementação, seja em arquitecturas de memória física partilhada, seja em arquitecturas de memória física distribuída.

O modelo de Invocação de Procedimentos Remotos procura a transparência na comunicação, a menos do comportamento face às falhas, por inspiração no modelo convencional de chamadas de procedimentos sequenciais. Dos três modelos, é este o que melhor se adapta a formas de estruturação dos programas[4], *e.g.* orientação para objectos, embora o modelo de mensagens também se possa adaptar, consideravelmente bem, a formas de estruturação de programas.

Por ser o mais básico e bem adaptado a multicomputadores, é o modelo de troca de mensagens que tem sido normalmente preferido como paradigma base da comunicação, neste tipo de arquitecturas, seja ao nível so sistema de operação, seja a um nível intermédio (como acontece no trabalho aqui exposto).

2.4 Interfaces de programação paralela e distribuída

2.4.1 As principais abordagens

Para disponibilizar um modelo de programação paralela existem duas abordagens principais:

i) definição de um novo modelo de linguagem de programação, com construções para a especificação do paralelismo e com definições mais ou menos rigorosas das suas semânticas;

ii) extensão de um modelo de uma linguagem de programação existente, com primitivas para o controlo de paralelismo e de comunicação entre processos.

A abordagem i) é mais elegante do ponto de vista formal, porque permite integrar, de uma forma coerente, os mecanismos de concorrência no modelo da linguagem. Existem múltiplos exemplos de experiências neste sentido (em [Bal90] é apresentada uma referência bastante completa), embora a maioria delas, talvez infelizmente, corresponda a linguagens que nunca sairam do laboratório e, por outro lado, aquelas que se tornaram mais divulgadas, *e.g. Ada* e *Occam*, não estão isentas de limitações.

A abordagem ii) é mais pragmática, e tenta aproveitar o *software* já existente, mas enfrenta a dificuldade de procurar adaptar os conceitos de paralelismo a uma linguagem que não foi concebida para esse efeito. Os graus de dificuldade que se podem encontrar nesta adaptação dependem fortemente dos paradigmas centrais da linguagem — *i.e.* imperativa, lógica, funcional e/ou orientada para objectos — conhecendo-se também diversas experiências, com mais ou menos sucesso, neste sentido [Bal90].

Independentemente da abordagem i) ou ii), uma interface de nível inferior ao da linguagem deve ser disponibilizada, com uma funcionalidade que permita a paralelização dos programas.

a) Esta interface pode ser oferecida por uma biblioteca de programas específica que, sobre um sistema de operação determinado, suporta os mecanismos exigidos pelas construções da linguagem ou pelas extensões à linguagem, com a vantagem da facilidade de transporte para outros ambientes de operação (ver figura 2.1).

   Contudo, é frequentemente difícil emular de forma *correcta* e *eficiente* os mecanismos de suporte à linguagem, sobre um sistema de operação que ofereça funcionalidades limitadas

[4] Em oposição ao modelo de memória partilhada, em que as estruturas globais são visíveis a todos os processos.

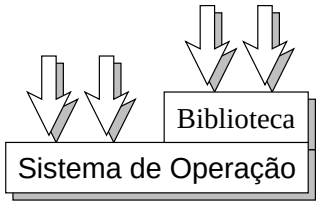

Figura 2.1: Biblioteca sobre o S.O.

quanto ao grau de controlo do paralelismo e das formas de comunicação entre processos, *e.g.* comunicação síncrona/assíncrona, não determinismo na recepção, *e.g.* em instruções do tipo *select()* do UNIX.

b) Em alternativa, a interface descrita em a) pode ser suportada por uma plataforma de programação paralela e distribuída que oferece uma conjunto de serviços de gestão de processos e de comunicação, definidos por uma máquina virtual para o paralelismo, a qual pode ser implementada sobre uma diversidade de sistemas de operação (ver figura 2.2).

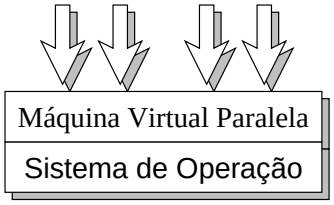

Figura 2.2: Plataforma de programação

Nos últimos anos, algumas experiências têm sido feitas neste sentido — de entre as quais se destaca o *PVM* [G+94, Lou93b] e o *MPI* [For94b] — tendo, como grande vantagem (para além da portabilidade) o facto de que tal camada já disponibiliza um conjunto adequado de primitivas que permite a implementação de modelos específicos de linguagem, ou simplesmente a extensão de linguagens recorrendo à máquina virtual paralela (pela introdução das primitivas desta ao nível da linguagem, como procedimentos e/ou funções em linguagens procedimentais, ou como predicados em linguagens lógicas).

Quanto ao aspecto da eficiência, esta última abordagem exige que na concepção da camada que virtualiza a máquina paralela, se prevejam mecanismos de correspondência (*mapping*), mais ou menos directos, para implementação sobre máquinas paralelas específicas[5].

c) É possível uma terceira abordagem para suportar os mecanismos de controlo do paralelismo e da comunicação: o *recurso directo* ao sistema de operação subjacente. Elimina-se, deste modo, as possíveis penalizações da eficiência que ocorrem nas abordagens a) e b), mas assume-se, à partida, que as primitivas de acesso ao sistema de operação são adequadas para o suporte do modelo de alto nível desejado (ver figura 2.3). Como exemplo desta abordagem, veja-se [Bal90] que ilustra a utilização do sistema de operação *Amoeba* [Tan90] para a programação de diversas aplicações, nas quais explora o paralelismo.

Em geral, esta abordagem pode sofrer as limitações que decorrem do facto de o sistema de

[5] O *PVM*, por exemplo, pode funcionar sobre uma rede heterogénea de estações de trabalho UNIX, mas também pode ser executado em multicomputadores dedicados, onde se conseguem obter melhores desempenhos (em parte devido à menor latência das comunicações).

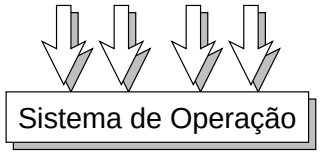

Figura 2.3: Acesso directo ao S.O.

operação não disponibilizar os serviços adequados e não ser possível alterá-lo[6] de forma a modificar/ampliar as suas funcionalidades, para que melhor se adeque ao modelo de programação desejado.

d) No sentido de resolver as limitações de c), surge uma direcção de investigação que emergiu durante a década de 80 no domínio dos sistemas de operação distribuídos [Bal88] [RM87], e que se tem revelado promissora como base para a concepção e implementação de *software* de sistema para multicomputadores dedicados. Esta abordagem consiste na definição de um núcleo minimalista de sistema de operação, sobre o qual são executados servidores (que são processos de um utilizador fictício, *i.e.* não são executados em modo *supervisor*), que disponibilizam os serviços necessários a cada configuração *hardware* da arquitectura (que está a gerir) e a cada ambiente de utilização (ver figura 2.4).

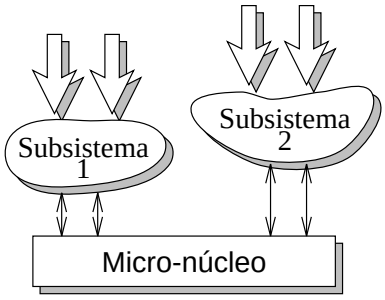

Figura 2.4: Uso de subsistemas

A temática dos sistemas baseados em micro-núcleos — sua organização, constituição, abstracções fundamentais e aplicações — é explorada em [Lou93a].

Com base nesta abordagem, torna-se possível disponibilizar, de forma flexível e eficiente os mecanismos de abstracção necessários ao suporte de um modelo, em particular, no controlo das unidades de concorrência (processos e *thread*) e da gestão das comunicações. Como exemplo deta abordagem, para suporte de *software* sobre multicomputadores, tem-se o caso do *Chorus* [RM87], já transportado para arquitecturas de multicomputadores baseados em *Transputers* (*T800* e *T9000*).

2.4.2 Requesitos postos aos sistemas de suporte à execução paralela e distribuída

Os serviços disponibilizados por uma interface de programação paralela podem oferecer, como se discutiu anteriormente, um maior ou menor grau de transparência para o utilizador. Subjacente a uma interface, deve existir um sistema de suporte à execução que disponibilize as funcionalidades que a seguir se enumeram, de forma sucinta.

[6] Porque não se dispor do código fonte, por exemplo.

Gestão das unidades de concorrência Este aspecto abrange as múltiplas dimensões ligadas à gestão da execução concorrente, nomeadamente as seguintes:

i) suporte da designação ou nomeação global dos processos e de possíveis formas de estruturação das unidades de concorrência, seja na forma de múltiplos *threads* locais a um processo, seja na forma de múltiplos processos que sejam membros de um mesmo *grupo*;

ii) suporte do carregamento e execução de processos, *i.e.* de afectação das unidades de concorrência a processadores reais, e correspondentes estratégias (fixas ou estáticas, dinâmicas – com criação em tempo de execução — e/ou permitindo a migração de processos); inclui possíveis estratégias e mecanismos de suporte do controlo dinâmico da carga de processos do sistema;

iii) suporte de mecanismos de controlo do estado das unidades de concorrência, incluindo mecanismos de controlo do(s) fluxo(s) de execução interno(s) à unidade (mecanismos para o controlo da criação, destruição, suspensão e activação de processos) e de obtenção de informação sobre o seu estado.

Gestão da comunicação Este aspecto abranje as estratégias e os mecanismos de controlo que suportam o modelo de comunicação oferecido a nível da interface de programação, em particular, as dimensões apontadas na secção 2.3.1. Visa esconder ao utilizador dessa interface os aspectos de realização física subjacente, em particular suportando a virtualização da rede física de interconexão dos processadores, os serviços de entrada/saída e de acesso ao sistema de ficheiros, e a implementação das primitivas de comunicação e de entrega de sinais.

Gestão da heterogeneidade Em arquitecturas não homogéneas, *e.g.* em redes locais que incluam multicomputadores dedicados, há que suportar os serviços de identificação dos nós e os seus diferentes atributos (tipo de CPU, capacidade de memória, topologia da interconexão) seja como recursos atómicos, *e.g.* um multicomputador ligado a uma máquina hospedeira pode ser identificado pelo nome desta, ou como recursos individualizados, *e.g.* designar individualmente os vários processadores de um multicomputador. Suporta também formas de representação interna e de conversão de formatos de dados.

Certos sistemas, por compatibilidade com ambientes de operação existentes, suportam a coexistência da interface de programação paralela definida pelo modelo, sobre sistemas de operação distintos, usando a abordagem do modelo cliente/servidor.

Aspectos de monitorização Estes aspectos englobam a detecção de eventos relevantes, durante a execução paralela de programas, e a aquisição de informação sobre o acesso aos recursos reais da arquitectura — *e.g.* processadores, memória, ligações físicas de comunicação — e sobre as entidades lógicas do modelo de computação usado — *e.g.* processos, canais lógicos de comunicação. A informação coligida é utilizada por utensílios de avaliação de desempenho e para a visualização dos comportamentos da computação paralela.

Associada a esta dimensão de monitorização, existe o aspecto de *debugging* que exige, não só a obtenção de informação sobre o estado individual de cada unidade de concorrência, como também sobre a configuração global do sistema, em termos dos processos envolvidos e suas interacções. Por outro lado, o *debugging* exige mecanismos de controlo do estado (referidos atrás) dos processos, seja a nível individual, seja a nível global.

2.5 Conclusões

Neste capítulo discutiram-se os aspectos centrais presentes nos ambientes de execução paralela de programas. Fez-se uma breve resenha dos conceitos fundamentais dos modelos de programação e discutiram-se as principais abordagens para o desenvolvimento de interfaces de programação paralela, tendo-se revisto as suas funcionalidades típicas.

O objectivo deste capítulo foi o de expor, de uma forma sucinta mas concreta, quais as infraestruturas para a realização implementacional de modelos de programação paralela de mais alto nível.

No capítulo 6 faz-se um estudo pormenorizado da abordagem seguida no trabalho experimental que acompanhou a elaboração desta dissertação. Antes, porém, estudam-se as dimensões do paralelismo em programas em lógica e um modelo de suporte à sua execução, cuja implementação se descreve no capítulo 6.

Capítulo 3

As dimensões do paralelismo na programação em lógica

3.1 Introdução

Uma linguagem diz-se lógica quando é fundamentada por um formalismo lógico, cuja expressividade é transposta para a linguagem. Num programa escrito numa linguagem lógica, apenas se expressam os dados (entidades do problema) e relações entre estes, sendo da responsabilidade da "máquina"[1] a escolha das estratégias a seguir para explorar as soluções para um golo, através do uso das relações indicadas no programa. Esta perspectiva das *linguagens ƀgicas* distingue-se da das *linguagens procedimentais*, onde o programa, para além de descrever um problema, tem de dar indicações explícitas de quais as estratégias a tomar para a resolução do problema e de que acções desenvolver para as aplicar — em geral exprimindo formas de sequenciação das instruções do programa, ainda que a lógica da solução do problema não o exija, e/ou explicitando formas de controlo da execução, *e.g.* ciclos de instruções, que revelam o modelo de execução da máquina física subjacente.

O uso da *programação em lógica* permite ao programador abstrair-se da estratégia de resolução do problema, focando a sua atenção nos dados e nas relações que estes mantêm entre si. A capacidade de decisão e escolha das estratégias para a resolução do problema obriga a que a "máquina" executora seja bastante mais complexa do que nas linguagens procedimentais, sendo também necessário que tenha uma boa capacidade de análise semântica do programa, de forma a conseguir decidir qual a melhor estratégia a seguir para determinar a solução em tempo útil.

Uma forma de compensar as falhas na (ou inexistência da) análise da semântica dos programas em lógica e obter melhorias consideráveis no desempenho, é dotar a linguagem com predicados extra-lógicos que, não fazendo parte da especificação do problema, dão indicações mais ou menos precisas das estratégias que o executor deverá seguir[2] para resolver o problema duma forma eficiente.

Quando a resolução de um programa em lógica envolve a avaliação de golos num espaço de pesquisa consideravelmente grande, é muitas vezes rentável, *i.e.* consegue-se um melhor desempenho, se se fizer uma procura concorrente em vez de sequencial, seja através da avaliação concorrente de diferentes regras sobre os mesmos dados, seja através da avaliação da mesma regra sobre diferentes partições dos dados, ou ainda por uma combinação das duas formas. A problemática da execução concorrente de programas em lógica será explorada ao longo deste capítulo.

[1] Entenda-se máquina, neste contexto, como entidade responsável pela execução do programa e não pelo conjunto de dispositivos físicos que constituem o *hardware*.

[2] Existe ainda uma grande diferença entre *indicar as estratégias* que o executor deverá seguir — opção em uso nas linguagens de programação em lógica — e *indicar os passos para implementar as estratégias* — opção necessária nas linguagens procedimentais.

Na próxima secção é feita uma súmula da *lógica das cláusulas de Horn*, base de algumas linguagens de programação em lógica; na secção 3.3 é apresentada uma linguagem de programação em lógica baseada na *lógica das cláusulas de Horn*: o Prolog; na secção 3.4 são estudadas diferentes abordagens de paralelização de linguagens de programação em lógica; na última secção são feitas algumas considerações sobre como e porquê se deve estender o Prolog com mecanismos de controlo da execução.

3.2 A Lógica das Cláusulas de Horn

A interpretação de programação em *Lógica Simbólica* foi introduzida no contexto da *Lógica das Cláusulas de Horn* (LCH) por Kowalski [Kow74b]. Esta interpretação iniciou um domínio nas ciências da computação, normalmente denominado de *Programação em Lógica* e forneceu a base teórica da linguagem de programação Prolog [Rou75].

Para introduzir os conceitos e definições que serão utilizados ao longo deste texto, apresenta-se nesta secção um resumo da Lógica das Cláusulas de Horn. Em [Kow74a, Llo84] é possível encontrar uma exposição mais detalhada deste tema.

3.2.1 A sintaxe

Os símbolos usados na definição da linguagem da LCH estão agrupados nos seguintes conjuntos:

- conjunto (finito) de símbolos de funções "$\mathcal{F}$";
- conjunto (finito) de símbolos de predicados "$\mathcal{P}$";
- conjunto (infinito numerável) de símbolos de variáveis "$\mathcal{V}$";
- conjunto de símbolos e operadores lógicos:
  - implicação "$\leftarrow$";
  - conjunção "$\wedge$" ou ",";
  - quantificador existêncial "$\exists$";
  - quantificador universal "$\forall$".

Cada símbolo de função tem associada uma aridade (≥ 0) que indica o número de argumentos dessa função. Cada símbolo de predicado tem também uma aridade (≥ 0) associada.

Num determinado contexto, *i.e.* dentro de um determinado domínio, uma variável denota um símbolo arbitrário e uma constante denota um símbolo fixo, ambos do contexto em que estão inseridos.

Termo: Um *termo* é uma variável ou uma expressão resultante da aplicação de uma função n-ária, *e.g.* $f(t_1, \ldots, t_n)$, a n termos. Se $n = 0$, então a função escreve-se apenas como f e diz-se uma *constante*.

Fórmula atómica ou **átomo:** Uma *fórmula atómica* ou *átomo* é a expressão que resulta da aplicação de um símbolo de predicado com aridade n a outros tantos termos, *e.g.* $p(t_1, \ldots, t_n)$. Se $n = 0$, então o predicado escreve-se apenas como p.

Cláusula: Uma cláusula, no contexto de lógica de primeira ordem (onde, para além dos operadores atrás indicados, existem operadores de negação "$\neg$", de disjunção "$\vee$" e de equivalência $\leftrightarrow$") é uma disjunção de átomos negados e não-negados, onde os átomos a_i $(0 \leq i \leq n)$ (conclusões) aparecem na sua forma de predicados e os átomos p_j $(0 \leq j \leq m)$ (premissas) aparecem na sua forma de predicados negados

$$\forall X_1 \ldots X_k : \qquad a_1 \vee \ldots \vee a_n \vee \neg p_1 \vee \ldots \vee \neg p_m \tag{3.1}$$

onde $X_1 \ldots X_k$ $(k \geq 0)$ são todas as ocorrências de variáveis na cláusula.

A expressão (3.1) pode escrever-se, de uma forma simplificada, como

$$a_1, \ldots, a_n \leftarrow p_1, \ldots, p_m \tag{3.2}$$

estando a quantificação universal das variáveis omitida e implícita. O operador binário virgula (","), estando à esquerda da implicação ("$\leftarrow$") denota a disjunção dos respectivos operandos (átomos), estando à direita da implicação denota conjunção.

Cláusula de Horn: Uma cláusula de Horn é um caso particular das cláusulas do tipo (3.2), em que $0 \leq n \leq 1$. Nesta condição existe, no máximo, um átomo a que é a conclusão e que é normalmente chamado de *cabeça da cláusula*.

Uma *cláusula de Horn* é, portanto, uma expressão da forma

$$a \leftarrow p_1, \ldots, p_m \qquad (m \geq 0) \tag{3.3}$$

em que a e p_i $(0 \leq i \leq m)$ são átomos, e em que todas as ocorrências de variáveis na cláusula estão, implicitamente, quantificadas universalmente.

Uma cláusula de Horn em que se tem $n = 1$ e $m \geq 0$ diz-se uma *cláusula definida*. Se $m > 0$, então os átomos $p_1, \ldots, p_m$ contituem o *corpo da cláusula*. Se $m = 0$, *i.e.* a cláusula não tem corpo, a cláusula diz-se unitária e escreve-se

$$a \leftarrow \qquad \text{ou apenas} \qquad a \tag{3.4}$$

Nos casos em que $n = 0$, *i.e.* a cláusula não tem cabeça, a cláusula diz-se negativa, e escreve-se

$$\leftarrow p_1, \ldots, p_m \qquad (m > 0) \tag{3.5}$$

O caso em que ambos, $n = 0$ e $m = 0$ a cláusula diz-se *nula* ou *vazia*, e escreve-se usando o símbolo

$$\leftarrow \tag{3.6}$$

Programa em LCH: Um *programa em LCH* pode ser definido por um conjunto $\mathbf{P}$ finito e não vazio de cláusulas de Horn. Uma cláusula de Horn negativa $\mathbf{G}$ indica os predicados que se pretendem provar no universo definido por $\mathbf{P}$.

3.2.2 A semântica declarativa

A semântica declarativa de um programa em LCH define um programa em lógica em termos dos predicados que o conjunto de cláusulas do programa implica logicamente.

Uma cláusula de Horn, com a forma $a \leftarrow p_1, \ldots, p_m$, tem a seguinte leitura declarativa:

a é verdadeiro se p_1 e ... e p_m também forem verdadeiros, para quaisquer variáveis $X_1, \ldots, X_k$ dessa cláusula.

A leitura de uma cláusula de Horn unitária, com a forma $a \leftarrow$, é a seguinte:

a é verdadeiro para quaisquer variáveis $X_1, \ldots, X_k$ dessa cláusula.

A leitura de uma cláusula de Horn negativa, com a forma $\leftarrow p_1, \ldots, p_m$, é a seguinte:

p_1 e ... e p_m são falsos para quaisquer variáveis $X_1, \ldots, X_k$ dessa cláusula.

Finalmente, a leitura declarativa de uma cláusula vazia, com a forma $\leftarrow$, é:

$\perp$ (contradição)

3.2.3 A semântica operacional

A semântica operacional de um programa em LCH define um programa em lógica em termos dos golos que são refutáveis usando o *prinápio da resolução* [Llo84]. Para decidir sobre a aplicabilidade ou não de uma cláusula no processo de resolução, é utilizado um mecanismo de *unificação* de átomos [Llo84]. A equivalência das semânticas declarativa e operacional foi formalmente definida em [vEK77].

A semântica operacional da LCH define uma leitura procedimental dum programa (*interpretação de programa*) em LCH, como

- uma cláusula '$p \leftarrow p_1, p_2, \ldots, p_n$' é interpretada como a declaração de um procedimento, com nome definido pela cabeça da cláusula (p), cujos parâmetros formais especificam os argumentos do procedimento, e em que o corpo da cláusula corresponde a um conjunto de chamadas de procedimento. A chamada de procedimento é efectuada pelo passo de resolução que substitui o golo pelo corpo da cláusula, em que a *passagem de argumentos* e *retorno de resultados* são efectuados pelo mecanismo de unificação;
- uma cláusula '$p \leftarrow$' é interpretada como um procedimento vazio. A execução dum procedimento vazio pode ter efeitos nos parâmetros de retorno, resultante da unificação dos argumentos do golo com os da cabeça da cláusula.
- uma cláusula negativa '$\leftarrow p_1, p_2, \ldots, p_n$' é interpretada como uma instrução para resolver p_1 e p_2 e ... e p_n;
- a cláusula vazia '$\leftarrow$' é interpretada como a instrução para parar a execução do programa.

3.3 A linguagem de programação Prolog

A interpretação de programação da LCH está na base da linguagem de programação em lógica Prolog [Rou75, CM81], cujas implementações convencionais seguem uma estratégia de computação baseada num modelo de execução sequencial. Nestas implementações, o átomo mais à esquerda duma cláusula negativa, é sempre o primeiro a ser considerado para resolução, sendo a primeira cláusula que unifique com esse átomo, pela ordem textual em que estas aparecem no programa, aquela vai ser usada na resolução. A estratégia de pesquisa usa um mecanismo de retrocesso (*backtracking*) quando

a resolução do átomo seleccionado não é possível, ou quando são pedidas soluções alternativas para o golo inicial.

Um programa em Prolog é, portanto, constituído por um conjunto de cláusulas que modelam o problema. Uma cláusula negativa, que modela o golo inicial, desencadeia a execução do programa.

3.3.1 O Prolog puro

O Prolog puro corresponde à linguagem da *lógica das cláusulas de Horn*, permitindo modelar exactamente os mesmos problemas e obter exactamente as mesmas soluções.

Sintaticamente, o Prolog usa uma notação ligeiramente diferente, em que

- '$\leftarrow$' é substituído por ':-';
- '$\wedge$' é substituído por ',' (vírgula);
- todas as cláusulas são terminadas por '.' (ponto);
- como na LCH todas as variáveis estão quantificadas universalmente, essa quantificação é omitida, estando implícita;
- o golo tem um prefixo '?-' para se distinguir do programa.

pelo que o programa (e golo inicial) em LCH

$$\begin{array}{ll} \forall X & a(X) \leftarrow b(X) \wedge c(X) \\ b(d) & \\ c(d) & \\ \forall Y & \leftarrow a(Y) \end{array}$$

se transforma em

$$\begin{array}{l} a(X) \text{ :- } b(X),\ c(X). \\ b(d). \\ c(d). \\ \text{?- } a(Y). \end{array}$$

3.3.2 Independência da estratégia de resolução

O Prolog, por convenção, usa uma estratégia de resolução sequencial, com pesquisa em profundidade e da esquerda para a direita. Esta estratégia faz com que as cláusulas que modelam o problema a representar não possam ser recursivas à esquerda, sob pena da máquina de inferência (executor de Prolog) entrar num ciclo infinito de busca em profundidade.

Como a semântica operacional da LCH é omissa quanto à estratégia de selecção do golo a resolver e da cláusula com que esse golo vai ser resolvido, é possível implementar estratégias alternativas que, por exemplo, façam uma busca em largura em vez de em profundidade[3] ou que tentem primeiro os ramos menos profundos[4].

Caso se esteja a correr a máquina de inferência numa arquitectura paralela, é possível, também devido à *neutralidade da estratégia de selecção* atrás referida, resolver múltiplos golos em paralelo (caso se consiga garantir a consistência dos valores das variáveis).

[3] Esta estratégia, usada livremente, eliminaria o problema da *recursidade à esquerda*, mas introduziria custos de gestão de memória potenciamente incomportáveis.

[4] A determinação de quais os ramos menos profundos pode ser aproximada por processos baseados em heurísticas.

3.4 Abordagens de paralelização de programas em lógica

Como nas linguagens lógicas o resultado da computação é, potencialmente, independente da estratégia usada (seja ela baseada num algoritmo sequencial ou paralelo), estas apresentam-se como boas candidatas ao suporte de programação paralela e concorrente.

Desde o início da década de 80 têm vindo a desenvolver-se cinco grandes formas de exploração do paralelismo em linguagens lógicas, que exploram diversas estratégias de paralelização, nomeadamente:

i) linguagens com paralelismo OU (*OR-Parallel languages*);

ii) linguagens com paralelismo E (*AND-Parallel languages*);

iii) linguagens com paralelismo E/OU (*AND/OR-Parallel languages*);

iv) linguagens com paralelismo E em Cadeia (*Stream AND-Parallel languages*);

v) linguagens concorrentes e Orientadas para Processos Distribuídos (*Process Oriented languages*).

Estas cinco abordagem de paralelização de programa em lógica são apresentadas, sucintamente, nas próximas cinco secções. Em [KW92] é possível encontrar uma exposição mais detalhada, com estudo de casos, sobre estas diferentes abordagens de paralelização de programas em lógica.

Existem ainda outras três classes de linguagens têm, nos últimos anos, sido objecto de investigação:

i) linguagens que integram “Objectos + Lógica + Concorrência” [Eli, Eli92];

ii) linguagens lógicas concorrentes com Comunicação baseada em Estruturas Partilhadas [Cia93a, Cia93b, BC90];

iii) linguagens lógicas concorrentes baseadas em Restrições (*Concurrent Constraint languages*) [Sar89, SKL90].

A abordagem i) será também apresentada depois das cinco atrás descritas. As abordagens ii) e iii), por manifesta falta de espaço, não serão revistas. É possível, no entanto, encontrar em [Cia93b] e [Sar89] descrições detalhadas dos temas ii) e iii) respectivamente.

3.4.1 Linguagens com paralelismo OU

Uma computação com paralelismo OU, pode-se representar por uma árvore OU, *i.e.* um grafo em forma de árvore em que os nós representam disjunções lógicas, estando a execução sequencial dos nós E implícita nos arcos.

A invocação de um predicado pode desencadear diversos percursos de procura (de soluções) alternativos, representados pelos nós OU da árvore e por arcos dirigidos para os nós mais recentes da computação, em que cada arco representa uma inferência lógica.

Existem dois tipos diferentes de nós OU: determinísticos e não determinísticos. Os nós determinísticos representam computações sem oportunidade para paralelismo e, numa árvore OU, distinguem-se por apenas darem origem a um único arco. Os nós não-determinísticos representam computações que podem ser feita de uma forma sequencial ou concorrente (possivelmente paralela) e, numa árvore OU, podem ser distinguidos por darem origem a vários (≥ 2) arcos.

Em [KW92] é desenvolvido este tópico do paralelismo OU e são, também, estudados alguns casos de linguagens lógicas que exploram este tipo de paralelismo.

3.4.2 Linguagens com paralelismo E

O paralelismo E genérico pode ser incomportável, devido à complexidade gerada pela gestão das dependências entre os diversos argumentos dos múltiplos golos de uma cláusula. O paralelismo E restrito (ou independente) foi desenvolvido, tal como o paralelismo E/OU (ver secção seguinte), como uma forma alternativa ao paralelismo genérico nos E. Neste modelo, de paralelismo E restrito, dois golos de uma conjunção só são resolvidos em paralelo se forem independentes.

Existem duas abordagens claramente distintas para este modelo: uma abordagem *estática* e uma *dinâmica*. Existem também sistemas híbridos, *i.e.* que combinam as abordagens estática e dinâmica.

A *abordagem estática* baseia-se numa análise estática do programa, *i.e.* em tempo de compilação, para a detecção dos fluxos de dados entre os vários golos do corpo de uma cláusula. A abordagem dinâmica faz testes durante a execução do programa, para determinar a viabilidade e aplicabilidade da execução dos golos em paralelo.

Em [KW92] é desenvolvido este tópico do paralelismo E e são, também, estudados alguns casos de linguagens lógicas que exploram este tipo de paralelismo.

3.4.3 Linguagens com paralelismo E/OU

A combinação de paralelismo E com paralelismo OU apresenta-se como o maior desafio para um modelo computacional de programas em lógica, visto que a exploração de paralelismo a todos os níveis do espaço de procura pode, se não for devidamente controlada, originar uma explosão de eventos concorrentes.

É reconhecido que a computação de um modelo genérico que suporte paralelismo E/OU genérico, será demasiado "pesada" para ser realizável. É, portanto, necessário optar por um compromisso que reduza o peso das computações desnecessárias, restringindo os nós que darão origem a computações concorrentes.

As linguagens que usam este tipo de paralelismo distinguem-se de acordo com a forma como limitam a explosão de computações, sendo umas mais adequadas a implementações em arquitecturas multiprocessadoras de memória partilhada e outras a arquitecturas multiprocessadoras de memória distribuída.

Os modelos de paralelismo E/OU restrito combinam uma forma de paralelismo E restrito com paralelismo OU, podendo este ser "ansioso" (*eager*) ou "ocioso" (*lazy*). O paralelismo OU "ocioso" apenas calcula soluções a pedido, uma de cada vez, enquanto o paralelismo OU "ansioso" não espera, começando imediatamente a determinar soluções.

Em [KW92] é desenvolvido este tópico do paralelismo E/OU e são, também, estudados alguns casos de linguagens lógicas que exploram este tipo de paralelismo.

3.4.4 Linguagens com paralelismo E em Cadeia

Este modelo de paralelismo de programas em lógica considera variáveis comuns, numa conjunção de golos, como canais de comunicação entre os golos. É necessário que os golos partilhem uma relação de produtor/consumidor para que possam ser executados em paralelo usando este modelo de comunicação. Este tipo de paralelismo pode ser utilizado como uma forma de implementar, em programação em lógica, modelos de computação reactiva, processos perpétuos e outros paradigmas de computação paralela.

Estas linguagens usam um mecanismo de sincronização entre os golos a serem executados em paralelo, baseado num mecanismo de controlo do fluxo de dados. Conseguem-se obter cadeias de

dados (*streams*) com recurso a termos parcialmente instanciados, partilhados entre o produtor e o consumidor.

Para definir qual é a direcção do fluxo de dados, fundamental para a sincronização entre os processos, é necessária uma metodologia de programação adequada, e/ou anotações que permitam identificar se o fluxo de dados, numa variável partilhada, é de entrada ou de saída. Durante a execução dos golos do corpo de uma cláusula, um golo não pode ser executado enquanto todos os seus argumentos de entrada não estiverem instanciados.

As linguagens baseadas em paralelismo E em Cadeia são normalmente conhecidas como "linguagens Não-Determinísticas de Escolha Única" (*Commited Choice Non-Deterministic languages*). Estas linguagens dizem-se de escolha única porque abandonaram o não-determinismo associado ao paralelismo OU, *i.e.* das múltiplas alternativas de uma cláusula que estão em condições de ser resolvidas, apenas uma é escolhida (de uma forma não-determinística). Quando uma das alternativas é escolhida, essa opção torna-se definitiva, não havendo qualquer possibilidade de retrocesso (ao contrário do que acontece no Prolog convencional). Assim, estas estratégias de pesquisa não são completas, *i.e.* não pesquisam exaustivamente todo o espaço de soluções.

Do ponto de vista da sincronização, estas linguagens exigem requisitos diferentes para determinar como os processos se irão sincronizar [Sha83]. Umas optam por utilizar anotações no programa, *i.e.* indicações explícitas de se as variáveis partilhadas entre os golos são de entrada ou de saída [CG83, CG84], enquanto outras optam por definir metodologias de programação, implícitas no código do programa [CHR].

Em [KW92] é desenvolvido este tópico do paralelismo E em Cadeia e são, também, estudados alguns casos de linguagens lógicas que exploram este tipo de paralelismo.

3.4.5 Linguagens concorrentes Orientadas para Processos Distribuídos

Até muito recentemente (1989) apenas uma das linguagens de programação em lógica que explora o paralelismo se poderia incluir nesta classe: o *Delta Prolog* [PN84, CFP89]. As linguagens orientadas para processos opõem-se às apresentadas até agora, no sentido de que estas aplicam combinações de *paralelismo E e OU* para extrair paralelismo de um programa em lógica, comunicando os vários processos, implicitamente, através de variáveis partilhadas, enquanto que as primeiras se baseiam em processos sequenciais que são criados explicitamente e que comunicam através da transmissão, também explícita, de mensagens.

Estas linguagens são as que apresentam maior diversidade na forma e comportamento dos processos durante a execução. Do ponto de vista de comunicação, estas podem ser caracterizadas segundo duas classes: comunicação unidireccional (a grande maioria) e comunicação bi-direccional. Do ponto de vista do retrocesso, estas linguagens podem dividir-se também em dois grupos: as que suportam o retrocesso distribuído e as que apenas suportam retrocesso local, *i.e.* uma comunicação é considerada como um predicado extra-lógico e, como tal, falha em retrocesso, não influenciando os demais processos do sistema.

O *Port Prolog* [CMC87], referido na secção 1.2.2, está inserido nesta classe de linguagens. As variantes propostas em [LB91] e nos capítulos 4 e 5 estão também nesta classe de linguagens.

Em [KW92] são estudados alguns casos de linguagens lógicas que exploram este tipo de paralelismo.

3.4.6 Linguagens concorrentes Orientadas para Objectos

Estas linguagens têm como objectivos principais:

i) trazer a declaratividade das linguagens lógicas para o modelo da programação orientada para objectos, e trazer os benefícios da estruturação — *e.g.* modularidade, encapsulamento de dados — das linguagens orientadas para objectos para o modelo das linguagens lógicas;

ii) permitir a expansão da concorrência em sistemas contituídos por múltiplos componentes, cada um modelado como um objecto — passivo ou activo — com um estado e conhecimento próprios e que comunica explicitamente com outros objectos através da invocação de predicados de interface, sendo esta comunicação suportada por uma troca de mensagens.

As duas principais abordagens correspondem a linguagens que só definem *objectos* como contruções de programação ou a linguagens que definem *objectos* como entidades passivas (semelhantes a módulos que encapsulam conjuntos de cláusulas), e processos como entidades activas que suportam a resolução de golos, nos contextos de cláusulas definidos pelos objectos que invocam.

Deste modo, é possível considerar esta classe de linguagens como uma forma evolutiva das linguagens incluídas nas classes anteriores, às quais se acescentam os mecanismos de estruturação característicos dos modelos orientados para objectos.

3.4.7 Conclusões

Da breve resenha apresentada nesta secção (3.4) sobre algumas variantes de exploração do paralelismo em linguagens lógicas, é possível observar-se que, dada a diversidade dos modelos, se torna inviável definir um único modelo de execução paralela, de nível intermédio, que satisfaça todos os requisitos postos por aquelas abordagens de paralelização.

3.5 Mecanismos de suporte à execução de modelos lógicos

Nesta secção discutem-se as principais noções que devem ser suportadas pelo nível intermédio sobre o qual se implementa um modelo de programação paralela em lógica. Restringe-se a discussão aos *modelos de linguagens orientados para processos distribúdos*, por serem estes os modelos que se pretendem suportar sobre a proposta apresentada nesta dissertação.

Há, em geral, três aspectos fundamentais a considerar:

i) um sistema de inferência que suporte o processo de avaliação de golos;

ii) funcionalidades para a gestão e controlo de processos criados para a avaliação paralela de golos;

iii) funcionalidades para a comunicação e sincronização entre processos.

Os modelos de linguagem em que estamos interessados são obtidos por extensões ao modelo do Prolog, sob a forma de predicados extra-lógicos, que suportam as funcionalidades ii) e iii). Deste modo a computação definida por cada processo de avaliação de um golo é representada por um processador virtual dedicado, que representa internamente o estado da computação associada às derivações produzidas a partir do golo inicial, na presença de um dado programa (conjunto de cláusulas).

A execução dos correspondentes passos de derivação é da responsabilidade de um executor de Prolog, que segue uma estratégia de computação sequencial, com pesquisa orientada em profundidade e baseada em retrocesso. Para a implementação do executor de Prolog, pode seguir-se uma abordagem baseada num interpretador ou num compilador que gera código intermédio para uma máquina abstracta especializada para o modelo de execução do Prolog.

As extensões referidas em ii) e iii) não modificam esta estratégia de pesquisa, e podem ser tornadas acessíveis ao executor, quer no caso do interpretador, *e.g. C-Prolog* [Per83], quer no caso da máquina abstracta, *e.g.* WAM [War83, War88].

Podem, deste modo, visualizar-se as seguntes camadas:

L – nível do modelo de programação paralela;

I – nível do modelo de suporte à execução;

S – nível do modelo de sistema de operação e arquitectura real.

O nível **L** é definido pelas abstracções de programação paralela em lógica, em relação ao qual a única hipótese que assumimos é ser baseada em processos distribuídos (ver secção 3.4.5), *i.e.* que não partilham variáveis lógicas, associados a fluxos independentes de resolução de golos, em espaços de trabalho e de cláusulas disjuntas.

O nível **S** corresponde ao ambiente de operação disponível sobre uma determinada arquitectura real.

O nível **I** é definido pelas três funcionalidades identificadas anteriormente, incluindo, em particular:

i) mecanismos de controlo de um executor sequencial de Prolog;

ii) mecanismos para a criação e gestão dinâmica de instâncias de executores independentes, cada um definido operacionalmente por uma máquina do tipo WAM, com espaços de memória distintos e privados, com um conjunto de registos próprios e com programas (conjunto de cláusulas) associados;

iii) mecanismos para a activação daqueles executores, iniciando processos de resolução de golos e para a obtenção das correspondentes soluções;

iv) mecanismos para a gestão comunicação entre executores, com base em troca de mensagens — interpretadas como termos Prolog — em modos de operação síncrona (bloqueante) ou assíncrona (não bloqueante); necessários para o envio e processamento de sinais, associados a uma noção de interrupção de um executor Prolog.

Diversos modelos podem ser considerados para contemplar os aspectos iii) e iv). Uma possibilidade é definir uma interface de comunicação de cada executor com o seu ambiente exterior, que generaliza os predicados de entrada/saída habituais no Prolog, de modo a contemplar a troca de mensagens e o envio[5] através de portas de comunicação — uma forma restrita de caixas de correio (*mailboxs*) — como se discutirá no capítulo 4.

Quanto ao aspecto ii), a abordagem mais simples consiste em associar uma instância independente de um executor de Prolog, com um espaço de cláusulas privado a cada processo de resolução de golos, sem que isto impeça a sua extensão futura para múltiplos processos de resolução de golos — *threads* — internamente a um mesmo executor.

Quanto ao ponto i), diversas alternativas são possíveis, dependendo principalmente das necessidades sentidas para dotar o nível intermédio com as potencialidades consideradas necessárias.

[5]Com base num mecanismo de envio e recepção de mensagens (termos Prolog) é possível implementar um outro de envio de golos e recepção de soluções.

3.6 Uma abordagem para extensões ao Prolog

A adição de predicados *extra-lógicos* à linguagem de programação Prolog retira-lhe a correspondência directa com a *Lógica das Cláusulas de Horn* (ver secção 3.3), mas dá-lhe funcionalidade necessárias, e em alguns casos fundamentais, para que seja viável a sua utilização corrente.

Uma forma de extensão possível da linguagem é através da adição de predicados de controlo da máquina de inferência (o executor do Prolog), do que o predicado *"!"* (*cut*), que permite ao programador limitar o espaço de procura da solução de um golo, é um exemplo clássico.

Uma segunda forma de extensão possível é disponibilizar, ao nível da linguagem, acesso aos recursos da arquitectura (através do sistema de operação da máquina) onde está a ser executado o programa, para possibilitar a interacção deste com o exterior. Os predicados *write/1* e *read/1* são um exemplo desta estratégia, que disponibilizam uma funcionalidade básica e necessária para o desenvolvimento de aplicações sobre esta linguagem.

Independentemente da classe em que se englobam as extensões feitas à linguagem, é condição importante que estas sejam independentes da arquitectura onde foram inicialmente implementadas e, se possível, sejam também independentes do sistema de operação usado na máquina em questão.

A aproximação apresentada na secção 3.4.5 (em que o modelo apresentado nesta dissertação está incluído), sugere a adição de predicados para suporte, entre outras construções, da comunicação explícita entre processos. Os novos predicados podem ser extra-lógicos, *e.g. Port Prolog*, mas não é forçoso que assim seja, *e.g. Delta Prolog*. Estes novos predicados deverão ser, também, tão independentes da arquitectura e do sistema de operação quanto possível, de forma a minorar o esfoço quando se tenta fazer o transporte da linguagem para uma nova arquitectura e/ou sistema de operação.

Uma forma de conseguir esta independência, é através da implementação de camadas intermédias, que localizem as dependências da arquitectura e do sistema de operação num grupo limitado e bem definido de funções. A linguagem é depois implementada sobre a camada intermédia, acedendo directamente ao sistema de operação para satisfação de serviços genéricos, *e.g.* operações de entrada/saída sobre ficheiros, e recorrendo à camada intermédia para satisfação dos serviços específicos, *e.g.* criação de processos.

3.7 Conclusões

Neste capítulo discutiram-se as principais abordagens de paralelização de programas em lógica. Identificaram-se as funcionalidades desejáveis para suportar modelos orientados para processos distribuídos e a sua integração em executores sequenciais de Prolog.

No capítulo 4 propõe-se um modelo que disponibiliza um conjunto de predicados adequados ao suporte das funcionalidades atrás referidas.

Capítulo 4

Proposta de um modelo de extensão ao Prolog

4.1 Introdução

Neste capítulo apresenta-se uma proposta de extensão da linguagem de programação Prolog com predicados extra-lógicos, que disponibilizam o acesso directo a funcionalidades de nível de sistema de operação — *e.g.* criação de processos, comunicação e envio de sinais — que normalmente não se encontram disponíveis em linguagens de programação em lógica, e em particular, no Prolog. Esta extensão ao modelo convencional do Prolog permite a utilização desta linguagem para a realização de aplicações para as quais não está vocacionada à partida, *e.g.* para aplicações de natureza distribuída em que se explora explicitamente o paralelismo.

Na próxima secção são apresentadas e caracterizadas as abstracções que fundamentam o modelo proposto, seguindo-se uma descrição pormenorizada da sintaxe e semântica procedimental dos novos predicados introduzidos na linguagem e, por fim, apresenta-se um exemplo que ilustra como resolver um simples problema, usando o modelo proposto.

4.2 Abstracções de concorrência e comunicação

O modelo que se propõe neste capítulo assenta em quatro abstracções básicas: processos, portas, mensagens e sinais. Estas abstracções diferem naturalmente das que normalmente associamos aos mesmos nomes, no contexto dos sistemas de operação, motivo pelo qual se apresenta, de seguida, uma caracterização das mesmas.

4.2.1 Processo

Os programas em Prolog podem ser compilados para uma máquina intermédia específica — denominada *máquina virtual*[1] — com um conjunto de instruções próprio e optimizada para uma execução eficiente, em termos temporais e espaciais, dos programas feitos nesta linguagem [War83, War88, AK90]. Apesar de existirem algumas implementações físicas (em *hardware* e *firmware*) das máquinas Prolog, na realidade a maneira mais simples de executar programas em Prolog, de forma eficiente e flexível, é através da implementação por *software* de uma *máquina virtual da WAM* e da compilação do código fonte em Prolog para o código intermédio dessa máquina virtual.

[1] Noção de *máquina virtual*: simulação de uma máquina e do seu conjunto de instruções com base numa outra máquina diferente.

Processo sequencial convencional

Designamos por *processo* a entidade básica que é responsável pela computação definida por um programa, sendo caracterizado pelo conjunto de recursos necessários à execução do programa — *e.g.* memória, dispositivos de entrada e saída — e por um fluxo de execução que utiliza esses recursos, através da interpretação e execução da sequência de instruções que formam o programa[2]. Um fluxo de execução, por sua vez, caracteriza-se pelo valor dos registos internos do CPU e pelo conjunto de dados em memória, num determinado instante (denomina-se a isto, o *estado do processo*).

Executor de programas Prolog

A unidade básica de computação, para o caso do modelo que se propõe, é aquilo a que se chama um *executor de Prolog*. Um executor de Prolog corresponde, na realidade, a uma instância da máquina virtual intermédia, *e.g.* WAM, que executa uma sequência de instruções resultantes da compilação de um programa Prolog. Os recursos de que o executor de Prolog necessita para controlar o processo de avaliação de golos, na presença de um conjunto de cláusulas, são definidos pela semântica da mencionada máquina virtual e não se revêem aqui (por falta de espaço), estando este tema desenvolvido em [AK90]. Numa implementação concreta, esses recursos são projectados (*mapped*) nos recursos reais disponibilizados pelo sistema de operação subjacente, *e.g.* registos de CPU, espaço de memória e canais de entrada/saída.

Deste modo, o estado do executor de Prolog é caracterizado pelo valor dos registos da sua máquina virtual e controlado pela lógica interna desta máquina, mas a representação física desse estado, a nível do ambiente do sistema de operação e da máqunia real, é controlada por um processo convencional[3].

No texto que se segue, sempre que não seja óbvio se o termo "processo" se refere a um *executor de Prolog* ou a um *processo convencional*, serão usados os termos "executor" ou "executor de Prolog" para identificar o primeiro e "processo" ou "processo convencional" para identificar o segundo.

A cada instância do executor está associado um único processo convencional (correspondência biunívoca). O sistema de operação subjacente afecta cada processo a um processador real do multicomputador, permitindo possivelmente a partilha de um mesmo processador real por múltiplos processos, sejam estes associados a executores de Prolog ou a executores de outro tipo de processos. Cada processo convencional é uma entidade independente das demais, com segmentos de código, dados e pilha e com um fluxo de execução próprios.

Do ponto de vista do modelo, é indiferente se existe apenas um ou mais do que um executor de Prolog em cada processador. Do ponto de vista do desempenho, a única consequência é a redução da velocidade de processamento em cada executor, devida à partilha do tempo do processador por todos os processos que residem num mesmo nó.

Assume-se que o escalonamento dos processos que residem num mesmo processador é da responsabilidade de uma entidade externa ao executor, normalmente o sistema de operação da máquina[4].

---

[2] Em sistemas modernos, é frequente encontrar sistemas onde é possível existir mais do que um fluxo de interpretação e execução dentro de um único processo, em que a cada um desse fluxos se chama "processo leve" ou *thread*. Esta situação, em que um processo pode ter mais do que um fluxo de execução, não é considerada no modelo proposto, embora este seja compatível com uma extensão nesse sentido.

[3] Nos sistemas com *threads* tem-se, em geral, um único processo convencional a controlar múltiplas instâncias de máquinas virtuais e a gerir as correspondências estre os estados Prolog e as suas representações físicas.

[4] No caso de uma arquitectura baseada em *Transputers* o escalonamento de processos é, normalmente, suportado pelo *firmware* do próprio processador [Lim88b].

4.2.2 Porta de comunicação

Num sistema em que a comunicação entre processos é baseada em mensagens, é possível usar diferentes mecanismos para especificar quem é o emissor e/ou o receptor da mensagem, podendo apresentar-se como um mecanismo de nomeação directa, indirecta, ou como uma combinação de ambos (ver capítulo 2).

A uma *porta* corresponde um canal de comunicação protegido (implementado como uma fila limitada de mensagens) para onde as mensagens podem ser enviadas e guardadas até serem recebidas, apresentando-se assim como uma abstracção do destinatário da comunicação. Cada porta é uma entidade única no sistema (ambiente de execução) e pode ser identificada por um nome (normalmente uma cadeia de caracteres).

A porta apresenta-se como um mecanismo intermédio na especificação dos destinatário de uma comunicação entre dois processos. Este mecanismo de nomeação indirecta permite o desenvolvimento de sistemas com múltiplos processos concorrentes que cooperam entre si, evoluindo na resolução de um problema comum, sem que nenhum deles tenha conhecimento ou acesso directo aos demais processos.

Uma das grandes vantagens que este tipo de nomeação apresenta é a de permitir que as aplicações não necessitem de ter conhecimento da topologia do *hardware* da arquitectura, facilitando assim o transporte das mesmas aplicações para outros tipos de arquitecturas com diferentes topologias, bem como a migração das portas para processos diferentes, nos mesmos ou noutros processadores, quando se exige uma reconfiguração dinâmica do sistema.

Um processo, ao criar uma porta, passa a ser o seu dono e obtém um descritor da mesma. Este será o único descritor em todo o sistema que poderá ser usado para receber mensagens da porta, *i.e.* só o processo criador da porta pode receber mensagens através dela. Um único processo pode criar várias portas, que ficam activas simultaneamente, correspondendo a vários canais distintos para recepção de comunicações.

Esta abstracção, em que um processo pode ter múltiplas portas que lhe ficam associadas, permite também diversas interpretações interessantes para a implementação de modelos de comunicação de mais alto nível. Por exemplo, uma porta pode ser assimilada a um ponto de entrada num processo, e utilizada para suportar uma interface de comunicação com o exterior, num contexto de um modelo orientado para objectos, através da qual chegam pedidos de invocação dos diferentes métodos de um objecto. Por outro lado, é possível associar uma facilidade de entrega urgente de mensagens a uma porta, desencadeando a interrupção do fluxo do processo ou a invocação de um *thread* para o processamento da mensagem.

A flexibilidade deste conceito e a sua adequação para o suporte de sistemas de múltiplos processos distribuídos, justificam a opção pela sua inclusão no modelo proposto. Trata-se, por outro lado, de uma abstracção suficientemente comum ao nível dos sistemas de operação [Tan92], para que a sua implementação, em muitos sistemas específicos, não implique penalizações de desempenho na execução de programas.

Um processo pode "abrir" uma porta (conhecendo o seu nome único e global), obtendo assim um descritor que lhe concede acesso indirecto ao processo que a criou, para quem pode enviar mensagens através dessa porta. É possível a um processo "abrir" várias vezes a mesma porta, com a consequente obtenção de múltiplos descritores locais dessa mesma porta.

4.2.3 Mensagem

Uma mensagem corresponde a uma sequência de *bytes* que podem, a um nível de abstracção superior, corresponder a formas de estruturação da informação. Uma mensagem é sempre enviada por um

processo, denominado *o emissor* e é recebida por um outro, denominado *o receptor*. Em geral, a forma como o emissor especifica quem é o receptor pode variar de sistema para sistema.

No modelo que se apresenta, a mensagem é a unidade lógica de base para troca de informação entre executores de Prolog, correspondendo a uma sequência de caracteres que são interpretados como um termo Prolog. O receptor da mensagem é indicado através de um mecanismo de nomeação indirecta, orientado para portas, em que o emissor dirige a mensagem para uma porta, sendo através desta que o receptor (o dono da porta) receberá a mesma.

Um aspecto central, na definição de um modelo de comunicação entre processos distribuídos, reside na semântica das operações de envio e de recepção de mensagens, quanto à forma de sincronização entre emissores e receptores e ao controlo do não-determinismo na chegada de mensagens.

Quanto à primeira dimensão do problema, as opções tomadas no modelo (ver secção 4.3) permitem a máxima flexibilidade: tanto a opção síncrona ou bloqueante como a opção assíncrona ou não bloqueante, são possíveis, por indicação explícita no programa e aplicáveis, quer ao envio quer à recepção de mensagens.

O modelo contempla, para além disso, uma integração do mecanismo de comunicação por mensagens e do mecanismo de interrupções de executor de Prolog, tal como foi proposto inicialmente em [CMC87], mas com funcionalidade acrescida. Esta integração possibilita uma forma de controlo do não-determinismo, que promove um estilo de programação reactiva[5], controlado por eventos, *i.e.* chegadas de mensagens, que desencadeiam a invocação de predicados definidos no programa Prolog (ver secção 4.2.4).

Quanto à segunda dimensão do problema, para suportar o controlo explícito de não-determinismo, o modelo inclui uma construção que permite a um processo aguardar a recepção de uma mensagem numa de entre um conjunto especificado de portas. O meio exterior ao processo determina quais as portas "activas", *i.e.* aquelas às quais chegam mensagens, e o mecanismo interno do executor selecciona, de forma não determinística, uma das portas "activas". Esta construção é fundamental para o suporte de construções de mais alto nível, do tipo do operador de escolha não-determinística do *CSP* [Hoa78] ou do operador "*::*" do *Delta Prolog* [PMCA86].

4.2.4 Sinal

Um sinal é uma notificação, dirigida a um processo, indicando que ocorreu um evento — os sinais podem ser considerados "interrupções por *software*" — havendo vários tipos de sinais diferentes, para vários tipos de eventos também diferentes.

Geralmente os sinais e o seu tratamento ocorrem de forma assíncrona com a computação que está a ser efectuada, ou seja, um processo não sabe antecipadamente quando é que vai ocorrer um sinal, para se predispor a tratá-lo. O sinal tem, como destino, um processo do sistema e como origem um outro processo do mesmo sistema.

O envio de um sinal a um processo provoca neste uma suspensão da computação actual e a invocação assíncrona de uma rotina alternativa que irá processar esse sinal[6]. Quando esta rotina terminar o processamento, será retomado o fluxo de execução anterior, no ponto em que ia quando foi suspenso.

Durante a execução de uma "rotina de tratamento de sinais", a recepção de um sinal pode originar um de três comportamentos distintos:

[5] Um processo pode ser encarado como um agente que reage à recepção de mensagens, por invocação automática de predicados dedicados ao seu processamento. Este comportamento é similar ao de um objecto que exibe concorrência interna no processamento dos predicados de invocação dos seus métodos.

[6] A suspensão pode ser atrasada por um certo intervalo de tempo, necessário para que o processo seja interrompido de forma coerente.

i) o novo sinal é ignorados;

ii) a actual "rotina de processamento de sinais" é suspensa e é activada uma nova rotina para processar o sinal recebido;

iii) o novo sinal é guardado e será atendido quando a actual "rotina de processamento de sinais" terminar.

A utilização da alternativa i) pode ser problemática, por ser posível que alguns sinais se percam, *i.e.* sejam enviados mas não sejam recebidos pelo destinatário, sem que nenhum dos processos disso tenha conhecimento. A alternativa ii), é das três apresentadas a mais genérica, pois permite associar níveis de prioridade aos sinais, mas pode complicar seriamente o processo de implementação, por obrigar a que todas as rotinas de tratamento de sinais sejam reentrantes. Na alternativa iii), que se apresenta como um compromisso entre i) e ii), os sinais são processados sequencialmente e apresentam-se como um mecanismo de comunicação fiável e de fácil implementação.

No modelo proposto, o envio de um sinal corresponde a enviar um termo Prolog para um executor de Prolog, existindo, portanto, um número ilimitado de variantes possíveis de sinais. O executor que recebe o sinal suspenderá o processamento (inclusivé se estiver bloqueado num envio ou recepção de uma mensagem) e invocará um predicado, se previamente indicado pelo programador, ao qual é passado como parâmetro o termo recebido. Após o processamento, a execução será retomada no ponto em que foi interrompida.

Se um receptor está a tratar um sinal, *i.e.* a resolver um golo enviado num sinal, não atende mais sinais, só voltando a ficar receptivo a sinais quando o processamento do anterior terminar. Como se pretende que os sinais que cheguem nestas circunstâncias não se percam, é necessário que o sistema que suporta a implementação do modelo, guarde os sinais pendentes. Em alternativa é necessário que seja criada uma fila (no executor) de sinais recebidos e ainda não tratados e, sempre que o executor termina de tratar um sinal, trate o mais antigo dos sinais nessa fila antes de voltar ao fluxo de execução principal.

4.3 Definição dos predicados de interface

Neste trabalho propõe-se um conjunto de predicados que, na sua essência, definem uma interface de um executor de Prolog com uma máquina de controlo de concorrência e comunicação. Pretende-se, com esta interface, oferecer um nível de independência dos executores face ao nível do sistema de operação.

A interface é definida sob a forma de um conjunto de predicados de controlo (predicados extra-lógicos). A sua definição operacional faz-se em termos das abstracções de concorrência — executores e processos — e de comunicação — portas, mensagens e sinais — apresentados na secção 4.2.

Cada um dos predicados, abaixo apresentados, está englobado numa das seguintes categorias:

i) comunicação por mensagens;

ii) gestão de executores de Prolog;

iii) geração e tratamento de sinais;

iv) controlo do estado do executor Prolog;

v) informação do estado dos executores de Prolog e da configuração do sistema.

Como as acções efectuadas por todos os predicados, a seguir apresentados, são irreversíveis, quando em execução para trás, *i.e.* em retrocesso (*backtracking*), os predicados falham sempre.

Os nomes dos novos predicados são palavras reservadas, tendo quase todos o prefixo "**sys_**". Aqueles que têm o prefixo "**sys_**" são sempre bem sucedidos quando em execução para a frente, devolvendo 0 (zero) em `Status` se realizaram a acção devida, ou devolvendo um número positivo, correspondente ao código do erro, caso tal não tenha sido possível.

Nas restantes secções deste capítulo e no capítulo 5, todos os nomes de argumentos dos predicados apresentados têm o prefixo "+" ou "−", indicando que são argumentos de entrada e de saída, respectivamente. Por exemplo, em sys_at_halt(+Goal, −Status) o argumento `Goal` é de entrada, tendo que estar instanciado na invocação, e o argumento `Status` é de saída, tendo de corresponder a uma variável livre na mesma invocação.

4.3.1 Predicados de gestão de portas de comunicação

Os predicados seguintes indicam os modos de criação, inicialização (ou abertura), fecho e destruição de portas de comunicação. As definições dos predicados aqui apresentadas correspondem fielmente à versão actualmente implementada no protótipo, pelo que se fazem, pontualmente e em certos casos, referências a valores particulares de alguns argumentos, válidos na "versão actual".

A evolução futura dos modelos poderá ditar extensões aos valores de certos argumentos, em particular os que correspondem a modos opcionais, indicados por `Flags`. Optou-se por fazer uma apresentação completa, apesar de resultar menos sucinta do que uma descrição que omitisse estes pormenores.

sys_create_port(+Name, +Flags, −Port, −Status)

Cria uma porta com o nome indicado em `Name`. Este nome tem de ser único no sistema, senão o predicado devolve um código de erro em `Status`, com um dos valores abaixo indicados. Na versão actual o argumento `Flags` apenas admite como valor o átomo **p_noflag**. No caso de `Status` retornar um valor diferente de 0 (zero), `Port` não tem qualquer informação útil. Caso contrário, `Port` devolve um descritor (número inteiro) da porta, a utilizar nas operações de recepção de mensagens da porta.

O executor que criou a porta é o único que pode receber mensagens através dela (ver sys_recv_msg/4 na secção 4.3.2).

A terminação de um executor associado a uma instância da máquina Prolog desencadeia automaticamente todas as acções necessárias para a destruição (ver sys_destroy_port/3 nesta secção) das portas a ele associadas.

O modelo de designação de portas adoptado é convencional: cada porta tem um nome global, único no universo de executores, e pode ter múltiplos nomes locais — correspondentes aos valores devolvidos em `Port` — a cada executor, consoante o número de operações de abertura que se invoquem.

Este predicado pode retornar em `Status` um dos seguintes códigos de erro:

`EOPTION` – O argumento `Flags` contém um valor inválido ou que não é admitido neste predicado;

`EFULL` – Não é possível obter mais descritores de portas;

`EEXIST` – Já existe uma porta com o mesmo nome que o indicado.

> **Exemplo:** "**sys_create_port(foo, p_noflag, P, S)**" Será criada uma porta com o nome `foo`, devolvendo em `P` o identificador da porta e em `S` o resultado da operação (sucesso ou não).

sys_open_port(+Name, +Flags, −Port, −Status)

Este predicado tem o efeito de colocar uma porta, previamente criada, num estado susceptível de permitir a sua posterior utilização para o envio de mensagens. Retorna um descritor (número inteiro), que representa um "canal" que se abre para a comunicação através da porta, similar aos canais lógicos para acesso aos ficheiros, normais num sistema Prolog.

Apesar das operações de criação e abertura de porta não afectarem o estado corrente dos canais *standard* de entrada e saída de um executor de Prolog, é possível associar canais de ficheiros a canais de mensagens, obtendo-se uma generalização real dos mecanismos de entrada/saída e conseguindo-se tornar a comunicação por mensagens tão transparente quanto o acesso a ficheiros.

O predicado abre uma porta com o nome indicado em `Name` e retorna um descritor da mesma em `Port`. O argumento `Flags` pode tomar um dos valores **p_wait** ou **p_nowait**.

O argumento `Flags` com o valor **p_wait** indica que o executor invocador deverá ser bloqueado[7] até que a porta em questão (com o nome `Name`) seja criada[8]. O argumento `Flags` com o valor **p_nowait** indica que, caso a porta em questão (com o nome `Port`) não exista, deverá retornar imediatamente um código de erro em `Status`.

É necessário abrir a porta para se poderem enviar mensagens para ela (ver sys_send_msg/4 na secção 4.3.2). Uma porta pode ser aberta várias vezes pelo mesmo executor ou por executores diferentes, obtendo-se de cada vez um descritor diferente e independente dos demais existentes.

Este predicado pode retornar em `Status` um dos seguintes códigos de erro:

`EOPTION` – O argumento `Flags` contém um valor inválido ou que não é admitido neste predicado;

`EFULL` – Não é possível obter mais descritores de portas;

`ENOENT` – Não existe nenhuma porta no sistema com o nome indicado.

> **Exemplo:** "**sys_open_port(bar, p_wait, P, S)**" Espera que a porta com o nome `bar` seja criada e abre-a, localmente ao executor invocador, devolvendo em `P` o identificador da porta e em `S` o resultado da operação (sucesso ou não).

sys_close_port(+Port, +Flags, −Status)

Fecha a porta associada ao descritor `Port`. O argumento `Flags` apenas admite como valor o átomo **p_noflag**.

Para ser permitido fechar uma porta, o descritor `Port` tem de ser um dos obtidos, por este executor, na invocação do predicado sys_open_port/4, senão o predicado devolve erro em `Status`. O facto de se fechar uma porta não impede que posteriormente se volte a abri-la (com sys_open_port/4).

Fechar uma porta não tem qualquer efeito sobre as mensagens nela pendentes, *i.e.* as mensagens ainda não recebidas pelo destinatário poderão ser recebidas mais tarde.

Este predicado pode retornar em `Status` um dos seguintes códigos de erro:

`EOPTION` – O argumento `Flags` contém um valor inválido ou que não é admitido neste predicado;

`EBADF` – O argumento `Port` não corresponde a um descritor válido;

[7] O bloqueio do executor corresponde à suspensão do processo de resolução de golos corrente, com preservação do estado do executor de Prolog.

[8] Se o executor invocador é também o suposto criador da porta e ainda não invocou sys_create_port/4, verifica-se um bloqueio interno, que não é identificado nem tratado pelo sistema.

`EPERM` – O descritor `Port` não dá permissões para fechar a porta (porque foi obtido com sys_create_port/4).

Exemplo: "**sys_close_port(4, p_noflag, S)**" Fecha o descritor de porta `4`, devolvendo em `S` o resultado da operação (sucesso ou não).

sys_destroy_port(+Port, +Flags, −Status)

Destrói a porta do sistema associada ao descritor `Port`. O argumento `Flags` apenas admite como valor o átomo **p_noflag**. Apenas quem criou a porta é que a pode destruir, *i.e.* o descritor `Port` tem de ser o obtido na invocação do predicado sys_create_port/4, senão o predicado devolve erro em `Status`.

A porta só é efectivamente destruída quando já não existirem descritores da porta abertos para além de `Port`, mas após a execução deste predicado não é mais possível abrir a porta (com sys_open_port/4). Quando a porta é efectivamente destruída, todas as mensagens nela pendentes são eliminadas.

Este predicado pode retornar em `Status` um dos seguintes códigos de erro:

`EOPTION` – O argumento `Flags` contém um valor inválido ou não admitido neste predicado;

`EBADF` – O argumento `Port` não corresponde a um descritor válido;

`EPERM` – O descritor `Port` não dá permissões para destruir a porta (porque foi obtido com sys_open_port/4).

Exemplo: "**sys_destroy_port(4, p_noflag, S)**" Destrói a porta associada ao descritor `4`, devolvendo em `S` o resultado da operação (sucesso ou não).

4.3.2 Predicados de controlo de comunicação

Os predicados seguintes definem o modo de utilização das portas para o envio, recepção simples de mensagens e recepção não determinística e disjuntiva de mensagens de um conjunto de portas.

sys_send_msg(+Port, +Flags, +Term, −Status)

Envia uma cópia do termo Prolog indicado em `Term` para a porta indicada em `Port`. Se `Term` incluir variáveis livres, elas corresponderão a novas variáveis, também livres, no termo recebido pelo destinatário. O campo `Flags` indica se o envio deve ser síncrono, tomando o valor **p_wait**, ou assíncrono, tomando o valor **p_nowait**.

Os termos oriundos de um mesmo emissor são entregues no destinatário pela ordem de emissão.

Este predicado pode retornar em `Status` um dos seguintes códigos de erro:

`EOPTION` – O argumento `Flags` contém um valor inválido ou não admitido neste predicado;

`EBADF` – O argumento `Port` não corresponde a um descritor válido;

`EPERM` – O descritor `Port` não dá permissões para enviar mensagens (porque foi obtido com sys_create_port/4).

Exemplo: "**sys_send_msg(2, p_wait, f(o(o)), S)**" Espera que o dono da porta associada ao descritor `2` esteja pronto para receber a mensagem. Depois envia-lhe a estrutura `f(o(o))` e devolve em `S` o resultado da operação (sucesso ou não).

sys_recv_msg(+Port, +Flags, −Term, −Status)

Instancia `Term` com o termo Prolog recebido da porta indicada em `Port` (pelo que `Term` tem que ser uma variável livre na altura da invocação do predicado). O argumento `Flags` indica qual o comportamento do predicado caso não exista nenhuma mensagem na porta especificada no momento da invocação: toma o valor **p_wait** caso o invocador deva ficar bloqueado até que a mensagem chegue, ou toma o valor **p_nowait** caso este não deva ficar bloqueado. Nesta última situação, `Term` virá instanciado com o átomo reservado **$no_message**.

Os termos oriundos de um mesmo emissor são entregues no destinatário pela ordem de emissão.

Este predicado pode retornar em `Status` um dos seguintes códigos de erro:

`EOPTION` – O argumento `Flags` contém um valor inválido ou que não é admitido neste predicado;

`EBADF` – O argumento `Port` não corresponde a um descritor válido;

`EPERM` – O descritor `Port` não dá permissões para receber mensagens (porque foi obtido com sys_open_port/4).

> **Exemplo:** "**sys_recv_msg(3, p_wait, T, S)**" Espera que alguém envie um termo para a porta associada ao descritor `3`, devolvendo em `T` o termo recebido e em `S` o resultado da operação (sucesso ou não).

sys_select_port(+Port_list, +Flags, −Port, −Status)

Testa se existe alguma mensagem pendente, *i.e.* pronta a ser recebida, em alguma das portas indicadas na lista de identificadores locais de portas `Port_list`. O argumento `Flags` indica se o executor deve esperar, com suspensão do processo de resolução de golos, até que exista uma mensagem em alguma das portas (caso `Flags` tenha o valor **p_wait**), ou se deve retornar imediatamente (caso `Flags` tenha o valor **p_nowait**).

Se não existem mensagens pendentes em nenhuma das portas indicadas e `Flags` tomou o valor **p_nowait** na invocação do predicado, `Port` retorna o átomo reservado **$p_noport**. Se existirem mensagens pendentes em apenas uma das portas indicadas em `Port_list`, é devolvido em `Port` o descritor da mesma. Se existirem mensagens pendentes em mais do que uma das portas indicadas em `Port_list`, o descritor a retornar em `Port` é escolhido de forma não-determinística (dependente do sistema e/ou da implementação). No caso de `Status` retornar um valor diferente de 0 (zero), `Port` não tem qualquer informação útil.

Este predicado apenas testa se existe, ou não, uma mensagem pronta a ser recebida numa das portas indicadas. A mensagem não é consumida, devendo, para tal, ser invocado sys_recv_msg/4 com o descritor de porta retornado em `Port`. Não há qualquer hipótese de haver corridas não determinísticas entre dois processos, porque só há um receptor para cada porta: o seu dono.

Este predicado pode retornar em `Status` um dos seguintes códigos de erro:

`EOPTION` – O argumento `Flags` contém um valor inválido ou não admitido neste predicado;

`EBADF` – Um dos descritores de portas indicados em `Port` não é válido;

`EPERM` – Um dos descritores de portas indicados em `Port_list` não dá permissões para receber mensagens (porque foi obtido com sys_open_port/4).

> **Exemplo:** "**sys_select_port([2,8,4,3], p_wait, P, S)**" Bloqueia o invocador até que alguma das portas `2`, `3`, `4` ou `8` tenha uma mensagem em condições de ser recebida, devolvendo um destes identificadores (o descritor local de uma porta com uma mensagem) em `P` e o resultado da operação (sucesso ou não) em `S`.

4.3.3 Predicados de geração e tratamento de sinais

Este mecanismo de comunicação apresenta-se como complememtar da comunicação por mensagens. Devido às suas características de tratamento assíncrono, deve ser usado com cuidado, sendo particularmente indicado para a notificação de acontecimentos excepcionais no sistema (ver secção 4.2.4).

Os predicados a seguir expostos permitem o envio e tratamento de sinais, ao nível do Prolog. Com eles, um executor de Prolog pode enviar sinais a outro e definir predicados Prolog como "rotinas de tratamento de sinais", que serão invocados automaticamente sempre que este receba um sinal.

sys_send_sig(+Eid, +Term, −Status)

Envia, de modo assíncrono, um sinal `Term`, que é na realidade um termo Prolog, para o executor identificado por `Eid` (ver secção 4.3.4).

O destinatário do sinal deverá ter associado, previamente, um predicado ao tratamento do sinal recebido (ver sys_sig_handler/2 nesta secção) ou então este será ignorado. Quer os sinais sejam tratados pelo destinatário, quer este os ignore, o emissor não é notificado do facto.

Este predicado pode retornar em `Status` um dos seguintes códigos de erro:

`ENOTPROCESS` – O argumento `Eid` não corresponde a um identificador de executor válido.

> **Exemplo:** "**sys_send_sig(443,foo(b,a,r),S)**" O termo `foo(b,a,r)` será enviado ao executor com o identificador `443`, devolvendo em `S` o resultado da operação.

set_sig_handler(+Functor/+Arity, +Goal)

Define o predicado `Goal` como sendo o responsável pelo tratamento dos sinais indicados em `Functor/Arity`. `Goal` será o nome de um predicado com a mesma aridade que os sinais que vai tratar, *i.e.* com aridade `Arity`, e os seus argumentos serão unificados com os do sinal recebido. Devolve em `Status` o resultado da operação (sucesso ou não)[9].

A definição de um predicado, como responsável pelo tratamento de um determinado tipo de sinal, elimina qualquer outra definição que exista anteriormente para o mesmo tipo de sinal.

O predicado de tratamento de sinais não tem qualquer limitação quanto ao seu tempo de execução, pode fazer uso de qualquer dos predicados que recorrem a chamadas ao sistema e não é interrompido por outro sinal que entretanto chegue. No entanto, o novo sinal será tratado imediatamente a seguir ao anterior e antes de retomar o fluxo principal de execução.

Se a unificação dos argumentos do sinal com os do predicado por ele citado falhar ou não for encontrada nenhuma solução para o dito predicado, o predicado de tratamento de sinais falha (sem mais consequências). Isto significa que o processo de resolução, no destinatário do sinal, é retomado no ponto em que fora interrompido, e que o emissor do sinal (cuja invocação de sys_send_sig/3 já fora bem sucedida) não é notificado do falhanço do predicado de tratamento no destinatário. Uma alternativa poderia ser definir sys_send_sig/3 como uma primitiva síncrona, que bloqueasse o emissor do sinal até que a activação do predicado de tratamento fosse bem sucedida ou, no caso de falhanço, devolvesse um código de erro ao emissor.

O tratamento de um sinal termina quando é encontrada a primeira solução para o predicado que lhe está associado. Como não há variáveis partilhadas entre a resolvente corrente e o predicado de

[9] Entenda-se como resultado da operação o resultado da invocação do predicado set_sig_handler/2 e não do predicado `Goal` (que só será invocado posteriormente, após a recepção de um sinal do tipo indicado).

tratamento de sinais, no destinatário, a execução deste último não pode afectar o estado da resolvente corrente, no ponto em que a execução foi interrompida, pelo que o efeito de um predicado de tratamento de sinais pode ser:

i) fazer assert/1 e/ou retract/1 de cláusulas no espaço do receptor;

ii) interactuar com outros processos via troca de mensagens, eventualmente com o próprio emissor do sinal;

iii) parar a execução (via halt/0) ou invocar um outro predicado de controlo.

> **Exemplo:** "**set_sig_handler(foo/2, bar)**" Sempre que o executor invocador receber um sinal do tipo `foo/2`, *e.g.* foo(a1, b2), invocará o predicado bar(X,Y), em que os dois argumentos `X` e `Y` serão unificados com os dois argumentos do sinal, *e.g.* bar(a1, b2).

unset_sig_handler(+Functor/+Arity)

A invocação deste predicado desfaz a associação (obtida com set_sig_handler/2) dos sinais do tipo indicado com um predicado do sistema.

A partir da invocação deste predicado, os sinais do tipo indicado serão ignorados, pois não está definido nenhum predicado para os tratar.

> **Exemplo:** "**unset_sig_handler(foo/2)**" A partir da invocação deste predicado, os sinais do tipo `foo/2` passam a ser ignorados, por falta de "rotina de tratamento de sinais".

sig_disable(+Functor/+Arity)

Após a invocação deste predicado, todos os sinais recebidos que tenham o mesmo nome e aridade que os especificados por `Functor/Arity` são suspensos, *i.e.* ficam pendentes. Os sinais pendentes num executor serão tratados assim que este reactivar o tratamento desse tipo de sinais (com sig_enable/1).

Este predicado só deve ser usado para suspender, temporariamente, o processamento de um determinado tipo de sinais e para os quais existe uma rotina de tratamento definida. Se se pretender ignorar definitivamente os sinais de um determinado tipo, dever-se-á usar o predicado unset_sig_handler/1 (já apresentado).

A invocação deste predicado para um sinal para o qual não exista uma rotina de tratamento especificada é ignorada, não tendo, portanto, qualquer efeito.

> **Exemplo:** "**sig_disable(foo/3)**" O executor invocador suspenderá o tratamento dos sinais do tipo `foo/3`, não invocando a respectiva função de tratamento quando estes são recebidos.

sig_enable(+Functor/+Arity)

Permite que os sinais com o mesmo nome e a mesma aridade que os especificados por `Functor/Arity` sejam atendidos. A invocação deste predicado é ignorada caso não tenha sido definida, previamente, uma rotina de tratamento para esse tipo de sinal (com set_sig_handler/1).

O uso deste predicado só é necessário para reactivar sinais temporariamente inibidos (com sig_disable/1). Se um sinal já está a ser considerado, uma segunda invocação deste predicado para o mesmo tipo de sinal é ignorada.

Exemplo: "**enable_sig(foo/3)**" O executor invocador passará a receber os sinais do tipo `foo/3` e a invocar a respectiva função de tratamento. Caso existam sinais pendentes, estes serão imediatamente tratados.

sig_alarm(+Time, +Term)

A invocação deste predicado provocará a geração de um sinal `Term` que será enviado ao próprio executor invocador. Esse sinal será gerado `Time` segundos depois da invocação[10] (`Time` é especificado através de um número inteiro ou decimal).

Exemplo: "**sig_alarm(1.5, foo(bar))**" A invocação deste predicado fará com que o sistema gere um sinal do tipo `foo(bar)` dentro de 1.5 segundos, aproximadamente, dirigido ao executor invocador.

wait_for_sig(+Functor/+Arity)

Suspende o processo invocador até que este receba um sinal do tipo indicado. Qualquer sinal que seja recebido enquanto o executor está suspenso neste predicado e que não seja do tipo indicado, fica pendente se houver uma rotina de tratamento definida e é ignorado caso contrário.

Exemplo: "**wait_for_sig(foo(b,a,r))**" O executor invocador ficará suspenso neste predicado até que receba um sinal do tipo `foo(b,a,r)`.

4.3.4 Predicados de gestão dos executores de Prolog

Este conjunto de predicados permite gerir dinamicamente os executores que constituem o ambiente de execução paralela, criando ou destruindo instâncias de executores de Prolog nos vários nós de um multicomputador.

Qualquer executor de Prolog pode criar uma nova instância de um executor de Prolog, no mesmo ou em outro nó. Estes dois executores de Prolog são dois processos totalmentes independentes entre si, não partilhando quaisquer relações lógicas, *e.g.* relação pai/filho, nem espaços de endereçamento.

sys_create_wam(+Node, +Flags, +In_dev, +Out_dev, +Cmd_line, −Eid, −Status)

Cria uma instância de um executor de Prolog (WAM[11]) no nó `Node` devolvendo um identificador único da mesma em `Eid`. Se não for possível criar o novo processo no nó especificado,*e.g.* `Node` não designa um nó válido do sistema, retorna um código de erro em `Status`.

Na versão actual o argumento `Flags` apenas admite como valor o átomo **p_noflag**.

Os argumentos `In_dev` e `Out_dev` indicam, de forma independente entre si, quais são os dispositivos de entrada e de saída do novo executor de Prolog a criar, podendo indicar um dispositivo de entrada/saída do ambiente de operação do computador ou então uma porta do sistema. Para indicar um dispositivo físico do computador o argumento deverá ser uma estrutura

[10] Admitindo que a granularidade do relógio é dependente do sistema, o tempo especificado será aproximado tanto quanto possível.

[11] No caso do protótipo desenvolvido.

com a forma `dev(+System_device)`, onde `System_device` identifica o dispositivo de entrada/saída; para indicar uma porta do sistema o argumento deverá ser uma estrutura com a forma `port(+Port_name)`, onde `Port_name` indica o nome da porta.

Se se indicar uma porta como dispositivo de entrada, ela será criada automáticamente durante a fase de inicialização do executor de Prolog (pelo que não deverá já existir outra porta com o mesmo nome). No caso de se indicar uma porta como dispositivo de saída do novo executor de Prolog, assume-se que ela foi (ou será) criada por um outro executor, pelo que o novo executor a vai tentar abrir com sys_open_port/4, esperando que ela seja criada caso não exista ainda.

O argumento `Cmd_line` é um átomo com argumentos de linha de comando para o executor a criar, caso este os aceite.

Na figura 4.1 é apresentado o ciclo principal de um executor de Prolog. Salienta-se a sua generalidade pelo facto de os canais *standard* de entrada e saída (*stdin* e *stdout*) tanto poderem corresponder a ficheiros (o que inclui o acesso à consola) como a portas do sistema.

```
:- repeat,
       read( Goal ),              % Recebe do cliente um golo "Goal"
                                  %   para resolver, pelo <stdin>, que
                                  %   pode ser um terminal ou uma porta
       try( Goal, Solution ),     % Procura uma solução para o golo
       write( Solution ),         % Envia a solução para o cliente, pelo
                                  %   <stdout>, que pode ser um terminal
                                  %   ou uma porta
   fail.                          % Procura todas as soluções por
                                  % retrocesso e depois espera por novos
                                  % pedidos
```

Figura 4.1: Ciclo principal de um "executor de Prolog"

Este predicado pode retornar em `Status` um dos seguintes códigos de erro:

`EOPTION` – Um dos argumentos `Flags`, `In_dev` ou `Out_dev` contém um valor inválido ou não admitido neste predicado;

`EBADNODE` – O argumento `Node` indica um nó inválido no sistema.

> **Exemplo:** "**sys_create_wam(5, p_noflag, dev('/dev/tty3'), port(foo_bar), '-q', P, S)**" Cria uma instância do executor (WAM) no nó `5`. Este executor receberá o argumento de linha de comando '-q', e utilizará como canal principal de entrada o dispositivo `/dev/tty3` e como canal principal de saída a porta `foo_bar`. Devolve em `P` o identificador único associado ao novo executor e em `S` o resultado da operação (sucesso ou não).

sys_destroy_wam(+Eid, −Status)

Destrói o executor de Prolog identificado por `Eid`. O processamento que este esteja a fazer é interrompido, mas os efeitos de todas as suas acções que interagiram com o exterior, *e.g.* mensagens enviadas para outros executores, não são cancelados.

Todas as mensagens e sinais pendentes ainda não tratados por esse executor serão ignorados. Qualquer executor de Prolog pode destruir outro, desde que tenha o seu identificador `Eid`.

Este predicado pode retornar em `Status` um dos seguintes códigos de erro:

`EOPTION` – O argumento `Eid`, contém um valor inválido ou não admitido neste predicado, *i.e.* não é um *identificador de executor* válido.

> **Exemplo:** "**sys_destroy_wam(34563, S)**" Destrói o executor de Prolog com o identificador `34563`, devolvendo em `S` o resultado da operação (sucesso ou não).

sys_at_halt(+Goal, −Status)

Insere o termo `Goal` numa lista de predicados a serem executados quando o executor invocador terminar (ou for destruído). `Goal` deverá indicar um predicado com 0 (zero) argumentos, *i.e.* deverá ter aridade 0 (zero). Devolve em `Status` o resultado da operação (sucesso ou não)[12].

Os predicados são executados pela ordem inversa da ordem pela qual foram inseridos na lista (a lista tem uma organização em pilha).

Este predicado pode retornar em `Status` um dos seguintes códigos de erro:

`EINVAL` – O termo especificado em `Goal` não tem aridade 0 (zero).

> **Exemplo:** "**sys_at_halt(foo_at_halt, S)**" Insere o predicado `foo_at_halt/0` na lista (pilha) de predicados a serem executados quando o executor terminar, devolvendo em `S` o resultado da operação.

frac_sleep(+Time)

Suspende o executor de Prolog durante o número de segundos indicados em `Time`. `Time` poderá ser um número inteiro ou um número decimal[13].

A chegada de um sinal pode interromper a execução deste predicado, que será continuada quando o processamento do sinal terminar.

> **Exemplo:** "**frac_sleep(1.3)**" Suspende o executor invocador durante 1.3 segundos, aproximadamente.

4.3.5 Predicados de controlo do estado do executor de Prolog

Estes predicados permitem salvaguardar o estado de um executor de Prolog e, posteriormente, repô-lo.

save_state(−State)

Salvaguarda o estado da executor de Prolog e devolve um identificador desse estado em `State`. Conceptualmente cria um ponto de escolha (*choice point*) que apenas será usado através da invocação explícita de restore_state/1 e que será ignorado em retrocesso simples.

O estado `State` salvaguardado deixa de ser válido se o processo de execução levou a que se retrocedesse para além da invocação deste predicado[14].

[12] Entenda-se como resultado da operação o resultado da invocação do predicado sys_at_halt/2 e não do predicado indicado em `Goal` (que só será executado posteriormente).

[13] Admitindo que a granularidade do relógio é dependente do sistema, o tempo especificado será aproximado tanto quanto possível.

[14] Quando o executor retroceder para além da invocação de save_state/1, é desfeita a unificação da variável de argumento deste predicado, invalidando naturalmente o estado salvaguardado.

Exemplo: "**save_state(W)**" A invocação deste predicado leva a que o argumento `W` seja unificado com um descritor do estado actual do executor de Prolog, de forma a que possa ser reposto posteriormente.

restore_state(+State)

Equivale a retroceder até ao ponto salvaguardado pelo save_state/1 correspondente, repondo o estado do executor de Prolog desse ponto. O estado `State` continua definido depois de um restore_state/1, pelo que pode ser reutilizado posteriormente, para forçar o executor a voltar, de novo, ao ponto salvaguardado. As acções irreversíveis, *e.g.* mensagens enviadas, não são canceladas.

Exemplo: "**restore_state(W)**" A invocação deste predicado leva a que o estado descrito pelo argumento `W` seja reposto, forçando um retrocesso até ao ponto salvaguardado pelo save_state/1 correspondente.

4.3.6 Predicados de informação do estado

Estes predicados fornecem ao programa informação sobre o (estado do) sistema referente à configuração — em termos de portas — e à composição — em termos de executores — do estado da arquitectura virtual disponibilizada pelo modelo intermédio. Por outro lado, também disponibilizam alguma informação sobre a configuração da arquitectura real controlada pelo sistema de operação subjacente (identificadores de nós do multicomputador[15] e códigos de erro dependentes do sistema de operação subjacente).

wam_error(+Errno, −Text)

Devolve em `Text` a mensagem de erro associada ao código `Errno`. Os valores admitidos para `Errno` são dependentes do sistema de operação subjacente, pelo que este predicado apenas deve ser invocado com valores instanciados pelos predicados com o prefixo "**sys_**" no argumento `Status`.

Exemplo: "**wam_error(205,E)**" Devolve em `E` um átomo, *e.g.* **not a process**, que é a mensagem de erro do sistema de operação subjacente correspondente ao código `205`.

get_eid(−Eid)

Devolve em `Eid` um inteiro, correspondente ao *identificador de executor* do executor invocador. O *identificador de executor* não é necessáriamente o mesmo que o usado pelo sistema subjacente que suporta todo o ambiente para identificar o processo em causa (ver secção 4.2.1 onde são explicadas as diferenças entre *processo* e executor de Prolog).

Exemplo: "**get_eid(P)**" Devolve em `P` um inteiro, *e.g.* `1423201`, que identifica univocamente o executor invocador no ambiente.

[15] O identificador de nó é um argumento visível na criação de executor de Prolog.

get_node(−Node)

Devolve em `Node` um inteiro, correspondente ao identificador do nó em que o executor invocador se encontra em execução.

> **Exemplo:** "**get_node(N)**" Devolve em `N` um inteiro, *e.g.* `2`, indicando, neste caso, que o executor invocador se encontra a correr no nó `2`.

get_all_nodes(−Node_list)

Devolve na lista `Node_list` os identificadores de todos os nós do sistema[16]. Estes identificadores não têm, obrigatoriamente, que ser os mesmos que o sistema subjacente usa para identificar os mesmos nós.

> **Exemplo:** "**get_all_nodes(L)**" Devolve em `L` uma lista de inteiros, *e.g.* `[0,1,2,3]`, indicando, neste caso, que existem 4 nós no sistema, identificados pelos números de `0` a `3`.

sys_node_state(+Node, −State, −Status)

Este predicado devolve em `State` uma lista de inteiros com os *identificadores de executor* dos executores de Prolog afectos ao nó indicado em `Node`[17].

`State` é uma lista com a forma [Eid, . . .] onde

Eid – é um *identificador de executor* de um executor de Prolog afecto ao nó em questão.

> **Exemplo:** "**sys_node_state(3,L,S)**" Devolve em `L` uma lista de inteiros, *e.g.* `[1254601, 9836400]`, indicando, neste caso, que no nó `3` existem dois executores de Prolog, com *identificadores de executor* dados por `1254601` e `9836400`.

sys_wam_state(+Eid, −State, −Status)

Este predicado devolve em `State` informação sobre um executor de Prolog específico, identificado por `Eid`. É possível obter o *identificador de executor* de um executor de Prolog através do recurso aos predicados sys_node_state/3, get_eid/1, sys_create_wam/4 ou através da sua transferência numa mensagem.

`State` é uma estrutura com a forma (Node, State, Port_name, . . .]), onde

Node – indica qual o nó onde o executor se encontra;

State – indica qual o estado corrente do executor, podendo tomar um dos valores:

- **running** – o executor está a resolver um golo;
- **blocked(recv, Port_name)** – o executor está bloqueado para recepção de uma mensagem pela porta `Port_name`;

[16] O conceito de "todos os nós" é dependente do sistema e, no caso específico do *Trollius*, o sistema usado no protótipo e descrito no capítulo 6, corresponde a todos os nós do multiprocessador (não incluindo, portanto, o computador hospedeiro).

[17] Os demais processos que estejam afectos ao mesmo nó e que não sejam executores de Prolog, são ignorados por este predicado.

blocked(send, Port_name) – o executor está bloqueado para envio de uma mensagem para a porta `Port_name`;

Port_name – indica que o executor é "dono" da porta com o nome `Port_name`.

Exemplo: "**sys_wam_state(4368703,L,S)**" Devolve em `S` o resultado da operação e em `L` uma estrutura, *e.g.* `(3, blocked(send, some_port), [my_first_port, my_other_port])`, indicando, neste caso, que o executor indicado se encontra no nó `3`, bloqueado no envio de uma mensagem para a porta com o nome `some_port` e que é dono das portas `my_first_port` e `my_other_port`.

sys_port_state(+Port, −State, −Status)

Este predicado devolve em `State` uma estrutura com informação sobre a porta indicada em `Port`. É possível indicar uma porta através do seu nome ou de um descritor obtido com sys_create_port/4 ou sys_open_port/4 (ver secção 4.3.1).

`Port` pode ainda tomar o valor especial **$all_ports**, devolvendo em `State`, neste caso, uma lista de estruturas com informação sobre todas as portas do sistema[18] (este valor especial destina-se principalmente a *debugging*).

`State` é uma estrutura (ou lista de estruturas, caso se tenha indicado **$all_ports** em `Port`) com a forma `(Port_name, Owner, Node, [Client_Eid, ...])`, onde

Port_name – indica o nome da porta à qual se refere a informação;

Owner – indica qual o identificador do executor que criou a porta (o "dono" da porta);

Node – indica qual o nó do sistema em que se encontra a porta (e o executor que a criou).

Client_Eid – identificador de um executor que seja cliente desta porta, *i.e.* que tenha aberto a porta com sys_open_port/4.

Exemplo: "**sys_port_state(5,L,S)**" Devolve em `S` o resultado da operação e em `L` uma estrutura, *e.g.* `(the_port_name, 3476502, 2, [5423409, 2345411])`, com informação sobre a porta associada ao descritor `5` do executor invocador. No exemplo apresentado, a estrutura indica que a porta associada ao descritor `5` do executor invocador tem o nome `the_port_name`, foi criada pelo executor com o *identificador de executor* `3476502`, reside no nó `2` e os executores com os *identificadores de executor* `5423409` e `2345411` são seus clientes, *i.e.* podem enviar para mensagens para essa porta.

4.4 Exemplo simplificado

Apresenta-se, em seguida, um exemplo com dois programas Prolog: o programa **hello** e o programa **world**. Com este exemplo pretende-se ilustrar o funcionamento de alguns dos predicados propostos no modelo apresentado neste capítulo.

Depois de carregado o programa **hello** no espaço de cláusulas de um executor Prolog (recorrendo a consult/1), é invocado o predicado hello com um argumento inteiro positivo.

O executor que está a resolver o predicado hello irá criar tantos executores quantos os indicados no argumento que recebeu, criando para cada um deles uma porta de comunicação. Após este processo de inicialização, o executor vai estar atento, simultaneamente, a todas as portas, à espera que

[18] "Todas as portas do sistema" inclui também as reservadas (para os sinais, por exemplo).

qualquer um dos executores recém criados lhe envie uma mensagem. Quando tiver uma mensagem pronta a ser recebida, o executor invoca o predicado respectivo para a receber. Quando tiver recebido uma mensagem de cada um dos executores que criou, o executor principal destruirá todas as portas e termina.

Os executores secundários apenas consultam o sistema, de forma a saberem qual o seu *identificador de executor*, e enviam-no numa mensagem para o executor principal (a mensagem tem ainda um texto de apresentação).

O argumento de linha de comando '-g', usado em sys_create_wam/4, força a que os predicados indicados sejam resolvidos antes de entrar no ciclo principal de execução. No caso particular do exemplo apresentado, porque o último golo do argumento de linha de comando é o predicado halt/0, o executor nunca chegará a executar o seu ciclo principal.

```
hello(N) :-
    write('Vou lançar '), write(N), write(' executores!'), nl,
    start(N, LP),   % Criar as portas e lançar os processos
    wait(N, LP),    % Receber uma mensagem de cada um deles
    end(LP).        % E destruir as portas

start(0, []).
start(N, [P|LP]) :-
    atomterm(N1, N),   % Converte um número num átomo
    sys_create_port(N1, p_noflag, P, 0), % Criar porta
    sys_create_wam(1, p_noflag,        % Criar novo executor no nó 1,
                   dev('/dev/null'),   %   sem canal de entrada
                   port(N1),           %   e com a saída redirigida
                                       %   para a porta recém criada
                   '-g consult(world), % Golos a serem executados
                       world,          %   quando o executor é lançado
                       halt',

                   Eid, 0),
    write('Criei executor com o EID='), write(Eid), nl,
    N2 is N-1,
    start(N2, LP).

wait(0, _):-
    write('Já recebi todas as mensagens!'), nl.
wait(N, LP):-
    sys_select_port(LP, p_wait, P, 0), % Espera pela mensagem de um
                                       %   dos executores
    sys_recv_msg(P, p_nowait, M, 0),   % Lê a mensagem (que já está
                                       %   disponível)
    print(P, M),                       % E imprime-a no ecrã
    N1 is N-1,
    wait(N1, LP).

end([]).
```

```
end([P|LP]) :-
    sys_destroy_port(P, p_noflag, 0), % Destroi as portas de
                                      % comunicação com os executores
                                      % "world"
end(LP).

print(P, M):-
    sys_port_state(P, (N, _, _, _), 0).
    write('Mensagem da porta '), write(N), write(': '), write(M), nl.
```

Exemplo 4.1: O programa "hello"

```
world :-
    get_eid(Eid),                % Vê qual o seu identificador
    atomterm(E1, Eid),           % Converte-o num átomo
    name(E1, EL),                % E depois numa lista
    append("Hello World!  From ", EL, M1), % Acrescenta-o a uma frase
                                      %   de apresentação
    name(M2, M1),                % Converte tudo num átomo
    write(M2).                   % E envia (implicitamente numa
                                 % mensagem) para o executor "hello"
```

Exemplo 4.2: O programa "world"

4.5 Conclusões

Neste capítulo descreveram-se as abstracções e as primitivas que caracterizam a proposta de um modelo que estende o Prolog para permitir concorrência e comunicação.

O modelo apresentado situa-se a um nível intermédio de um sistema computacional, no sentido em que é definido sob a forma de predicados extra-lógicos que suportam uma interface de qualquer sistema Prolog com um ambiente de operação (sequencial, paralelo e/ou distribuído) subjacente. Os mecanismos do modelo não interferem com a semântica básica da linguagem Prolog, nem com a estratégia convencional de pesquisa sequencial, em profundidade, que é preservada, bem como todos os operadores extra-lógicos, como o de corte e os de *assert/1* e *retract/1*.

O modelo também não se compromete com a abordagem de implementação do executor de Prolog, sendo compatível com sistemas baseados em interpretadores ou em compiladores e máquina abstracta (embora o uso desta segunda abordagem, no protótipo implementado e descrito no capítulo 6, tenha influenciado o nome de alguns predicados do modelo).

As funcionalidades oferecidas pelo modelo, bem como a sua flexibilidade, são ilustradas no capítulo seguinte, para a implementação de um sistema de distribuição e execução de golos Prolog.

Capítulo 5

Avaliação da funcionalidade do modelo

5.1 Introdução

No secção 1.2 argumentou-se que a existência de um sistema de nível intermédio entre o modelo de programação em lógica e o modelo disponibilizado pelo sistema de operação da máquina, contribui significativamente para a obtenção, num curto espaço de tempo, de uma versão final dum sistema que suporta o modelo de programação em lógica, através de uma rápida prototipagem, implementação e facilidade de alteração.

Com o objectivo de justificar esta argumentação, apresenta-se neste capítulo um sistema que suporta um ambiente de execução distribuída de programas em Prolog e sugestões de como o implementar sobre o modelo apresentado no capítulo anterior. O sistema de execução distribuída de programas em Prolog apresentado neste capítulo pode ser encarado como uma aplicação do modelo proposto no capítulo anterior.

Na secção 5.2 será apresentado o modelo operacional e as primitivas do sistema; na secção 5.3 será apresentado um modo como este ambiente de execução distribuída de Prolog poderá ser usado para, por exemplo, implementar predicados (ou operadores, neste caso) de mais alto nível ainda. Na secção 5.4 serão apresentadas sugestões de como implementar os predicados propostos com base no modelo inicial. Na última secção, serão feitas algumas considerações sobre o sistema descrito.

5.2 Sistema de execução distribuída de Prolog

Como exemplo de aplicação do modelo proposto no capítulo 4, propõe-se um sistema de execução distribuída de programas em Prolog, em que o programador não necessite de estar a controlar os recursos, *i.e.* processadores usados para a paralelização dos golos, nem de enviar e receber mensagens explícitamente. Na próxima secção, serão apresentados os três tipos de processos intervenientes no sistema e, na seguinte, serão apresentados os predicados do sistema proposto.

5.2.1 Intervenientes no sistema

Existem no sistema três tipos de processos: o *gestor de executores* (chamado **gestor** no resto do texto), os processos *servidores de golos* (chamados **servidores** no resto do texto) e os processos *clientes* (chamados **clientes** no resto do texto).

Gestor: é da responsabilidade do gestor (único no sistema) controlar a carga dos vários nós do multicomputador, garantindo uma equilibrada distribuição de carga, através de uma correcta

distribuição dos servidores pelos múltiplos nós. É também da sua responsabilidade a criação de novos servidores (caso o sistema de operação da máquina suporte a criação dinâmica de processos) e a destruição dos que já não são necessários.

Cliente: o executor inicial pode, quando assim o entender, solicitar ao gestor que lhe disponibilize um executor de Prolog que esteja desocupado, de forma a que este executor (um *servidor*) possa, em paralelo com o executor cliente, percorrer uma parte do espaço de soluções de um golo, obtendo-se assim (potencialmente) uma ou mais soluções num menor tempo, *i.e.* obtendo-se um aumento no desempenho do sistema. O acesso aos predicados que disponibilizam, ao cliente, as operações sobre os servidores, *e.g.* requisição e libertação de servidores, é obtido através da consulta de um ficheiro contendo a definição desses predicados.

Servidor: um servidor é, na realidade, um executor de Prolog que tem um ciclo inicial adaptado, de forma a que o seu comportamento se resuma a resolver golos, quando tal lhe é solicitado. Entre os golos que o servidor pode ser solicitado para resolver, encontram-se golos que desencadeiam o carregamento de cláusulas no seu espaço de trabalho, disponibilizando assim os contextos desejados para a resolução dos golos. Um servidor pode, a dada altura do seu processo de resolução de um golo, solicitar ao gestor um outro servidor, assumindo assim um papel de cliente para o novo servidor (mas mantendo o seu papel de servidor para com o seu cliente inicial).

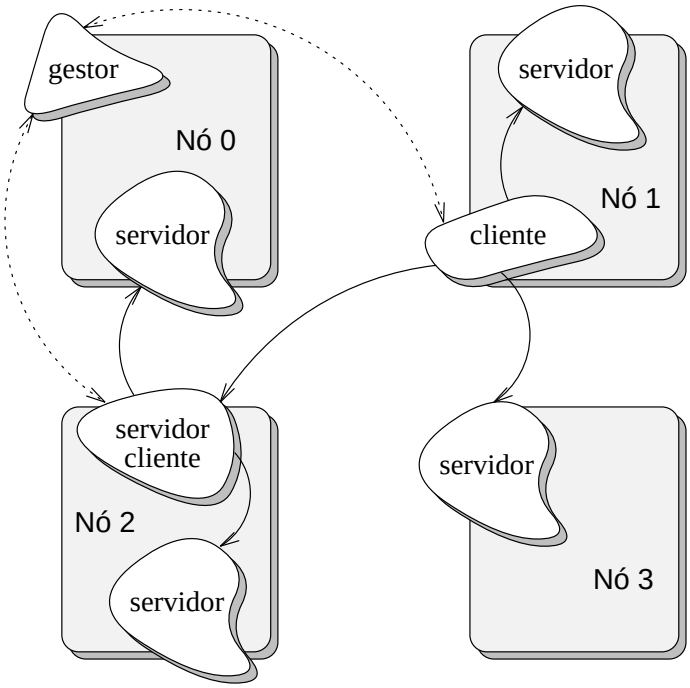

Figura 5.1: Sistema de execução distribuída de programas em Prolog

A figura 5.1 representa uma situação exemplificativa do comportamento descrito atrás para os três tipos de executores: o gestor de executores encontra-se no nó 0; o executor inicial (indicado na figura como *cliente*) encontra-se no nó 1 e tem três executores servidores, um no nó 1, outro no nó 2 e outro no nó 3. O servidor do nó 2, por sua vez, é também cliente de mais dois servidores, um no mesmo nó 2 e outro no nó 0.

Os clientes e servidores estão dispersos pelos vários nós do multicomputador, de forma a que a carga seja distribuída tão uniformemente quanto possível. Esta condição de equilíbrio de carga não é fundamental, mas revela-se importante para garantir o equilíbrio global do sistema.

5.2.2 Predicados do sistema

Os predicados apresentados nesta secção definem um sistema de execução distribuída de programas em Prolog e, durante a sua execução, falham em retrocesso ou no caso de se verificar uma situação anómala não prevista.

get_worker(−WorkerId)

Com a invocação deste predicado, um executor de Prolog pode solicitar um servidor ao gestor de executores, recebendo o *identificador de servidor* do servidor que lhe foi atribuído em `WorkerId`[1]. Este predicado falha no caso de o gestor não ser capaz de disponibilizar um servidor ao invocador deste predicado.

É possível definir um novo predicado tal que, quando em processamento normal (para a frente) invoque este (get_worker/1) e que, quando em retrocesso, liberte (destrua) o servidor pedido.

A forma como o gestor processa um pedido de um servidor é dependente de uma estratégia interna ao sistema, podendo-se optar pela criação dinâmica do servidor, quando necessário, ou por manter um reservatório (*pool*) de servidores, que serão atribuídos à medida que vão sendo solicitados (ver secção 5.4.1).

send_goal(+Goal, +WorkerId)

Após obter o identificador de um servidor, o cliente recorre a este predicado para lhe enviar o golo que aquele há-de resolver.

Este predicado apenas envia o golo a resolver, devendo posteriormente o cliente ir buscar as soluções do golo enviado, recorrendo ao predicado solution/3 (ver descrição nesta mesma secção).

spawn(+Goal, −WorkerId)

O cliente pode, se assim o entender, recorrer a este predicado para lançar a resolução concorrente de um golo. O golo dado é lançado num servidor cujo identificador é devolvido em `WorkerId`. Se não for possível atribuir um servidor ao processo invocador, este predicado falhará.

Este predicado é apenas uma variante dos dois anteriores, que os combina numa forma mais natural e intuitiva (*user friendly*). Assim, aplica-se-lhe, também, a nota referente a send_goal/2 sobre a obtenção, explícita e por pedido, das soluções para os golos.

solution(+WorkerId, +Flags, −Solution)

Após enviar um golo ao servidor para ser resolvido por este, é necessário pedir-lhe que este devolva uma solução encontrada. Com recurso a este predicado, o cliente solicita ao servidor a solução para o golo previamente enviado (com send_goal/2 ou spawn/2). Este predicado, tal como todos os outros apresentados neste capítulo, falha quando em retrocesso.

O argumento `Flags` pode tomar os valores **s_wait** ou **s_nowait** e o argumento `Solution` é instanciado com a solução devolvida pelo servidor. A solução que o servidor devolve é, na realidade, o mesmo golo que lhe foi enviado resolver, mas com as variáveis instanciadas de acordo com a

[1] O *identificador de servidor* é independente do *identificador de executor* mencionado no capítulo anterior, *i.e.* pode não ser o mesmo.

solução que foi encontrada. Se não existir uma solução para o golo dado, *i.e.* o golo falhou, então o servidor devolve o átomo **fail**.

Caso o servidor ainda esteja a calcular a solução quando o predicado é invocado, *i.e.* não exista ainda uma solução disponível, o comportamento deste depende do valor de `Flags`: se o valor for **s_wait** a invocação do predicado bloqueia o invocador até haver uma solução disponível; se o valor for **s_nowait** então o argumento `Solution` virá imediatamente instanciado com o átomo reservado **$not_available**.

Cada servidor, uma vez solicitada a resolução de um golo, fica comprometido a pesquisar todo o espaço de soluções para esse golo, dado o conjunto de clásulas que o servidor conhece (carregado inicialmente ou explícitamente). Só se o cliente o explicitar, através do controlo que se discute adiante, é que o servidor abandonará a pesquisa exaustiva. Enquanto isto não for feito, por cada invocação de solution/3 o servidor devolverá uma solução, até esgotar o espaço de pesquisa.

free_worker(−WorkerId)

Quando um servidor não é mais necessário, o cliente pode (e deve) libertar o servidor. Invocando este predicado, o cliente está a comunicar ao gestor que o servidor indicado em `WorkerId` não é mais necessário.

A forma como o gestor processa a libertação de um servidor é dependente da estratégia do sistema, podendo-se optar pela destruição do servidor ou pela sua inserção num reservatório (*pool*) de servidores livres, que serão atribuídos à medida que vão sendo solicitados (ver secção 5.4.1).

reset_worker(+WorkerId)

Um cliente pode não estar interessado em que o servidor continue a calcular soluções para um golo que lhe tenha enviado, mas pode ainda necessitar dele para lhe resolver outros golos. Em vez de libertar o servidor que já detém e solicitar um novo ao gestor, é possível o cliente recorrer a este predicado, para forçar o servidor a voltar ao estado inicial em lhe foi atribuído, *i.e.* aguardar que lhe enviem um golo para resolver, "esquecendo" todo o processamento que já tinha feito entretanto.

A eliminação dos efeitos do processamento no estado da máquina Prolog é feita de uma forma semelhante à eliminação dos efeitos do processamento quando em retrocesso e, tal como neste caso, as acções irreversíveis, em particular as que envolvem comunicação com outros servidores e sobre a base de dados de clásulas, não são desfeitas. É possível definir um novo predicado que, antes de invocar este, liberte (destrua) todos os servidores entretanto pedidos, de forma a evitar deixar "resíduos" no sistema.

Este predicado só existe por uma questão de optimização do desempenho, pois o cliente conseguiria funcionalidade semelhante, libertando este servidor e requisitando um novo.

reset_goal(+WorkerId)

A invocação deste predicado força o servidor a "esquecer" todas as solucões que já tenha calculado, de forma a que este recomeça a resolver o golo pesquisando todo o espaço de soluções (o servidor recomeça o processamento do golo como se, estando no seu estado inicial, tivesse acabado de o receber para resolução, nesse instante).

Este predicado só existe por uma questão de optimização do desempenho, pois o cliente conseguiria funcionalidade semelhante libertando este servidor, requisitando um novo, e enviando-lhe o último golo que havia enviado ao anterior servidor para ele resolver.

5.3 Exemplo de aplicação

O conjunto de predicados propostos na secção anterior, apesar de serem de nível superior aos dos propostos no modelo do capítulo 4, ainda obrigam o programador a controlar explicitamente a atribuição de tarefas aos servidores e a recolha das soluções. É possível implementar ainda mais uma camada de paralelização de golos, sobre os predicados propostos na secção anterior, conduzindo à obtenção de um mecanismo de paralelização de nível superior: por exemplo, o operador "//" do Delta Prolog [PN84, Cun85].

Este operador permite a especificação da execução concorrente (se possível) duma conjunção de golos Prolog, como por exemplo, p(A, B) // q(B, C), em que a variável B é logicamente partilhada pelos predicados p/2 e q/2, tal como é exigido pela semântica do *Delta Prolog*. Esta condição é independente do facto de os predicados estarem a ser executados em sequência ou em paralelo e, neste caso, no mesmo ou em processadores diferentes.

Esta aproximação por camadas permite ao programador, tal como exemplifica a figura 5.2, obter um compromisso entre o nível da funcionalidade obtida e a eficiência da implementação, optando por mecanismos de paralelização de mais alto nível e possivelmente menos eficientes, ou optando por mecanismos de paralelização de mais baixo nível e mais eficientes[2].

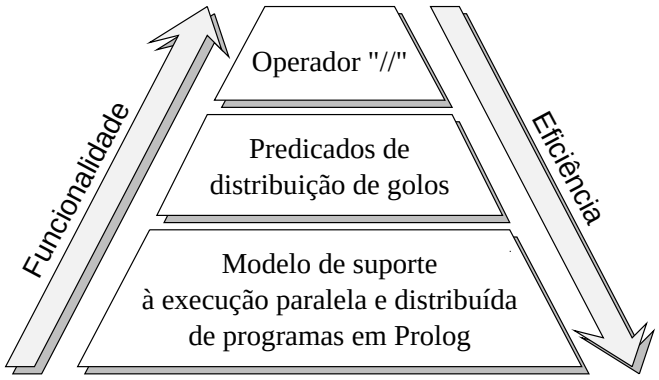

Figura 5.2: "Eficiência" *vs.* "funcionalidade" numa aproximação por camadas

O código que a seguir se se apresenta descreve a implementação do operador "//" do *Delta Prolog*, sobre o sistema de distribuição de golos apresentado na secção anterior, podendo ser considerado como um exemplo da sua utilização e funcionalidade.

```
:- op( 220, xfy, // ).

G1 // G2 :-
        spawn( G2, Server_id ),        % Se o "spawn/2" teve sucesso,
        !,                             %   então os golos serão
        '$doPar'( G1, G2, Server_id ). %   executados em paralelo
G1 // G2 :-
        call( G1 ),             % Se o "spawn/2" falhou, os golos
        call( G2 ).             %   serão executados sequencialmente

'$doPar'( G1, G2, Server_id ) :-
```

[2] Mais uma vez se salienta que o programador é sempre livre de refazer o seu programa, recorrendo directamente às camadas de nível inferior, à partida mais eficientes e oferecendo maior grau de flexibilidade.

```
        call( G1 ),                    % Resolve o golo "G1" e pede uma
        '$solutions'( G2, Server_id ). %   solução de "G2" ao servidor
'$doPar'( _, _, Server_id ) :-
        free_server( Server_id ), % Como todo o espaço de soluções já
        fail.                     % foi percorrido, destrói o servidor
                                  % e falha

'$solutions'( G, Server_id ) :-
        solution( Server_id, s_wait, S ), % Pede uma solução ao servidor
        '$processSol'( S, G, Server_id ). %   e processa-a

'$processSol'( fail, _, Server_id ) :-
        !,                        % Se o servidor não tem mais soluções
        reset_goal( Server_id ), %   força-o a remomeçar a percorrer o
        fail.                     %   espaço de soluções desde o início.
'$processSol'( G, G, _ ).          % Se tem uma solução, então unifica a
                                   %   solução obtida com o golo que tinha
                                   %   enviado, de forma a que as
                                   %   variáveis comuns aos golos "G1" e
                                   %   "G2" tenham o mesmo valor
'$processSol'( _, G, Server_id ) :-   % Se as variáveis comuns a "G1" e
        $solutions( G, Server_id ).   %   "G2" não foram instanciadas
                                      %   com os mesmos valores, ou se
                                      %   foi pedida uma nova solução
                                      %   (por retrocesso), então vai
                                      %   pedir a próxima solução ao
                                      %   servidor, ficando o cliente
                                      %   à espera
```

Exemplo 5.1: Implementação do operador "//"

5.4 Sugestões de implementação do sistema de distribuição de golos

Nesta secção serão apresentadas algumas sugestões de como implementar os predicados propostos na secção 5.2 sobre o modelo proposto no capítulo 4. Na secção 5.4.1 são dadas indicações de como implementar o *gestor de executores*; na secção 5.4.2 são dadas indicações de como implementar o *servidor*; na secção 5.4.3 serão apresentadas sugestões, com mais pormenor, de como implementar todos os predicados que o processo *cliente* pode invocar.

5.4.1 Sugestões para a implementação do "gestor de executores"

O gestor tem à sua responsabilidade a gestão e controlo dos servidores, bem como o controlo da carga nos vários nós do sistema, de forma a tentar garantir uma distribuição equitativa.É uma exigência do sistema aqui proposto, que todos os processos sejam capazes de comunicar com o gestor[3].

Por uma questão de simplicidade e facilidade de implementação, o gestor é apenas mais um executor de Prolog, que executa um programa específico de controlo do sistema da execução distribuída

[3]No caso de o sistema se basear num mecanismo de comunicação por portas, é necessário que o gestor esteja permanentemente atento a uma porta cuja designação seja do conhecimento geral.

de golos Prolog. O gestor deverá estar num ciclo infinito, atento a três tipos de mensagens: mensagens de *pedido de servidor*, mensagens de *libertação de servidor* e mensagens de *existência de um novo servidor*.

Quando recebe uma mensagem de pedido de servidor, o gestor deverá responder ao cliente com o identificador de um servidor, que ficará atribuído ao cliente até este o libertar. A estratégia a adoptar para gestão dos servidores poderá variar, conforme as necessidades e limitações do sistema de operação e da arquitectura subjacentes.

Se o sistema de operação admite criação dinâmica de processos, e se esta não reduz apreciavelmente o desempenho, a satisfação de um pedido de servidor pode ser realizada através da criação dinâmica de um novo executor, forçando-o a executar um código específico para que assuma "uma personalidade" de servidor (ver secção 5.4.2), enquanto que a libertação de um servidor levará a que o gestor o destrua.

Se for necessário garantir um bom desempenho do sistema, a criação dinâmica de executores não deverá ser uma boa opção, pois geralmente é uma operação pesada e demorada, excepto se houver suporte para processos leves (*threads*). Se o sistema não admite criação dinâmica de processos[4] a estratégia atrás apresentada não é possível de implementar.

Uma solução alternativa para a gestão dos servidores é obrigar à existência de um número pré-determinado de executores de Prolog em estado de "hibernação", *i.e.* sem fazerem nada, como se não existissem, sendo os seus identificadores do conhecimento do gestor. Quando um cliente solicita um servidor, este devolve-lhe o identificador de um dos que estão em "hibernação" e envia uma notificação ao servidor para que este "acorde". Quando um cliente liberta um servidor, o gestor coloca-o na sua lista de servidores disponíveis e envia-lhe uma notificação para o colocar em "hibernação".

Esta simulação do estado de "hibernação", que força o servidor a deixar de estar operacional, pode ser obtida forçando-o a ficar bloqueado na recepção de uma mensagem. O grande inconveniente desta solução é que, a dada altura, os servidores poderão estar todos ocupados, pelo que o requisitar de um novo servidor pelo cliente vai falhar. Note-se que nestes casos, como se ilustra no exemplo, o cliente pode recorrer à avaliação sequencial de golos, o que, em situações em que o sistema já está muito sobrecarregado com um grande número de processos, pode ser preferível, face a gerar novos servidores.

A solução que apresenta o melhor compromisso entre funcionalidade e eficiência resume-se a uma combinação equilibrada das duas soluções anteriores (caso o sistema de operação o admita): sempre que um processo liberta um servidor, este não é destruído, mas sim colocado na lista de disponíveis; sempre que um processo requisita um servidor é-lhe disponibilizado um dos que já existem, mas que não estão atribuídos, ou então é criado um novo servidor para atribuir ao cliente, caso não existam mais servidores disponíveis.

5.4.2 Sugestões para a implementação do "servidor de golos"

O servidor deverá, inicialmente, entrar em contacto com o gestor e indicar-lhe que se encontra disponível para atender pedidos, através do envio de uma mensagem com o formato `worker(WorkerPid, WorkerPortName)`. Nesta mensagem, o argumento `WorkerPid` indica o identificador do servidor e o argumento `WorkerPortName` indica a porta por onde este receberá os golos a resolver.

O servidor deverá consultar o ficheiro onde se encontra a definição dos predicados de cliente (ver secção seguinte), se pretender vir a ser cliente de outros servidores.

[4] Muitos sistemas de operação para multicomputadores optam por não admitir a criação dinâmica de processos.

O servidor, recorrendo ao predicado save_state/1, deverá salvaguardar o seu estado inicial e o estado imediatamente após receber o golo que irá resolver, de forma a suportar a implementação dos predicados reset_worker/1 e reset_goal/1, respectivamente. Se o sistema não admitir criação dinâmica de processos, o servidor deverá ainda salvaguardar o seu estado antes do sítio onde notifica o gestor que se encontra disponível, de forma a suportar a implementação do predicado free_worker/1.

A implementação dos predicados reset_worker/1 e reset_goal/1 corresponde, no servidor, à reposição do estado que lhes ficou associado, recorrendo ao predicado restore_state/1.

A implementação do predicado free_worker/1 poderá ser, também, obtida com recurso aos predicados save_state/1 e restore_state/1, caso o sistema não admita a criação dinâmica de processos; ou então poderá corresponder à destruição efectiva do servidor, caso contrário.

5.4.3 Sugestões para a implementação dos predicados do "cliente"

O cliente necessita de consultar um ficheiro onde se encontram definidos, em Prolog, os predicados necessários para que possa solicitar a execução concorrente de golos (neste sistema), já apresentados na secção 5.2.2.

Por constituir a interface entre o programador e o sistema, vai ser dada uma especial atenção à implementação deste predicados.

get_worker(−WorkerId)

Para um cliente solicitar um servidor, envia ao gestor de executores uma mensagem do tipo get(MyPortName), onde `MyPortName` indica o nome da porta para onde o gestor deve responder ao pedido.

O gestor, por sua vez, responde ao cliente enviando-lhe uma mensagem do tipo get(WorkerPid, WorkerPortName), em que `WorkerPid` indica o identificador único do servidor que foi atribuído ao cliente e `WorkerPortName` indica o nome da porta de serviço do servidor, *i.e.* a porta por onde este deverá receber o golo a resolver.

Sempre que se requisita um novo servidor, o cliente cria uma nova porta para contactar com este, *i.e.* para receber mensagens deste. Esta opção tem como inconveniente o de obrigar à existência e manutenção de um número considerável de portas, mas facilita de forma significativa, a programação das demais primitivas, pois permite uma indicação explícita (através da escolha da porta) de qual o servidor que se pretende contactar (para receber mensagens).

Para que os demais predicados do sistema possam conhecer a porta destinada ao novo servidor, a informação relevante sobre este é inserida na base de dados do cliente, numa estrutura com a forma:

```
'$worker'( id(WorkerPid),
           send(WorkerPortName, ToWorker),
           recv(MyNewPortName, FromWorker) ).
```

Nesta estrutura, o argumento `WorkerPid` identifica o servidor em causa, os argumentos `WorkerPortName` e `ToWorker` indicam o nome e o identificador da porta (obtido recorrendo a sys_open_port/4) por onde o servidor receberá mensagens, os argumentos `MyNewPortName` e `FromWorker` indicam o nome e o identificador da porta (obtido recorrendo a sys_create_port/4) por onde o cliente receberá as mensagens vindas do servidor.

As acções levadas a cabo para a requisição de um novo servidor, *i.e.* se este é criado dinamicamente ou se é retirado de um reservatório (*pool*), são da responsabilidade do gestor, sendo completamente desconhecidas para o cliente.

send_goal(+Goal, +WorkerId)

Este predicado é apenas uma máscara para um envio de uma mensagem ao servidor, em que o golo é embebido numa estrutura do tipo solve(Goal).

spawn(+Goal, −WorkerId)

Este predicado tem uma funcionalidade semelhante à conjunção de get_worker/1 com send_goal/2.

solution(+WorkerId, +Flags, −Solution)

Para implementar este predicado o cliente deverá receber uma mensagem do seu servidor (indicado em `WorkerId`). O argumento `Flags` é convertido no equivalente para o predicado sys_recv_msg/4, *i.e.* convertendo **s_nowait** em **p_nowait** e **s_wait** em **p_wait**, e recebendo em `Solution` uma estrutura do tipo solution(Solution), onde `Solution` está instanciado com a solução calculada, conforme apresentado em 5.2.2.

free_worker(−WorkerId)

Para avisar o gestor de que um servidor não é mais necessário, o cliente envia uma mensagem do tipo free(WorkerPid), em que `WorkerPid` identifica o servidor a libertar.

As acções desencadeadas sobre o servidor quando este é libertado, *i.e.* se este é destruído ou colocado num reservatório (*pool*) de servidores disponíveis, são da exclusiva responsabilidade do gestor, sendo completamente desconhecidas para o cliente.

reset_worker(+WorkerId)

Para implementar este predicado, o cliente deverá notificar o servidor, assincronamente, de que este deverá repor o seu estado inicial, tal como descrito em 5.2.2.

A notificação assíncrona do servidor é obtida através do envio de um sinal, usando sys_send_sig/3, com a estrutura reset(worker). Para que os *buffers* fiquem limpos, após a notificação do servidor, o cliente fica num ciclo a receber mensagens, ignorando o seu conteúdo, da porta associada a esse servidor. Este ciclo será interrompido quando o cliente receber uma estrutura do tipo reset_worker(ok) (enviada pelo servidor, quando este recebe o sinal), indicando que já não existem mensagens em trânsito nem armazenadas em *buffers*[5].

reset_goal(+WorkerId)

Para implementar este predicado, o cliente deverá notificar o servidor, assincronamente, de que este deverá considerar, de novo, todo o espaço de soluções para a resolução do golo que lhe foi enviado, tal como descrito em 5.2.2.

A notificação assíncrona do servidor é obtida através do envio de um sinal, usando sys_send_sig/3, com a estrutura reset(goal). Para que os *buffers* fiquem limpos, após a notificação do servidor, o cliente fica num ciclo a receber mensagens, ignorando o seu conteúdo, da porta associada a esse servidor. Este ciclo será interrompido quando o cliente receber uma estrutura do tipo

[5] Pressupõe-se que as mensagens são entregues pela mesma ordem pela qual foram enviadas, tal como indicado na secção 4.3.2, na definição dos predicados sys_recv_msg/4 e sys_send_msg/4.

reset_goal(ok) (enviada pelo servidor, quando este recebe o sinal), indicando que já não existem mensagens em trânsito nem armazenadas nos *buffers*.

5.5 Conclusões

Neste capítulo ilustrou-se a funcionalidade das primitivas do modelo proposto no capítulo 4, através da descrição de uma camada de gestão de paralelização (e distribuição) de golos. Discutiu-se a flexibilidade que esta camada proporciona aos *executores clientes*, permitindo-lhes recorrer a diversas estratégias para a composição sequencial e paralela de golos, sem preocupação com a gestão efectiva dos recursos (processadores virtuais e reais, e camadas de comunicação) necessários para esse efeito. Também se ilustraram as potencialidades do modelo proposto, para realizar modelos de concorrência, de nível superior, com semânticas mais declarativas, através do exemplo do operador "*//*" do *Delta Prolog*.

A discussão das potencialidades de implementação do sistema de gestão dos executores e servidores, com diferentes estratégias alternativas, mais ou menos adequadas à gestão eficiente dos recursos, também ilustrou a flexibilidade conseguida, sem, contudo, sobrecarregar os clientes/utilizadores com a necessidade de conhecerem/especificarem essas estratégias.

Capítulo 6

Implementação do modelo proposto

6.1 Introdução

A avaliação da funcionalidade do modelo proposto no capítulo 4 foi feita através da sua implementação num multicomputador, *i.e.* multiprocessador de memória distribuída, e implementando sobre este o sistema de distribuição de golos descrito no capítulo 5.

Neste capítulo apresenta-se uma descrição da componente experimental deste trabalho. Esta descrição, não sendo exaustiva, tenta focar os pontos relevantes para a implementação do modelo proposto, quer na perspectiva do modelo operacional, *i.e.* organização e interacção dos executores de Prolog e estruturas de dados relevantes (independentes da linguagem de implementação e do sistema de suporte à execução), quer na de alguns pormenores de implementação (dependentes da linguagem de implementação e/ou do sistema de suporte à execução).

Na próxima secção é apresentada a arquitectura física que suportou todo o trabalho de desenvolvimento e teste; na secção 6.3 é descrito o sistema de operação usado na máquina paralela (onde se implementou o protótipo) e descrvem-se, também, as extensões feitas a este para melhor se adequar ao suporte do modelo proposto; na secção 6.4 é feita uma descrição do modelo operacional escolhido; na secção 6.5 apresentam-se os pormenores de implementação do modelo; por fim, na seccção 6.6 faz-se uma breve avaliação do trabalho experimental desenvolvido e das opções tomadas.

6.2 Arquitectura física utilizada na experimentação

Para suporte ao desenvovimento do protótipo do modelo proposto no capítulo 4 e do sistema de distribuição de golos proposto no capítulo5, foi utilizado um multicomputador *Meiko CS/1* (*Meiko Computing Surface 1*) [Mei90a, Mei90b] com 16 nós, disponível no DI/UNL, em que cada nó é constituído por um *Transputer T800* com 4 Mbytes de memória. Como este computador não dispõe de consola nem de suportes magnéticos de informação próprios, *e.g.* discos rígidos, é necessário um computador hospedeiro que partilhe com ele os seus recursos, permitindo-lhe assim acesso à sua consola e ao seu sistema de ficheiros.

O sistema de operação usado para gerir o multicomputador atrás referido, e que serviu de base para a implementação das funcionalidades e abstracções propostas no modelo, foi o *Trollius* versão 2.2 [Ohi92d]. O *Trollius*, tal como é descrito na seccção 6.3, é um sistema de operação para multicomputadores que controla directamente os recursos da máquina paralela e que é executado com recurso a um emulador no computador hospedeiro.

Na máquina paralela, o *Trollius* é o responsável pela gestão de todos os recursos desta, pelo suporte de mecanismos de comunicação e pela emulação, em todos os nós, de serviços não locais, como o acesso ao sistema de ficheiros e à consola do computador hospedeiro. No computador hospedeiro, o emulador do *Trollius* apenas tem que fazer a correspondência das funcionalidades que pretende disponibilizar nas funcionalidades disponibilizadas pelo sistema de operação do computador hospedeiro, necessitando também de suportar e gerir a comunicação com o multicomputador. O computador que serve de hospedeiro ao multicomputador é uma estação de trabalho *Sun*, que tem como sistema de operação o *SunOS 4.1.3* e executa um emulador do *Trollius*[1].

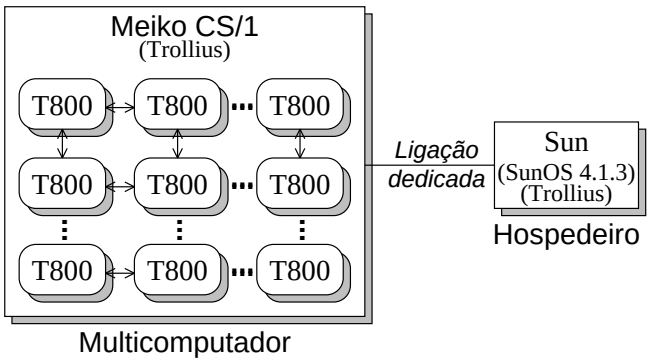

Figura 6.1: Arquitectura usada no protótipo

Internamente, o *Transputer T800* possui um CPU (unidade de processamento central) e um FPU (unidade de cálculo em vírgula flutuante) separados, uma unidade de gestão de processos (que permite fazer o escalonamento dos processos por *firmware*), dois temporizadores, uma interface de gestão da memória externa, uma memória interna de 4 Kbyte e quatro canais de comunicação, cada um com dois DMA (unidade de acesso directo à memória) que conseguem transferir informação a uma velocidade até 20 Mbit/s. Os 8 DMA permitem ao processador fazer comunicações bidireccionais, em paralelo, pelos seus quatro canais, ao mesmo tempo que executa cálculos, sendo assim possível, em casos extremos, o processador estar envolvido na transferência simultânea de 160 Mbit/s (receber 80 Mbit/s e enviar mais 80 Mbit/s) e em cálculos internos. Esta possibilidade é extremamente interessante, pois permite reduzir ao mínimo o peso das comunicações na arquitectura física e nas aplicações que sobre ela são executadas.

6.3 O sistema de operação Trollius

6.3.1 Apresentação

Devido à falta de utilitários de desenvolvimento de programas nos multicomputadores, um grupo de investigadores americanos decidiu desenvolver um sistema de operação capaz de se adaptar a um grande número de máquinas presente e futuras. O projecto nasceu em 1985, no *Cornell Theory Center* e foi mantido durante largo tempo pela *Ohio State University*, tendo sido anunciado recentemente (1994) o abandono do seu desenvolvimento, em favor da LAM (*Local Area Multicomputer*) [Ohi94]. O *Trollius*, inicialmente, desenvolvido para arquitecturas baseadas em *Transputers* [Ohi92c, Ohi92a] foi, no entanto, transportado para outras arquitecturas, como as máquinas VOLVOX IS-860 baseadas no processador i860 de Intel [DGJP93].

[1] Este emulador do *Trollius* sobre o UNIX tem algumas funcionalidades limitadas, visto que não pode controlar directamente o *hardware*.

O *Trollius* [Ohi92d, Ohi92b, Ohi92c, Ohi92a] baseia-se num modelo de cliente/servidor, típico dos sistemas baseados em micro-núcleos [Lou93a]. Pretende formar um sistema de base, sobre o qual se desenvolvem funcionalidades de mais alto nível. O sistema utiliza as capacidades de multiprocessamento dos *Transputers* de forma a que, a pedido dos processos do utilizador, sejam invocados os servidores necessários — *e.g.* encaminhamento de mensagens, gestor de memória — para que os serviços sejam realizados. No coração do sistema *Trollius* reside um núcleo de reduzidas dimensões com a principal função de gerir o *rendez-vous* entre processos na troca de mensagens.

O *Trollius* é constituído pelos seguintes componentes: um sistema de operação para o multicomputador; um emulador do sistema, sobre o sistema de operação do computador hospedeiro; um conjunto de programas utilitários acessíveis a partir duma interface de *linha de comando*, que permitem ao utilizador controlar e monitorizar o ambiente de execução; uma biblioteca de funções acessíveis a partir da linguagem de programação *C* e *Fortran*, que disponibilizam o acesso aos serviços do sistema.

Em termos breves, as principais características do *Trollius* são:

Portabilidade – o *Trollius* pode adaptar-se a todos os computadores hospedeiros UNIX e a todos os multicomputadores baseados em *Transputers*;

Adaptabilidade – o código fonte do sistema está disponível em linguagem *C*, sendo robusto, claro e bem documentado; está assim facilitado o caminho para a alteração do sistema, se necessária;

Facilidade de utilização – a interface de *linha de comando* com o utilizador é semelhante à do UNIX[2];

Facilidade de programação – o ambiente de programação é uniforme em todos os nós do sistema (incluindo no computador hospedeiro), garantindo o *Trollius* para todas as mensagens trocadas, os seguintes serviços:

i) a gestão dos *buffers*;

ii) o encaminhamento;

iii) a partição em pacotes, se necessário, e a respectiva junção no destinatário;

iv) o controlo de fluxo;

v) a criação de circuitos virtuais.

6.3.2 Arquitectura física do Trollius

O *Trollius* é um sistema de operação que (idealmente) é capaz de controlar aquilo a que os autores chamam de *Local Area Multicomputer*, que na realidade corresponde a um conjunto de computadores interligados através de uma rede local. Alguns desse computadores podem ser multicomputadores, que estão integrados no *Local Area Multicomputer* através de um ou mais computadores hospedeiros. Na terminologia dos autores, aos nós internos aos multicomputadores chama-se *nós ITB (In The Box)* e aos nós formados por computadores convencionais chama-se *nós OTB (Out The Box)*. A figura 6.2 ilustra esta arquitectura.

Na realidade o sistema nunca chegou ao estado de evolução ambicionado inicialmente, restringindo-se sempre, até à sua última versão, a um único multicomputador com um único computador hospedeiro.

[2]No caso do *Trollius* sobre computador hospedeiro, a interface com o utilizador é a normalmente oferecida pelo sistema de operação do computador hospedeiro.

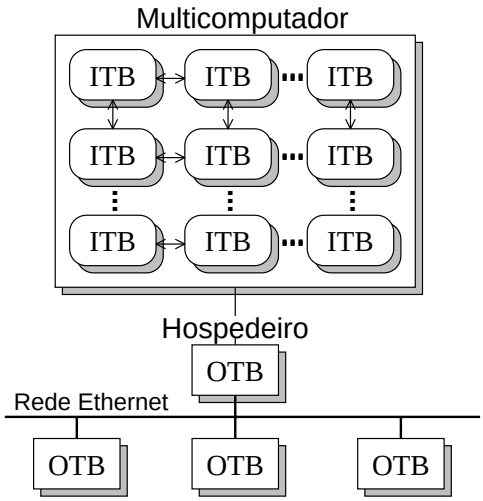

Figura 6.2: O ambiente de execução do *Trollius*

6.3.3 Configuração do ambiente de execução

A configuração da arquitectura física do multicomputador gerido pelo *Trollius*, *e.g.* quantos e quais os processadores que o *Trollius* irá gerir, é definida na altura de arranque do sistema, não sendo possível posteriores adaptações e/ou reconfigurações. A configuração do ambiente lógico de execução é dinâmica, sendo possível a qualquer processo, quando em execução, cria outros processos, no mesmo ou noutro nó.

Criação de processos no mesmo nó

Na criação de processos no mesmo nó em que reside o processo que fez o pedido, é possível distinguir dois tipos de pedidos, baseados na comparação entre os códigos de ambos os processos: os processos a lançar têm o mesmo código, *i.e.* são instâncias do mesmo programa, ou têm código diferente.

No segundo caso, a criação de processos é feita como se de uma criação remota se tratasse. No primeiro caso, existem três alternativas que devem ser ponderadas em termos de peso (da criação do processo) e da funcionalidade obtida. Os processos a criar podem ser do tipo: *He (Hélio)*, *Na (Sódio)* e *Fe (Ferro)*.

Em qualquer uma das três alternativas apresentadas, o processo pai e o processo filho partilham o mesmo código. Em qualquer dos casos também, o processo filho é designado como uma chamada de uma função, que admite argumentos, do código do pai.

As três alternativas de processos apresentadas diferenciam-se pelos três pontos seguintes:

i) partilha ou cópia do segmento de dados;

ii) segmento de pilha atribuído automaticamente pelo sistema, ou indicado pelo pai;

iii) possibilidade ou impossibilidade de aceder aos serviços do *Trollius*.

A tabela 6.1 compara os três tipos de processos aqui apresentados, de acordo com a sua funcionalidade.

Da análise da tabela, depreende-se que os processos tipo *He* e *Na* não são mais que *threads* do mesmo processo. Os processos do tipo *Fe* partilham uma relação semelhante à dos processos que, em

Tipo	Pai e filho partilham o segmento de dados	Criação automática do segmento de pilha	Acesso aos serviços do *Trollius*
He (hélio)	Sim	Não	Não
Na (sódio)	Sim	Sim	Não
Fe (ferro)	Não	Sim	Sim

Tabela 6.1: Alternativas para criação de processos locais no *Trollius*

ambiente UNIX, são criados com *fork()*, com a diferença de que, no *Trollius*, o novo processo começa a executar a função indicada no momento da ciração do processo.

Criação de processos em nós remotos

A criação de processos em nós remotos é executada pelos servidores do sistema, sendo obtida com base na transferência de mensagens e na criação de processos locais. Para a criação de processos em nós remotos, o *Trollius* disponibiliza duas alternativas diferentes: replicação de um processo existente num nó remoto ou criação de um novo processo num nó remoto.

A tabela 6.2 expõe as principais características de ambas as alternativas.

Função	Fonte	Destino	Argum. iniciais	Partilha de código	Partilha de dados
rploadgo()	Ficheiro no computador hospedeiro	Qualquer nó	Sim	Não	Não
rpspawn()	Processo activo	Qualquer nó excepto o do processo de origem	Sim	Não	Não
rpspawn()	Processo activo	O nó do processo de origem	Sim	Sim	Não

Tabela 6.2: Alternativas para criação de processos remotos no *Trollius*

Na tabela existem duas entradas para a função *rpspawn()*, porque o funcionamento desta é dependente de o nó de destino do novo processo ser ou não o mesmo que o nó do processo origem. No caso em que o nó de destino do novo processo é o mesmo que o nó do processo origem, esta primitiva não faz mais do que criar um processo do tipo *Fe* (*Ferro*) atrás apresentado e invocar a função principal deste, *e.g.* a função `main()` de um programa em *C*.

É de notar que os processos criados com *rpspawn()*, ainda que em nós remotos, recebem sempre uma cópia do segmento de dados do processo origem, mas começam a executar, sempre, a função inicial do programa.

6.3.4 Comunicação

A comunicação é uma das funcionalidades mais completas e mais bem conseguidas neste sistema. A nível físico, todas as comunicações no mesmo nó se efectuam através das instruções *in* e *out* dos *Transputers*[3]. Estas instruções, se o emissor e o receptor se encontram no mesmo processador,

[3]Excepção feita para as comunicações com o computador hospedeiro, que têm de recorrer a um protocolo específico (que no computador hospedeiro é implementado com base num servidor ou num *device driver*).

apenas fazem uma cópia de um bloco de memória e se os processos intervenientes se encontram em processadores distintos, mas ligados físicamente, fazem a transferência de informação através dos canais físicos do processador. Processos que residam em processadores que não estejam interligados fisicamente, não podem comunicar directamente.

Comunicação intra-*Transputer*

Quando dois processos pretendem comunicar, têm que conhecer um endereço comum de um canal de comunicação. Para que possam transferir a mensagem, ambos os processos fazem um pedido a um processo do sistema (o processo *Kernel*[4]), indicando-lhe o endereço comum a ambos. Este fará com que os dois processos se "encontrem" e transfiram a mensagem como pretendido.

A este nível, o *Trollius* propõe dois tipos de funcionalidades de comunicação:

Funções bloqueantes – Os processos receptor e emissor permanecem bloqueados até que a mensagem seja transferida de/para outro processo[5].

Funções não bloqueantes – A tentativa é abandonada imediatamente se a comunicação não se pode realizar.

Comunicação inter-*Transputer*

De forma a assegurar a possibilidade de comunicação entre quaisquer dois processos, independentemente da sua localização, o *Trollius* mantém tabelas de encaminhamento de mensagens entre qualquer par de nós do multicomputador (estas tabelas são estáticas e definidas durante o processo de lançamento do sistema).

As camadas de comunicação do *Trollius* Os diferente níveis de comunicação no *Trollius* são inspirados nos níveis OSI, tendo o nível inferior (o primeiro apresentado, em baixo) um maior desempenho e o superior (o último apresentado) uma maior funcionalidade:

Kernel – A comunicação apenas se pode realizar entre quaisquer dois processos que residam no mesmo processador;

Físico – A este nível é possível a troca de mensagens, com *rendez-vous*, entre dois processos que se encontrem em processadores adjacentes, *i.e.* ligados entre si por um canal físico de comunicação;

Dados – O envio de mensagens já pode ser não bloqueante, sendo o sistema responsável pelo seu armazenamento até serem solicitadas pelos destinatários;

Rede – Este é o nível de comunicação que apresenta o melhor compromisso entre a funcionalidade com o desempenho. Usando as primitivas de comunicação deste nível, é possível enviar mensagens bloqueantes ou não-bloqueantes entre quaisquer dois processos, ainda que não adjacentes, sendo da responsabilidade do sistema o seu encaminhamento do emissor ao receptor. O sistema também garante a partição da mensagem em pacotes, se necessário, e a respectiva junção no destinatário.

[4] O canal associado ao processo *Kernel*, que permite a comunicação com este, está fixado no código fonte do *Trollius*.
[5] A mensagem pode ser transferida de/para um processo intermediário, que faça a gestão dos *buffers*.

Transporte – Com o principal objectivo de facilitar o *debugging* dos programas, este nível disponibiliza um modelo de comunicação do tipo *request-to-send*, em que a mensagem só é realmente enviada quando o receptor estiver disponível para a receber, evitando assim o transbordamento (*overflow*) dos *buffers*.

Descritor de mensagem Uma mensagem corresponde a um conjunto de informação, que irá ser transferido entre dois processos. No *Trollius* uma mensagem é definida por um descritor (estrutura com o nome *nmsg*) que se apresenta na figura 6.3 (em código tipo *C*).

```
struct nmsg {
    int  nh_dl_event;  /* Evento de encaminhamento */
    int  nh_node;      /* Nó destino da mensagem */
    int  nh_event;     /* Evento (etiqueta) da mensagem */
    int  nh_type;      /* Tipo (de evento) da mensagem */
    int  nh_length;    /* Tamanho da mensagem */
    int  nh_flags;     /* Opções */
    int  nh_data[8];   /* Dados (inteiros) */
    char *nh_msg;      /* A mensagem a enviar */
}
```

Figura 6.3: Descritor de mensagem no *Trollius*

O descritor de mensagem é usado, da mesma forma, por todos os níveis de comunicação (à exepção do nível *kernel*, cujos pormenores não serão discutidos aqui, para o que se recomenda a consulta deq [Ohi92d, Ohi92b, Ohi92c, Ohi92a]). Dum ponto de vista geral, cada um dos campos atrás apresentados tem a seguinte funcionalidade:

nh_dl_event – Este campo é usado pelo processo responsável pelo envio de mensagens aos níveis *físico* e *dados*. Normalmente é o evento (ver mais à frente **nh_event**) associado ao processo responsável pela gestão dos canais físicos de comunicação. Este campo pode, em geral, ser ignorado pelo utilizador, *i.e.* o sistema preenche-o com os valores correctos quando necessário.

nh_node – O processo que envia a mensagem coloca neste campo o nó onde o processo destinatário se encontra. O processo receptor não usa este campo. A filtragem das mensagens dentro de um mesmo nó é feita com base nos campos **nh_event** e/ou **nh_type** (descritos a seguir).

nh_event – Neste campo indica-se um número inteiro e positivo, que corresponde ao evento de sincronização no nó de destino (equivale a uma etiqueta que é associada à mensagem). A mensagem será entregue a qualquer processo no nó de destino que pretenda receber uma mensagem e tenha especificado o mesmo valor para o evento.

nh_type – Este é o tipo de evento de sincronização no nó de destino. Corresponde a um segundo nível de filtragem de mensagens, que é aplicado *depois* da sincronização em **nh_event**, *i.e.* só é testado se existe sincronismo neste campo, se houve previamente sincronismo no campo **nh_event**. Dois processos sincronizam-se a este nível, se pelo menos um deles indicou o valor 0 (zero) no respectivo campo, ou se o resultado da operação *E binário* aplicada a estes campos (dos descritores do emissor e do receptor) for diferente de 0 (zero). A utilização deste campo é facultativa, podendo o utilizador indicar sempre o valor 0 (zero), que elimina este segundo nível de sincronização.

nh_length – No emissor, este campo indica qual o tamanho, em *bytes*, da mensagem a enviar (apontada por **nh_msg**). No receptor, indica qual o tamanho máximo que a mensagem a receber pode ter. Se o receptor indicar um valor inferior ao tamanho da mensagem, apenas esse número de *bytes* é transmitido. Após a comunicação, este campo é afectado com o número de *bytes* realmente transferidos entre os dois processos, tanto no emissor como no receptor.

nh_flags – Tal como o nome indica, este campo destina-se a indicar opções (que são associadas a cada um dos *bits* deste campo). Serve, entre outras, para indicar o tipo dos dados que a mensagem apontada por **nh_msg** contém.

nh_data – Este campo, com tamanho e tipo associado de 8 números inteiros, destina-se à transferência de informação entre os interlocutores, estando o seu uso totalmente livre para o utilizador. O sistema assume que o conteúdo deste *buffer* são números inteiros, pelo que lhe aplica as regras de conversão de formatos internos, quando necessário.

nh_msg – Apesar de declarado como um apontador para um vector de caracteres, uma mensagem pode conter qualquer tipo de dados, devendo este tipo ser indicado em **nh_flags**. O sistema fará as conversões dos dados da representação interna do emissor para a do receptor, se tal for necessário. Se a mensagem contiver informação estruturada, *i.e.* que não seja um conjunto de dados todos do mesmo tipo, então é necessário que seja o utilizador a fazer, explicitamente, a conversão estre as respectivas representações internas, recorrendo a funções que o *Trollius* disponibiliza para o efeito.

Circuitos virtuais e comunicação por grupos Quando a comunicação entre dois interlocutores não é espontânea, *i.e.* existe um fluxo de comunicação mais ou menos permanente entre ambos, é possível pedir ao *Trollius* que estabeleça um circuito virtual entre os processos em causa, permitindo-lhes assim comunicar, enviando mensagens, sem o peso extra do encaminhamento das mesmas.

O *Trollius* suporta, também, um sistema básico de comunicação por grupos, que permite associar a um *descritor de mensagem* um conjunto de *descritores de mensagem*. A utilização das primitivas de comunicação normais com estes novos descritores permite o envio de uma mensagem para múltiplos destinatários, e a combinação das mensagens de múltiplos emissores num único receptor.

6.3.5 As acções de entrada e saída sobre ficheiros

O *Trollius* disponibiliza, a todos os processos do utilizador que são executados em nós do multicomputador, um serviço de entrada/saída de/para ficheiros, sendo todos os pedidos transferidos para o computador hospedeiro e aí satisfeitos de forma transparente para o processo cliente.

Os serviços de entrada/saída sobre ficheiros, disponíveis ao processos em nós remotos, são idênticos e compatíveis com os seus equivalentes UNIX: *read()*, *write()*, *open()*, *close()*, *lseek()* e *getc()*.

6.4 Abstracções de concorrência e de comunicação

O modelo descrito no capítulo 4 apresenta quatro tipos de abstracções básicas: executor de Prolog, porta, mensagem e sinal. Os predicados também sugeridos no modelo foram divididos em seis classes: predicados de *gestão de portas de comunicação*, predicados de *controlo de comunicação*, predicados de *geração e tratamento de sinais*, predicados de *gestão de executores de Prolog*, predicados de *controlo do estado dos executores de Prolog* e predicados de *informação*.

Nas secções seguintes indica-se de que modo cada uma das abstracções e classes de predicados foram implementadas.

6.4.1 Executor de Prolog

O conceito de executor de Prolog, como descrito na secção 4.2.1, é suportado por um programa executado sobre o sistema de operação *Trollius*, que implementa uma máquina virtual que interpreta código compilado para a WAM. Sendo o executor um processo sob o controlo do sistema de operação *Trollius*, a implementação das abstracções que foram propostas no modelo apenas exige a sua correspondência com as do sistema de operação, através de extensões e alterações ao executor.

Um executor pode ser caracterizado pelo conjunto de recursos de que a WAM necessita para execução — *i.e.* os segmentos de código, pilha e *heap* — e pelos canais de entrada/saída que lhe permitem a comunicação com os outros processos e com a consola. A figura 6.4 exemplifica de que modo os recursos da WAM são projectados (*mapped*) nos recursos de um processo convencional.

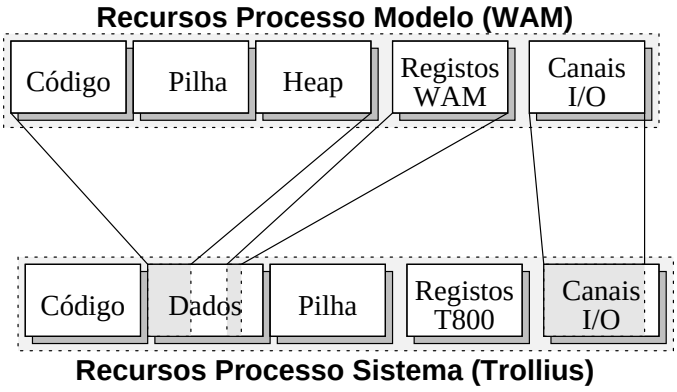

Figura 6.4: Projecção dos “recursos de um executor” nos “recursos de um processo”

Os segmentos e os registos do executor (WAM) são projectados no segmento de dados do processo, os primeiros em vectores/tabelas e os segundos em variáveis globais. Os canais de entrada/saída do executor de Prolog têm correspondência directa nos canais de entrada/saída do processo.

No sistema *Trollius*, o *identificador de processo (PID)* é local ao nó, *i.e.* é possível existirem dois processos com o mesmo *PID* em nós diferentes, de tal forma que para se identificar univocamente um processo, é necessário indicar o seu *PID* e o nó onde se encontra. No modelo proposto o *identificador de executor* (***E**xecutor **ID**entifier*) deve identificar univocamente um processo, de forma independente da sua localização, pelo que se optou por adoptar a seguinte fórmula para geração de um *EID*:

$$\mathbf{EID = PID * 100 + NID}$$

Onde **EID** representa o *identificador de executor*, **PID** o *identificador de processo* e **NID** o *identificador do nó* onde o executor está residente[6].

[6]Esta estratégia assume que o multicomputador não tem mais de 100 nós (o que é verdade para o multicomputador usado no desenvolvimento do protótipo) mas, caso esta condição não se verifique, a adaptação resume-se à alteração de uma constante no código fonte do executor de Prolog e recompilação deste.

6.4.2 Porta

O modelo de comunicação por portas, ou suas variantes, é suportado por muitos núcleos de sistemas de operação para multicomputadores, *e.g. Mach* [B+88] e *Chorus* [RM87], por ser um modelo bastante intuitivo e de fácil utilização. Quando os sistemas de operação não disponibilizam este modelo de comunicação directamente é, geralmente, possível implementá-lo sobre os mecanismos de comunicação existentes, programando um ou mais servidores, de nível utilizador.

Como o *Trollius*, na versão usada para implementação do protótipo (versão 2.2), não suporta directamente o modelo de comunicação por portas, foi necessário estendê-lo com uma camada de comunicação adicional, de forma a disponibilizar este modelo de comunicação aos executores de Prolog (ver figura 6.7, página 73).

Extensão ao Trollius para suporte de comunicação por portas

Para a implementação dos predicados de comunicação apresentados no modelo, foi necessário estender o sistema de operação *Trollius* de forma a suportar comunicação por portas, em complemento do modelo de comunicação original (orientado para nós e eventos).

Para suportar o modelo de comunicação por portas, foi necessário assegurar o estabelecimento e manutenção de pares

(nome-da-porta / (nó-receptor, evento))

tais que, a cada nome de porta se associa um nó e um evento do sistema de comunicação original do *Trollius*. A gestão destes pares está entregue a um processo único, em execução num dos nós do multicomputador[7], mais especificamente o nó com identificador 0 (zero)[8]. Os eventos usados pelas portas foram escolhidos entre os valores possíveis e consideram-se reservados, estando os limites inferior e superior definidos como constantes num ficheiro do código fonte do q*servidor de portas*.

Como a gestão do modelo de portas não se limita à associação dos pares atrás apresentados, optou-se por integrar, no *servidor de nomes*, funcionalidades e estruturas de dados complementares, transformando-o assim num *servidor de portas*, de forma a que toda a informação necessária à gestão das portas ficasse concentrada num único processo, evitando assim, para se obter a satisfação de um serviço, a troca de múltiplas mensagens entre um pretenso "servidor de portas" e um "servidor de nomes".

Descritor de porta no processo "cliente": do ponto de vista de um processo utilizador de portas, um *descritor de porta* é um número inteiro. Este número indica o índice de uma tabela, interna ao processo, onde se têm informações locais sobre a porta em questão. Esta informação pode ser descrita pela estrutura (em código tipo *C*) apresentada na figura 6.5.

Com esta informação guardada localmente, toda a comunicação entre processos é directa, *i.e.* não passa pelo servidor de portas. Este apenas intervém nas acções de criação, de destruição, de abertura, de fecho de portas e, como é natural, na obtenção de informação sobre as portas.

[7] Devido às características do mecanismo de comunicação do *Trollius*, o nó onde se executa o *servidor de nomes* tem de ser fixo.

[8] Uma alternativa seria pôr este servidor em execução no computador hospedeiro, mas a comunicação com os processos do sistema *Trollius* que estão em execução no computador hospedeiro é muito mais pesada, o que degradaria de forma considerável o desempenho destas primitivas.

```
struct portclient {
    int port_node;      /* Identificador do nó onde se encontra */
                        /*     a porta */
    int port_event;     /* Número do evento associado à porta */
    int port_tpd;       /* Descritor desta mesma porta */
                        /*     no "servidor de portas" */
    int port_tflags;    /* Outras informações associadas à porta */
};
```

Figura 6.5: O descritor de porta no processo “cliente”

Descritor de porta no “servidor de portas”: para poder garantir a coerência de todo o sistema de comunicação por portas, foi necessário guardar, no servidor, informação extra associada às portas. Esta informação, que o servidor de portas mantém, pode ser descrita pela estrutura (em código tipo *C*) apresentada na figura 6.6.

```
struct pdesc {
    int4 p_tpd;             /* Descritor único (global) da porta, no */
                            /*     sistema Trollius */
    int4 p_tflags;          /* Outras informações associadas à porta */
    int4 p_owner;           /* Identificador de processo (PID) */
                            /*     do dono da porta */
    int4 p_node;            /* Nó onde se encontra a porta */
    int4 p_event;           /* Evento associado à porta */
    char p_name[PNAMEMAX]; /* O nome da porta */
    LIST *p_users;          /* Informacao (PID's e nós) dos processos */
                            /*     que estão a usar esta porta */
};
```

Figura 6.6: O descritor de porta no “servidor de portas”

Quando um processo pretende criar uma porta, é comunicado ao servidor o respectivo nome e, caso esse nome não esteja já a ser usado por outra porta, toda a estrutura será preenchida. Quando um processo pretende abrir uma porta, indica o respectivo nome ao servidor e este, para além de lhe devolver toda a informação necessária para preencher a sua tabela local (apresentada na figura 6.5), actualiza a lista de processos utilizadores dessa porta, com informações sobre o novo processo.

6.4.3 Mensagem

O envio de mensagens, tal como já foi dito na secção anterior, não passa pelo *servidor de portas*, envolvendo apenas os dois processos interlocutores: o emissor da mensagem e o receptor da mesma.

A nível do executor de Prolog, enviar uma mensagem para uma porta envolve a conversão dos parâmetros passados ao predicado, em dados tipo *C*, e a invocação das funções equivalentes no *Trollius* estendido. Como o parâmetro associado à mensagem é um termo Prolog, a conversão é obtida através da invocação de uma forma especial do predicado write/1, que escreve o argumento numa cadeia de caracteres em memória, em vez de no ecrã.

A nível do *Trollius*, enviar uma mensagem para uma porta corresponde a fazer uma consulta à estrutura de dados local, de forma a obter o nó e o evento associados a essa porta, recorrendo de

seguida aos mecanismos, já existentes no *Trollius*, para fazer a entrega da mensagem.

6.4.4 Sinal

Um *sinal*, tal como apresentado no capítulo 4, corresponde ao envio assíncrono de uma mensagem para uma porta especial (com nome reservado) e pela recepção e tratamento, também assíncronos, pelo destinatário.

A porta especial, por onde os processos recebem os sinais, é criada automaticamente durante a fase de inicialização dos processos. O nome (reservado) da porta é composto pela cadeia de caracteres "**$sig-**" concatenada com o "**EID**" do processos, *e.g.* o executor com *identificador de executor* **325412** terá uma porta de sinais com o nome **$sig-325412**.

Para forçar a recepção assíncrona da mensagem, a mensagem é precedida do envio de um sinal, controlado pelo *Trollius*. É durante o tratamento desse sinal, no receptor, que a mensagem é recebida e guardada numa fila de mensagens contendo os sinais pendentes, que serão tratadas assim que possível.

Na rotina de tratamento de sinais, o processo fica temporariamente incapaz de receber outros sinais, do mesmo ou de outro processo no sistema. Como o *Trollius* garante que os sinais enviados nestas circunstâncias não se perdem, ficando armazenados pelo sistema, de forma a poderem ser entregues assim que o processo destinatário desinibir os sinais, não existe qualquer perigo de perder sinais.

6.5 Predicados da interface

6.5.1 Predicados de gestão de portas e de controlo da comunicação

Após ter estendido o *Trollius* com um mecanismo de comunicação por portas, tal como descrito na secção 6.4.2, a implementação dos predicados de comunicação ficou bastante simplificada, limitando-se a verificar a validade dos argumentos passados aos predicados, *e.g.* se os argumentos de retorno correspondem realmente a variáveis livres, e a transferi-los como argumentos (depois de devidamente convertidos em representações internas válidas) para as funções de biblioteca acima descritas.

Foi assim possível usar um mecanismo de comunicação independente do sistema de operação que, caso não tenha já um mecanismo de comunicação por portas, pode ser implementado como uma extensão ao sistema de operação, separando toda a implementação da comunicação em três camadas, tal como apresentado na figura 6.7: a do sistema de operação, a de comunicação por portas (que pode eventualmente ser vazia, caso o sistema de operação suporte, de base, este modelo de comunicação) e a dos predicados de comunicação do Prolog.

6.5.2 Predicados de geração e tratamento de sinais

A nível do executor de Prolog, o envio de um sinal corresponde ao envio de um sinal do *Trollius* seguido do envio de uma mensagem. O processamento do envio de uma mensagem já foi descrito na secção anterior e, para um processo enviar um sinal *Trollius* a outro processo, apenas tem de conhecer o seu *identificador de executor* e fazer uma chamada ao sistema (*Trollius*), com esse identificador como parâmetro.

O executor de Prolog usado na implementação do protótipo é uma versão alterada do original [Dia90], com funcionalidade acrescida, para suportar a execução concorrente de golos Prolog, através

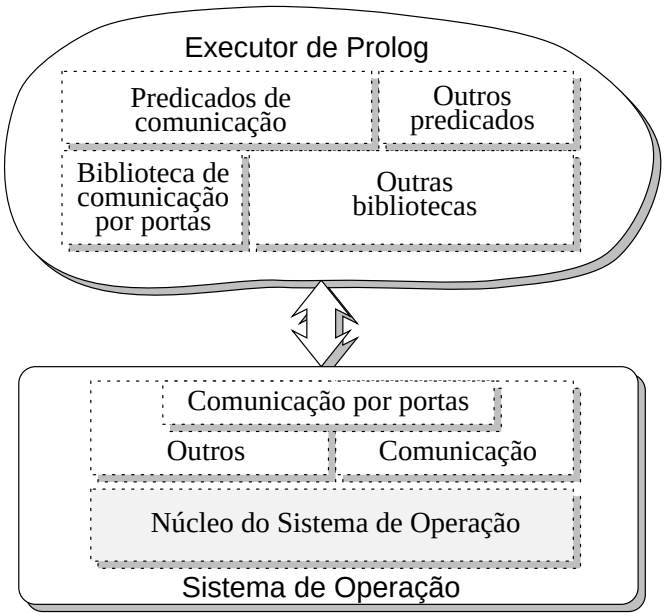

Figura 6.7: Camadas de comunicação no protótipo

de co-rotinas ou *threads* Prolog [Mar92]. O tratamento assíncrono de sinais foi implementado recorrendo a esta funcionalidade, mas apenas por questões de simplicidade da implementação[9].

Sempre que um executor de Prolog executa a instrução da WAM *CallInst*[10], verifica antes, se existem mensagem de sinais por processar e, caso existam, é criado um *thread* Prolog para resolver o golo enviado na mensagem. O *thread* Prolog criado para executar o golo enviado no sinal, assume uma prioridade mais elevada do que a do fluxo de execução principal, de forma a que tenha o exclusivo do processador (neste executor) até terminar.

Quando o fluxo de execução corrente está associado a um *thread* (e portanto a um sinal), não são feitas as verificações na instrução *CallInst*, evitando assim que uns sinais possam interromper outros sinais. Se se pretendesse a implementação de um mecanismo de sinais com níveis de prioridade, *i.e.* os sinais de nível mais elevado interromperem sinais de nível inferior, dever-se-ia verificar se algum dos sinais pendentes tinha um nível superior e, nesse caso, tratá-los sequencialmente da mesma forma, *i.e.* através da criação de mais um *thread* com uma prioridade superior à do *thread* corrente.

6.5.3 Predicados de gestão de executores de Prolog

Se o sistema de operação que suporta a implementação do modelo admite a criação dinâmica de processos, a implementação dos predicados de criação e destruição de executores de Prolog (os predicados sys_create_wam/7 e sys_destroy_wam/2) é baseada na criação de processos executores de Prolog, quando necessários. Se o sistema de operação não admite a criação dinâmica de processos, esta pode ser simulada com o recurso a um *gestor de executores*, que simule a criação dinâmica, através da gestão de um reservatório (*pool*) de executores. O protótipo implementado recorre ao sistema de operação (o *Trollius*) para a criação dinâmica de executores, quando necessário.

Para implementar o predicado sys_at_halt/2 foi necessário interceptar a função de terminação do

[9] Em [CMC87, LB91] foi implementado um mecanismo de sinais em Prolog sem recurso a *threads* Prolog.

[10] Na realidade a instrução *ExecuteInst*, que é apenas uma optimização da instrução *CallInst* para implementar a *optimização da recursividade à cauda*, também é tida em consideração.

executor, *i.e.* a instrução halt/0, de forma a que, antes de terminar, executasse todos os predicados que haviam sido indicados nas diversas invocações de sys_at_halt/2.

Os predicados a executar são adicionados à base de dados do executor, mas encapsulados numa estrutura especial com a forma **$at_halt(Goal)**, onde `Goal` indica o predicado a executar quando o executor terminar. Os predicados são adicionados à cabeça, *i.e.* antes dos demais com o mesmo nome e a mesma aridade, de forma a que a recolha e execução dos mesmos se faça pela ordem inversa da que foram adicionados.

O predicado frac_sleep/1 faz a aproximação necessária ao argumento, para concordar com a granularidade do relógio admitida pelo sistema de operação, recorrendo depois a este, para que o executor seja bloqueado durante o tempo indicado.

6.5.4 Predicados de controlo do estado do executor de Prolog

Os predicados de controlo de estado são apenas dois, um que salvaguarda o estado do executor de Prolog, criando um *ponto de possível retrocesso*, e outro que repõe o estado salvaguardado no *ponto de retrocesso*.

Para conseguir este efeito, o predicado save_state/1 está implementado em Prolog — é carregado automaticamente num ficheiro de inicialização do executor — e tem duas cláusulas alternativas. Assim, quando é invocado, o executor cria um *ponto de escolha* para que, no caso da primeira alternativa falhar, possa tentar a segunda. É neste ponto de escolha que fica salvaguardado o estado do executor.

A segunda alternativa do predicado save_state/1 está feita de tal modo que falha se não está a ser executada por imposição do predicado restore_state/1, não influenciando em nada o processo de resolução caso esta condição não se verifique.

Quando é invocado o predicado restore_state/1, é forçado o retrocesso até ao *ponto de escolha* criado no save_state/1 correspondente, mas agora a segunda alternativa deste predicado sucede invocando-se a si próprio recursivamente[11], de forma a salvaguardar mais uma vez o estado do executor.

6.5.5 Predicados de informação

A implementação dos predicados de informação é baseada, no que se refere ao estado do sistema subjacente, em primitivas afins do sistema *Trollius*. Apesar de estes predicados se basearem em primitivas de funcionalidade básica, comuns a quase todos os sistemas de operação para multicomputadores, apresentam-se como aqueles que, entre os propostos neste trabalho, são potencialmente mais dependentes do sistema de operação.

Pretendeu-se oferecer, com estes predicados, alguma visibilidade sobre a afectação efectiva dos executores aos processadores reais, de forma a apoiar possíveis utensílios de *debugging* e de monitorização do estado do sistema.

Um predicado que necessitou de tratamento suplementar foi o sys_wam_state/3 pois, como o envio bloqueante de um termo corresponde, na realidade, ao envio do termo e à recepção de uma confirmação, a chamada ao sistema *Trollius* pode indicar que o executor está bloqueado na recepção de uma mensagem (uma confirmação), quando se pretende, na realidade, que o predicado informe que o executor está bloqueado num envio. Para eliminar este problema, foi necessário filtrar os resultados

[11] Esta invocação recursiva não tem influências negativas no espaço ocupado na pilha do executor, uma vez que este está equipado com uma *optimização de recursividade à cauda*.

obtidos com a chamada ao sistema *Trollius* de forma a corrigi-lo. A filtragem foi também necessária para eliminar as indicações de estado de todos os processos que não correspondem a executores de Prolog.

6.6 Conclusões

Pela descrição da implementação do protótipo apresentada nesta capítulo, conclui-se que o modelo proposto é realizável sobre um multicomputador baseado em *Transputers*.

Da descrição da implementação dos predicados, é possível extrair a conclusão de que estes não recorrem a funcionalidades demasiado específicas da arquitectura (*Meiko CS/1*) ou do sistema de operação (*Trollius*) usados, permitindo antever poucas dificuldades no transporte do mesmo modelo para uma arquitectura e/ou sistema de operação diferentes.

Os predicados que são, potencialmente, mais dependentes do sistema, *i.e.* os predicados de informação, são também aqueles que com mais facilidade se reescrevem, pois a sua semantica operacional é clara, e a sua implementação resume-se em invocar o sistema de operação para obter as informações necessárias e depois “tratá-las”, de forma a serem apresentáveis a um executor de Prolog.

Capítulo 7

Conclusões e trabalho futuro

O trabalho relatado nesta dissertação insere-se numa linha de investigação mais vasta, para a qual o desenvolvimento de sistemas de suporte à execução paralela e distribuída de programas é um objectivo fundamental. Em particular os modelos de programação em lógica são considerados como muito interessantes, dado o estilo declarativo e a clareza da programação que promovem, se nelas for possível contemplar formas de paralelismo, disponibilizadas pela geração corrente de arquitecturas de computadores.

A investigação de modelos de programação em lógica, dotados de construções para a especificação explícita do paralelismo e da comunicação, tem sido um preocupação dominante desde os anos 80, a nível da comunidade internacional e, em particular, no Departamento de Informática da Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa. O estudo desses modelos está fora do âmbito desta dissertação, mas forneceu a motivação para a concepção e implementação do modelo intermédio proposto.

Este modelo suporta funções de controlo dinâmico de executores de Prolog, diversas formas de comunicação entre processos e um esquema genérico de processamento de interrupções a nível do executor de Prolog, para além de disponibilizar funções de controlo do estado interno do executor. Suporta também funções para a obtenção de informações sobre o estado de um executor individual e sobre o estado global de um sistema distribuído de múltiplos executores. Para a definição destas funcionalidades, analisaram-se os requisitos postos por modelos baseados em processos distribuídos.

As duas conclusões principais deste trabalho, que se referem à avaliação do modelo proposto e à viabilidade da sua realização sobre arquitecturas paralelas, serão analisadas de seguida.

O trabalho realizado mostrou que, em termos de funcionalidades do modelo, as suas primitivas oferecem flexibilidade suficiente para suportar linguagens lógicas distribuídas de mais alto nível, conforme ilustrado pela realização do operador paralelo "//" na secção 5.3.

Mostrou-se também que o modelo é adequado para desenvolvimento de aplicações a um nível de programação intermédio, *i.e.* uma forma de "programação de sistemas distribuídos" com base em "executores de Prolog comunicantes". Uma experiência simplificada foi descrita no capítulo 5, no qual se realizou uma camada de distribuição de golos Prolog, assente sobre um serviço de gestão de múltiplos executores e, cujo objectivo, é permitir ao utilizador explorar as estratégias de paralelização mais adequadas a cada aplicação. A prevista utilização do modelo proposto em dois projectos recentes (PROLOPPE [P$^+$94] e PADIPRO [eJCC94]) contribuirá para avaliar e apurar melhor as opções tomadas neste trabalho.

Salienta-se ainda o facto de o modelo ter sido concebido sob a forma de extensões a um sistema Prolog convencional (seja baseado num interpretador seja num compilador e máquina abstracta), o que o potencia para o suporte de uma plataforma intermédia e integradora (do ponto de vista de gestão

das unidades de concorrência e das comunicações) em sistemas heterogéneos.

A investigação futura indicará, certamente, quais as direcções mais adequadas para melhorar e/ou corrigir as funcionalidades oferecidas pelo modelo. Em particular serão consideradas as extensões para suporte de concorrência interna a um processo, sob a forma de *threads* Prolog, e as extensões para comunicação por grupos.

En termos de viabilidade de realização do modelo proposto sobre arquitecturas paralelas, a experiência relatada sobre o sistema *Trollius* e uma arquitectura baseada em *Transputers* confirma que foi conseguido um razoável grau de independência face à arquitectura real. O *Trollius* revelou-se um sistema adequado para o suporte do modelo proposto. Contudo, as limitações que o ambiente de operação do *Trollius* impõe, consistindo numa configuração física com uma única máquina hospedeira ligada a um multicomputador, necessitam de ser removidas, de modo a ser possível contemplar a execução sobre uma rede local heterogénea com múltiplas estações UNIX e múltiplos multicomputadores.

Ao confrontar o modelo proposto com tais ambientes heterogéneos, ter-se-ão, possivelmente, de analisar mais profundamente os requesitos do sistema de suporte à execução (apresentados na secção 2.4.2), nos seus aspectos de configuração dinâmica e tratamento da heterogeneidade. Nesse contexto, analisar-se-ão plataformas e sistemas de operação subjacentes, alternativos ao *Trollius* e que possam proporcionar, não só facilidade no transporte do modelo, como conduzir a uma possível implementação mais eficiente.

Uma outra direcção, ainda que relacionada com a anterior, envolve o estudo das realizações do modelo directamente sobre micro-núcleos, e o desenvolvimento dos mecanismos de apoio à monitorização da execução de programas que permitam suportar utensílios de avaliação de desempenho, *debugging* e visualização das computações paralelas.

Estas direcções de investigação serão exploradas no contexto dos referidos projectos e, possivelmente, enquadradas em trabalhos de preparação de novas teses de mestrado e de teses de doutoramento.

Agradecimentos

Quero agradecer a todos aqueles que, directa ou indirectamente, contribuíram para a realização deste trabalho.

Em particular, gostava de agradecer ao meu orientador, José Cunha, pelo muito apoio dado e grande disponibilidade que sempre demonstrou ao longo de todo este tempo.

Ao Rui Marques, pelos muito e importantes conselhos dados, em particular sobre a versão do NanoProlog usada.

Ao Pedro Medeiros, pela bibliografia que me disponibilizou, pelas muitas conversas que tivemos e pela disponibilidade que sempre demonstrou.

Ao Vitor Duarte, pela co-instalação do LaTeX na minha estação de trabalho.

A todos os meus colegas do Departamento de Informática, pelo seu contributo pelo óptimo ambiente de trabalho e boa camaradagem.

À Teresa, pelo seu apoio e grande compreensão demonstrada durante todo este (longo) tempo em que não teve a minha atenção.

Bibliografia

[Agh86] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, 1986.

[AK90] Aït-Kaci. The WAM: A (real) tutorial. Relatório técnico, Digital, Paris Research Laboratory, Janeiro 1990.

[AS83] G. R. Andrews e F. B. Schneider. Concepts and notations for concurrent programming. *ACM Computer Surveys*, 15(1), 1983.

[Ass94] Luís Assunção. Contribuição para o estudo de modelos de memória partilhada distribuída. Proposta de Tese de Mestrado, Departamento de Informática, Universidade Nova de Lisboa, Portugal, Setembro 1994.

[B+88] Robert V. Baron et al. MACH Kernel interface manual. Relatório técnico, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, USA, Fevereiro 1988.

[Bal90] H. Bal. *Programming Distributed Systems*. Prentice-Hall, 1990.

[BC90] A. Borgi e P. Ciancarini. The Concurrent language Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1), 1990.

[BL84] A. Birrel e R. Levin. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1), 1984.

[CA84] J. C. Cunha e J. N. Aparicio. Delta Prolog implementation — progress report No. 1. Relatório Técnico RT-6/84, Departamento de Informática, Universidade Nova de Lisboa, Portugal, Dezembro 1984.

[Car91] Manuel F. Carvalhosa. Concepção e implementação de uma máquina abstracta para o Delta Prolog. Dissertação de Mestrado, Instituto Superior Técnico, Lisboa, Portugal, Outubro 1991.

[CFP89] J. C. Cunha, M. C. Ferreira, e L. M. Perreira. Programming in Delta Prolog. In *Proceedings of International Conference on Logic Programming*, Portugal, 1989. Departamento de Informática, Universidade Nova de Lisboa.

[CG83] Keith L. Clark e Steve Gregory. Parlog: A parallel logic programming language. Relatório técnico, Department of Computing, Imperial College, London, UK, Maio 1983.

[CG84] K. Clark e S. Gregory. Parlog: parallel programming in logic. Relatório Técnico DOC 84/4, Imperial College London, Abril 1984.

[CHR] D. Cohen, M. HuntBach, e G. A. Ringwood. Logical Occam. Relatório técnico, Department of Computer Science, Queen Mary and Westfield College, Londres, UK.

[Cia93a] Paolo Ciancarini. Coordinating rule-based software processes with ESP. Relatório Técnico UBLCS-93-8, Laboratory of Computer Science, University of Bologna, Itália, Abril 1993.

[Cia93b] Paolo Ciancarini. Distributed programming with logical tuple spaces. Relatório Técnico UBLCS-93-7, Laboratory of Computer Science, University of Bologna, Itália, Abril 1993.

[CM81] W. F. Clocksin e C. S. Melish. *Programming in Prolog*. Springer-Verlag, 1981.

[CMC87] José Cunha, Pedro Medeiros, e Manuel Carvalhosa. Interfacing Prolog to an operating system environement: Mechanisms for concurrency and parallelism control. Relatório técnico, Departamento de Informática, Universidade Nova de Lisboa, Portugal, Abril 1987.

[CMP88] José A. Cunha, Pedro D. Medeiros, e Luís M. Pereira. PARALOGISM Final Technical Report. Relatório Técnico Projecto 4136, ESPRIT Parallel Computing Action, Fevereiro 1988.

[Cun85] J. A. Cunha. Issues in Delta Prolog implementation: Report on a research stay at the Argonne National Laboratory. Relatório Técnico RT-30/85, Departamento de Informática, Universidade Nova de Lisboa, 1985.

[Cun88] José A. Cunha. *Execução Concorrente de uma Linguagem de Programação em Lógica*. Dissertação de Doutoramento, Departamento de Informática, Universidade Nova de Lisboa, Portugal, Setembro 1988.

[Cun91] J. C. Cunha. Fundamentos de sistemas distribuídos. Relatório Técnico DI-RT-60/91, Departamento de Informática, Universidade Nova de Lisboa, Portugal, 1991.

[Cun94] José A. Cunha. Research directories in parallel and distributed processing. Relatório técnico, Departamento de Informática, Universidade Nova de Lisboa, Portugal, 1994. 1st SEPP Project Meeting, University of Westminster.

[DGJP93] Frédéric Desprez, Cyrille Gavoille, Bruno Jargot, e Makan Pourzandi. Tests des performances des communications de la machine VOLVOX IS-860. Relatório Técnico LIP-TR-93-02, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, França, Março 1993.

[Dia90] Artur Miguel Dias. Implementação de um sistema de programação em lógica contextual. Dissertação de Mestrado, Instituto Superior Técnico, Lisboa, Portugal, Setembro 1990.

[eJCC94] Luís M. Pereira e José C. Cunha. Parallel Distributed Prolog and Aplications—PADIPRO. External European Research Project, 1994.

[Eli] A. Eliëns. Distributed logic programming for artificial intelligence. *AI Communications*.

[Eli92] A. Eliëns. *DLP: a Language for Distributed Logic Programming*. Willey, 1992.

[Fly72] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972.

[For94a] HPF Forum. High Performance Fortran language specification, 1994.

[For94b] Message Passing Interface Forum. Mpi: A message-passing interface standard. Relatório Técnico Computer Science Department Technical Report CS-94-230, University of Tennessee, Knoxville, TN, May 5 1994.

[G+94] Al Geist et al. *PVM 3 User's Guide and Reference Manual*. Engineering Physics e Mathematics Division, Oak Ridge National Laboratory, USA, Setembro 1994.

[Han78] P. Brinch Hansen. Distributed Processes: a concurrent programming concept. *Communications of the Association for Computing Machinery*, 21(11), 1978.

[Hoa78] C. A. R. Hoare. Communicating sequencial processes. *Communications of the Association for Computing Machinery*, 21(8), 1978.

[Kow74a] R. Kowalski. Logic for problem solving, 1974.

[Kow74b] R. Kowalski. Predicate Logic as a programming language. In *Proceedings of IFIP Congress '94*, páginas 569–574, Amesterdão, Holanda, 1974.

[KW92] P. Kacsuk e Michael Wise, editores. *Implementations of Distributed Prolog*. Willey, 1992.

[LB91] João M. Lourenço e José L. Borges. Ambiente de execução de processos Prolog para arquitecturas baseadas em Transputers. Relatório Técnico RT-63/91, Departamento de Informática, Universidade Nova de Lisboa, Portugal, Dezembro 1991.

[Lim88a] INMOS Limited. *Occam 2 Reference Manual*, 1988.

[Lim88b] INMOS Limited. *The Transputer Instruction Set: a compiler writer's guide*. Prentice-Hall, 1988.

[Llo84] J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1984.

[Lou93a] João M. Lourenço. Conceitos e funcionalidades dos sistemas baseados em micro-núcleos. Relatório Técnico RT-3/93, Departamento de Informática, Universidade Nova de Lisboa, Portugal, 1993.

[Lou93b] João M. Lourenço. Introdução ao sistema PVM (Parallel Virtual Machne). Relatório Técnico RT-4/93, Departamento de Informática, Universidade Nova de Lisboa, Portugal, 1993.

[Mar92] Rui F. Marques. Sistema de suporte à execução distribuída de programas Prolog compilados para uma máquina intermédia. Relatório de projecto de Licenciatura em Eng. Informática, Departamento de Informática, Universidade Nova de Lisboa, Portugal, Dezembro 1992.

[Med89] Pedro Medeiros. Comunicação em sistemas de operação distribuídos. Provas de Aptidão Pedagógica e Capacidade Científica RT-28/89-DI-UNL, Departamento de Informática, Universidade Nova de Lisboa, Portugal, Julho 1989.

[Mei90a] Meiko Limited, Birstol. *CSTools Communication Library for C Programmers*, 1990.

[Mei90b] Meiko Limited, Birstol. *CSTools for SunOS*, 1990.

[Mon83] Luís Monteiro. *Uma Lógica para Processos Distribuídos*. Dissertação de Doutoramento, Departamento de Informática, Universidade Nova de Lisboa, Portugal, 1983.

[Mon86] Luís Monteiro. A theory of distributed programming in logic. Relatório técnico, Departamento de Informática, Universidade Nova de Lisboa, Portugal, 1986.

[Nel81] B. Nelson. Remote Procedure Calls. Relatório Técnico CSL-81-9, Xerox Parc, Palo Alto, 1981.

[Ohi92a] Ohio Super Computer Center, Columbus, USA. *Ohio Trollius 2.2 Release Notes*, Março 1992.

[Ohi92b] Ohio Super Computer Center, Columbus, USA. *Trollius Command Reference*, Março 1992.

[Ohi92c] Ohio Super Computer Center, Columbus, USA. *Trollius Intalation Guide*, Março 1992.

[Ohi92d] Ohio Super Computer Center, Columbus, USA. *Trollius Tutorial Introduction in C*, Março 1992.

[Ohi94] Ohio Supercomputer Center, The Ohio State University, Columbus, USA. *LAM for C Programmers*, versão 2.2.7 edição, Janeiro 1994.

[P+94] Luís M. Pereira et al. Programação Paralela em Lógica com Extensões—PROLOPPE. Projecto JNICT, 1994.

[Per83] Fernando Pereira, editor. *C-Prolog Users Manual*. Department of Artificial Inteligence, Edinburgh, 1983.

[PMCA86] L. M. Pereira, L. Monteiro, J. Cunha, e J. N. Aparicio. Delta prolog: a distributed backtracking extension with events. In Ehud Shapiro, editor, *Third International Conference on Logic Programming*, páginas 69–83, Londres, Julho 1986.

[PN84] Luís M. Pereira e R. Nasr. Delta-Prolog: a distributed logic programming language. In *Proceedings of Fifth Generation Computer Systems, ICOT84*, páginas 283–291, Tokyo, 1984.

[Rai93] Sanjay Raina. *Emulation of a Virtual Shared Memory Architecture*. Dissertação de Doutoramento, Facuty of Engeneering, University of Bristol, UK, Setembro 1993.

[RM87] M. Rozier e J. L. Martins. *Distributed Operating Systems: Theory and Practice*, capítulo The CHORUS distributed operating system: some design issues, páginas 262–287. Springer-Verlag, Berlin, 1987.

[Rou75] P. H. Roussel. *Prolog: manuel de reference et d'utilization*. Groupe d'Intelligence Artificielle, Univ. de Aix-Marseille II, 1975.

[Sar89] Vijay A. Saraswat. *Concurrent Constraint Programming Languages*. Dissertação de Doutoramento, Carnegie-Mellon University, USA, Janeiro 1989.

[SC90] Luís F. Silva e João P. Cabral. Definição e implementação de software de sistema para a execução de programas numa arquitectura distribuída baseada em transputers. Relatório Técnico RT-1/90-DI/UNL, Departamento de Informática, Universidade Nova de Lisboa, Portugal, Janeiro 1990.

[Sha83] Ehud Y. Shapiro. A subset of Concurrent Prolog and its implementation. Relatório Técnico TR-003, Institute for New generation Computer technology, Janeiro 1983.

[SKL90] Vijay A. Saraswat, Ken Kahn, e Jacob Levy. Janus: A step towards distributed constraint programming. Relatório técnico, Xerox PARC and Technion Haifa, 1990.

[Tan90] A. S. Tanenbaun. Experiences with the Amoeba Distributed Operating Sistem. *Communications of the Association for Computing Machinery*, 33(12):46–63, 1990.

[Tan92] A. S. Tanenbaun. *Modern Operating Systems*. Prentice-Hall, 1992.

[vEK77] M. H. van Emden e R. A. Kowalski. The semantics of predicate logic as a programming languge. *Jounal of the ACM*, 24(4):733–742, Outubro 1977.

[War83] David D. H. Warren. An abstract Prolog intruction set. Nota Técnica 309, SRI International, Menlo Park, CA, Outubro 1983.

[War88] David D. H. Warren. Implementations of Prolog. In *International Conference and Sysmposium on Logic Programming*, volume Tutorial No. 3, Seattle, WA, Agosto 1988.

[Y+87] M. Young et al. The duality of memory and communication in the implementation of a multiprocessor operating system. *ACM*, Outubro 1987.