Introduzindo o Paralelismo OU na Programação em Lógica com Restrições

Patrícia Kayser Vargas - Cláudio Fernando Resin Geyer
{kayser, geyer}@inf.ufrgs.br
Instituto de Informática - UFRGS
Caixa Postal 15064, Porto Alegre - RS
Telefone: (051) 316-6802 - Fax: (051) 319-1576

Resumo

O objetivo da programação em lógica com restrições (*Constraint Logic Programming* ou CLP) é obter maior eficiência do que as linguagens em lógica durante a execução através do uso de restrições para descartar opções inválidas, diminuindo o espaço de busca do problema. A execução em paralelo é uma das opções para melhorar ainda mais o seu desempenho. Esse artigo propõe um modelo multi-seqüencial de exploração de paralelismo OU em CLP, onde cada trabalhador possui uma máquina CLP completa sobre domínios finitos. Além disso, propõe-se uma política de escalonamento de tarefas distribuída. O protótipo pclp(FD) será baseado nesse modelo. Ele facilitará a inclusão de novas características no futuro. Para tal, serão utilizados objetos distribuídos.

Abstract

The aim of *Constraint Logic Programming* (CLP) is speedup logic languages during execution through constraint using to discard invalid options, decreasing search space. Parallel execution is one option to increase even more its performance. This paper proposes an OR Parallel multisequential model for CLP, where each worker has a complete CLP engine over finite domain. Besides, one distributed scheduling has been proposed. The pclp(FD) prototype will be based in this model. It will facilitate new features extension in the future. To do that, it will be used distributed object.

1. Introdução

A programação em lógica apresenta como grande atrativo o seu alto poder de expressão que permite a modelagem de uma série de problemas de forma intuitiva. Uma das linhas de pesquisa mais promissoras é a programação em lógica com restrições (*Constraint Logic Programming* ou CLP) [JAF 94] [FRU 93]. As linguagens em lógica como Prolog efetuam a resolução de um problema utilizando a operação de unificação. CLP substitui a operação de unificação pela resolução de restrições (*constraints solving*) sobre um domínio particular. O objetivo é prover maior eficiência durante a execução através do uso de restrições para descartar opções inválidas, diminuindo o espaço de busca.

Apesar do bom desempenho das linguagens CLP, algumas classes de problemas ainda não podem ser resolvidas em tempo viável. Uma opção para melhorar a sua eficiência é a execução em paralelo. Atualmente, a exploração de paralelismo é uma boa alternativa devido ao grande número de plataformas disponíveis principalmente redes de computadores.

A partir da década de 80 foram desenvolvidos vários estudos de paralelização automática de Prolog. O paralelismo OU é um dos tipos de paralelismo implícito que podem ser explorados. Várias propostas para a exploração de paralelismo OU utilizam o modelo multi-seqüencial, que se caracteriza pela existência de múltiplas máquinas Prolog que trabalham de forma seqüencial. Neste modelo, também é necessário controlar o compartilhamento de variáveis e a distribuição de tarefas entre as máquinas Prolog.

O paralelismo OU também pode ser explorado em CLP. Acredita-se que esse seja o tipo mais importante devido ao não determinismo inerente à vários problemas resolvidos com CLP, tais como escalonamento de horários e alocação de recursos.

Deste modo, esse trabalho propõe uma forma de exploração de paralelismo OU em CLP, que utiliza muitos dos princípios existentes em implementações paralelas de Prolog. Propõe-se um modelo multi-seqüencial de exploração de paralelismo OU em CLP, onde cada trabalhador possui uma máquina CLP sobre domínios finitos. Além disso, propõe-se uma política de escalonamento de tarefas distribuída. Desse modelo,

será feito um protótipo que facilite o reuso de trabalhos consolidados, assim como a agregação de futuras características, através do uso de objetos distribuídos, e no qual seja alcançado um bom nível de eficiência.

O restante do texto está organizado do seguinte modo: inicialmente, serão apresentados algumas considerações sobre a paralelização em Prolog e em CLP; na seção posterior, o modelo proposto será detalhado; e na seção seguinte, serão feitas observações sobre a implementação do protótipo pclp(FD) (*parallel constraint logic programming over Finite Domains*). Finalmente, serão apresentadas as conclusões e trabalhos futuros

2. Exploração de Paralelismo em Prolog e em CLP

2.1 Paralelismo em Prolog

As primeiras implementações de Prolog foram feitas na década de 70. Desde o princípio procurou-se formas de implementação que aumentem a sua eficiência tais como novas técnicas de compilação e de execução paralela. Em 1983, David H.D. Warren projetou a máquina abstrata de Warren (WAM) [AIT 90], representando um grande avanço. A WAM é formada por pilhas e por um conjunto de registradores utilizados na sua manipulação, assim como uma área de código. Atualmente ela é o padrão de fato para a compilação de programas Prolog, e serviu de base para uma série de extensões da programação em lógica como as linguagens concorrentes e implementações paralelas de Prolog.

Para explorar o paralelismo, há duas abordagens possíveis: (1) Paralelismo Implícito: o sistema executa em paralelo de forma transparente levando em conta o paralelismo inerente aos programas em lógica; (2) Paralelismo Explícito: o programa contém primitivas específicas para o controle do paralelismo.

O paralelismo explícito exige que o programador domine algoritmos e primitivas para controle do processamento paralelo, no entanto, é possível ter domínio total sobre a paralelização, o que pode levar a obtenção de melhor desempenho. Uma das vantagens do paralelismo implícito é permitir que o programador se concentre apenas no problema a ser solucionado, ignorando os aspectos relacionados a paralelização, o que pode levar à diminuição do tempo de desenvolvimento da aplicação. Os avanços nas

técnicas de análise abstrata estão facilitando a implementação dessa última abordagem por fornecer uma série de informações essenciais para a execução eficiente de programas Prolog.

Existem basicamente dois tipos de paralelismo implícito, que podem ser explorados de forma isolada ou combinada:

- Paralelismo OU: execução paralela das cláusulas que compõem um predicado, possuindo maior aplicação para programas não determinísticos. Os dois trabalhos mais importantes são Muse [ALI 90] e Aurora [LUS 90].
- Paralelismo E: execução paralela dos objetivos que compõem uma cláusula. Existem dois tipos: **E dependente** onde os objetivos que compartilham variáveis podem ser executados em paralelo, estabelecendo uma relação de produtor-consumidor; e **E independente** onde apenas os objetivos que não apresentam nenhuma variável livre comum podem ser executados em paralelo, evitando conflitos de ligação.

As formas de exploração de paralelismo OU em Prolog já encontram-se relativamente consolidadas. A principal desvantagem dos primeiros modelos era a geração de um grande número de processos com granulosidade pequena. Atualmente, os modelos multi-seqüenciais minimizam os custos de criação de processos, comunicação e verificações em tempo de execução, mostrando-se mais promissores [KER 92]. Nesses modelos a computação é realizada por processos trabalhadores que possuem uma máquina Prolog seqüencial eficiente (normalmente baseada na WAM). Alternativas não tentadas por um trabalhador representam trabalho OU potencial, isso é, podem ser entregues para outros trabalhadores resolverem. Assim, uma tarefa corresponde a alternativas não exploradas e um contexto. O escalonamento de uma tarefa é um fator importante na determinação da eficiência do sistema. Existem várias políticas de escalonamento, que podem tanto ser centralizadas (um único nodo é responsável pela distribuição das tarefas) como distribuídas (existe mais de um nodo responsável pelo escalonamento).

Existem diferentes técnicas no paralelismo OU multi-seqüencial para a exportação do contexto de uma tarefa. Essas técnicas disponibilizam as pilhas de execução da WAM para os trabalhadores e podem ser classificadas em três tipos:

- compartilhamento de pilhas: neste esquema uma parte das pilhas é compartilhada enquanto outra parte é privada (usada para armazenar as diversas ligações condicionais às variáveis das partes compartilhadas).
- cópia de pilhas: um processo que está recebendo trabalho copia as pilhas com o estado da computação anterior a alternativa não explorada.
- recomputação de pilhas: um trabalhador especializado indica a cada trabalhador um caminho pré-determinado da árvore de busca.

2.2 CLP

Para algumas classes de problemas, a árvore de resolução dos programas Prolog se torna muito grande, acarretando em grande uso de memória para manter as pilhas que controlam a execução. Entretanto, uma árvore é formada em grande parte por ramos que não levam ao resultado, mas que só são detectadas no momento que falham e causam retrocesso (*backtracking*). Ainda que existam mecanismos para diminuir o espaço de busca, como o predicado *cut*, muitos problemas não podem ser resolvidos em tempo viável. Normalmente, esses problemas são do tipo combinatoriais, como, por exemplo, alocações de recursos e composição de horários (*time tabling*).

Os termos restrições (*constraints*) e propagação de restrições (*constraint propagation*) têm sido amplamente empregados em inteligência artificial para designar diferentes tipos de técnicas, normalmente, utilizadas para tratar problemas combinatoriais. A programação em lógica com restrições (*Constraint Logic Programming* ou CLP) [JAF 94] [FRU 93] [VAR 97] é uma extensão da programação em lógica onde introduz-se a noção de restrições nos domínios das variáveis e o mecanismo de unificação utilizado em Prolog é substituído por uma operação mais geral chamada satisfação de restrição (*constraint satisfaction*). Normalmente a sintaxe dessas linguagens é similar a Prolog.

Os estudos teóricos iniciais mais importantes datam da década de 70 e início de 80. As primeiras implementações começaram a surgir no final da década de 80. Uma das primeiras linguagens em lógica com restrições foi CHIP [DIN 88]. Ela influenciou grande parte das atuais linguagens CLP implementando os mecanismos de resolução de restrições através da utilização de técnicas de propagação de restrições.

2.3 Paralelismo em CLP

Apesar de mais eficiente que a programação em lógica, é preciso buscar caminhos que melhorem o desempenho de CLP e permitam a resolução de um maior número de problemas reais, como por exemplo a execução paralela. Em CLP também é possível explorar as duas formas implícitas de paralelismo citadas anteriormente: Paralelismo OU e Paralelismo E.

Existem diversos trabalhos recentes sobre exploração do paralelismo OU na programação em lógica com restrições, em sua maioria, paralelizando linguagens derivadas da linguagem CHIP. [RÉT 96] busca formas de execução distribuída das linguagens concorrentes com restrições, enquanto [YAN 95] implementou uma linguagem com restrições paralela utilizando a linguagem concorrente KL1. Já [TON 95] preocupa-se com escalabilidade, utilizando computadores massivamente paralelos.

Trabalhos que estão mais relacionados com os objetivos desse trabalho são [PRS 93] e [MUD 94]. A linguagem ElipSys [PRS 93] é uma linguagem em lógica na qual é possível explorar paralelismo e restrições de forma independente, isto é, de forma isolada ou combinada. É possível explorar tanto paralelismo OU quanto E, mas fica a cargo do programador quando deve ser feita a paralelização. Já [MUD 94] explora apenas paralelismo OU, mas de forma automática. Além disso, o alvo do protótipo é uma rede heterogênea de computadores. Ambos utilizam um modelo OU multi-seqüencial, mas [PRS 93] usa a abordagem de cópia de pilhas, enquanto [MUD 94] usa recomputação de pilhas.

Apesar dos vários trabalhos existentes, ainda não há um consenso de quais técnicas são mais adequadas para exploração do paralelismo em CLP.

3. Modelo de Exploração de Paralelismo OU em CLP

O paradigma CLP é mais eficiente que a programação em lógica. No entanto, há interesse em obter implementações mais eficientes, tanto para melhorar o desempenho de aplicações existentes quanto para permitir o desenvolvimento de novas aplicações que exijam muito processamento. O projeto OPERA do II/UFRGS, dentro do qual esse trabalho está sendo desenvolvido, tem por objetivo estudar as diversas formas de

exploração de paralelismo na programação em lógica. Já foram desenvolvidos diversos trabalhos como por exemplo: paralelismo OU em uma máquina transputer [GEY 92], paralelismo E em uma rede de estações [YAM 94] [WER 94], simulação de escalonamento dinâmico de tarefas [COS 97], análise de granulosidade [BAR 95] [VAR 95] e interpretação abstrata [CAS 97]. Todos esses trabalhos buscam técnicas para aprimorar o desempenho da programação em lógica preferencialmente em ambientes distribuídos. Deste modo, o estudo da programação em lógica com restrições em paralelo está de acordo com os objetivos do projeto, uma vez que o uso de restrições pode ser considerado uma nova técnica de otimização da programação em lógica, ao mesmo tempo que será explorado o paralelismo. A idéia é aproveitar o conhecimento adquirido anteriormente com Prolog e utilizar na modelagem e implementação de um ambiente de execução paralelo para uma linguagem CLP. Os principais detalhes sobre o modelo serão descritos nas próximas seções.

3.1 Modelo Multi-seqüencial

Optou-se por modelar o paralelismo OU por acreditar-se ser o de maior aplicação para CLP uma vez que existem muitos programas combinatoriais que possuem não-determinismo inerente. No entanto, há a possibilidade de ampliar o modelo para incluir a exploração de paralelismo E. O modelo apresentado é multi-seqüencial pois existem vários trabalhadores cooperando para a obtenção do resultado final em menor tempo. Os trabalhadores possuem a capacidade de resolver programas em lógica com restrições. A máquina CLP, existente em cada um deles, deve ser uma extensão da WAM para que técnicas de paralelização Prolog possam ser utilizadas de modo mais direto.

Existem várias linguagens CLP implementadas. Como o objetivo deste trabalho não é a proposta de uma nova linguagem, a melhor alternativa é aproveitar soluções já desenvolvidas. Optou-se pela utilização do sistema clp(FD) [COD 96] como linguagem a ser paralelizada pelos seguintes motivos:

- código fonte está disponível e é bem documentado e estruturado, facilitando a sua alteração;

- trabalha sobre domínios finitos que permite a modelagem e solução de problemas de satisfação de restrições discretas;
- sistema *clp(FD)* estendeu o sistema *wamcc*, que é uma implementação eficiente da máquina abstrata de Warren (WAM). O fato de ser baseado na WAM facilita a utilização de uma abordagem multi-seqüencial na exploração do paralelismo OU.

O sistema clp(FD) é fortemente influenciado pelas linguagens CHIP e cc(FD). A noção de domínios finitos (FD) foi introduzida na programação em lógica pela linguagem CHIP, onde a resolução de restrições é feita por propagação e técnicas de consistência originadas de CSP (*Constraint Satisfaction Problems*). Um domínio em FD é um conjunto finito (não vazio) de números naturais, isto é, um intervalo. A linguagem cc(FD) também é baseada em domínios finitos, e foi proposta por Pascal van Hentenryck em 1991, apresentando grande transparência sobre os detalhes de implementação. Sua compilação é baseada em uma extensão da WAM, tendo uma única restrição *X in r* onde *X* é uma variável e *r* é um intervalo. Intuitivamente, *X in r* significa que "*X* deve sempre pertencer a *r*". Restrições complexas são definidas em termos dessa restrição.

Como já foi dito, clp(FD) é uma extensão da wamcc para FD, sendo que a arquitetura e a estrutura de dados básicas da WAM não foram alteradas. A wamcc [COD 95] traduz um programa WAM para a linguagem C, com o objetivo de criar uma implementação simplificada, eficiente e portável da WAM de modo a facilitar a extensão para outros trabalhos.

3.2 Cópia Incremental

Deseja-se criar um modelo que seja adequado para execução em ambiente distribuído, onde considera-se que existem trabalhadores em processadores distintos sem memória comum. Por isso, dos três tipos de técnicas para exportação do contexto de uma tarefa no paralelismo OU multi-seqüencial, apresentados anteriormente (seção 2.1), o compartilhamento de pilhas não é viável devido a inexistência de memória comum entre diferentes processadores. Então, a escolha deve ser feita entre as outras duas abordagens: recomputação e cópia.

As pilhas em CLP são maiores do que em Prolog [MUD 94], o que apresenta uma vantagem para a recomputação. Entretanto, as operações de resolução de restrições em CLP são relativamente caras, o que eleva o custo de recomputação.

Optou-se pela cópia de pilhas que evita o desperdício de computação útil e possui a implementação mais simples. Também considerou-se nessa decisão o fato de que na transmissão de mensagens em uma rede a latência é mais cara que a banda, isto é, diminuir o tamanho de uma mensagem traz ganho de desempenho menos significativo do que a diminuição no número total de mensagens. Considerando-se que nas técnicas de cópia e de recomputação o número de mensagens é aproximadamente o mesmo, o tempo total de latência também será aproximadamente o mesmo. Assim, o ganho da recomputação na diminuição da banda não é tão significativo.

Porém, deve-se considerar que, além do fato das pilhas serem maiores por manipularem restrições, em problemas reais a árvore de busca pode ser muito profunda e portanto as pilhas a serem copiadas se tornam ainda maiores. A utilização de um mecanismo de **cópia incremental**, como o proposto por esse modelo, constitui-se de uma importante otimização, diminuindo o custo da cópia de pilhas. A cópia incremental baseia-se no princípio de que trabalhadores que já realizaram alguma tarefa podem ter em suas pilhas várias informações comuns a um trabalhador que esteja exportando trabalho.

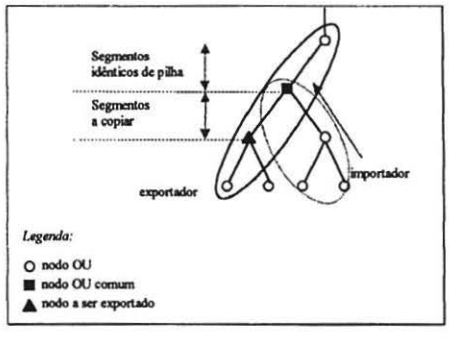

Figura 1. Exemplo de árvore de execução.

A Figura 1 apresenta um exemplo de árvore de execução. Supondo que um trabalhador importou trabalho e já terminou a execução, isto é, está sem trabalho. O conteúdo de suas pilhas de execução permanece intacta. Caso um outro trabalhador tenha tarefas

para exportar e eles possuam um nodo OU comum (houve exportação de tarefas anteriormente), o trabalhador exportador somente precisará enviar o contexto de execução no intervalo entre o nodo comum e o nodo a ser exportado. O importador deve descartar os nodos mais recentes (através de retrocesso) até o nodo comum e acrescentar o que o exportador lhe enviar.

3.3 Escalonamento Distribuído de Tarefas

O ambiente pclp(FD) proposto é formado por *n* nodos trabalhadores localizados em *n* processadores distintos. Cada nodo trabalhador possui uma máquina CLP e uma interface para um escalonador. Um nodo trabalhador poderá assumir, em um determinado instante, um dos seguintes estados:

- *idle*: está sem trabalho, aguardando que algum outro trabalhador peça ajuda;
- *busy*: está trabalhando, mas ainda não possui quantidade de trabalho (pontos de escolha não explorados) que justifique a exportação;
- *overloaded*: está trabalhando e está sobrecarregado, por isso pode exportar tarefas a outros trabalhadores.

O trabalhador informa o escalonador sobre suas transições de estado. Note que o estado *overloaded* deve ser determinado por experimentação na implementação. A Figura 2 ilustra a transição de estados dentro de um trabalhador.

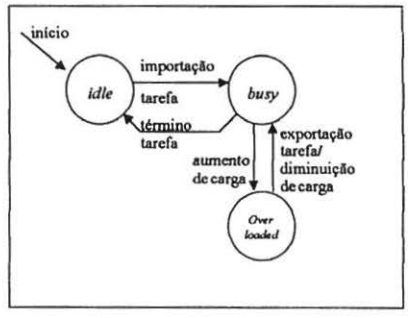

Figura 2. Diagrama de transição de estados.

A função do escalonador é distribuir tarefas entre os nodos trabalhadores. Uma tarefa a ser exportada corresponde ao ponto de escolha (*choice point*) mais antigo, isto é, o mais

próximo da raiz. Essa definição de tarefa baseia-se na heurística que quanto mais próximo da raiz maior a quantidade de trabalho ainda a ser explorada [KER 92]. Através da análise de granulosidade seria possível efetuar uma escolha mais precisa [BAR 95], mas tal análise não será considerada no momento.

A forma como o escalonamento é implementado não influi na concepção do trabalhador. A interface será a mesma para um escalonador centralizado ou para um distribuído. Com um escalonador centralizado utiliza-se um número menor de mensagens, mas diminui-se a escalabilidade e a tolerância a falhas. Com um distribuído tem-se o oposto. Acredita-se que o escalonamento distribuído apresenta a melhor relação entre custo e benefício. Por isso, nessa seção propõe-se uma política de escalonamento distribuído, conforme descrito a seguir.

Em um nodo trabalhador inicial, a máquina CLP inicia o processamento a partir da raiz da árvore de execução CLP enquanto o escalonador cadastra todos os demais trabalhadores como *idle* na sua **lista de nodos *idle***. A organização e manutenção desta lista serão discutidas no decorrer dessa seção.

Quando o seu nível de trabalho torná-lo *overloaded*, o trabalhador informará o escalonador. Inicialmente, todos os demais trabalhadores estão com as pilhas vazias, e portanto o escalonador poderia escolher qualquer um para exportar a tarefa. Assim, o escalonador escolhe o trabalhador que estiver no topo da sua lista de *idle*, e envia a tarefa.

No decorrer do processamento, sempre que um escalonador detectar que o nodo trabalhador estiver *overloaded*, ele enviará uma mensagem para cada um dos nodos da sua lista de *idle* com o objetivo de exportar uma tarefa (Figura 3a). Em princípio, todos os nodos *idle* iriam responder afirmativamente ao pedido (Figura 3b).

O escalonador exportador precisará escolher um dos nodos para enviar a tarefa. Um critério possível seria dar trabalho ao primeiro trabalhador que respondesse ao pedido de exportação. No entanto, em muitos casos não seria a escolha mais adequada. Como o sistema pode ser utilizado em ambientes distribuídos, principalmente em redes de computadores, não existe tempo fixo para envio e recebimento de mensagens. Deste modo, o exportador não deve esperar que todos os importadores respondam, pois caso

contrário pode haver muita demora na tomada de decisão, comprometendo o desempenho do sistema.

Assim, de posse da maioria (metade mais uma) das respostas ou após um *timeout* estabelecido pelo sistema, o exportador irá pesar os seguintes critérios:

1. custo de comunicação: é calculado a partir: (a) do tempo de resposta do trabalhador, porque considera-se que se ele recebeu e enviou a mensagem de exportação em pouco tempo, possivelmente receberá a mensagem com a tarefa em pouco tempo também; e (b) da quantidade de contexto a ser enviado, que pode ser detectada através do último nodo OU comum.
2. capacidade computacional do trabalhador: é formada por dois tipos de informação: (a) uma estática fornecida pelo sistema que indica a capacidade do processador e (b) uma dinâmica indicando a carga atual do processador onde o trabalhador se encontra (os trabalhadores devem aproveitar quando estiverem em estado *idle* para obterem tal informação).

Seguindo esses critérios, o escalonador exportador enviará a tarefa ao trabalhador que apresentar o menor custo de comunicação e maior capacidade computacional. Note que é enviado o contexto (pilhas) e uma lista com os nodos *idle*. O exportador também enviará uma mensagem para os demais dispensando-os de ajudá-lo (Figura 3c). Essa mensagem de resposta evita que um trabalhador fique indefinidamente ocioso a espera de um trabalho que jamais lhe será concedido.

De fato, o trabalhador não fica ocioso enquanto espera a resposta, pois nesse meio tempo ele cadastra mensagens de exportação. Caso receber uma resposta negativa, pode escolher entre as mensagens recebidas, outro exportador cujo ponto de escolha comum mais recente.

Outro ponto importante é a lista de *idle*: ela é ordenada pela capacidade computacional do processador onde o trabalhador *idle* se encontra. Isso é especialmente importante no momento inicial onde a primeira tarefa, que provavelmente possui alta granulosidade, será resolvida pelo trabalhador existente no processador mais potente. Além disso, a aplicação pode estabelecer um limite máximo de n pedidos de importação. Deste modo, somente os n primeiros da lista seriam notificados.

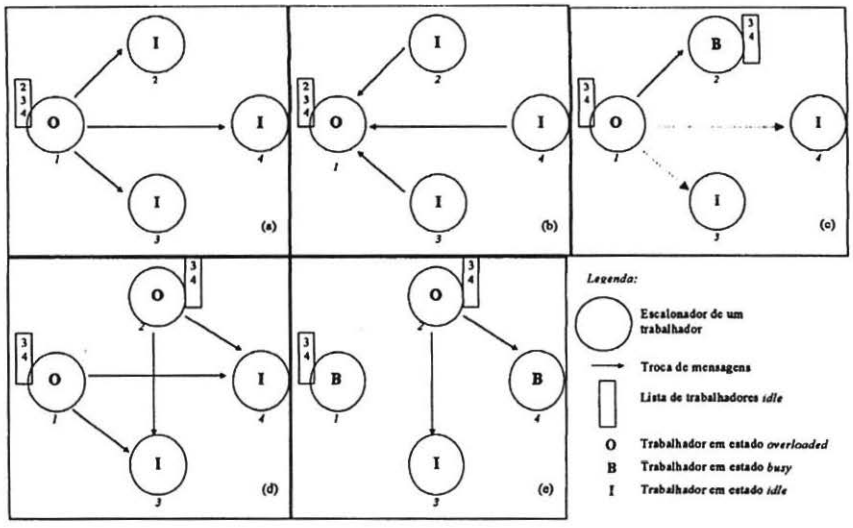

Figura 3. (a) pedido de ajuda; (b) resposta ao pedido de ajuda; (c) escolha do importador; (d) gerência de vários pedidos de ajuda; (e) *busy* ignora pedido de ajuda;

A partir desse momento, mais de um trabalhador pode estar sobrecarregado. Por isso, é preciso estabelecer critérios de escolha para os trabalhadores *idle*. Por exemplo, na Figura 3d, os trabalhadores 1 e 2 desejam exportar tarefas.

Sistemas distribuídos utilizam um número maior de mensagens para manter informações atualizadas sobre o sistema, mas tais mensagens precisam ser minimizadas para não comprometer o desempenho do sistema. Optou-se pela manutenção de uma lista de *idle* para evitar mensagens para todos os nodos quando da exportação. Assim, nodos exportadores enviam, além da tarefa, a sua lista de *idles*.

Quando um nodo fica *idle* ele informa a todos os demais trabalhadores que deseja receber trabalho. Os trabalhadores em estado *idle* ignoram a resposta, enquanto os em *busy* e em *overloaded* incluem o novo nodo *idle* na sua lista. No entanto, quando um trabalhador se tornar *busy*, o seu escalonador não informará ninguém. Por exemplo, a Figura 3e ilustra a possibilidade de um trabalhador *busy* receber um pedido de ajuda.

Optou-se por não propagar a informação de mudança de *idle* para *busy* porque (a) deseja-se minimizar a troca de mensagens e (b) há a possibilidade do trabalhador voltar para o estado *idle* antes que outro nodo trabalhador solicite a sua ajuda.

4. Implementação do Protótipo

pclp(FD) (parallel constraint logic programming over Finite Domains) é o sistema a ser implementado. Como já foi dito, o projeto OPERA se preocupa com ambientes distribuídos. A opção natural de implementação em um ambiente distribuído é a utilização de bibliotecas para troca de mensagens. No entanto, notou-se que esse tipo de programação implica a necessidade de preocupação com detalhes de mais baixo nível como sincronização, empacotamento de dados, etc. Protótipos com alto nível de complexidade mostram-se mais suscetíveis a erros e possuem manutenção mais difícil.

A necessidade de criar um modelo flexível e um protótipo que facilite a manutenção e permita a interação com outros trabalhos motivaram a concepção desse modelo. Acredita-se que a orientação a objetos seja a ferramenta ideal para atingir esses objetivos. Além disso, escolheu-se como linguagem para implementação Java, a fim de obter portabilidade e a possibilidade de extensões futuras para aplicações na internet.

Assim, o nodo trabalhador é um objeto que possui a capacidade de comunicar-se com outros trabalhadores através do objeto escalonador. Não entraremos em detalhes a cerca da implementação do objeto escalonador por limitações de espaço.

O objeto trabalhador encapsula o resolvedor de restrições clp(FD), e o seu construtor precisa inicializar a máquina clp(FD), que consiste basicamente na chamada de uma função do sistema legado que carrega o programa Prolog para pilha de código e inicializa as pilhas e os registradores. Definiu-se que existe um único objeto trabalhador em cada processador, e, por isso, também considera-se que não existe uma área de memória comum entre trabalhadores. O trabalhador possui ainda métodos para exportar e importar tarefas, implicando o acesso ao conteúdo das pilhas da máquina abstrata.

Em orientação a objetos pode-se ter a noção de objetos observados [PRE 96], isto é, quando houver alterações no objeto observado, os observadores são informados. Na implementação do escalonador distribuído aproveita-se esse mecanismo para informar o escalonador sobre mudanças de carga sem a presença de um objeto espião específico.

5. Conclusão

Esse trabalho possui como principal contribuição um modelo de exploração de paralelismo OU flexível. Os mecanismos de cópia incremental e escalonamento distribuído propostos são adequados tanto para CLP sobre domínios finitos quanto para outros tipos de linguagens CLP baseadas na WAM, e até mesmo para Prolog.

O protótipo a ser concluído usufruirá dos benefícios da orientação a objetos, ou seja, facilidade de manutenção e de extensão. Deste modo, a interface entre trabalhador e exportador é traduzida na implementação de forma bastante natural.

Quanto ao escalonamento, acredita-se que a solução proposta seja eficiente para um número não muito grande de processadores. Para explorar computadores massivamente paralelos, uma solução mista, isto é, hierárquica, seria melhor porque que não apresenta o gargalo do escalonamento centralizado nem o grande número de mensagens do distribuído. Outra possibilidade é a definição de uma política de escalonamento tolerante a falhas.

6. Referências Bibliográficas

[AIT 90] AIT-KACI, Hassan. **The WAM: A (Real) Tutorial**. Paris: Digital Equipment Corporation, Research Laboratory, Jan 1990.

[ALI 90] ALI, Khayri A. KARLSSON, Roland. The Muse Or-Parallel Prolog Model and its Performance". NACLP 1990. **Proceedings...** pp. 757-776.

[BAR 95] BARBOSA, Jorge Luís Victória; **GRANLOG: Um Modelo para Análise Automática de Granulosidade na Programação em Lógica.** Porto Alegre: CPGCC-UFRGS, 200p,1995. (dissertação de mestrado)

[CAS 97] CASTRO, Luís Fernando Pias de. Um Modelo de Analisador Estático Baseado na Interpretação Abstrata Direcionado à Paralelização de Programas em Lógica Porto Alegre: CPGCC-UFRGS (dissertação de mestrado em andamento)

[COD 95] CODOGNET, Philippe; DIAZ, Daniel. wamcc: Compiling Prolog to C. In: 12th International Conference on Logic Programming, Tokyo, Japan. **Proceedings...** MIT Press, 1995, p.317-331.

[COD 96] CODOGNET, Philippe; DIAZ, Daniel. Compiling Constraints in clp(FD). **Journal of Logic Programming**, New York, v.27, n.1, 45p., 1996.

[COS 97] COSTA, Cristiano André da. **Uma Proposta de Escalonamento Distribuído para Exploração do Paralelismo na Programação em Lógica.** Porto Alegre: CPGCC-UFRGS (dissertação de mestrado em andamento)

[DIN 88] DINCBAS, M.; HENTENRYCK, P. Van; SIMONIS, H.; AGGOUN, A.; GRAF, T., BERTHIER, F. The Constraint Logic Programming Language

CHIP. In: International Conference on Fifth Generation Computer Systems 1988. **Proceedings...** ICOT, 1988, p.693-702..

[FRU 93] FRÜHWITH, Thom et all. **Constraint Logic Programming: An Informal Introduction**. Müchen, Germany: ECRC, 25p, February 1993. (relatório técnico ECRC-93-5 - disponível em ftp://ftp.ecrs.de:/pub/ECRC_tech_reports)

[GEY 92] GEYER, Cláudio Fernando Resin; Briat, Jacques. et al. OPERA: Or-Parallel Prolog System on Supernode. In: Implementations of Distributed Prolog, John Wiley & Sons Ltda, 1992.

[JAF 94] JAFFAR, Joxan; MAHER, Michael J. Constraint Logic Programming: A Survey. **The Journal of Logic Programming**, New York, v.19/20, p.503-581. May/July 1994.

[KER 92] KERGOMMEAUX, Jacques Chassin; CODOGNET, Philippe. **Parallel Logic Systems**. France: Institut Grenoble/IMAG. Maio 1992 (relatório de pesquisa)

[LUS 90] LUSK, E. WARREN; H., HARIDI, S. The Aurora Or-Parallel Prolog System. University of Bristol, 1990. (relatório técnico TR-90-07)

[MUD 94] MUDAMBI, Shyam; SCHIMPF, Joachim. Parallel CLP on Heterogeneous Networks. In: Eleventh International Conference on Logic Programming, June 13-18, 1994, Santa Marherita Ligure, Italy. **Proceedings...** MIT Press, p.124-141, 1994

[PRE 96] PREE, Wolfgang; et al. Application of Design Patterns in Commercial Domains. In: 10th European Conference on Object-Oriented Programming. Austria. **Proceedings...** 1996.

[PRS 93] PRESTWICH, Steven. **ElypSys Programming Tutorial.** ECRC, 30p., April 1993 (relatório técnico)

[RÉT 96] RÉTY, Jean-Huges. **A Distributed Concurrent Constraint Programming Language**. S.L: s.n. March 1996.

[TON 95] TONG, Bo-Ming; LEUNG, Ho-Fung. Concurrent Constraint Logic Programming on Massively Parallel SIMD Computers. In: 1993 International Symposium in Logic Programming, Vancouver, Canada, **Proceedings...** MIT Press, 1993

[VAR 95] VARGAS, Patrícia Kayser. **Implementação de um Analisador de Granulosidade para Prolog**. Porto Alegre: CIC/UFRGS, 1995. 71p. (projeto de diplomação)

[VAR 97] VARGAS, Patrícia Kayser. **Um Estudo sobre as Linguagens em Lógica com Restrições**. CPGCC/UFRGS: Porto Alegre, 1997. 40p. (trabalho individual)

[WER 94] WERNER, Otília. **Uma Máquina Abstrata Estendida para o Paralelismo E na Programação em Lógica.** CPGCC/UFRGS: Porto Alegre, 1994. 170p. (dissertação de mestrado)

[YAM 94] YAMIN, Adenauer Corrêa **Um Ambiente para Exploração de Paralelismo na Programação em Lógica.** Porto Alegre: CPGCC-UFRGS, 1994. 212p. (dissertação de mestrado)

[YAN 95] YANG, Rong Implementing a Finite Domain Constraint Solving System. ICLP95 – Workshop on Parallel Logic Programming, Kanagawa, Japan. **Proceedings...** June 1995