Prolog Paralelo em Rede de Computadores

Adenauer C. Yamin[1]
Otilia Werner[2]
Cláudio F. R. Geyer[3]

Universidade Federal do Rio Grande do Sul
Instituto de Informática
Pós-graduação em Ciência da Computação
Caixa Postal 15064
CEP 91501-970 - Porto Alegre, RS, Brasil
Tel.: (051)336-8399 Fax: (051)336-5576

Resumo

Este artigo descreve o modelo de implementação em rede local de computadores de uma proposta de exploração do paralelismo da linguagem Prolog. O trabalho é fundamentado no modelo E/OU do projeto Opera, que integra o paralelismo OU multi-seqüencial e o paralelismo E-restrito (RAP). São tratados tópicos pertinentes à arquitetura de processos, à política de escalonamento e à gerência de memória.

Abstract

This paper describes the implementation model in a local area network of a parallelism exploitation proposal in Prolog language. The work is based in the AND/OR model of the Opera project, wich integrates the multi-sequential OR parallelism and the Restricted AND parallelism (RAP). We include topics belonging to process architecture, scheduling policy and memory management.

[1]Engenheiro Eletricista (UCPel/RS,1981), Especialista em Informática na Educação (UFPel/RS); Mestrando do CPGCC/UFRGS; Processamento Paralelo, Arquitetura de Computadores; E-mail: adenauer@inf.ufrgs.br

[2]Bacharel em Informática (PUC/RS,1991); Mestrando do CPGCC/UFRGS; Linguagens de Programação, Processamento Paralelo; E-mail: werner@inf.ufrgs.br

[3]Professor UFRGS/CPGCC; Dr. em Informática (Université Joseph Fourier, Grenoble, França); Processamento Paralelo, Programação Distribuída, Sistemas Operacionais; E-mail: geyer@inf.ufrgs.br

1 Introdução

Uma estratégia que atualmente se mostra promissora para aumentar a velocidade de execução de programas Prolog é a combinação de uma avançada técnica de compilação com a execução paralela.

O modelo Opera E/OU [GEY92b] tem uma proposta de exploração integrada do paralelismo E e OU da Programação em Lógica empregando estes dois recursos. A premissa é que o modelo que explorar os dois tipos de paralelismo terá maior desempenho do que aquele que explorar somente um (apesar dos custos de gerenciamento inerentes à integração).

Um dos trabalhos da atual fase do Projeto Opera consiste na implementação em rede local de computadores de um ambiente de execução para exploração do paralelismo E na Programação em Lógica. O ambiente de execução, em linhas gerais, consiste de uma máquina abstrata para processamento de programas Prolog, de uma arquitetura de processos, de um método para escalonamento de serviços e de um protocolo para gerência da memória distribuída.

Dois são os principais fatores que estimularam o uso de uma rede local na implementação do protótipo de paralelismo E do modelo Opera E/OU. O primeiro é da ordem de aproveitamento de recursos. Este fator leva em consideração a existência, no ambiente acadêmico, de um bom número de máquinas parcialmente ociosas [DOU91]. Tal ociosidade se deve a trabalhos de elevada interação com o usuário (leitura e escrita de textos), característicos neste ambiente. Existem casos pesquisados que revelam situações mais extremas, onde um total correspondente a um terço do montante de equipamentos em determinados períodos está sem nenhum uso mesmo durante horas úteis do dia [THE89]. O objetivo é viabilizar uma alternativa para o aproveitamento destas estações total ou parcialmente livres como "servidores de processamento".

O segundo fator está ligado à disponibilidade, e leva em conta o fato de as redes de computadores serem hoje a "arquitetura paralela" mais difundida na comunidade científica e isto, dentre outros aspectos, facilita o intercâmbio de experiências entre instituições.

Este artigo tem a seguinte organização: na seção 2 são introduzidos conceitos pertinentes ao paralelismo na programação em lógica. A seção 3 descreve como é explorado o paralelismo no modelo Opera E/OU. A seção 4 trata da proposta de implementação do paralelismo E do modelo Opera E/OU em redes de computadores. Na seção 5 são comentados alguns trabalhos relacionados. Finalizando, a seção 6 apresenta as conclusões.

2 Exploração do Paralelismo na Programação em Lógica

A Programação em Lógica apresenta algumas características que a tornam adequada ao paralelismo. Tais características envolvem a possibilidade de execução das cláusulas e objetivos de forma não-seqüencial, o não-determinismo na escolha de caminhos que levem à solução e a bidirecionalidade da variável lógica. Aliada a todos estes aspectos, está a grande motivação para exploração do paralelismo: a ineficiência da execução de programas construídos segundo o paradigma lógico.

2.1 Fontes de Paralelismo

Segundo [HER86], há dois tipos básicos de paralelismo que podem ser explorados em programas lógicos:

1. Paralelismo OU: um processo pode resolver toda cláusula cuja cabeça unifique com um dado objetivo (neste caso, diz-se que esta cláusula está ativa). Deste modo, se várias cláusulas unificarem com um objetivo, vários processos serão executados em paralelo (cada um resolvendo uma cláusula).

2. Paralelismo E: um processo pode resolver cada objetivo do corpo de uma cláusula ativa. Assim, de forma ideal, será lançado um processo para a resolução de cada objetivo em paralelo.

Além destes, há outros tipos de paralelismo, que são considerados de mais baixo nível:

1. Paralelismo *Stream*: vários processos podem avaliar estruturas de dados complexas incrementalmente, em paralelo com o processo que as está produzindo. Assim, cada vez que o processo A produz um valor para uma variável compartilhada pelo processo B, o primeiro envia este valor ao segundo e continua a computação, da mesma maneira que o processo B segue processando depois de receber o valor.

2. Paralelismo *Search*: o programa pode ser dividido em conjuntos disjuntos de cláusulas de modo que vários processos possam buscar, em paralelo, cláusulas cujas cabeças unifiquem com um dado objetivo, cada processo trabalhando sobre um conjunto diferente.

3. Paralelismo de Unificação: quando ocorre a unificação de um objetivo com a cabeça de uma cláusula, vários pares de termos (argumentos) correspondentes podem ser unificados em paralelo.

2.2 Detecção e Controle do Paralelismo

Os programas lógicos oferecem muitas fontes de paralelismo, mas a detecção do potencial para paralelismo em um dado programa é um problema que pode ser abordado de maneiras diferentes. Há, no mínimo, duas maneiras de detectar este potencial: pelo menos teoricamente, o paralelismo implícito pode ser descoberto e gerenciado automaticamente pelo sistema de compilação e execução. Este método apresenta vantagens, tais como liberar o programador das tarefas de sincronização, tópicos de concorrência e comunicação, e possibilitar a melhoria de desempenho de maneira transparente ao usuário.

Por outro lado, a tarefa de descobrir o paralelismo existente no programa pode ser deixada a cargo do programador, estendendo-se a linguagem de controle de forma que esta inclua construções ou anotações que invoquem e manipulem explicitamente a execução paralela.

Outra característica que deve ser analisada é a escolha dos tipos de paralelismo explorados em um dado sistema. De forma ideal, todas as possíveis fontes de paralelismo devem ser exploradas, mas o gerenciamento e controle desta abordagem a torna não-trivial e o *overhead* introduzido por ela pode invalidar qualquer ganho de desempenho obtido através da execução paralela.

3 O Modelo Opera E/OU

O projeto Opera integra os paralelismos E e OU da programação em lógica, explorando o paralelismo implícito da linguagem. A linguagem Prolog, portanto, mantém-se inalterada, o que possibilita a execução de programas já existentes, sem necessidade de modificações, e também não introduz complexidade no desenvolvimento de programas novos.

O paralelismo OU explorado é o multi-seqüencial, segundo o qual ocorre a cópia da resolvante completa quando um processador requisita trabalho. O paralelismo OU é especialmente útil em programas fortemente não-determinísticos ([GEY91], [GEY92a]).

A motivação para exploração do paralelismo E está no fato de que a quase totalidade dos programas lógicos apresentam-no, enquanto que o OU pode não estar presente em alguns programas. O paralelismo E, ao contrário do OU, pode ser bastante vantajoso para programas fortemente determinísticos.

O paralelismo E explorado é o RAP (*Restricted AND Parallelism*) ([DEG84], [GEY92b]), que combina uma análise de dependência de dados e geração de CGE's (*Conditional Graph Expressions*) em tempo de compilação e uma avaliação de tais expressões em tempo de execução. Para a geração destas expressões é usado um conjunto de primitivas que verificam se os sub-objetivos podem ser executados em paralelo ou se precisam ser executados de forma seqüencial. Os testes que as primitivas realizam em tempo de execução são muito simples, gerando um *overhead* mínimo.

Shen e Hermenegildo, em [SHE91], estudaram a natureza dos paralelismos OU e E-independente (ou RAP) em programas Prolog e concluíram que tais programas tendem a exibir uma forma ou outra de paralelismo, não tendo encontrado muitos programas que apresentassem, ao mesmo tempo, quantidades significativas de paralelismo OU e E. Esses aspectos tornam óbvia a afirmação de que um sistema que explora ambas as formas de paralelismo melhora o desempenho de uma gama maior de programas do que aquele sistema que explora uma única forma.

O modelo E/OU do Projeto Opera, quando detectada a presença de ambos paralelismos, prevê dois casos distintos. O primeiro caso ocorre quando se encontra paralelismo E abaixo do OU: o paralelismo OU é disparado no modo multi-seqüencial entre *clusters* (subredes) e, dentro de cada *cluster*, a gerência do paralelismo acontece como se puramente E (vide figura 1). O número máximo de ramos OU explorados simultaneamente será igual ao número de *clusters* alocados na rede para a arquitetura do Opera. O paralelismo OU volta a ser disparado a medida em que são esgotados os ramos OU, ou seja, a medida em que os *clusters* forem concluindo o processamento.

O segundo caso é aquele que apresenta paralelismo OU abaixo do paralelismo E, tornando a situação mais complexa. Naturalmente esta possibilidade pode ser desativada no sistema e os nós OU, criados dentro dos literais E paralelos, podem ser explorados através de backtracking. Porém, resultados de simulação ([HER91]) indicam que, assim procedendo, uma significativa quantidade de paralelismo OU pode ser perdida. Porém, considerando que a implementação desta alternativa é bastante sofisticada, inicialmente ela não será contemplada no protótipo.

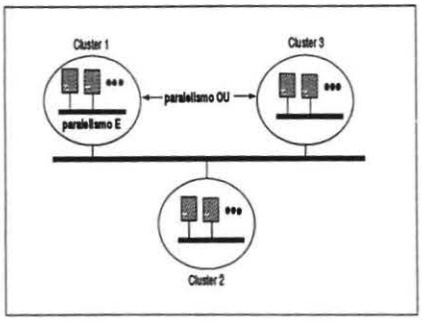

Figura 1: Topologia Opera E/OU

4 Paralelismo E em Rede de Computadores

4.1 O Ambiente de Execução

As estações de trabalho da rede utilizada empregam o sistema operacional Unix. O padrão de rede empregado é o Ethernet com uma taxa nominal de transferência de 10 M bits por segundo. A configuração é formada por estações SUN rodando SunOs versão 4.1.1. O número de estações a serem empregadas é determinado na inicialização do sistema. Este *hardware* caracteriza uma arquitetura multiprocessadora sem memória comum, onde a comunicação entre processadores é feita por troca de mensagens (MIMD - *Multiple Instructions Multiple Data* ; tipo NORMA - *No Remote Memory Access*).

A comunicação entre processos alojados em diferentes processadores é feita utilizando primitivas construídas com sockets BSD (*Berkeley Software Distribution*), enquanto que processos em um mesmo processador se comunicam empregando memória compartilhada e sinais (interrupção de *software*).

A opção por ferramentas de mais baixo nível (*sockets*) na programação das primitivas de comunicação remota (entre processadores) é decorrência da preocupação com dois aspectos; o primeiro é o **desempenho** global do protótipo, e o segundo é manter um bom nível de **portabilidade** para outras máquinas paralelas.

Na comunicação local entre processos (mesmo processador), o uso de sinais e de memória compartilhada tornam mais complexa a interface entre módulos, porém oferecem uma possibilidade não onerosa de comunicação assíncrona, a qual é indispensável para minimizar o *overhead* introduzido pela gerência do paralelismo.

Um dos custos desta opção por maior controle e eficiência sobre o *software* foi a inviabilização do uso de ferramentas para desenvolvimento de sistemas distribuídos/paralelos como PVM (*Parallel Virtual Machine*, Universidade do Tennesee), P4 (*Portable Programs for Parallel Processor*, Laboratório Nacional de Argonne) ou até mesmo linguagens como SR (*Synchronizing Resources*, Universidade do Arizona), que simplificam de sobremaneira o projeto e a manutenção do *software*.

4.2 A Arquitetura de Processos

A arquitetura empregada no modelo Opera E/OU na exploração do paralelismo E tem dois tipos básicos de agentes: *Mestre* e *Trabalhador* (vide figura 2). O *mestre* é o processo responsável pela gerência da arquitetura e deve ser disparado no segmento de rede onde ocorrerá o processamento. Uma vez instalado ele controla a execução disparando processos e escalonando servicos. Cada *trabalhador*, por sua vez, é formado por três processos, a saber:

- **Solver**: processo responsável pela execução de programas Prolog; é ele que contém a máquina abstrata Prolog;
- **Spy**: processo reponsável por manter o agente *mestre* informado sobre o estado dos *trabalhadores* e;
- **Communicator**: processo que efetua todo o tratamento das comunicações assíncronas recebidas pelo *trabalhador*.

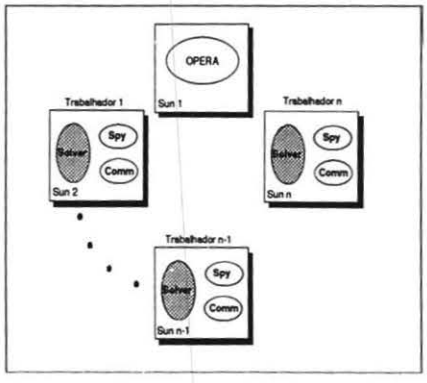

Figura 2: A Arquitetura de Processos

Inicialmente o processo *mestre* verifica se a arquitetura já está operacional. Caso não esteja, os *trabalhadores* são instalados nos processadores. A arquitetura de processos uma vez operacional, ela assim permanece dando suporte às várias aplicações até que o usuário delibere seu encerramento.

Os processadores onde serão instalados os *trabalhadores* estão definidos em uma tabela que é de responsabilidade do usuário. Nesta tabela existem informações sobre o poder computacional dos equipamentos, as quais serão empregadas durante o escalonamento de serviço.

Cada programa Prolog utiliza um determinado número de processadores (normalmente um subconjunto daqueles que formam a arquitetura). Estando a arquitetura operacional, o processo *mestre* lê os parâmetros sobre os quais será executada a aplicação e os distribui através de uma mensagem a todos os processadores envolvidos.

Esta técnica de manter a arquitetura montada e em repouso, somente encerrando ou instalando *trabalhadores* para casos particulares, é extremamente útil para sistemas paralelos que tenham custo elevado para disparo remoto de processos. Particularmente em nosso caso, uma rede de estações Unix, o uso de mensagens introduziu redução significativa também no tempo necessário para encerrar processos nos *trabalhadores*. O ganho de tempo com o uso de mensagens decorre de não ser necessário utilizar comandos Unix para login remoto (*rlogin*, *rsh*) cujo desempenho é bastante inferior.

É importante observar que este uso de mensagens assíncronas exige um servidor de comunicações sempre ativo em cada processador (processo *communicator*).

4.3 A Dinâmica do Sistema

Cada programa Prolog, uma vez concluído o trabalho de compilação, é considerado pronto para ser entregue ao processo *mestre*, que gerenciará seu processamento a partir de parâmetros do usuário. O processo *mestre* desdobra sua atuação em cinco fases bem distintas.

Fases do Processo *mestre*:

- **instalação**: caso a arquitetura de processos não esteja montada, ou precise ser modificada, é nesta fase que os *trabalhadores* serão instalados em algumas estações e encerrados em outras;
- **distribuição**: nesta fase são distribuídos os parâmetros necessários para execução nos *trabalhadores* (processos *solver*) envolvidos com a aplicação;
- **escalonamento**: nesta fase o processo *mestre* se transforma em *Scheduler* e permanece distribuindo trabalho até que a execução esteja encerrada em todos os *trabalhadores* envolvidos na aplicação;
- **encerramento**: esta fase consiste em exibir o resultado final de um programa Prolog e enviar sinal de sincronismo aos *trabalhadores* envolvidos preparando o sistema para outra execução;
- **desinstalação**: nesta fase são encerrados em todos os processadores da arquitetura os processos que compõem o *trabalhador*

Estados do processo *Solver*

Na exploração do paralelismo E, a avaliação da carga de trabalho em determinada máquina é feita dinamicamente. É utilizada a heurística de associar carga ao número de literais empilhados [GEY92b]. Dependendo da sua carga, os *trabalhadores* são enquadrados em um de três estados. O limite entre os estados são específicos de cada aplicação:

- **idle**: neste estado, não há trabalho para ser executado. O *trabalhador* está aguardando uma autorização para busca de uma tarefa em outro *trabalhador* (importação).
- **quiet**: neste estado, o *trabalhador* está ativo, mas não dispõe de trabalho em volume suficiente que justifique a cessão de parte deste para outro *trabalhador* (exportação).

- **overloaded**: neste estado, o *trabalhador* está ativo e existe volume de trabalho que viabiliza exportação.

Sincronismo Entre Escalonador e Trabalhadores no Paralelismo E

Na proposta de um escalonador para uma arquitetura paralela sem memória comum ([BRI90]), destaca-se a escolha de indicadores para avaliação de carga nos processadores. A correta tomada de decisão por parte de um escalonador depende deste ter informações as mais precisas possíveis sobre o estado global da carga de trabalho na arquitetura. Nos tópicos a seguir está presente o compromisso entre uma coleta de informações suficientemente precisa e o seu custo computacional em funcão desta exatidão.

- Quando o *trabalhador* estiver no estado *overloaded*, o processo *Spy* periodicamente (o período pode ser determinado pelo usuário para cada aplicação) envia a situação da sua carga ao *Scheduler* (processo *mestre*).
- Quando o *trabalhador* entrar no estado *quiet* ou *idle*, o processo *Spy* informa a transição de estado e interrompe o envio periódico de informações sobre a carga ao *Scheduler*. O propósito é evitar o congestionamento da rede minimizando o tráfego de comunicações gerenciais, neste caso em detrimento de manter no escalonador um estado mais preciso do sistema. O uso de heurísticas e a perda de precisão do estado global no escalonador introduz evidentes prejuízos para uma política de *trace* centralizada.
- No momento em que o escalonador tiver nas suas *Tabelas de Estado do Sistema* um *trabalhador* assinalado como *idle* e outro como *overloaded*, são disparadas as respectivas autorizações de importação/exportação de trabalho.
- Enquanto durar o processo exportação/importação, os *trabalhadores* tem seus estados bloqueados no escalonador para evitar múltiplas autorizações.
- Se, no momento da efetivação da operação de exportação/importação, o atual estado do *trabalhador* exportador não for mais *overloaded*, será enviado ao *trabalhador* importador um sinal de NOWORK. Esta é a maneira de contornar situações decorrentes da avaliação heurística e periódica da carga de trabalho.
- Ao término da operação de exportação/importação (mesmo com final NOWORK), os *trabalhadores* envolvidos enviam seus estados ao escalonador para minimizar inconsistências nas *Tabelas de Estado do Sistema*, caracterizando os trabalhadores como livres para outras transações.

4.4 A Gerência de Memória e o Escalonamento de Tarefas no Paralelismo E

No paralelismo E, os literais que serão executados se comportam como procedimentos, não existindo obrigatoriamente uma seqüência ou relação entre os literais que são buscados pelos processos *Solver*, isto é, na execução paralela podemos ter diferentes cláusulas sendo processadas nos diversos *trabalhadores*. Por este motivo, o literal que um *trabalhador* vai executar não depende somente da semântica procedimental da linguagem, mas também

de outros parâmetros pertinentes à implementação do modelo, os quais serão introduzidos adiante.

A seguir avaliaremos a relação entre escalonamento e gerenciamento de memória, tendo em vista os seguintes objetivos: minimizar o tempo inativo dos processadores, otimizar o uso da memória, evitar ao máximo o uso de rotinas para *garbage collection* e garantir uma distribuição de carga proporcional à capacidade de resposta de cada *trabalhador*.

Resultados da Execução de um Literal

Quando ocorrer uma exportação de serviço, o *trabalhador* que cede o literal para execução remota fica na condição de *trabalhador-pai* daquele que importa que, a partir de então, é visto como *trabalhador-filho*. A seguir, de modo suscinto, temos os procedimentos possíveis quando da execução de um literal.

- **Falha**: neste caso, o *trabalhador* envia ao pai uma mensagem de falha e, aos *trabalhadores-filhos*, uma mensagem que encerra nestes a execução corrente (mensagem de *Kill*). Os *trabalhadores* envolvidos procedem a liberação de área nos processos *solver* (pilhas das máquinas abstratas Prolog).
- **Sucesso**: o *trabalhador* envia uma mensagem de sucesso ao *trabalhador-pai* e fica aguardando deste:
  - Uma mensagem solicitando os resultados obtidos, ou
  - Uma mensagem de *kill*, encerrando a execução do literal pendente e autorizando liberação de área nas pilhas de dados utilizadas pelo seu processo *solver*, ou
  - Uma mensagem de *Redo* fazendo com que seja explorada outra cláusula-alternativa para o literal já processado.

Estados do Trabalhador Filho

Para minimizar o tempo inativo, enquanto aguarda determinações (mensagens) do *trabalhador-pai*, após ter obtido sucesso no processamento de um literal, o *trabalhador* pode buscar outro literal para execução. Disto podem decorrer as seguintes situações:

- O processamento do novo literal armazenado nas pilhas da máquina abstrata Prolog é terminado antes do literal que está aguardando mensagem de *Kill* ou *Redo*. Esta é a melhor situação e o mecanismo normal de retrocesso (*backtracking*) consegue recuperar toda a memória alocada.
- O processamento do novo literal não é esgotado antes do anteriormente empilhado (também fica aguardando mensagem). Desta situação decorrem duas possibilidades:
  - É recebida mensagem de *Kill* para o primeiro literal pendente: neste caso o processamento irá continuar com uma faixa sem uso nas pilhas da máquina abstrata. Esta faixa será liberada quando ocorrer retrocesso no literal empilhado por último.
  - É recebida mensagem de *Redo* para o primeiro literal pendente: neste caso, poderá faltar área de trabalho (contígua) para o *Redo*, o que é uma situação de gerenciamento de memória de difícil administração.

Algumas Alternativas para Otimizar o Uso da Memória

As alternativas a seguir tem o objetivo de evitar os problemas de alocação de memória citados:

- Buscar novos literais somente do *trabalhador-pai* (isto garante que o último literal importado terá seu processamento esgotado primeiro);
- Não havendo mais possibilidade de importar literais no *trabalhador-pai*, verificar se o literal pendente tem nodo OU (possibilidade de *Redo*). Caso não possua, pode ser buscado literal em qualquer *trabalhador* (com este procedimento, na pior hipótese, teremos parte do processamento com uma faixa de memória que não pode ser liberada).
- Não existindo mais literais disponíveis no *trabalhador-pai* e sendo o literal empilhado primeiro passível de *Redo*, a alternativa é criar um novo conjunto de pilhas, nas quais poderá ser processado qualquer outro literal disponível para execução paralela no sistema, sem risco de sobreposição à área contígua.

  Na atual fase da implementação, se utiliza a primeira alternativa.

4.5 Outras Características da Arquitetura

Política do Escalonador

O escalonador é centralizado para cada subrede (*cluster*) onde está sendo explorado o paralelismo E. Na distribuição de serviços são adotados os seguintes critérios:

- Na importação são priorizados os *trabalhadores* mais poderosos;
- Na exportação são priorizados os processadores com maior carga de trabalho.

Aspectos Operacionais

A execução começa em um processador escolhido como *trabalhador* principal, e é a partir da exportação de trabalho deste que acontece a exploração do paralelismo E. Também este *trabalhador* devolverá a resposta final ao processo *mestre*.

A arquitetura é facilmente escalonável, sendo possível acrescentar ou retirar elementos processadores a partir de definições do usuário, não necessitando recompilações do *software*.

O fluxo de trabalho é controlado por demanda, ficando a iniciativa de busca de tarefas a cargo dos *trabalhadores* inativos.

Estruturas de Entrada e Saída

O objetivo das estruturas de entrada é facilitar a definição de parâmetros para execução das diferentes aplicações Prolog. A associação de parâmetros a cada execução tem reflexos diretos no trabalho do escalonador. O tema escalonamento na exploração do paralelismo na programação em lógica é um problema em aberto ([GUP89], [GUP90]) e objeto de diversas pesquisas.

As estruturas de saída, por sua vez, contemplam o registro do processamento efetuado, o que é indispensável para a avaliação do comportamento dinâmico do modelo face às diferentes aplicações.

O projeto Opera dispõe das seguintes estruturas de entrada:

- *System Resource Table* (SRT): o objetivo desta tabela é definir quais e como são as máquinas da rede que formam a arquitetura do sistema. Ela define um *Processor ID* para cada máquina e registra seu poder computacional, que é uma informação básica do algoritmo de escalonamento empregado.
- *Application Characteristic Table* (ACT): esta tabela, por sua vez, contém informações específicas de cada aplicação, dentre as quais destacamos: quais máquinas definidas na SRT serão utilizadas, em qual máquina será iniciado o processamento, qual a freqüência de atividade do processo espião e qual o limite entre as faixas de *quiet/overloaded.*

Já as estruturas de saída são:

- *Trace Text Results*: neste arquivo texto ficam registrados os parâmetros empregados na execução, bem como a situação da carga dos trabalhadores sempre que ocorrer uma transição de estado em qualquer dos processos *solver* da arquitetura.
- *Trace Graphic Results*: o sistema gera uma massa de dados que será processada por uma ferramenta de visualização. Dentre os diversos gráficos gerados, temos número de processadores, carga no processador e períodos de atividade no processador ao longo da execução.

5 Trabalhos Relacionados

Entre as propostas de exploração de paralelismo na Programação em Lógica, temos aquelas que tratam do paralelismo OU ([ALI90], [BRI90], [WAR87] e [LUS88]), outras que tratam do paralelismo E ([CON85], [HER86a], [LIN88a] e [HER90]) e ainda aquelas que propõem a integração de ambos ([BAR88], [GUP89], [KAL89] e [HER91]). As implementações destes modelos, na sua maioria, foram previstas para máquinas com memória compartilhada.

A proposta de implementação de paralelismo E do modelo E/OU do projeto Opera, no que tange ao *hardware* utilizado, identifica-se com as propostas de [CIA92] e [VAN88], uma vez que ambas utilizam o ambiente de rede de computadores.

Segundo [CIA92], DeltaProlog é a única linguagem lógica concorrente que explora as duas formas de não-determinismo: *commited-choice*(*don't care*) e *backtrackable choice* (*don't know*). Esta linguagem é uma extensão da linguagem Prolog e suporta comunicação assíncrona, sendo que seu modelo é baseado em CSP (*Communicating Sequential Processes*). DeltaProlog se mostra muito eficiente para aplicações distribuídas.

Em [VAN88] é apresentado um sistema que explora paralelismo E em rede de computadores, através do uso de mensagens. O *hardware* utilizado para a validação do modelo foi um conjunto de Sun 3/50, conectadas por Ethernet (10 Mbps). O objetivo principal do autor é melhorar o desempenho de programas que não foram designados para paralelismo

e, ao mesmo tempo, implementar um sistema Prolog distribuído de maneira coerente e modular sobre um sistema Prolog seqüencial existente. Foram avaliados os resultados de quatro programas-exemplo: um compilador Prolog, um programa que implementa Fibonacci de maneira recursiva, um *quicksort* e um programa simples de xadrez. Todos os programas apresentaram ganho de desempenho, especialmente aqueles que envolvem mensagens pequenas e são *CPU-bound*, ou seja, aqueles em que o tempo de uso da CPU justifica o custo da comunicação.

6 Conclusão

Foi apresentada a proposta de exploração do paralelismo E em rede local de computadores. A implementação é baseada no modelo E/OU do Projeto Opera.

A arquitetura proposta é facilmente escalonável, prioriza os trabalhadores mais poderosos e é controlada por demanda. Com estas características, se busca adequar o modelo Opera E/OU a uma rede de estações heterogênea quanto ao poder computacional dos equipamentos e sujeita a cargas de trabalho diversificadas provenientes de outros usuários, ressaltando-se que este é o perfil típico da maioria das instalações.

A maioria dos trabalhos publicados que contemplam a exploração de paralelismo na programação em lógica são voltados para arquiteturas com memória compartilhada. O trabalho de [VAN88], uma das poucas implementações em redes de computadores, mostrou ser possível explorar, com ganho de desempenho, o paralelismo E. Nesta direção, os resultados obtidos até agora no projeto Opera são bastante satisfatórios.

7 Agradecimentos

Aos professores Ana Maria de A. Price e Philippe O. A. Navaux do Instituto de Informática da UFRGS pelo apoio recebido. A Jacques Briat, Michel Favre e J. Chassin de Kergommeaux do IMAG/LGI, Grenoble, França pela contribuição técnica. Aos bolsistas de Iniciação Científica Patrícia Kayser, Vinicius L. do Amaral, Marcos B.T. Casal e Roberto G. Ribeiro pelos trabalhos realizados. A CNPq/CAPES pelo suporte parcial aos trabalhos.

8 Referências Bibliográficas

Referências

[ALI90] ALI, Khayri A.M.; KARLSSON, Roland. The Muse Or-Parallel Prolog Model and its Performance. In: NORTH AMERICAN CONFERENCE ON LOGIC PROGRAMMING. Oct, 1990. **Proceedings...** : MIT Press, 1990. p.757-776.

[BAR88] BARON, Uri et al. The Parallel ECRC Prolog System PEPSys:An Overview and Evaluation Results. In: INTERNATIONAL CONFERENCE ON FIFTH GENERATION COMPUTER SYSTEMS. May, 1988, Tokyo. **Proceedings...** Tokyo: ICOT Press, 1988. p.841-849.

[BRI90] BRIAT, J.; FAVRE, M.; GEYER, C. et al. **Opera: OR-Parallel Prolog System on Supernode**. Implementations of Distributed Prolog. John Wiley & Sons Ltd, England, 1992.

[CIA92] CIANCARINI, Paolo. Parallel Programming with Logic Languages: a Survey. **Computer Languages**, Oxford, v.17, n.4, p.213-240, Oct. 1992.

[CON85] CONERY, J. S.; KIBLER, D.F. AND Parallellism and Nondeterminism in Logic Programs. **New Generation Computing**, Berlin, v.3, n.1, p.43-70, 1985.

[DEG84] DEGROOT, Doug. Restricted And-Parallelism. In: INTERNATIONAL CONFERENCE ON FIFTH GENERATION COMPUTER SYSTEMS. 1984. **Proceedings...** Tokyo: ICOT Press, 1984. p.471-478.

[DOU91] DOUGLIS, Fred; OUSTERHOUT, John. Transparent Process Migration: design alternatives and Sprite implementations. **Software - Practice and Experience**, New York, v.21, n.8, p. 757-785, Aug. 1991.

[GEY91] GEYER, Cláudio F.R. **Une Contribution a L'Etude du Parallelisme OU en Prolog sur des Machines sans Mémoire Commune**. PhD Thesis, Université Joseph Fourier, Grenoble, 1991.

[GEY92a] GEYER, Cláudio F.R., et al. Otimizações Importantes para Paralelismo OU em Prolog sobre Máquinas com Memória Distribuída. CLEI'92, Sep. 1992, Las Palmas.

[GEY92b] GEYER, Cláudio F.R.; YAMIN, Adenauer C.; WERNER, Otilia. Projeto Opera: Um Modelo E/OU para Prolog. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, Rio de Janeiro, 29/09-02/10. 1992. **Anais**. Rio de Janeiro, 1992. 390p. p.269-281.

[GUP89] GUPTA, G.; JAYARAMAN, B. A Model for And-Or Parallel Execution of Logic Programs. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, August 1989, Illinois. **Proceedings...**, 1989.

[GUP90] GUPTA, Gopal; JAYARAMAN, Bharat. Optimizing And-Or Parallel Implementation. In: NORTH-AMERICAN CONFERENCE ON LOGIC PROGRAMMING, April 1990. Austin. **Proceedings...** p.605-623.

[HER86a] HERMENEGILDO, M. **An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel**. PhD Thesis, University of Texas at Austin, 1986.

[HER90] HERMENEGILDO, M.; GUPTA, G. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, June 1990. **Proccedings...** MIT Press, 1990. p.253-268.

[HER91] HERMENEGILDO, M.; GUPTA, G. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 1991. **Proceedings...** Berlin: Springer-Verlag, 1991. p.146-158 (Lecture Notes in Computer Science)

[KAL89] KALE, L.; RAMKUMAR, B. Compiled Execution of the REDUCE-OR Process Model on Multiprocessors. In: NORTH AMERICAN CONFERENCE ON LOGIC PROGRAMMING, October 1989. **Proceedings...**

[LIN88a] LIN, Yow-Jian; KUMAR, Vipin. An Execution Model for Exploiting AND-Parallelism in Logic Programs. **New Generation Computing**, Berlin, v.5, p.393-425, 1988.

[LIN91] LIN, Zheng. A Distributed Fair Polling Scheme Applied to OR-Parallel Logic Programming. **International Journal of Parallel Programming**, New York, v.20, n.4, p.315-339, Aug. 1991.

[SHE91] SHEN, Kish; HERMENEGILDO, Manuel V. A Simulation Study of Or- and Independent And- parallelism. p.135-151. 1991.

[THE89] THEIMER, Marvin M.; LANTZ, Keith A. Finding idle machines in a workstation-based distributed system. **IEEE Transactions on Software Engineering**, New York, v.15, n.11, p.1444-1458, Sept.1989.

[VAN88] VAN ROY, Peter; CARLTON, Mike. A Distributed Prolog System with AND Parallelism. **IEEE Software**, p.43-51, Jan. 1988.

[WAR87] WARREN, David H.D. Or-Parallel Execution Models of Prolog. In: INTERNATIONAL JOINT CONFERENCE ON THEORY AND PRACTICE OF SOFTWARE DEVELOPMENT, Nov.1987, Pisa. **Proceedings...** Berlin: Springer-Verlag, 1987. p.243-259.