PARALELIZAÇÃO DE ALGORITMOS DE CONSISTÊNCIA DE ARCOS EM UM CLUSTER DE PC'S

Marluce Rodrigues Pereira

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Inês de Castro Dutra, Ph.D.

Prof. Maria Clicia Stelling de Castro, D.Sc.

Prof. Cláudio Luis de Amorim, Ph.D.

Prof. Vítor Santos Costa, Ph.D.

Prof. Cláudio Fernando Resin Geyer, Dr.

RIO DE JANEIRO, RJ - BRASIL

AGOSTO DE 2001

PEREIRA, MARLUCE RODRIGUES

Paralelização de Algoritmos de Consistência de Arcos em um Cluster de PC's [Rio de Janeiro] 2001

VII, 96 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2001)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1 - Programação Lógica com Restrições

2 - Algoritmos de Consistência de Arcos

3 - Memória Compartilhada-Distribuída

I. COPPE/UFRJ II. Título (série)

Aos meus pais Helvécio e Conceição

Agradecimentos

Agradeço:
às minhas orientadoras Inês de Castro Dutra e Maria Clicia Stelling de Castro, pelo incentivo, confiança e apoio em todos os momentos;
à CAPES que deu suporte financeiro à realização deste trabalho;
à Alvaro Ruiz-Andino que desenvolveu a versão original do sistema utilizado nesta dissertação e gentilmente nos enviou o código fonte;
à Silvano Bolfoni Dias que cedeu seu trabalho sobre OpenMP realizado para a cadeira de Programação Concorrente;
ao Prof. Cláudio Luis Amorim por ter cedido o *cluster* de PC's para realizarmos os experimentos;
ao corpo docente, administrativo e técnico da COPPE, que sempre me prestou ajuda quando precisei;
à Deus por ter me dado saúde e força para enfrentar os obstáculos;
aos meus pais, Helvécio e Conceição, que sempre me incentivaram e apoiaram nas horas difíceis;
à Vítor Santos Costa, por ter ficado com Mariana, enquanto Inês e eu trabalhávamos;
aos meus irmãos, Helvécio, Márcio e Marci, que carinhosamente auxiliaram-me em todos os momentos que precisei;
ao meu namorado Marcelo, que sempre me incentivou e me compreendeu;
a todos os amigos que sempre se mostraram dispostos a ajudar quando eu precisei.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M. Sc.)

PARALELIZAÇÃO DE ALGORITMOS DE CONSISTÊNCIA DE ARCOS EM UM CLUSTER DE PC'S

Marluce Rodrigues Pereira

Agosto/2001

Orientadoras: Inês de Castro Dutra
Maria Clicia Stelling de Castro
Programa: Engenharia de Sistemas e Computação

Esta tese concentra-se no estudo e na análise de desempenho de um sistema que implementa consistência de arcos em paralelo. O sistema base utilizado, CSOS, foi originalmente implementado para plataformas de memória compartilhada. Neste trabalho, modificamos o sistema para ser executado em uma plataforma *software* DSM (*Distributed Shared Memory*), utilizando o protocolo TreadMarks numa rede de PC's. Utilizamos aplicações com características que variam o número total de restrições e a topologia do grafo de inter-dependência entre as variáveis. Este grafo de interdependência foi particionado de forma a melhorar a localidade da computação. Realizamos os experimentos em um cluster de 8 PC's conectados através de uma rede *fast-ethernet* e *giganet*. Nossos 1 mostram *speedups* superlineares para algumas aplicações devido ao nosso particionamento e ao *labeling* realizado em paralelo.

As contribuições mais relevantes deste trabalho são apresentar a paralelização de um sistema de satisfação de restrições para uma plataforma DSM, que possui uma relação de custo/benefício atrativa, e demonstrar que a implementação apresenta boa escalabilidade.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M. Sc.)

PARALLELIZATION OF ARC-CONSISTENCY ALGORITHMS ON A PC CLUSTER

Marluce Rodrigues Pereira

August/2001

Advisors: Inês de Castro Dutra

Maria Clicia Stelling de Castro

Department: Systems Engineering and Computer Sciences

This thesis focuses on the study and performance analysis of a parallel arc-consistency algorithm, the AC-5. The base system we use, CSOS, was originally parallelised for a logically shared memory platform, the Cray T3E. In this work, we modified this parallel implementation, PCSOS, to run on a distributed shared memory plataform, using the TreadMarks protocol on a cluster of 8 PCs connected via a Fast-Ethernet network. We used four applications whose characteristics vary according to the connectivity pattern of the constraint graph. One of our applications has a constraint graph which is totally connected, another has a constraint graph which is weakly connected, and the other two have strongly connected graphs of different degrees. We explored opportunities for parallelisation by implementing two kinds of partitioning for the *indexicals* and two kinds of *labeling*. Our results show that we can obtain linear speedups for one application and superlinear speedups for another application by doing distributed labeling. When the constraint graph is strongly connected we need to employ sophisticated techniques to implement better partitioning of indexicals and better distributed labeling. Our results also show that invalidate-based coherence protocols, as used by TreadMarks, compromises the performance of our applications, hindering scalability.

Índice

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Esta tese concentra-se no estudo e na análise de desempenho de um sistema que implementa consistência de arcos paralelizada. O sistema base utilizado, CSOS [1], foi originalmente implementado para plataformas de memória compartilhada (CRAY T3E)[36] e, neste trabalho, foi modificado para ser executado em uma plataforma *software* DSM (*Distributed Shared Memory*), utilizando-se TreadMarks [25].

Este capítulo apresenta a motivação, o objetivo e as contribuições deste trabalho. Além disso, apresentamos como este trabalho está organizado.

1.1 Motivação

Muitos problemas de Inteligência Artificial utilizam programação lógica com restrições sobre domínios finitos [9] para especificar e resolver problemas de satisfação de restrições e permitir a realização de otimizações. Os problemas de satisfação de restrições sobre domínios finitos são combinatórios, e, portanto apresentam um espaço de busca exponencial que pode ser podado através de algoritmos de consistência.

Um problema formulado como satisfação de restrições possui três componentes principais: variáveis, domínios e restrições. A representação do problema é feita através de equações e/ou inequações que relacionam as variáveis. Estas equações/inequações representam as restrições do problema. Cada variável é inicializada com seu domínio respectivo, ou seja, um conjunto de valores, que no caso de domínios finitos é discreto, finito e enumerável. A poda do espaço de busca ocorre na remoção de valores inconsistentes de acordo com as restrições impostas sobre as mesmas. Por exemplo, suponha o problema de se colocar N rainhas em um tabuleiro de xadrez $N \times N$ de tal forma que as rainhas não se ataquem [1]. As rainhas se atacam se estiverem na mesma linha, na mesma coluna, na mesma diagonal principal

ou na mesma diagonal secundária. Cada rainha deve ser colocada em uma linha do tabuleiro. O problema consiste em selecionar uma coluna para cada rainha, de forma que elas não se ataquem. Para fazer a representação em satisfação de restrições deste problema, associamos cada rainha a uma variável, que pode assumir valores de 1 a N, que são os valores das colunas que as rainhas podem ocupar. As restrições correspondem às condições necessárias e suficientes para que as rainhas não se ataquem. Sejam X e Y duas rainhas. A restrição para que a rainha Y seja colocada no tabuleiro de xadrez de forma que não ataque, na diagonal principal, a rainha X, colocada anteriormente, pode ser escrita da seguinte forma $Y \neq X + I$ e para que Y não ataque X na diagonal secundária a restrição é $Y + I \neq X$, onde $I \in 1, ..., N - 1$. Estas restrições devem ser definidas para todos os pares de variáveis. A restrição correspondente a não colocar rainhas na mesma coluna é dada por $Y \neq X$.

Uma classe importante dos algoritmos de consistência para resolver problemas de satisfação de restrições é a classe dos algoritmos de consistência de arcos.

Os algoritmos de consistência de arcos permitem eliminar valores inconsistentes do domínio das variáveis de forma a podar o espaço de busca. A primeira fase destes algoritmos seleciona uma variável e um valor do domínio (fase de *labeling*). Com esta variável e valor, os algoritmos passam pela segunda fase, de propagação, que poderá eliminar valores inconsistentes das outras variáveis, examinando cada restrição da variável que foi escolhida. O algoritmo termina quando é encontrada uma atribuição de um único valor para cada variável ou se não encontrar uma solução. Este tipo de algoritmo trata de restriç oes entre apenas duas variáveis.

Na literatura, diversos algoritmos de consistência de arcos foram propostos como AC-1 [26], AC-2 [29], AC-3 [29], AC-4[32], AC-5[19] e AC-6 [5]. Nosso trabalho concentra-se no algoritmo AC-5, que consiste de uma generalização dos algoritmos AC-1 a AC-4. O algoritmo AC-5 tem complexidade seqüencial similar a dos outros algoritmos: AC-1 é $O(ena^3)$, AC-2 e AC-3 são $O(ea^3)$, AC-4 é $O(ea^2)$, AC-5 é $O(ea)$ e AC-6 é $O(ea^2)$, onde e é o número de arcos (restrições), a é o tamanho do domínio e n é o número de variáveis. Portanto, utilizamos um algoritmo de complexidade baixa nos experimentos paralelos.

Utilizamos em nossos experimentos o sistema CSOS, implementado por Andino *et al.* [3], que implementa o algoritmo AC-5, além de resolver problemas de otimização utilizando Branch and Bound, algoritmos genéticos e outras técnicas de Inteligência Artificial. Nosso trabalho concentra-se apenas no algoritmo de con-

sistência e sua paralelização. O sistema CSOS oferece ao usuário uma interface de programação com restrições baseada em programação lógica [27]. Este modelo chamado de programação lógica com restrições é bastante adotado por programadores nas aplicações de otimização [22], pois permite a construção de códigos mais diretos e sucintos. Além disso, programação lógica com restrições livra o programador de ter que implementar a busca por soluções através de *backtracking*.

Andino *et al.* desenvolveram uma versão paralela do CSOS para uma arquitetura CRAY T3E com 34 processadores. Seus experimentos mostram *speedups* de até 18 para 34 processadores no melhor caso. Andino *et al.* concentram-se na distribuição estática e dinâmica de restrições entre os processadores sem maiores preocupações com a interdependência entre as restrições. Além disso, a fase de *labeling* é realizada de forma seqüencial, ou seja, cada variável somente é escolhida para ser rotulada após o término da execução do *labeling* para a variável anterior.

No nosso trabalho, exploramos o particionamento de restrições entre os processadores de acordo com a interdependência entre as restrições, além de realizar o *labeling* de cada subconjunto de restrições em paralelo. Realizando o particionamento das restrições entre os processadores e permitindo que o *labeling* de cada subconjunto seja feito em paralelo conseguimos *speedups* super-lineares para uma classe de aplicações, num *cluster* de 8 PC's. Um *cluster* se caracteriza por um conjunto de computadores interconectados por uma rede rápida com um objetivo comum.

Os experimentos realizados por Andino *et al.* utilizam uma plataforma para processamento paralelo de alto custo. Nosso trabalho procura explorar o potencial de plataformas de baixo custo (*hardware* + *software*).

Atualmente, as redes de PC's apresentam relação de custo/benefício muito atrativa e vêm obtendo bastante destaque. Assim, a utilização de redes de PC's na execução de sistemas paralelos apresenta-se muito viável. Isto nos motivou a mudar a plataforma do sistema implementado por [2] para uma rede de PC's. Este fato acarreta a mudança no modelo de programação paralela, realizando as devidas adaptações para expressar o paralelismo numa rede de PC's. Os sistemas *software* DSM possuem um modelo de programação bastante amigável. Neste trabalho utilizamos o sistema *software* DSM TreadMarks [25], que é um dos principais *softwares* DSM utilizados na comunidade científica e está disponível em nosso laboratório.

1.2 Objetivo

O nosso objetivo neste trabalho é analisar o desempenho do sistema paralelo de satisfação de restrições, implementado por [2], que adaptamos para uma plataforma DSM, utilizando TreadMarks. A finalidade desta adaptação é analisar o desempenho de sistemas de satisfação de restrições paralelos sobre uma rede de PC's utilizando o modelo de programação de um sistema *software* DSM.

1.3 Contribuições do autor

Este trabalho contribui apresentando a paralelização de um algoritmo de satisfação de restrições para um sistema *software* DSM, utilizando uma rede de PC's que possui uma relação de custo/benefício atrativa. Além disso, demonstramos que a implementação pode apresentar boa escalabilidade, permitindo o seu desenvolvimento em nível comercial.

1.4 Organização do texto

Este texto está organizado da seguinte forma. O capítulo 2 aborda os conceitos sobre programação lógica com restrições e consistência de arcos. No capítulo 3 apresentamos a paralelização de consistência de arcos. No capítulo 4 apresentamos os conceitos básicos de *software* DSM e TreadMarks. No capítulo 5 descrevemos os sistemas seqüencial e paralelo implementados por [2]. No capítulo 6 descrevemos as versões paralelas que propomos para uma plataforma *software* DSM. No capítulo 7 descrevemos a metodologia, o ambiente utilizado e analisamos os resultados experimentais. No capítulo 8 apresentamos as nossas conclusões e propomos trabalhos futuros.

Capítulo 2

Programação Lógica com Restrições e Consistência de Arcos

Este capítulo introduz programação lógica com restrições e sua utilização e apresenta a terminologia necessária ao entendimento deste trabalho. Com a programação lógica com restrições o espaço de busca pode ser cortado, diminuindo assim a quantidade de *backtracking*, que é a principal causa do baixo desempenho de execução de programas em lógica [37].

2.1 Programação lógica com restrições

A idéia principal da programação lógica com restrições ou CLP (*Constraint Logic Programming*) é trocar a unificação por um tratamento de restrições num domínio de restrições. Este esquema, chamado CLP(X), foi proposto por Jaffar e Lassez [21]. O X tem sido instanciado com diversos domínios de computação, como por exemplo, reais em CLP(R), racionais em CLP(Q), inteiros em CLP(Z) e domínios finitos em CLP(FD), cujo enfoque é dado por Codognet e Diaz [9].

Uma linguagem CLP(X) é definida pelas restrições do domínio X e pelo *solver* de restrição de domínio X. Um *solver* é um método para resolver problemas de satisfação de restrições (CSPs).

A adição de *solvers* aos sistemas de programação lógica muda o paradigma de busca pelas soluções, de *generate-and-test* (primeiro gera uma valoração e então testa se é a solução) e comportamento de *backtracking* padrão (enumera valores para uma variável de cada vez, gerando no final todas as soluções) para o que é chamado de *constrain-and-generate*. No método *constrain-and-generate* primeiro as restrições são aplicadas, em seguida uma solução é gerada por *labeling*. O procedimento de *labeling*

utiliza uma busca *backtracking* para encontrar uma solução para as restrições [31].

Um problema de satisfação de restrições (CSP) é um tipo especial de problema de busca que possui estados e domínios. Os estados são conjuntos de variáveis. O estado inicial é um conjunto de variáveis com valores possíveis iniciais. O estado final é um conjunto de variáveis com valores que respeitem as restrições do problema. O domínio é o conjunto possível de valores que uma variável pode assumir, que pode ser discreto ou contínuo e finito ou infinito.

O CSP define restrições sobre variáveis e um domínio que relaciona cada variável a um conjunto finito de valores. É possível determinar a satisfatibilidade de um CSP através de todas as combinações de diferentes valores, já que este número é finito. Entretanto, este procedimento de satisfatibilidade pode ser custoso, uma vez que grande parte dos CSPs têm espaço de busca exponencial [2]. Porém, existem aqueles que possuem complexidade polinomial.

Vários *solvers* de CSP que têm complexidade polinomial, no pior caso, baseiam-se na observação de que se o domínio de qualquer variável do CSP é vazio, então o CSP é insatisfatível. O objetivo destes *solvers* é transformar o CSP em outro equivalente com os domínios menores para as variáveis. Se qualquer um dos domínios do CSP equivalente se tornar vazio, este é insatisfatível. Conseqüentemente, o CSP original também é insatisfatível. Por equivalência, as restrições representadas pelo CSP original e o CSP equivalente têm o mesmo conjunto de soluções [31]. Para encontrar as soluções o *solver* passa por fases de eliminação de valores do domínio das variáveis, de acordo com as restrições.

Os *solvers* dependem do domínio das variáveis. No caso de restrições no domínio real (infinito), são utilizados métodos matemáticos como eliminação de Gauss e Fourier-Moutzkin ou métodos computacionais como o Simplex. No caso de domínios finitos, o método mais utilizado é o de consistência de arcos. Nesta tese nos concentramos em domínios finitos com método de consistência de arcos.

Para os problemas que possuem domínios finitos, são necessárias duas escolhas: a escolha da variável e a escolha do valor da variável. A escolha da variável pode ser *most-constrained*, ou seja, aquela de menor domínio ou *most-constraining*, que restringe ao máximo os domínios das outras variáveis. A escolha do valor da variável pode ser pelo princípio *least contraining*, onde o valor escolhido é aquele que afeta menos o conjunto de valores das outras variáveis.

Considerando apenas problemas de satisfação de restrições que possuem um con-

junto de variáveis, um domínio finito para cada variável e um conjunto de restrições unárias ou binárias, é possível representar o CSP por um grafo de restrições, onde cada nó representa uma variável e cada arco representa uma restrição entre variáveis representada pelo ponto final de cada arco. Qualquer CSP com restrições n-árias pode ser convertido para outro CSP binário equivalente [26].

Por exemplo, um problema de coloração de mapas pode ser considerado como um CSP. Neste problema, precisamos colorir cada região do mapa com uma cor (de um conjunto de cores), tal que duas regiões adjacentes não tenham a mesma cor. A figura 2.1 mostra um exemplo do problema de coloração de mapas e seu CSP equivalente. O mapa tem quatro regiões que devem ser coloridas com verde, azul ou vermelho. O CSP equivalente tem uma variável para cada uma das quatro regiões do mapa. O domínio de cada uma das variáveis é dado pelo conjunto de cores. Para cada par de regiões que são adjacentes no mapa, há uma restrição binária ($V_i \neq V_j$) entre as variáveis correspondentes que não permite atribuições idênticas para as duas variáveis. No grafo da figura 2.1, cada nó representa uma variável do problema e as arestas representam as restrições existentes entre as variáveis.

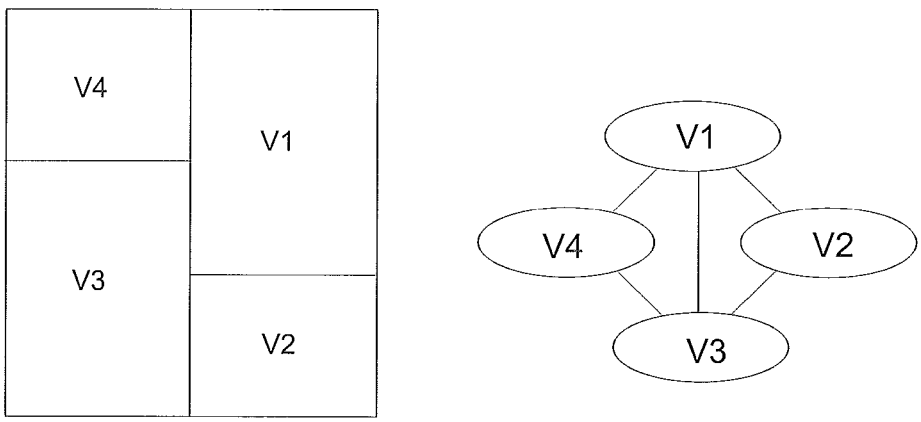

Figura 2.1: Exemplo de um problema de coloração de mapas e seu problema de satisfação de restrições equivalente [26]

Na implementação do *solver* podem ser identificadas duas fases principais: consistência de nós e consistência de arcos, que estão descritas a seguir.

A consistência de nós elimina aqueles valores dos domínios das variáveis que as tornam inconsistentes de acordo com as restrições, isto é, restrições que envolvam apenas uma variável (por exemplo, $V > 0$). Em seguida, na fase de consistência de arcos, para cada variável são consideradas as restrições da variável e o domínio de outra variável envolvida nas mesmas restrições. O domínio de uma outra variável

C ::=	X in r	
r ::=	t1..t2	intervalo
	{t}	um número
	R	parâmetro
	dom(Y)	domínio de outra variável
	r1 : r2	união
	r1 & r2	interseção
	−r	complemento
	r + ct	soma
	r − ct	subtração
	r * ct	multiplicação
	r / ct	divisão
t ::=	min(Y)	t é o valor mínimo de Y
	max(Y)	t é o valor máximo de Y
	ct	termo constante
	$t_1 + t_2 \mid t_1 - t_2 \mid t_1 * t_2 \mid t_1 < t_2 \mid t_1 > t_2$	operações inteiras
ct ::=	C	o termo é um parâmetro
	n | infinito	maior valor
	$ct_1 + ct_2 \mid ct_1 - ct_2 \mid ct_1 * ct_2 \mid$ $ct_1 < ct_2 \mid ct_1 > ct_2$	

Tabela 2.1: Sintaxe de restrições *X in r*.

então pode ser manipulado para podar os valores inconsistentes baseado nas restrições. O processo se repete até que nenhum valor do domínio de uma variável possa ser eliminado.

Como consistência de arcos se aplica apenas a restrições binárias, isto é, que relacionam apenas duas variáveis, é necessário, muitas vezes, realizar um pré-processamento das restrições para torná-las binárias. O pré-processamento pode ser realizado utilizando o esquema de *indexicals* [9].

O esquema de *indexicals* consiste em definir as restrições na forma *X in r*, onde *X* é uma variável e *r* é uma expressão. Desta forma, uma restrição mais complexa é representada por um conjunto de *indexicals*. Cada *indexical* descreve uma variável *X* dependente das demais variáveis. A sintaxe de restrições representadas por *indexicals* é mostrada na tabela 2.1 [9]. A entrada das restrições no sistema de restrições pode ser feita pelo usuário na forma de restrições mais complexas ou na forma de *indexicals*.

A estrutura onde o domínio mais atual é armazenado recebe o nome de *store*.

O valor inicial do *store* é definido pelo domínio inicial das variáveis. Quando um *indexical* é executado, o domínio da variável relativa àquele *indexical* é atualizado no *store* provocando mudanças no domínio de outras variáveis relacionadas a este *indexical.*

Suponha a restrição aritmética $v_1 = v_2 + 4$ sobre o domínio dos inteiros [2]. Ela pode ser transformada em dois *indexicals*, envolvendo apenas duas variáveis de cada vez:

$$I_1 = v_1 \; in \; min\,(v_2) + 4 \;\; .. \;\; max\,(v_2) + 4$$

$$I_2 = v_2 \; in \; min\,(v_1) - 4 \;\; .. \;\; max\,(v_2) - 4$$

Quando o limite inferior de v_2, $min\,(v_2)$, ou o limite superior, $max\,(v_2)$ se modifica, devido à eliminação de algum valor do domínio, o *indexical* I_1 remove valores inconsistentes do domínio de v_1. O *indexical* I_2 comporta-se similarmente. Suponha que os domínios iniciais de v_1 e v_2 sejam iguais a $[1, \ldots, 10]$. Ao aplicarmos o algoritmo de consistência de nós, o *indexical* I_1 é executado e o valor 1 é atribuído a $min\,(v_2)$ e o valor 10 a $max\,(v_2)$. Porém, o valor máximo do domínio de v_1 é 10. Assim, a restrição I_1 poda o domínio de v_1 para $[5, \ldots, 10]$ e I_2, da mesma forma, poda o domínio de v_2 para $[1, \ldots, 6]$. Se outro *indexical* relacionado a v_2 poda seu domínio para $[4, \ldots, 5]$, I_1 substituirá $min\,(v_2)$ por 4 e $max\,(v_2)$ por 5, podando o domínio de v_1 para $[8, \ldots, 9]$. Entretanto, esta modificação é propagada por causa da execução dos *indexicals* que dependem de v_1 na fase de consistência de arcos.

Como mencionado anteriormente, várias versões de algoritmos de consistência de arcos podem ser encontradas na literatura, como AC-1, AC-2, AC-3, AC-4, AC-5 e AC-6. A próxima seção enfoca estes algoritmos de consistência de arcos.

2.2 Algoritmos de consistência de arcos

Um arco (V_i, V_j) é consistente se para todo valor x no domínio de V_i existe algum valor y no domínio de V_j tal que $V_i = x$ e $V_j = y$ é permitido pela restrição binária entre V_i e V_j. O conceito de consistência de arcos é unidirecional, isto é, se um arco (V_i, V_j) é consistente não significa automaticamente que o arco (V_j, V_i) também é consistente [26]. Considere o grafo de restrições do problema de coloração de mapas ilustrado na figura 2.2, onde o domínio de $V_1 = \{azul, verde, vermelho\}$, $V_2 = \{azul, verde\}$ e $V_3 = \{verde\}$ e as arestas representam as restrições existentes entre V_1, V_2 e V_3.

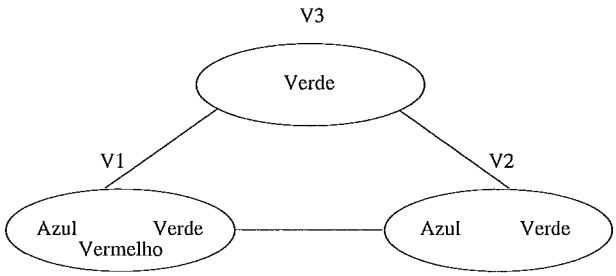

Figura 2.2: Grafo de restrições para um problema de coloração de mapas

No grafo de restrições, o arco (V_3, V_2) é consistente porque verde é o único valor no domínio de V_3, e para $V_3 = \textit{verde}$, existe no mínimo uma atribuição para V_2 que satisfaça a restrição entre V_3 e V_2. Se $V_2 = \textit{verde}$, então o arco (V_2, V_3) não pode se tornar consistente, pois não há valor no domínio de V_3 que seja permitido pela restrição entre V_2 e V_3.

Um arco (V_i, V_j) pode se tornar consistente pela eliminação daqueles valores do domínio de V_i que não satisfaçam todas as restrições existentes entre V_i e V_j.

Para tornar todos os arcos do grafo de restrições consistentes existem vários algoritmos de consistência de arcos.

Os algoritmos AC-1 e AC-3 de Mackworth [26], possuem um procedimento denominado REVISE que é mostrado na figura 2.3.

```
procedure REVISE(V_i, V_j);
1    DELETE ←false;
2    for each x ∈ D_i do
3       if there is no such v_j ∈ D_j
4          such that (x, v_j) is consistent,
5       then
6          delete x from D_i;
7          DELETE ←true;
8       end-if;
9    end-for;
10   return DELETE;
end-REVISE
```

Figura 2.3: Procedimento REVISE

O procedimento REVISE elimina valores inconsistentes dos domínios das variáveis, de acordo com as restrições. Este procedimento deve ser executado mais de uma vez para cada arco do grafo de restrições para que o grafo se torne consis-

tente. Isto significa que para cada valor v_i do domínio de uma variável V_k (D_{V_k}) é verificado se existe um valor v_j de uma variável V_l (D_{V_l}) que satisfaça a restrição (V_k, V_l). Se não existe, o valor v_i é eliminado de D_{V_k}. A fila de arcos é fixa. Como o procedimento REVISE pode reduzir o domínio de alguma variável V_k, cada arco (V_l, V_k) tem que ser revisado novamente, pois a poda de V_k pode ter removido o único valor que fazia algum dos elementos do domínio de V_l compatível com V_k.

O algoritmo AC-1 armazena todos os arcos do grafo de restrições numa fila. O procedimento REVISE é executado para cada arco. A desvantagem deste algoritmo é que uma revisão de um arco numa iteração força que todos os arcos sejam revisados na próxima iteração, mesmo que o número de arcos afetados pela revisão seja pequeno. Para corrigir este problema, Mackworth propôs um outro algoritmo, que é uma variação do AC-1, denominado AC-3.

O AC-3 re-executa o procedimento REVISE somente para aqueles arcos que são possivelmente afetados por uma revisão anterior.

O algoritmo AC-3 é uma versão mais simples e geral do algoritmo AC-2, também chamado de *Waltz filtering*. Desta forma, no algoritmo AC-3 a fase de consistência de nós é realizada de forma separada da consistência de arcos e além disso, a revisão dos arcos é feita para todos os arcos relacionados ao nó que está sendo tratado. No AC-2 a revisão dos arcos é realizada obedecendo a uma ordenação prévia imposta aos nós do grafo. Desta forma, o algoritmo AC-2 realiza mais revisões que o algoritmo AC-3 [30].

Um outro algoritmo de consistência de arcos encontrado na literatura é o AC-4 [32] que é baseado na noção de conjunto suporte, que é explicado a seguir. Considere um valor b de um nó i. Se b tiver um conjunto de suporte mínimo formado por valores de cada um dos outros nós j ($j \neq i$), ou seja, se existir algum valor dos outros nós que satisfaça a relação entre b e os outros nós, b é considerado um valor viável para o nó i. Mas, se existe um nó que nenhum dos seus valores satisfaça a relação requerida com b, então b pode ser eliminado do conjunto de possíveis valores para o nó i. O algoritmo atribui um contador para cada par arco-valor. Tais pares são denotados por $[(i,j), b]$, onde (i,j) indica o arco de i para j, e b é o valor do nó i. Além disso, para cada valor c do nó j, o conjunto S_{jc} é construído, onde $S_{jc} = \{(i,b) \mid c \text{ do nó } j \text{ suporta } b \text{ do nó } i\}$. Se c é eliminado do nó i, então os contadores de $[(i,j), b]$ devem ser decrementados para cada b suportado por c. Uma tabela M é usada para manter a informação de que valores foram eliminados de determinados objetos. Além disso,

é mantida também uma lista *List* para controlar a propagação de restrições. O algoritmo é mostrado na figura 2.4.

```
foreach value b of D_i
   foreach value c of D_j
      if value c of D_j attends the constraint (i, j) for value b of D_i
      then
         add (i, b) to S_jc
      else
         decrease the counter [(i, j), b]
   end-for
end-for
```

Figura 2.4: Algoritmo AC-4

Segundo [32], o algoritmo AC-1 tem complexidade $O(ena^3)$, o AC-3 tem complexidade $O(ea^3)$ e o AC-4 é $O(ea^2)$, onde e corresponde ao número de arcos (restrições), a é o tamanho do domínio e n é o número de variáveis. Os algoritmos AC-1 e AC-3 consideram que valores cortados anteriormente precisam ser avaliados novamente. Estes dois algoritmos armazenam uma fila de pares (i, j) que são todos consultados. O algoritmo AC-4 considera que valores cortados das variáveis não precisam ser novamente examinados, detectando inconsistências mais cedo.

Já o algoritmo AC-5, apresentado em [19], é um algoritmo genérico. As variáveis assumem valores de números naturais e possuem um domínio finito D_i associado. Todas as restrições são binárias e relacionam duas variáveis distintas. O AC-5 implementa um grafo, onde um arco representa uma restrição entre duas variáveis. Ele possui uma fila Q que contém elementos da forma $< (i,j), w >$, onde (i,j) é um arco e w é o valor que foi removido do domínio D_j, e justifica a necessidade de reconsiderar o arco (i,j).

O algoritmo AC-5 possui dois passos principais. No primeiro passo, todos os arcos são considerados uma vez e a consistência de nós é realizada para cada um deles. Uma fila Q é gerada com elementos da forma $< (k,i), w >$, onde (k,i) são arcos que chegam em i e w são elementos removidos ao se fazer a consistência de (k,i). No segundo passo, computa-se valores em D_i afetados pela remoção de w para cada elemento de Q, possibilitando a geração de novos elementos em Q e a repetição do processo.

Na implementação do algoritmo AC-5, utilizando restrições sobre domínios fini-

tos, os domínios das variáveis são conjuntos finitos de números naturais. Nosso trabalho concentra-se numa implementação do algoritmo AC-5.

O algoritmo de consistência de arcos AC-6 é uma especialização do algoritmo AC-4. O algoritmo AC-6 mantém a complexidade de tempo do AC-4, elimina a complexidade de espaço e verifica somente dados suficientes nas restrições para computar o domínio na consistência de arcos. O algoritmo AC-6 somente utiliza um valor por restrição para provar que um valor é viável: para cada valor, é verificado somente um conjunto suporte por restrição e verifica outro valor somente quando o valor suporte atual é retirado do domínio.

Outros algoritmos de consistência existem na literatura tais como *k-consistency* [26] e consistência de caminhos [32, 29], cuja discussão está fora do escopo desta tese. Estamos nos concentrando apenas em consistência de arcos, pois estes são computacionalmente menos complexos.

2.3 Particionamento e distribuição do *labeling* em consistência de arcos

A complexidade dos algoritmos de consistência de arcos pode ser alta por causa de sua natureza combinatória. Vimos na seção anterior que a base destes algoritmos é varrer todos os valores de uma variável e combinar com todos os outros valores das outras variáveis. Existem, no entanto, formas de tornar este problema combinatório apenas particionando o conjunto de *indexicals* e variáveis em sub-conjuntos. Desta forma, a combinação é realizada apenas dentro de cada subconjunto. A complexidade do algoritmo AC-5, por exemplo, que é $O(ea)$ ficaria reduzida a uma fração de ea, onde e é o número de restrições e a é o tamanho do domínio. Tal particionamento, no entanto, somente é possível de ser realizado, sem muita sofisticação, para grafos fracamente conectados, onde a interdependência entre as variáveis apresenta um padrão regular.

Uma forma de fazer particionamento de *indexicals* e *labeling* distribuído é através de um algoritmo de consistência de arcos concorrente, onde cada processo ou *thread* trabalha em seu subconjunto de *indexicals* e de variáveis. Neste tipo de algoritmo, como cada processo trabalha em seu próprio subconjunto de dados e podem ocorrer dependências entre os subconjuntos, introduz-se o problema de consistência **dentro** dos subconjuntos e **entre** os subconjuntos. Chamamos a consistência dentro

dos subconjuntos de consistência **local** e consistência entre subconjuntos de consistência **global**. Quanto maior o número de subconjuntos e mais irregular o padrão de interdependência entre as variáveis do grafo mais complexa será a detecção de consistência global.

Dependendo do grafo de restrições da aplicação que está sendo executada e do padrão de interdependência, este problema de consistência apresenta-se de forma mais marcante. Se o particionamento das variáveis do grafo for feito de forma que toda variável de um subconjunto S_1 seja dependente de uma variável de qualquer outro subconjunto S_2, as falhas serão consideradas globais. Porém, se em algum subconjunto S_1 existir algum conjunto de variáveis V, $V \subset S_1$, que não seja dependente de outro subconjunto qualquer S_2, as falhas deste conjunto V poderão ser locais ou globais. A detecção automática destes subconjuntos é viável, porém complexa, e está fora do escopo desta tese. Neste trabalho, quando o grafo apresenta um padrão de interdependência regular procuramos fazer a distribuição do *labeling* de forma manual.

2.4 Resumo

Este capítulo introduziu a programação lógica com restrições. Além disso, foi definido o que é um problema de satisfação de restrições e como as técnicas de consistência de arco podem ser utilizadas para resolvê-los. O esquema de *indexicals*, utilizado para transformar restrições n-árias em binárias, foi também descrito. Para terminar, foram apresentados os algoritmos de consistência de arcos: AC-1, AC-2, AC-3, AC-4, AC-5 e AC-6.

O próximo capítulo descreve como pode ser feita a paralelização do algoritmo de consistência de arcos, baseada na proposta do algoritmo AC-5.

Capítulo 3

Paralelização de Consistência de Arcos

Os algoritmos de consistência de arcos possuem a característica de poderem ser paralelizados. O uso do esquema de *indexicals* facilita a implementação paralela porque os *indexicals* podem ser divididos em subconjuntos. Assim, cada processador pode receber um subconjunto de dados e executar o algoritmo de consistência de arcos seqüencialmente sobre seus dados locais. Desta forma, é possível obter melhor desempenho para as aplicações, além de permitir a escalabilidade das massas de dados de entrada das aplicações.

Este capítulo apresenta alguns modelos de paralelização que podem ser usados para se implementar algoritmos de consistência de arcos.

3.1 Memória compartilhada × troca de mensagens

Existem dois modelos básicos de programação paralela: o baseado em memória compartilhada e o baseado em troca de mensagens. O paralelismo, em ambos os modelos, pode ser expresso através de suporte do sistema operacional, através de linguagens paralelas ou através de bibliotecas especializadas. O modelo de memória compartilhada proporciona a visão do espaço global de endereçamento para vários processos. Já o modelo de troca de mensagens possui primitivas de comunicação entre processos.

Algumas das bibliotecas que suportam paralelismo são: TreadMarks [35], PVM (*Parallel Virtual Machine*) [15], MPI (*Message-Passing Interface*) [34] e OpenMP [11]. Cada uma delas está descrita a seguir.

TreadMarks é um sistema *software* DSM (*Distributed Shared Memory*) que implementa o modelo de programação de memória compartilhada, executando sobre um *hardware* de memória distribuída. A implementação de TreadMarks visa reduzir a quantidade de comunicação necessária para manter as memórias distribuídas consistentes, já que fisicamente as máquinas não compartilham memória [25]. TreadMarks esconde do programador a troca de mensagens que ocorre entre os processos. A sua biblioteca de funções pode ser ligada tanto a aplicações implementadas em linguagem C quanto em linguagem Fortran. Um exemplo de implementação de um programa paralelo utilizando TreadMarks está ilustrado no apêndice A.

As bibliotecas PVM e MPI implementam o modelo de memória distribuída.

PVM é um conjunto integrado de ferramentas e bibliotecas que emulam estruturas de computação concorrente sobre computadores interconectados que podem possuir arquiteturas variadas. Um dos objetivos principais do sistema PVM é permitir que uma coleção de computadores possa ser usada cooperativamente para computação concorrente ou paralela [15]. PVM pode ser chamado a partir de um programa implementado em linguagem C ou Fortran.

MPI tenta estabelecer um padrão de passagem de mensagens para permitir portabilidade. Possui uma biblioteca de funções que pode ser chamada a partir de um programa implementado em linguagem C ou Fortran. O objetivo deste grupo de funções é possibilitar paralelismo através da passagem de mensagens. Uma função de passagem de mensagens transmite dados de um processo para outro, explicitamente. MPI é projetado para rodar na maioria das plataformas [34].

OpenMP é uma especificação para um conjunto de diretivas de compilador, biblioteca de rotinas e variáveis de ambiente que podem ser usadas para especificar paralelismo com modelo de memória compartilhada. OpenMP é, também, uma extensão para as linguagens Fortran, C e C++, mas não introduz nenhum construtor que necessite de características específicas de Fortran 90 ou C++ [11].

Nosso trabalho concentra-se na paralelização de consistência de arcos através da utilização de TreadMarks.

3.2 Paralelização e particionamento em consistência de arcos

Os algoritmos de consistência de arcos possuem a característica de permitirem que a execução de *indexicals* seja feita sobre um subconjunto de dados. Assim, cada processador pode executar separadamente um subconjunto de *indexicals*, estabelecendo comunicação com os demais processadores apenas quando houver dependência entre os *indexicals*. Esta comunicação é realizada através do *store* que é uma estrutura de dados compartilhada por todos os processadores. Cada *indexical* comporta-se como um processo concorrente que atualiza o *store* cada vez que é afetado por uma outra alteração anterior no *store*. O *store* deve ser acessado por todos os processadores para que toda mudança ocorrida neste, provocada por um determinado processador e que afete os *indexicals* de um outro processador, seja vista pelos outros processadores e as atualizações sejam realizadas.

Existem três tipos de particionamento para resolver problemas de satisfação de restrições paralelos: particionamento de variáveis, de restrições e de domínios. Nosso trabalho utiliza o particionamento de variáveis e de restrições, que são os *indexicals*. Além disso, a fase de *labeling* também foi paralelizada.

Em nosso trabalho, o particionamento de variáveis é realizado de forma que as variáveis que possuam muitas restrições a elas relacionadas fiquem preferencialmente no mesmo processador. Desta forma, podemos reduzir a comunicação entre os processadores. Na fase de *labeling* as variáveis podem ser distribuídas uniformemente entre os processadores para que o procedimento de *labeling* seja realizado em paralelo por todos os processadores. Ou seja, cada processador realiza o *labeling* sobre o seu grupo de variáveis. Esta distribuição uniforme pode não ser a melhor forma de realizar *labeling* distribuído, porque pode causar desbalanceamento de carga entre os processadores já que um conjunto de variáveis pode causar mais falhas que outro devido às restrições relacionadas a cada variável. No entanto, nos nossos experimentos, esta divisão mostrou-se satisfatória. Pode-se melhorar os resultados utilizando-se um esquema mais sofisticado e inteligente de distribuição de variáveis para *labeling*, baseado no grafo de dependências.

O particionamento de *indexicals* entre processadores é realizado observando-se a dependência entre os *indexicals*. *Indexicals* são independentes quando não possuem variáveis em comum e são dependentes quando relacionam a mesma variável.

A análise da aplicação com relação à dependência entre os *indexicals* permite a realização de particionamentos que explorem melhor o paralelismo da aplicação. Existem várias formas de particionamento de *indexicals*. Neste trabalho, utilizamos o particionamento *Round-robin* e o particionamento por blocos. No particionamento *Round-robin* o primeiro *indexical* é atribuído ao primeiro processador, o segundo *indexical* é atribuído ao segundo processador, e assim por diante. O particionamento por blocos consiste em dividir o conjunto de *indexicals* em blocos de *indexicals* consecutivos que são atribuídos aos processadores. Desta forma, somente há comunicação entre as variáveis de ligação entre os blocos.

O projeto de um modelo de execução paralelo no contexto de CSP depende da política de escalonamento de restrições, isto é, da atribuição de restrições aos processadores. O escalonamento pode ser feito estaticamente ou dinamicamente. Uma política de escalonamento estático é apropriada para um sistema de memória distribuída porque reduz a comunicação. Neste tipo de escalonamento as restrições são distribuídas entre processadores em tempo de compilação. Na política de escalonamento dinâmico o particionamento dos *indexicals* é feito em tempo de execução. Este particionamento pode não ser apropriado para uma arquitetura de memória distribuída, porque a quantidade de trabalho para cada processador pode ser menor do que o custo de comunicação.

Na literatura encontramos várias formas de paralelização de algoritmos de consistência de arcos.

Uma das formas de paralelização de algoritmos de consistência de arcos existentes é utilizando *hardware* dedicado. Gu e Sosic [18] implementaram um algoritmo paralelo baseado no algoritmo AC-1 para uma arquitetura maciçamente paralela. O algoritmo paralelo foi implementado em *hardware* e alcançou bons resultados para aplicações de tempo real.

Cooper e Swain implementaram em um circuito digital um algoritmo de consistência de arcos maciçamente paralelo, baseado no AC-4. Implementaram, também, um algoritmo com paralelismo intermediário para uma máquina SIMD *(Single Instruction Stream, Multiple Data Stream)*. Foram utilizados métodos para explorar as características do domínio do problema para melhorar o desempenho dos algoritmos de consistência de arcos [10].

Kasif e Delcher apresentaram diversas técnicas para alcançar execução paralela de redes de restrições (grafo de restrições). Obtiveram como resultado que a com-

plexidade paralela das redes de restrições é criticamente dependente das propriedades intrínsecas da rede e que não influencia sua complexidade seqüencial [24].

Zhang e Mackworth desenvolveram dois algoritmos para resolver problemas de satisfação de restrições em paralelo sobre domínios finitos, derivados dos algoritmos de consistência de arcos [38]. Apresentaram, também, alguns resultados de complexidade paralela, generalizando resultados em [23].

Nguyen e Deville apresentaram um algoritmo de consistência de arcos distribuído, baseado no AC-4. É um algoritmo paralelo projetado para computadores de memória distribuída usando comunicação por passagem de mensagens [33].

Em [4] foram propostas versões distribuídas para AC-3 e AC-6 para (CSP) binários, baseado em escalonamento estático.

Fabiunke, em [13], apresentou uma estrutura distribuída paralela para resolver CSPs baseados em idéias conexionistas de processamento de informação distribuída. Nesta estrutura, cada variável de um dado problema é associada a um agente simples, continuamente aplicando uma regra de manipulação de variável, com o objetivo de minimizar o conflito local para satisfazer todas as restrições que envolvem esta variável. Todos os agentes trabalham simultaneamente formando um sistema dinâmico recorrente que deveria se auto-organizar depois de algumas iterações para uma solução do problema.

Hermenegildo, em [20], apresentou alguns dos problemas encontrados para detecção de paralelismo em tempo de compilação para programas de lógica e restrições. Foram analisados detecção de independência, custo de modelos e análises, gerenciamento de memória, técnicas para gerenciamento especulativo e computações irregulares através do controle da granulosidade de tarefas e alocação de tarefas dinâmicas, entre outras.

Luo *et al.*, em [28], apresentaram algumas heurísticas de métodos de busca utilizadas para melhorar as estratégias de resolução de problemas paralelos e distribuídos. Estas estratégias são denominadas baseada em variável, baseada em domínio e em função. Os resultados experimentais mostraram que os métodos de heurística básica para problemas de satisfação de restrições podem ser aplicados, em algoritmos de busca distribuídos, localmente quando os algoritmos são baseados em variável ou globalmente quando os algoritmos são baseados em domínio ou baseado em função.

Ferris e Mangasarian apresentaram em [14] uma técnica para paralelizar res-

trições em um programa matemático. As restrições do programa matemático são distribuídas entre processadores juntamente com um operador Lagrangeano apropriado para cada processador, que contém informações Lagrangeanas sobre as restrições tratadas pelos outros processadores. A informação do multiplicador Lagrangeano é, então, trocada entre os processadores.

Geyer *et al.*, em [16], concentraram-se na descrição de técnicas para explorar paralelismo-e e paralelismo-ou em sistemas de programação lógica paralelos. Foram mostrados aspectos importantes de alguns sistemas sobre arquiteturas de memória compartilhada, memória distribuída e memória compartilhada distribuída.

Gregory *et al.*, em [17], mostraram alguns experimentos com o sistema de programação lógica paralelo e/ou com suporte para solução de restrições sobre domínios finitos.

Andino *et al.*, em [2], implementaram um sistema paralelo baseado em memória distribuída. Foi implementado o algoritmo de consistência de arcos sobre domínios finitos para uma plataforma com múltiplos processadores, CRAY T3E [36], onde utilizaram 34 processadores. O sistema seqüencial, denominado CSOS (*Constrained Stochastic Optimization Solver*), foi utilizado como sistema base para gerar a versão paralela correspondente, o PCSOS (*Parallel Constraint Satisfaction and Optimization System*). O sistema PCSOS é um ambiente de programação de restrições para domínios finitos que integra técnicas de resolução de restrições e otimização evolucionária. Além de paralelismo em consistência de arcos, o sistema PCSOS pode também explorar paralelismo em procedimentos de busca. Seus experimentos mostraram *speedups* de até 18 para 34 processadores, no melhor caso.

Em nosso trabalho, nos baseamos no sistema PCSOS. A implementação do algoritmo de consistência de arcos foi realizada sobre o *software* DSM TreadMarks. A plataforma de *hardware* utilizada é um *cluster* de PC's. Ele difere dos trabalhos citados anteriormente, porque além de explorar melhor o paralelismo existente em cada uma das aplicações utilizando particionamento de variáveis e *indexicals*, e *labeling* paralelo, procuramos utilizar uma plataforma com modelo de programação de memória compartilhada mais amigável ao programador (*software* DSM) em um *hardware* acessível economicamente.

3.3 Resumo

Este capítulo apresentou as oportunidades de paralelização de algoritmos de consistência de arcos, formas de particionamento e alguns trabalhos relacionados. No capítulo seguinte, abordamos os conceitos básicos que envolvem um sistema DSM e descrevemos TreadMarks, o sistema *software* DSM utilizado em nossos experimentos.

Capítulo 4

Protocolos *Software DSM*: Conceitos Básicos

Os sistemas que possuem memória compartilhada distribuída (*Distributed Shared Memory - DSM*) são compostos por múltiplos nós de processamento conectados por uma rede de interconexão. Cada nó pode conter um ou mais processadores, memória privativa, memórias *cache* e dispositivos de entrada e saída. A organização de um sistema DSM está ilustrada na figura 4.1 [25].

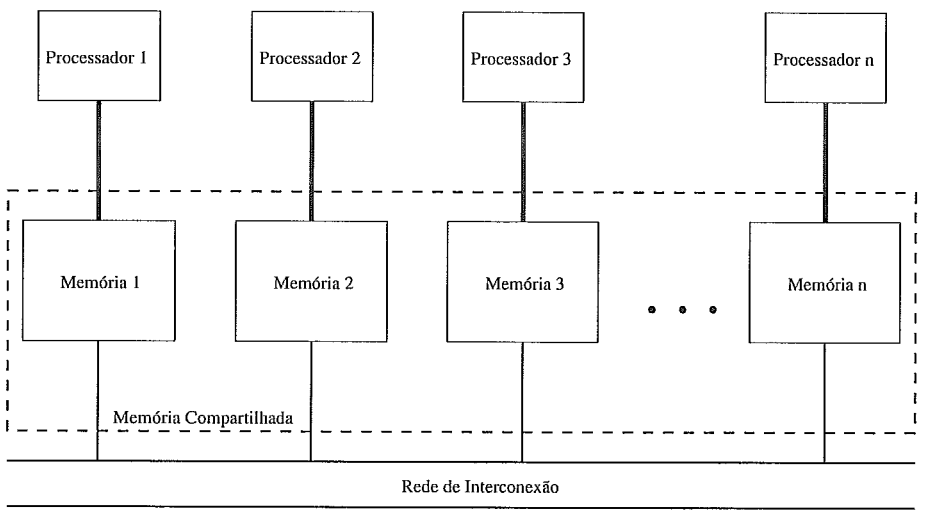

Figura 4.1: Memória Compartilhada Distribuída (DSM)

Nestes sistemas, a "memória compartilhada" é implementada por mecanismos de *hardware* e/ou *software*, que transformam os acessos às memórias remotas em mensagens pela rede.

Esses sistemas assumem que somente um processo é criado em cada processador. Nesse modelo, o compartilhamento dos dados é implementado com instruções de acesso às posições de memória. A coordenação e cooperação entre os processos é

realizada através da leitura e da escrita de variáveis compartilhadas e de ponteiros que referem-se aos endereços compartilhados. As escritas em dados compartilhados são realizadas de modo exclusivo e ordenadas com operações de sincronização. As barreiras e os *locks* são os dois tipos de implementação de operações de sincronização mais comuns.

As operações de barreira separam as fases de execução de uma aplicação. Um processo inicia uma nova fase somente depois que todos os processos tenham atingido uma barreira. Dessa forma, a operação de sincronização de barreira garante que todos os acessos de escrita e leitura feitos na nova fase estão ordenados em relação aos acessos realizados na fase anterior.

As operações de *locks* são utilizadas para ordenar os acessos dos processos envolvidos em seções críticas.

Neste capítulo apresentamos alguns conceitos básicos sobre sistemas DSM. São apresentados modelos de consistência de memória, protocolos de coerência e o suporte a múltiplos escritores. Além disso, apresentamos o sistema de memória compartilhada distribuída desenvolvido em *software* denominado TreadMarks [35].

4.1 Modelos de consistência

Os modelos de consistência de memória definem a ordem em que os acessos à memória compartilhada devem ocorrer. Estes modelos definem, também, a ordem em que estes acessos são observados pelos processadores. Os modelos de consistência podem ser caracterizados como fortes e relaxados segundo suas restrições às ordenações dos acessos.

O modelo de consistência seqüencial (ou consistência forte) define que um sistema multiprocessador é seqüencialmente consistente se e somente se o resultado de qualquer execução é o mesmo que seria obtido caso as operações de todos os processadores fossem executadas em alguma ordem seqüencial, e as operações de cada processador na ordem estabelecida pelo programa.

Os modelos de consistência relaxada, como utilizado em TreadMarks, procuram viabilizar o uso de algumas otimizações no acesso à memória compartilhada, e assim, com um modelo menos restritivo, obter melhora de desempenho do sistema.

Dentre os modelos de consistência relaxada podemos citar: *Processor Consistency* (PC), *Weak Consistency* (WC), *Release Consistency* (RC) e *Lazy Release*

Consistency (LRC) [8]. O modelo de consistência implementado em TreadMarks é o *Lazy Release Consistency.*

O modelo *Release Consistency* (RC) subdivide os acessos de sincronização em *acquires* e *releases.* Para garantir a coerência dos dados compartilhados a aplicação deve, portanto, utilizar sincronizações explícitas através de primitivas do sistema. As primitivas mais comuns fornecidas pelos sistemas DSM para sincronização são: *locks* (*acquires*) e *unlocks* (*releases*), para guardar seções críticas, e barreiras (um *release* seguido de um *acquire*), para sincronização global.

Um *acquire* e um *release* correspondem a uma operação de leitura e de escrita, respectivamente, a uma variável de sincronização. O *acquire* indica que o processador esta iniciando acessos a determinados dados compartilhados que pode depender de valores gerados por outro processador. Já o *release* indica que o processador está terminando os acessos aos dados compartilhados e pode ter gerado valores dos quais outro processador pode depender.

As condições para que um sistema esteja coerente segundo o modelo RC são:

- antes que uma operação de *release* tenha sido observada por qualquer processador, todos os acessos anteriores aos dados compartilhados devem ter sido observados por este processador;
- os acessos que seguem uma operação de *acquire* numa variável de sincronização devem esperar que o *acquire* tenha terminado e
- os acessos às variáveis de sincronização devem ser observados segundo os modelos SC ou PC.

O modelo RC relaxa as restrições de consistência em relação ao modelo WC, e ainda executa corretamente programas propriamente sincronizados, necessitando porém que os acessos de sincronização sejam mapeados em *acquires* e *releases.* Este fato faz com que a programação tenha que ser um pouco mais cuidadosa devido a inserção das operações de sincronização agora divididas em *acquires* e *releases.* Ele porém, apresenta uma redução sensível na latência de acesso à memória compartilhada, pois os pontos de espera são apenas nas operações de *release.*

O modelo *Lazy Release Consistency* (LRC) é uma versão mais relaxada do modelo RC, suportando o mesmo modelo de programação. LRC relaxa as condições de RC porque não necessita que numa operação de *release* os acessos anteriores sejam globalmente visíveis. Ele necessita apenas que os acessos anteriores ao *release*

sejam visíveis somente no processador que executar um *acquire*. Isto é, a propagação das modificações feitas por um processador P_i para outro processador P_j são atrasadas até o momento de um *acquire* subseqüente em P_j numa mesma variável de sincronização.

A comunicação é realizada somente entre os dois processadores envolvidos, *acquirer* e *releaser*.

Para determinar quais modificações um processador deve ter ciência no momento de um *acquire*, LRC estabelece uma ordem parcial dos acessos aos dados compartilhados, denominada *happened-before-1* (*hb*1). Esta ordem parcial *hb*1 é baseada na ordem seqüencial de execução de um processador e no encadeamento das operações *acquire* e *release* realizadas em processadores diferentes, mas sob a mesma variável de sincronização. Dois acessos à memória compartilhada a_1 e a_2 são ordenados por *hb*1, denotado por $a_1 \overset{hb1}{\rightarrow} a_2$, se:

- a_1 e a_2 são acessos do mesmo processador e a_1 ocorre antes de a_2;
- a_1 é um *release* no processador P_1, a_2 é um *acquire* na mesma variável de sincronização em P_2 e a_2 retorna o valor escrito por a_1.

A ordem *hb*1 é dada basicamente pelo fecho transitivo da ordem de execução de um processador com a ordem das operações de sincronização na mesma variável de sincronização. Se $a_1 \overset{hb1}{\rightarrow} a_2$ e $a_2 \overset{hb1}{\rightarrow} a_3$ então $a_1 \overset{hb1}{\rightarrow} a_3$.

No protocolo LRC pode-se ter ainda uma redução de comunicação se atrasarmos a troca de dados entre os processadores até o momento em que o processador *acquirer* tocar o dado compartilhado.

4.2 Protocolos de coerência

Para melhorar o desempenho, os sistemas DSM permitem a replicação dos dados compartilhados nas memórias privativas dos processadores, na tentativa de reduzir o número de acessos remotos. No entanto, tal replicação exige a utilização de um protocolo para manter a memória coerente. A coerência deve ser mantida segundo um modelo de consistência de memória. Para manipular a replicação dos dados compartilhados existem dois tipos de protocolo de coerência, os baseados em invalidações e os baseados em atualizações. Estes protocolos visam propagar para os demais processadores as modificações das posições de memória compartilhada feitas localmente a um processador.

No mecanismo de invalidação somente é enviada, aos demais processadores, a informação de que os dados foram modificados. Ao receber uma mensagem deste tipo os processadores remotos invalidam seus dados correspondentes. Uma tentativa de acesso a estes dados gera uma falha de acesso. Neste tempo, é buscada uma versão atual do dado modificado remotamente. Neste mecanismo, as falhas de acesso não são evitadas, porém somente são transmitidos dados que realmente serão necessários.

No protocolo de atualização os próprios dados modificados localmente por um processador são enviados aos demais processadores. Dessa forma, se um outro processador realizar um acesso ao dado compartilhado, este já estará disponível, não ocasionando falha de acesso. Este mecanismo minimiza as falhas de acesso, porém pode vir a provocar uma carga desnecessária de comunicação se novas modificações forem feitas antes que seja realizado um acesso ao dado.

4.3 Suporte a múltiplos escritores

Para minimizar o problema de falso compartilhamento (dois processadores utilizam dados diferentes que estão localizados na mesma unidade de coerência) quando a unidade de coerência do sistema é grande, como em muitos sistemas *software DSM* que utilizam como unidade a página, são utilizados protocolos de múltiplos escritores. Nestes protocolos são permitidas escritas concorrentes à mesma unidade de coerência, retardando a visão das modificações para um ponto de sincronização posterior entre os processadores. É importante ressaltar que as escritas concorrentes devem ser feitas a diferentes posições de memória. Em caso contrário, estaria caracterizada uma condição de corrida. Esta condição não é permitida em programas propriamente sincronizados. Para este tipo de programa, escritas feitas na mesma posição de memória devem ser ordenadas através de operações de sincronização.

O principal problema de se permitir a existência de múltiplos escritores na mesma unidade de coerência está em combinar todas as escritas realizadas para montar uma versão coerente dela.

Considerando a página como unidade de coerência, na implementação de protocolos com múltiplos escritores, podem ser utilizados os mecanismos de *twinning* e *diffing*. Inicialmente, todas as páginas compartilhadas são protegidas contra escrita. Quando um processador tenta modificar um dado compartilhado localmente, é gerada uma falha de acesso à página. Ao detectar a falha, o protocolo *software DSM*

faz uma cópia da página (*twin*) e a libera para escrita. Assim, as escritas locais ocorrem livremente na página e o *twin* mantém a versão original. No momento da propagação das modificações locais, o protocolo faz uma comparação entre a cópia atual da página e o *twin*, e cria uma estrutura, denominada *diff*, contendo as modificações realizadas. Dessa forma, para se obter uma página atualizada é necessário aplicar os diversos *diffs* criados pelos diferentes processadores que compartilham a mesma página e a modificaram. A aplicação dos *diffs* deve preservar a ordenação imposta pelas operações de sincronização, para evitar que uma escrita recente seja sobreposta por uma outra escrita mais antiga na mesma posição de memória.

4.4 TreadMarks

TreadMarks (TMK) [35] é um sistema de memória compartilhada distribuída desenvolvido em *software*. Sua implementação é em nível de usuário e utiliza as bibliotecas padrão do Unix para a criação de processos remotos, para a comunicação entre processos e para o gerenciamento de memória. A sua unidade de coerência é a página, sendo a coerência dos dados mantida com uso do protocolo de invalidações. Este protocolo é implementado através da propagação de notificações de escrita (*write notices*) em operações de sincronização. A *interface* de programação oferecida por TMK provê dois tipos de primitivas de sincronização: *lock/unlock* e barreira (*Tmk_lock_acquire*/ *Tmk_lock_release* e *Tmk_barrier*).

Para reduzir os efeitos de falso compartilhamento causado pela unidade de coerência grande (páginas de 4K*Bytes*), TreadMarks fornece suporte a múltiplos escritores através dos mecanismos de *twining* e *diffing*. Inicialmente todas as páginas são protegidas contra escrita. Quando há uma tentativa de modificação numa página, é gerada uma falha de acesso. O TMK intercepta esta falha, faz uma cópia da página (*twin*) e a libera para escrita. Quando for necessária a propagação das modificações feitas localmente a uma página, TMK faz uma comparação entre o *twin* gerado e a versão modificada, e cria um *diff* contendo todas as modificações locais feitas à página. A cada *diff* existe um *write notice* associado identificando o intervalo e o processador onde o *diff* deve ser criado. Os intervalos são segmentos de tempo na execução de um processador e iniciados cada vez que uma operação de sincronização é executada.

Com o objetivo de reduzir o tráfego de dados pela rede, principalmente o número

de mensagens, este sistema adota o modelo de consistência *Lazy Release Consistency* (LRC). O envio e a aplicação dos *diffs* são atrasados até o primeiro acesso à página realizado após uma operação de *acquire*. TMK utiliza a ordem parcial *happens-before-1* para ordenar os intervalos de diferentes processadores. Através de $hb1$ é possível determinar quais intervalos de outros processadores precedem o intervalo corrente de um processador P. As modificações realizadas nos dados compartilhados são associadas aos intervalos em que elas ocorreram e, assim, numa operação de *acquire* ocorrida num intervalo i, o processador deve ser notificado sobre todas as modificações associadas a intervalos anteriores a i segundo a ordem parcial $hb1$.

A notificação sobre as modificações ocorridas nos dados compartilhados é realizada através de notificações de escrita. Estas notificações indicam que uma determinada página foi modificada. Cada intervalo contém uma notificação de escrita para cada página modificada no segmento de tempo correspondente. Quando o processador P executa uma operação *acquire* num intervalo i, ele deve receber as notificações de escrita correspondentes a todos os intervalos anteriores a i segundo $hb1$. Ao receber uma notificação de escrita, o processador invalida a página correspondente. Os *diffs* relativos às modificações em questão só são recebidos na próxima falha de acesso de cada página.

Quando ocorre uma falha num acesso à página é necessário buscar um conjunto de *diffs* para torná-la válida, havendo então a comunicação com um ou mais processadores. Os *diffs* recebidos são aplicados à página, respeitando a ordenação parcial, para se obter uma versão coerente.

TreadMarks fornece suporte a múltiplos escritores visando diminuir os efeitos do falso compartilhamento e da fragmentação. Entretanto, para aplicações onde não há falso compartilhamento o *overhead* associado ao processamento dos *diffs* pode degradar o sistema. Esse *overhead* inclui o custo da criação de *twins* e de *diffs* e da aplicação dos *diffs*. Além disso, a busca dos *diffs* pode envolver vários processadores e gerar contenção na rede de interconexão. Uma das situações mais críticas seria aquela onde todos os processadores simultaneamente requisitam *diffs* [8].

4.5 Resumo

Este capítulo apresentou os conceitos básicos relacionados ao desenvolvimento de sistemas DSM. Além disso, descreveu as principais características de TreadMarks, que

é o sistema *software* DSM utilizado em nossos experimentos. No próximo capítulo descrevemos o sistema CSOS e as versões paralelas propostas neste trabalho.

Capítulo 5

O Sistema CSOS e as Versões Paralelas

Neste capítulo descrevemos o sistema CSOS (*Constrained Stochastic Optimization Solver*) [3] e sua versão paralela PCSOS, no qual baseamos nossos experimentos. São descritas, também, as alterações realizadas no PCSOS para se avaliar o desempenho deste *solver* usando um *cluster* de PC's, em um ambiente *software* DSM.

Os autores do CSOS apresentaram um *solver* que implementa o algoritmo de consistência de arcos usando o esquema de *indexicals*. O *solver* pode ser usado na modelagem de problemas de otimização discreta e verificação, tais como escalonamento, planejamento, alocação de horários entre outros. A parte principal do sistema é um instrumento de propagação que executa *indexicals* através da consistência de arcos. O sistema pode resolver problemas para valores atribuídos a variáveis e também por otimização, usando um algoritmo *branch and bound* ou um algoritmo genético [1].

Os *indexicals* a serem executados são gerados com domínio finito estendido de um programa *pseudo-Prolog*. Estes mantêm a sintaxe próxima à sintaxe da biblioteca de domínios finitos do Sicstus Prolog [7].

5.1 Descrição do CSOS

CSOS é a denominação do sistema seqüencial, proposto em [3]. O sistema PCSOS, também foi proposto em [3] e é baseado no CSOS. Possui implementações paralelas do algoritmo de consistência de arcos e de um algoritmo genético para resolver problemas de otimização de restrições. A entrada de dados é realizada da mesma forma tanto para a versão seqüencial quanto para as versões paralelas.

A entrada de dados é formada por um conjunto de *indexicals* armazenados em um arquivo. O CSP a ser resolvido deve ser programado em Prolog, usando restrições aritméticas. Em seguida, o programa Prolog é interpretado gerando um arquivo texto contendo os *indexicals*. Este arquivo texto é transformado em um formato binário, gerando o arquivo de entrada do sistema. O interpretador do programa Prolog e o compilador para o formato binário estão escritos em Prolog.

O sistema oferece um ambiente interativo para a entrada de dados, assim como para a inspeção de *indexicals* e domínios das variáveis durante a execução dos *indexicals* compilados. Além disso, existe ainda a fase de *labeling* e otimização da solução.

A fase de *labeling* consiste em verificar a existência de uma solução para as restrições. Durante esta fase um valor inicial é atribuído a uma variável, em seguida as restrições são executadas para eliminar valores dos domínios. Este processo é repetido sucessivamente até que cada domínio de cada variável contenha um único elemento ou nenhuma solução seja encontrada. Portanto, cada variável possui um único valor que satisfaça todas as restrições, formando, assim, uma solução do CSP. A fase de *labeling* foi implementada utilizando o princípio *most-constrained*, isto é, a próxima variável a ser rotulada é aquela com o menor domínio corrente. Na literatura este princípio é referenciado, algumas vezes, como *first-fail principle*. Os valores do domínio são avaliados do menor para o maior, ou seja, não utiliza nenhuma heurística específica. No apêndice podemos visualizar a tela do CSOS com a fase de entrada de dados e chamada do *labeling*.

Após encontrar a primeira solução é realizado um procedimento de *backtracking*, ou seja, os valores dos domínios das variáveis são restaurados, de forma que a última variável rotulada receberá o próximo valor ainda não usado para tentar uma nova solução.

Vários comandos estão disponíveis, além daqueles para compilar *indexicals*. São comandos para examinar estado de domínios, variáveis, fila de variáveis e *indexicals*. Há, também, comandos para enumeração e otimização, comandos para acrescentar *indexicals* no *prompt* do sistema, comandos para zerar ou mostrar estatísticas e uso de recursos, *trace*, além do comando para sair do sistema, história do sistema, executar uma operação do *shell* do sistema operacional, ajuda e *reset*. A análise de valores de variáveis durante a execução é possível devido às estruturas de dados usadas na implementação. As principais estruturas de dados usadas na implementação

do CSOS são descritas a seguir.

5.2 Estruturas de dados

As principais estruturas de dados utilizadas são listas encadeadas de variáveis e *indexicals*. A lista de *indexicals* é gerada logo após a entrada de dados no sistema. A lista de variáveis é gerada, também, à medida que os dados são lidos e os *indexicals* são armazenados, pois os primeiros *indexicals* são constituídos pelas variáveis e seus respectivos domínios, ou seja, restrições unárias que apenas delimitam o intervalo de valores das variáveis.

Estas estruturas são usadas tanto na implementação seqüencial quanto na paralela e, portanto, existem campos nas estruturas de dados relativos ao controle de paralelismo.

A estrutura de dados de uma variável armazena informações como nome da variável, domínio da variável, *indexicals* dependentes desta variável, entre outras.

O domínio de uma variável tem uma estrutura composta pelos valores mínimo e máximo do domínio e um vetor de *bits*.

A necessidade de representação do domínio utilizando o vetor de *bits* ocorre devido ao fato de alguma restrição eliminar, durante a consistência de arcos, valores intermediários de um intervalo. Desta forma, somente os campos mínimo e o máximo não são suficientes para se representar o domínio.

A estrutura de dados de um *indexical* é composta pelo *indexical* propriamente dito, lista de variáveis afetadas por este *indexical*, lista de *indexicals* dependentes de outros *indexicals*, informações referentes ao paralelismo, entre outras.

A figura 5.1 ilustra como estas estruturas estão implementadas.

Na implementação seqüencial, variáveis, domínios e *indexicals* estão alocados numa *heap*.

Na implementação paralela, as variáveis são alocadas de forma compartilhada, assim como a estrutura dos domínios. As estruturas dos *indexicals* são locais a cada processador.

A implementação apresenta, também, uma *trail*, onde são guardados os valores antigos dos domínios antes que a alteração no domínio de uma variável seja feita. Isto é necessário para se recuperar o valor anterior do domínio quando é realizada a volta a um passo anterior na execução.

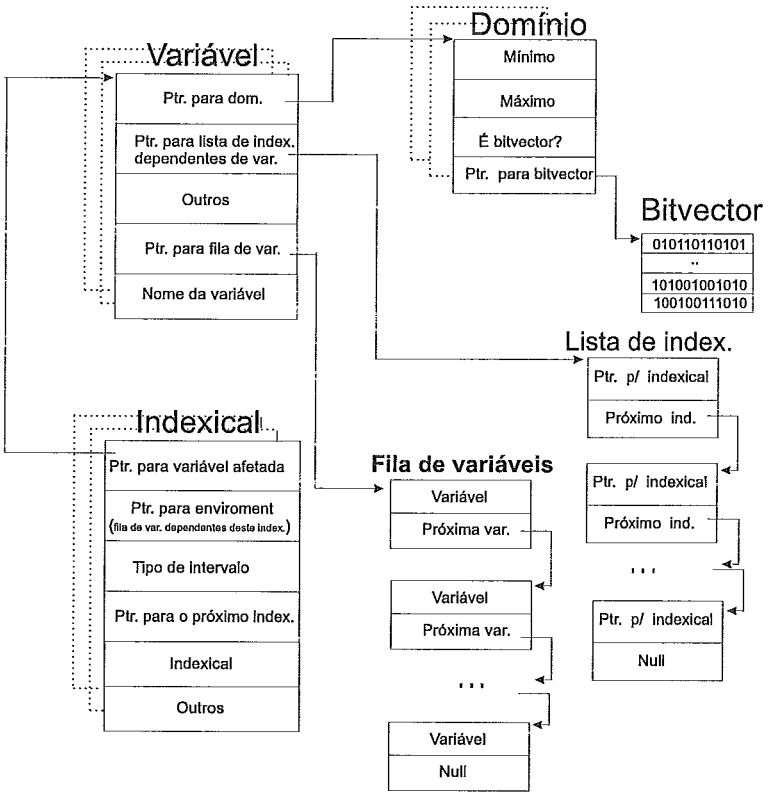

Figura 5.1: Estrutura de dados

As próximas seções descrevem como foram implementados os algoritmos de consistência de arcos seqüencial e paralelo no sistema.

5.3 Algoritmo seqüencial

O algoritmo de consistência de arcos seqüencial tem como argumentos de entrada um problema de satisfação de restrições na forma de variáveis, domínios e restrições. A partir daí a consistência de arcos deve ser realizada. O algoritmo é executado recursivamente até alcançar consistência de arcos ou inconsistência.

A implementação seqüencial do algoritmo AC-5 baseado no esquema de *indexicals*, feita por [3], possui dois passos principais: o de consistência de nós e o de consistência de arcos. No primeiro passo a consistência de nós é realizada para cada *indexical*. Nesta fase, somente *indexicals* com uma única variável são tratados. A variável associada ao *indexical* que está sendo executado é empilhada numa estrutura de dados (pilha) para utilização posterior na fase de consistência de arcos. A finalidade de empilhar as variáveis encontra-se na necessidade de se utilizar na fase de consistência de arcos a lista de *indexicals* suspensos, armazenada na estrutura

das variáveis. No segundo passo cada variável é desempilhada. Em seguida, é realizada a consistência de arcos para cada *indexical* que dependa daquela variável, possibilitando uma poda maior no domínio das variáveis.

A figura 5.2 mostra o algoritmo AC-5 implementado por [3] e utilizado em nosso trabalho.

```
function Arc-Consistent-CSP(<VarSet,DomSet,ConstrSet>):Store
   begin
1      Queue_Init(PropagationQueue,...);
2      while NOT Empty(PropagationQueue) do
3          QueuePop(PropagationQueue, v_i);
4          for each indexical I_j which depends on v_i do
5              if NOT Arc_Consistent (I_j,Store,PropagationQueue) then
6                  return FAILURE;
7              end-if;
8          end-for;
9          end-while
10     return Store;
   end
```

Figura 5.2: Algoritmo AC-5, apresentado por [3]

Os argumentos de entrada da função Arc-Consistent-CSP, *VarSet*, *DomSet* e *ConstrSet* são, respectivamente, o conjunto de variáveis, o conjunto de domínios das variáveis e o conjunto de restrições, isto é, o conjunto de *indexicals*.

A função Arc-Consistent-CSP realiza a fase de consistência de nós dos *indexicals* e gera uma fila de propagação (PropagationQueue) composta por variáveis, utilizando o procedimento *Queue_Init*, como pode ser visto na figura 5.2. O *loop* principal (linhas 2 a 9) da figura 5.2 itera até a fila de propagação ficar vazia ou ser detectada inconsistência.

Neste *loop*, inicialmente, o procedimento *Queue_Pop* retira uma variável de cada vez da fila de propagação. O *loop* que se encontra entre as linhas 4 e 8 da figura 5.2 é executado por todos os *indexicals* que se encontram na lista de *indexicals* suspensos da variável retirada da fila de propagação. A função *Arc_Consistent*, mostrada na figura 5.3 é responsável pela realização da consistência de arcos.

Esta função tem como entrada o *indexical* a ser executado, o *Store* e a fila de propagação. O *indexical* é representado por 'v_i *in* *E()*', onde v_i é a variável da qual o *indexical* é dependente. *E()* corresponde à expressão referente ao domínio da

variável v_i. Nesta função é chamada a função Update(), mostrada na figura 5.3, que realiza a interseção do novo domínio com o *Store*. Se a interseção for vazia significa que ocorreu uma falha. Caso contrário, a variável tratada por aquele *indexical* é colocada na fila de propagação de variáveis.

```
function Arc-Consistent('v_i in E()', Var Store, Var PropagationQueue):Boolean
    begin
        NewDomain := Eval(E(), Store);
        return(Update(NewDomain, v_i, Store, PropagationQueue) <> EMPTY);
    end;

function Update(NewDomain,v_i, Var Store, Var PropagationQueue):RESULT
    begin
        NewDomain := NewDomain ∩ Store[v_i];
        if Empty(NewDomain) then return EMPTY; end-if;
        if (NewDomain ⊂ Store[v_i]) then
          Store[v_i] := NewDomain;
          Queue_Push(v_i, PropagationQueue);
          return PRUNED;
        end-if;
        return NOT_PRUNED;
    end;
```

Figura 5.3: *Store* e atualização da fila de propagação [3]

A função Arc-Consistent-CSP retorna ou um *Store*, onde o domínio de cada variável foi podado, alcançando consistência de arcos, ou uma falha, se for detectada inconsistência (domínio de uma variável foi podado para um domínio vazio).

Após a consistência de arcos ser realizada é iniciada a fase de *labeling* do CSOS. Nesta fase, sobre a lista de variáveis é aplicada uma busca com *backtracking* para encontrar a solução para aquelas variáveis. Um valor é atribuído a uma das variáveis e a consistência de arcos é realizada novamente para as outras variáveis. Outro valor é escolhido para outra variável e a consistência de arcos é repetida, e assim sucessivamente. Quando se obtém uma configuração inconsistente, é realizado *backtrack* do último valor escolhido para a última variável e um novo valor é escolhido. Este processo continua até se obter uma solução ou até que o conjunto de restrições seja insatisfatível. Para se gerar todas as soluções o *backtrack* deve ser realizado para todos os valores possíveis que atendam todas as restrições.

A partir da implementação seqüencial, foram desenvolvidas versões paralelas do

algoritmo de consistência de arcos AC-5, que descrevemos a seguir. O algoritmo descrito na próxima seção corresponde à versão implementada por Andino *et al.* [3].

5.4 Algoritmo paralelo original

O algoritmo descrito nesta seção corresponde à versão implementada por Andino *et al.* [3]. Uma versão paralela do algoritmo de consistência de arcos pode ser obtida se distribuirmos *indexicals* entre os processadores. O escalonamento de *indexicals* nestes processadores pode ser estático ou dinâmico. No escalonamento estático as restrições são distribuídas entre os processadores em tempo de compilação, enquanto no escalonamento dinâmico as restrições são dinamicamente distribuídas entre os processadores. Isto é, o escalonamento é realizado em tempo de execução.

5.4.1 Algoritmo paralelo com particionamento estático

No escalonamento estático, o mapeamento de restrições nos processadores é gerado antes da execução da consistência de arcos. Cada processador possui sua própria informação de quais restrições executar. A única necessidade de coordenação vem da detecção de terminação, tarefa que pode ser realizada por qualquer um dos processadores.

O conjunto de restrições é particionado em n subconjuntos disjuntos, ou seja, cada restrição só pertence a um conjunto. Este particionamento induz a uma distribuição do conjunto de variáveis em n não necessariamente subconjuntos disjuntos. Este fato ocorre porque uma mesma variável pode estar presente em mais de uma restrição.

Cada processador recebe seu subconjunto de restrições e realiza a consistência de arcos seqüencial sobre o seu grupo de *indexicals*, atualizando a cópia local das variáveis afetadas. Como pode ocorrer que algumas variáveis sejam alocadas em diversos processadores, cada processador deve comunicar a modificação do domínio da variável alterada por ele aos demais processadores que possuam *indexicals* dependentes desta variável. Toda vez que um processador receber uma notificação de modificação do domínio de outro processador é realizada a interseção da cópia do domínio local com a cópia recebida, provocando novas alterações. A comunicação entre os processadores é também necessária em três casos: para detectar a termi-

nação do algoritmo, alcançar um ponto fixo global ou detectar inconsistência [2].

No algoritmo paralelo estático, inicialmente todo *indexical* atribuído ao processador é executado e cada variável associada ao *indexical* é empilhada. Em seguida, cada variável é desempilhada e para cada *indexical* dependente do domínio daquela variável é feita a consistência de arcos. Quando um ponto fixo local é alcançado, o processador notifica o processador mestre de seu estado. Isto é realizado por cada um dos processadores até alcançar o ponto fixo global, ou seja, todos os *indexicals* foram executados mas não ocorreu nenhuma inconsistência, ou até que inconsistência seja detectada. A inconsistência pode ocorrer por um domínio vazio resultante da execução de um *indexical* local, um domínio vazio resultante da interseção de um domínio local de uma variável com o domínio recebido de outro processador ou através da inconsistência detectada por outro processador.

O algoritmo termina quando todo processador alcança o ponto fixo local e não há mais mensagens pendentes na rede de comunicação. O processador escolhido para detectar terminação é o único responsável por este procedimento, mas ele faz propagação local como qualquer outro processador. Quando o processador escolhido para detectar terminação recebe as notificações de terminação local de todos os outros processadores, este notifica-os da terminação global.

5.4.2 Algoritmo paralelo com escalonamento dinâmico

O escalonamento dinâmico consiste em distribuir entre os processadores o conjunto de *indexicals* prontos. Ou seja, distribuir *indexicals* que tenham o domínio de suas variáveis modificado depois de sua última execução. Este modelo requer mecanismos para assegurar que a execução de uma restrição relacionada a uma variável seja feita depois que a mudança no domínio da variável tenha sido atualizada no *store* do processador que está executando a restrição. As maneiras para se alcançar a ordem de execução são: introduzir pontos de sincronização durante a execução (controle distribuído) ou incluir um processador mestre (controle centralizado).

Quando o controle é distribuído, considera-se a divisão da execução em ciclos de execução sincronizados. Um ciclo de execução consiste na geração do conjunto de *indexicals* prontos, da seleção e execução distribuída do conjunto de prontos e um ponto de sincronização. Todo processador deve gerar filas de propagação de restrições idênticas. Os pontos de sincronização entre ciclos de execução são introduzidos para gerar filas de propagação idênticas e o *store* deve ser replicado em

todo processador.

No algoritmo de consistência de arcos deste modelo, inicialmente todo *indexical* é empilhado. Em seguida, um *indexical* de cada vez é desempilhado e é feita a consistência de arcos até não haver mais *indexicals* ou ser detectada a inconsistência. A execução de um ciclo consiste nos seguintes passos:

- cada restrição na pilha é executada por um processador particular até a pilha ficar vazia;
- variáveis modificadas são empilhadas e as mudanças nos domínios das variáveis são transmitidas para todos os outros processadores;
- mudanças nos domínios de variáveis recebidas de processadores remotos são atualizadas e empilhadas;
- quando a pilha de propagação estiver vazia e depois da sincronização que assegura que todos os processadores tenham o mesmo valor nos domínios das variáveis, uma nova pilha de propagação é gerada com todas as restrições dependentes das variáveis empilhadas.

Quando o controle é centralizado, a ordem na execução de restrições é alcançada introduzindo um processador mestre que decide que processador executa cada restrição no conjunto de *indexicals* prontos. Assim, um modelo possível consiste em ter um processador mestre que executa a propagação de restrições de forma seqüencial, mantendo um único *store*. O processador mestre empilha restrições da pilha de propagação e requisita sua execução aos escravos. As restrições são enviadas com os domínios daquelas variáveis requeridas para a execução. Os domínios resultantes da execução são enviados de volta para o processador mestre que atualiza o *store* e a fila de propagação [2].

Andino *et al.* implementaram seus algoritmos paralelos num multiprocessador, CRAY T3E [36], com 34 processadores DEC Alpha com 400-MHz, com 128 MBytes de memória por processador e utilizando o sistema operacional UNICOS (UNIX-*like*). O CRAY T3E é um sistema com memória logicamente compartilhada, onde cada processador possui sua memória local, com uma rede de interconexão rápida, com topologia *3D torus*. Esta arquitetura possui um conjunto de registradores externos (*E-registers*) usados como fonte ou destino para toda comunicação remota. Através dos registradores externos, o T3E disponibiliza um conjunto de

operações atômicas de acesso à memória e facilita a troca de mensagens. É possível declarar um número ilimitado de variáveis de sincronização. Além disso, existem primitivas como `shmem_short_fadd()` e `shmem_short_finc()`, por exemplo, que são usadas, respectivamente, para adicionar um determinado valor à variável de um determinado processador e para incrementar uma variável de um determinado processador.

Esta arquitetura permite que se utilize os dois modelos básicos de programação paralela: o modelo de memória compartilhada e o modelo de memória distribuída. Andino *et al.* implementaram os seus algoritmos utilizando o modelo de programação distribuída, usando como canal de comunicação o suporte à memória compartilhada. Desta forma, conseguiram minimizar os *overheads* de sincronização e comunicação. Eles implementaram o modelo de escalonamento dinâmico e o estático. O modelo de escalonamento dinâmico utiliza um controle centralizado, onde o processador mestre decide que processador é responsável por executar um determinado conjunto de *indexicals*. Esta implementação gera uma seqüencialização da execução, não produzindo bons resultados. A implementação que utiliza o modelo de escalonamento estático, faz a distribuição estática dos *indexicals*. Para este modelo é que foram obtidos os resultados apresentados por Andino *et al.*.

Em nosso trabalho, nos concentramos no particionamento estático de *indexicals* pelos processadores, visto que a execução de cada *indexical* tem baixa granulosidade e o escalonamento dinâmico pode deteriorar o desempenho na plataforma paralela que utilizamos. Nas seções seguintes apresentamos as duas versões do algoritmo paralelo modificado para memória compartilhada distribuída, utilizando o *software DSM* TreadMarks.

5.5 Resumo

Este capítulo apresentou o sistema CSOS implementado por [3], com suas estruturas de dados e algoritmos. Além disso, descreveu também, a versão paralela, PCSOS, proposta por eles.

Capítulo 6

Algoritmo paralelo modificado

Baseado no sistema PCSOS, descrito no capítulo anterior, desenvolvemos duas versões do algoritmo paralelo, denominadas PCSOS_TMK e PCSOS_SHM_TMK. Na primeira implementação, as estruturas de dados do algoritmo desenvolvido por Andino *et al.* [3] são adaptadas para a utilização de memória compartilhada distribuída, utilizando o *software* DSM TreadMarks. Nesta implementação mantivemos o modelo de programação distribuída utilizada originalmente. Na segunda implementação, simplificamos o modelo de programação, utilizando somente o paradigma de memória compartilhada. Neste capítulo, também apresentamos as aplicações utilizadas, os particionamentos e *labelings* utilizados em cada uma delas.

6.1 Implementação PCSOS_TMK

A implementação PCSOS_TMK do algoritmo paralelo desenvolvido por [3], utilizando TMK, mantém a estrutura do algoritmo utilizado e as estruturas de dados da versão PCSOS. Porém, esta versão apresenta algumas particularidades na alocação das estruturas de dados. A alocação da memória compartilhada é realizada somente pelo processador mestre, previamente escolhido. Os demais processadores permanecem em uma barreira esperando que a região compartilhada seja alocada. A partir desta barreira todos os processadores passam a ter acesso à região de memória compartilhada. Todos os acessos à memória tanto de escrita quanto de leitura são protegidos por *locks*, pois precisamos garantir que cada processador tenha acesso exclusivo ao dado compartilhado e obtenha o valor mais atualizado do dado. A busca de um valor mais atual de uma variável é realizada no *loop* principal da consistência de arcos.

Para a realização da fase de *backtracking* é necessário recuperar o domínio pre-

viamente armazenado da variável que causou a falha. Uma vez que este domínio foi recuperado, fazemos uma interseção com o domínio compartilhado para obter um conjunto de novos valores para a realização do procedimento de *labeling*. Se o resultado desta interseção for vazio continuamos fazendo *backtracking*. Durante este processo precisamos de uma sincronização para obter o domínio mais atualizado da variável.

As estruturas de dados compartilhadas são: o vetor de variáveis do problema, o domínio das variáveis, as estruturas auxiliares utilizadas para transmitir atualização do domínio das variáveis da aplicação aos demais processadores (*store*) e outras variáveis para detectar a terminação do algoritmo.

Na versão original, cada variável pode ser alterada na memória de outro processador através da biblioteca SHMEM do UNICOS, específica para operações atômicas em memória compartilhada. Na nossa versão, este procedimento foi representado por acessos a estruturas de dados, indexadas por processador, para variáveis e domínios, alocadas na memória compartilhada. Todos os acessos de leitura e escrita nestas estruturas compartilhadas são protegidos.

A estrutura de dados que armazena as variáveis do problema é uma matriz indexada pelo número do processador e número da variável que está sendo armazenada. Portanto, cada processador possui um vetor com todas as variáveis do problema.

O domínio também é armazenado na memória compartilhada, pois os processadores devem obter valores atualizados por outros processadores em algum momento, e isto somente é possível utilizando variáveis compartilhadas, que são utilizadas para comunicação entre os processadores.

As estruturas de dados das variáveis, domínios e o *store* têm como padrão de compartilhamento um único escritor e múltiplos leitores.

Na figura 6.1 ilustramos a estrutura das variáveis, que chamamos de $AV[i][j]$, onde i representa o número do processador e j é o número da variável representada. Observe que cada processador P_i tem armazenado todas as variáveis do problema, de $V(0)$ a $V(k)$. Cada variável tem um ponteiro para a estrutura do seu domínio, que nesta figura está representado para a variável $V(1)$. Considerando um particionamento em que cada processador é responsável por manipular duas variáveis, observe que somente as variáveis que estão sombreadas são alteradas diretamente por um determinado processador. As demais variáveis são protegidas por *locks*, o que pode gerar troca de mensagens entre os processadores se estes precisarem acessar valo-

res atualizados destas variáveis. Assim, as variáveis $V(0)$ e $V(1)$ são manipuladas pelo processador P_0; as variáveis $V(2)$ e $V(3)$ são manipuladas pelo processador P_1 e assim sucessivamente para todos os processadores. Se houver algum *indexical* que relacione, por exemplo, as variáveis $V(0)$ e $V(2)$ haverá uma dependência entre os processadores P_0 e P_1. Toda vez que ocorrer alguma alteração no domínio da variável $V(0)$, por exemplo, o processador P_0 deve comunicar esta modificação ao processador P_1, para que este mantenha o valor de sua variável $V(0)$ atualizada.

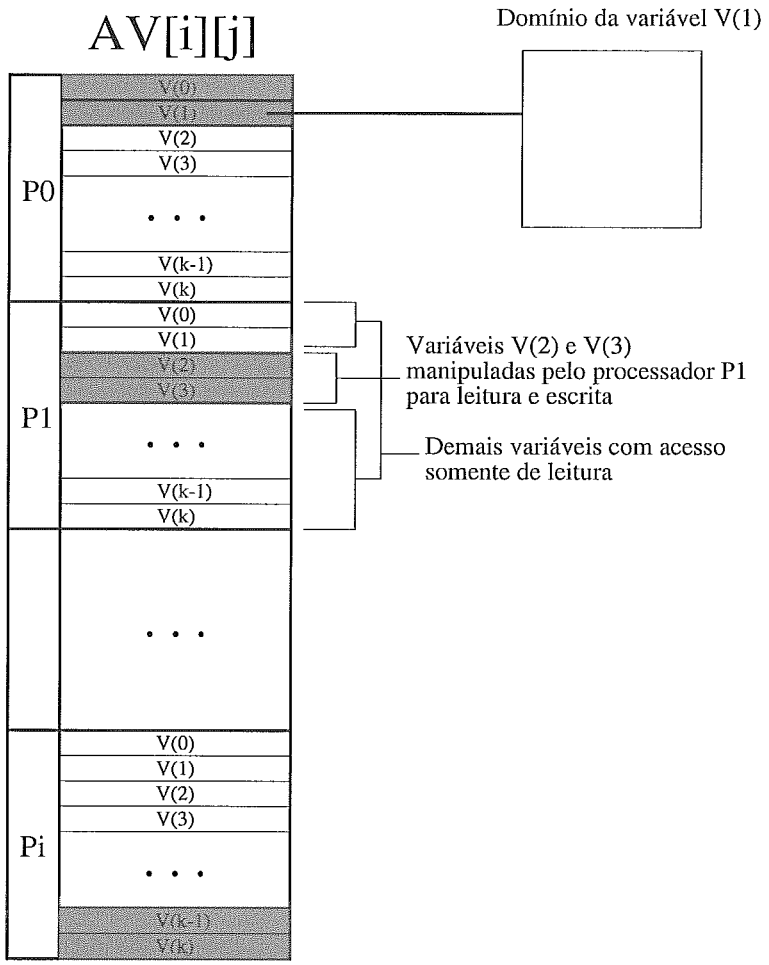

Figura 6.1: Estrutura de dados da variável e do domínio para o PCSOS_TMK

A atualização dos valores do domínio que foi alterado é realizada no *store* que está implementado utilizando-se duas filas alocadas de forma compartilhada. Estas filas são indexadas pelo número do processador, e associam a variável ao seu domínio modificado. Uma fila armazena a variável modificada e a outra armazena o domínio alterado.

Na figura 6.2 ilustramos o funcionamento do *store* para dois processadores, P_0 e P_1. As filas armazenam o endereço e o domínio das variáveis alteradas. Cada

processador possui um par destas filas. Assim, as filas do processador P_0 são escritas pelo processador P_1. Ou seja, cada vez que P_1 altera o domínio de cada uma de suas variáveis, a variável alterada é colocada na fila de variáveis de P_0 e o novo domínio é colocado na fila de domínios. Como estas duas filas são escritas ao mesmo tempo e estão protegidas pelo mesmo *lock*, há uma correspondência entre a variável alterada e o novo domínio. Durante a execução do algoritmo de consistência de arcos, o processador P_0, por exemplo, consulta estas filas para verificar se P_1 alterou o valor de alguma de suas variáveis. Se houver variável alterada na fila, P_0 faz a interseção de seu domínio com o novo domínio, obtido da fila. Assim, os dois processadores passam a ter suas variáveis atualizadas.

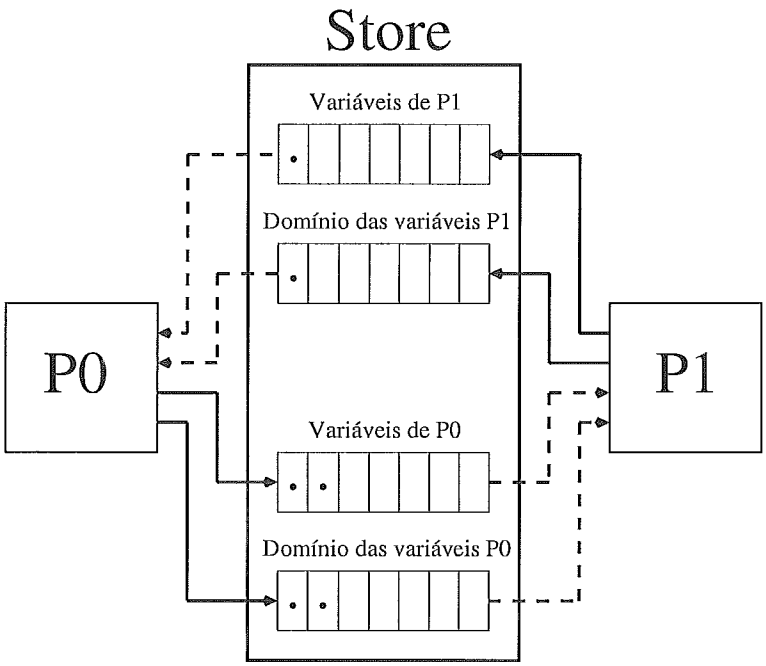

Figura 6.2: Estrutura de dados do *store* para dois processadores

A alocação dos *indexicals* é realizada por cada um dos processadores de forma privada. Ou seja, todos os processadores lêem o arquivo de entrada e, de acordo com o particionamento definido para a aplicação, os processadores armazenam somente aqueles *indexicals* que devem lhe pertencer.

O particionamento dos *indexicals* é diferenciado para cada aplicação, pois buscamos explorar o paralelismo inerente de cada aplicação. Quanto maior a dependência entre os *indexicals* maior será a comunicação entre os processadores. Os detalhes sobre os tipos de particionamento realizados para as aplicações utilizadas em nossos experimentos são apresentados na seção 6.3.

Na fase inicial da execução do programa, todos os processadores têm acesso à estrutura de dados das variáveis do problema já inicializada previamente pelo mestre. Após a leitura do arquivo de entrada e armazenamento de sua lista de *indexicals*, cada processador somente altera as variáveis relacionadas no seu grupo de *indexicals*.

Na fase seguinte é realizada a consistência de nós, onde todos os *indexicals* são executados e é gerada uma pilha de variáveis. Em seguida, na fase de consistência de arcos, as variáveis são, uma a uma, desempilhadas e todos os *indexicals* são novamente executados.

Ao executar um *indexical*, a alteração do domínio de variáveis que são relacionadas por *indexicals* deve ser transmitida aos outros processadores. Esta comunicação da alteração do valor do domínio é realizada através do *store*. Cada vez que o domínio é alterado o novo valor e a variável são armazenados no *store* pelo processador que fez a alteração. Antes de executar o *indexical* seguinte, é realizada uma consulta ao *store* para verificar se alguma variável teve seu domínio alterado pelos outros processadores. Caso haja alguma alteração, o processador lê o novo valor do domínio para a variável alterada e faz uma interseção com o valor do domínio localmente armazenado para aquela variável. Desta forma, os processadores passam a armazenar um valor único para o domínio das variáveis.

O procedimento de *labeling* é realizado em paralelo por todos os processadores. Cada processador acessa a sua lista de variáveis a serem rotuladas. No caso de aplicações em que há comunicação entre todas as variáveis esta lista é composta por todas as variáveis.

Na fase de *labeling*, cada processador escolhe uma variável na sua lista de variáveis, que possua o menor domínio e escolhe, também, o menor valor deste domínio. Em seguida, é realizada consistência de arcos para esta variável. Todos os *indexicals* relacionados a esta variável são executados e os domínios das variáveis podem ser podados. Se for gerado um domínio vazio, é realizado um procedimento que implementa o *backtracking*, onde os valores do domínio daquela variável que havia sido escolhida são restaurados. Uma nova variável e um novo valor são escolhidos e novamente é realizada consistência de arcos. Este procedimento realiza-se até conseguir uma atribuição de valores para todas as variáveis do domínio ou até concluir que não há solução.

6.2 Implementação PCSOS_SHM_TMK

Na implementação PCSOS_SHM_TMK buscamos melhorar a utilização do modelo de programação baseado em memória compartilhada. Realizamos esta implementação para compararmos o seu desempenho com a versão PCSOS_TMK, que é baseada no modelo de programação de troca de mensagens. A alocação da memória compartilhada é realizada da mesma forma que na implementação PCSOS_TMK, ou seja, é realizada somente pelo processador mestre, previamente escolhido. Os demais processadores permanecem em uma barreira esperando que a região compartilhada seja alocada.

A estrutura de dados das variáveis está implementada de forma diferente da implementação PCSOS_TMK. Ela é implementada como um vetor de todas as variáveis alocado de forma compartilhada. As variáveis são protegidas por *locks* distintos. Assim, cada vez que um processador alterar o domínio de uma variável, esta modificação será transmitida aos outros processadores quando estes acessarem a variável. Portanto, qualquer alteração no domínio das variáveis será realizada diretamente na variável compartilhada referente ao domínio, não havendo a utilização da filas de comunicação como na implementação PCSOS_TMK, ou seja, o *store* agora é o próprio conjunto de variáveis compartilhadas. Não há mais a divisão local de *stores* e atualização de domínios através de filas de comunicação. A figura 6.3 mostra que a variável $V(4)$, por exemplo, pode ser lida e escrita pelos processadores P_0 e P_1. O vetor de variáveis é único e encontra-se em região compartilhada. Cada variável é protegida por um *lock* para leitura e escrita.

A leitura do arquivo de entrada é realizada somente pelo processador mestre e o particionamento dos *indexicals* é realizado da mesma forma que na versão PCSOS_TMK. A alocação dos *indexicals* é realizada somente pelo mestre e de forma compartilhada. Os demais processadores esperam numa barreira até que o mestre termine a alocação. Como ocorre na versão PCSOS_TMK, os *indexicals* são distribuídos entre os processadores de acordo com uma máscara, que é determinada pelo mestre no momento da alocação.

Na fase de consistência de nós e de consistência de arcos, cada processador acessa a lista de todos os *indexicals*, que é única, mas executa somente o seu grupo de *indexicals* de acordo com a máscara determinada pelo mestre. A necessidade da lista de *indexicals* ser compartilhada advém do fato da estrutura de dados das

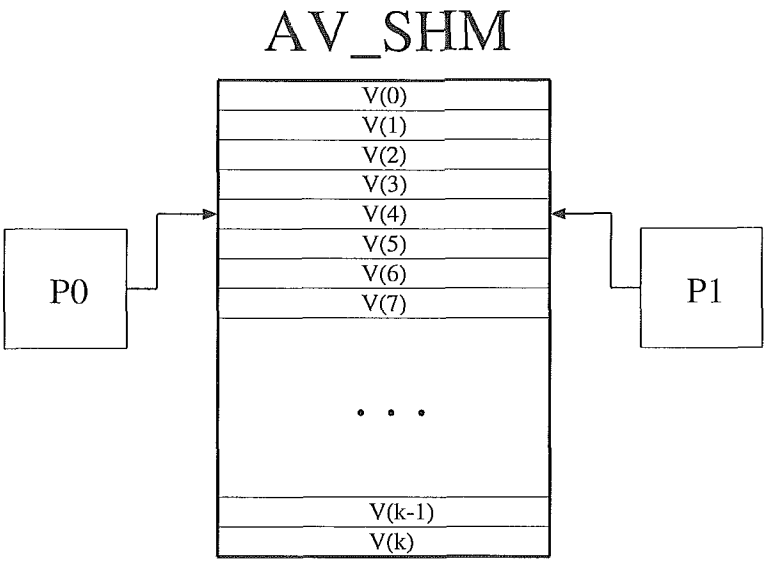

Figura 6.3: Estrutura de dados das variáveis para o PCSOS_SHM_TMK

variáveis possuir um campo onde é armazenada a lista de *indexicals* dependentes daquela variável. Durante a fase de consistência de arcos, cada uma das variáveis é desempilhada e esta lista de *indexicals* dependentes é percorrida. Portanto, se o endereço do *indexical* fosse privativo ao processador que executa aquele *indexical*, os demais não poderiam acessar esta região de memória, o que ocasionaria erro de execução.

As alterações no domínio de uma variável após a execução de um *indexical* são diretamente realizadas na região compartilhada da memória que foi alocada para a estrutura de variáveis e que está protegida por um *lock* associado àquela variável. Desta forma, toda vez que for feita uma leitura por outro processador este acessa o valor mais atual do domínio daquela variável. Este valor mais atual do domínio da variável é buscado cada vez que uma poda é realizada, ao contrário da versão PCSOS_TMK que busca um valor mais atual no *loop* principal da consistência de arcos. Para determinadas aplicações a poda é feita de forma mais freqüente, o que pode fazer com que o número de entradas na seção crítica seja maior para a versão PCSOS_SHM_TMK.

Na fase de *backtracking*, ao contrário da versão PCSOS_TMK, não é necessária a sincronização, neste momento, para a obtenção do domínio da variável. A sincronização já foi realizada na fase anterior quando o processador detectou a inconsistência dos domínios.

A fase de *labeling* também ocorre da mesma forma que a versão PCSOS_TMK, ou

seja, cada processador possui sua lista de variáveis para serem rotuladas, realizando *labeling* paralelo.

Nesta implementação houve uma simplificação do código original, pois muitas das rotinas usadas para consistência de dados não são necessárias. Por exemplo, não é necessário utilizar filas de comunicação para comunicar aos demais processadores que houve alteração no domínio de uma variável. A alteração do domínio é realizada na própria estrutura compartilhada do domínio, que é acessada por todos os processadores. Além disso, a quantidade de memória utilizada é menor, pois não há replicação da lista de *indexicals* e existe apenas uma lista de variáveis para todos os processadores.

A seção 6.3 apresenta as aplicações utilizadas e o particionamento feito para cada uma delas.

6.3 Conjunto de aplicações

Utilizamos como *benchmarks* quatro aplicações para testar nosso modelo de paralelização: *Arithmetic*, *Parametrizable Binary Constraint Satisfaction Problem* (PBCSP), *N-Queens* e *Sudoku* [3]. Estas são as mesmas estudadas por Andino *et al.* em seu trabalho no PCSOS. Escolhemos estas aplicações para que pudéssemos comparar nossos resultados com aqueles obtidos para PCSOS. Estas aplicações abrangem diferentes tipos de interdependência entre os *indexicals*.

Analisamos o padrão de interdependência das restrições em cada uma das aplicações para aproveitarmos melhor o paralelismo inerente e fazermos uma distribuição dos *indexicals* mais adequada.

6.3.1 *Arithmetic*

Arithmetic é um *benchmark* sintético composto por 16 blocos de relações aritméticas, $\{B_1, \ldots, B_{16}\}$. Cada bloco contém 15 equações e inequações envolvendo 6 variáveis. Os blocos B_i, B_{i+1} estão relacionados por uma equação adicional entre um par de variáveis, uma de B_i e outra de B_{i+1}. Os coeficientes são aleatoriamente gerados. O objetivo é encontrar um vetor solução inteiro. Esta aplicação possui 126 variáveis, 254 restrições e 1.468 *indexicals*.

Este tipo de aplicação é bastante utilizada em problemas de otimização baseados em sistemas de equações lineares muito grandes, onde se usa decomposição em blocos

para encontrar uma solução para o problema.

A característica desta aplicação nos permite fazer um particionamento dos *indexicals* de forma a minimizar bastante a comunicação entre os processadores. Distribuímos os blocos de *indexicals* consecutivos entre os processadores de acordo com o número de variáveis. Portanto, só existe comunicação entre as variáveis das fronteiras, isto é, existem *indexicals* que relacionam certas variáveis que estão alocadas em processadores diferentes. A figura 6.4 ilustra como foi realizada esta distribuição entre N processadores, sendo que cada um recebeu uma fatia K de variáveis. Observando a fronteira entre $P0$ e $P1$, podemos observar que existem *indexicals* em $P0$ e em $P1$ que relacionam as variáveis $V(K)$ e $V(K+1)$.

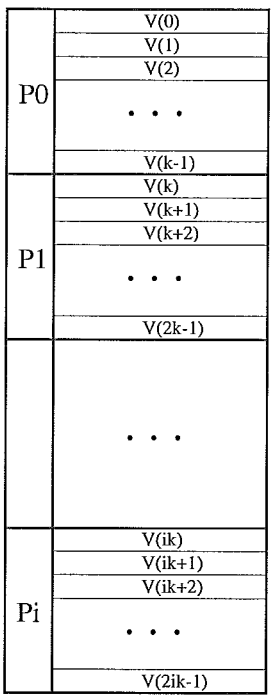

Figura 6.4: Particionamento dos *indexicals* para a aplicação *Arithmetic*

Na fase de *labeling* da execução seqüencial as variáveis a serem rotuladas correspondem ao conjunto de todas as variáveis da aplicação. Na execução paralela, o conjunto de variáveis a serem rotuladas por cada processador corresponde às variáveis previamente determinadas no particionamento dos *indexicals*.

Este particionamento dos *indexicals* visa reduzir a quantidade de comunicação entre os processadores. Porém, não leva em consideração o desbalanceamento de carga que esta distribuição pode causar, pois podem haver variáveis que possuem mais dependência do que outras. Desta forma, a fatia atribuída a um processador pode gerar a execução de mais *indexicals* do que a fatia dos outros processadores.

6.3.2 *Parametrizable Binary Constraint Satisfaction Problem (PBCSP)*

PBCSP é um problema sintético e permite estudar o desempenho de algoritmos de consistência de arcos variando os parâmetros do problema. Exemplos deste problema são gerados aleatoriamente dados os 4 parâmetros: número de variáveis, o tamanho dos domínios iniciais, densidade e *tightness*. A densidade corresponde ao número de restrições existentes para cada variável. O *tightness* define o grau de conexão entre os nós do grafo. Todas as restrições são binárias, isto é, elas envolvem somente duas variáveis. A densidade e o *tightness* são definidos da seguinte forma:

$$densidade = \frac{nc}{nv - 1} \qquad tightness = 1 - \frac{np}{ds^2}$$

onde *nv* é o número de variáveis, *nc* é o número de restrições envolvendo uma variável (é o mesmo para todas as variáveis), *np* é o número de pares que satisfazem a restrição e *ds* é o tamanho dos domínios iniciais (igual para todas as variáveis).

O grafo de restrições da aplicação PBCSP é fortemente conexo. Isto significa que há muita comunicação entre as variáveis.

A figura 6.5 ilustra um grafo de restrições para a aplicação PBCSP, cujos parâmetros de entrada são: 5 variáveis, densidade = 0,75, tamanho do domínio = 3, *tightness* = 0,85, número de soluções = 2 e semente = 1.

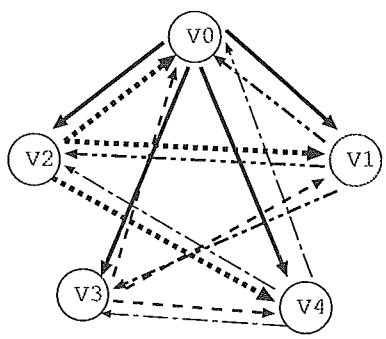

Figura 6.5: Representação das restrições para o problema PBCSP com 5 variáveis

Nesta figura, cada nó representa uma variável da aplicação e cada arco representa uma restrição entre duas variáveis.

O particionamento utilizado foi o *Round-robin*. Variamos as massas de dados desta aplicação para 100 e 200 variáveis.

O *labeling* utilizado na execução desta aplicação foi seqüencial, ou seja, o conjunto de variáveis executadas é o mesmo para todos os processadores.

6.3.3 *N-Queens*

N-Queens é um problema que consiste em colocar N rainhas em um tabuleiro de xadrez $N \times N$ tal que as rainhas não se ataquem. As rainhas se atacam se estiverem na mesma linha, na mesma coluna, na mesma diagonal principal ou na mesma diagonal secundária. Cada rainha deve ser colocada numa linha do tabuleiro. O problema consiste em selecionar uma coluna para cada rainha de forma que elas não se ataquem. Foi utilizado um exemplo com $N = 111$, que possui um razoável tempo de execução seqüencial.

A figura 6.6 ilustra um exemplo de um tabuleiro 4×4. A rainha representada pode atacar as demais na mesma linha, coluna ou diagonal.

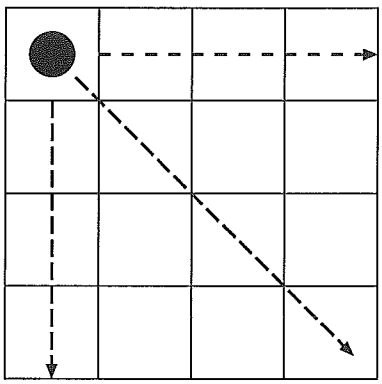

Figura 6.6: Restrições para uma rainha do problema *4-Queens*

A figura 6.7 apresenta uma solução para o problema *4-Queens*.

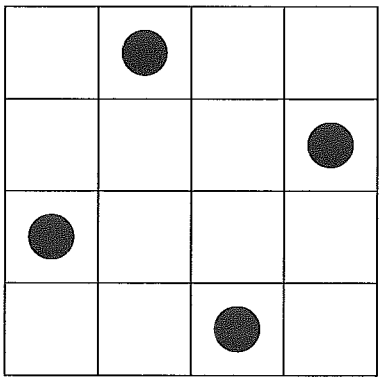

Figura 6.7: Solução para o problema *4-Queens*

Esta aplicação apresenta um grafo de restrições totalmente conectado. Isto é, todas as variáveis estão relacionadas às demais. Desta forma, os *indexicals* são bastantes dependentes. Esta característica gera muita comunicação entre os proces-

sadores na execução paralela da aplicação. Utilizamos o particionamento *Round-robin* para dividir os *indexicals* entre os processadores.

Na fase de *labeling*, tanto na execução seqüencial quanto na execução paralela, o conjunto de variáveis a serem rotuladas corresponde ao conjunto total de variáveis da aplicação para todos os processadores.

6.3.4 *Sudoku*

Sudoku é um problema japonês crypto-aritmético. Dado um *grid* de quadrados de dimensão 25×25, onde 317 deles estão preenchidos com um número entre 1 e 25, *Sudoku* preenche o restante dos quadrados tal que cada linha e coluna é uma permutação de números de 1 a 25. Entretanto, cada um dos 25 quadrados de dimensão 5×5 começando nas colunas (linhas) 1, 6, 11, 16, 21 deve também ser uma permutação dos números de 1 a 25. Esta aplicação possui 308 variáveis, 13.942 restrições e 27.884 *indexicals*.

Na figura 6.8 visualizamos o tabuleiro 25×25, onde a parte colorida corresponde às variáveis que já possuem valores atribuídos. O relacionamento existente entre as variáveis representadas nesta matriz ocorre nas linhas e nas colunas. Ou seja, todas as variáveis de uma mesma linha comunicam entre si e todas as variáveis de uma mesma coluna também apresentam comunicação. Além disso, todas as variáveis de uma linha comunicam com todas as variáveis das outras linhas na mesma coluna.

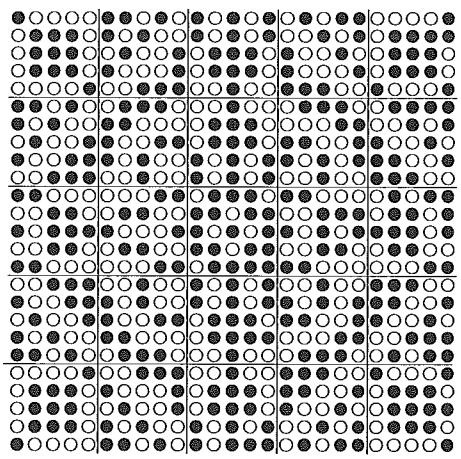

Figura 6.8: Representação da matriz de variáveis do *Sudoku*

O grafo de restrições desta aplicação é um grafo fortemente conectado. Portanto, é uma aplicação que apresenta elevada dependência entre os *indexicals*. Ao dividir os *indexicals* entre os processadores, vários *indexicals* relacionados a uma mesma

variável ficam em processadores distintos. Ao serem executados, as alterações dos domínios das variáveis devem ser transmitidas ao demais processadores, gerando muita comunicação entre os processadores.

Na execução paralela, particionamos os *indexicals* entre os processadores utilizando o método *Round-robin.*

Na fase de *labeling* o conjunto de variáveis utilizado para ser rotulado é o mesmo para todos os processadores e corresponde ao conjunto total de variáveis da aplicação.

6.4 Resumo

Neste capítulo apresentamos duas implementações do algoritmo paralelo, PCSOS_TMK e PCSOS_SHM_TMK, destacando suas diferenças. Foram apresentadas, também, as características de cada uma das aplicações utilizadas para testar estes sistemas, além do particionamento e *labeling* utilizados.

Capítulo 7

Resultados

Iniciamos este capítulo apresentando a plataforma computacional e as características principais das aplicações utilizadas em nossos experimentos. Em seguida, para cada aplicação analisamos os resultados obtidos em tempo de execução.

7.1 Plataforma utilizada

Nossos experimentos foram realizados utilizando uma plataforma de *hardware* que consiste de um *cluster* de 8 nós, onde cada nó é um dual-pentium-III com velocidade de processamento de 650 MHZ. Apesar dos nós conterem dois processadores, estamos utilizando somente um processador em cada nó. Além disso, cada nó possui 512 MBytes de memória. Todos os nós rodam o sistema operacional *Red Hat Linux release 6.2*, com *Kernel 2.2.14-5.0*. A rede de interconexão entre os nós é uma rede *Fast-Ethernet* com velocidade de 100 Mbps. Avaliamos o potencial de paralelismo dos sistemas PCSOS_TMK e PCSOS_SHM_TMK sobre esta plataforma de *hardware* e *software*. A linguagem empregada na implementação destes sistemas é a linguagem C e o compilador utilizado é o gcc. Os sistemas PCSOS_TMK e PCSOS_SHM_TMK têm aproximadamente 7.000 linhas de código. As aplicações estão implementadas em liguagem C ou Prolog. Em todas as aplicações contamos apenas o tempo para encontrar a primeira solução do problema e descartamos o tempo de leitura do arquivo de entrada e a inicialização dos dados compartilhados.

7.2 Características das aplicações

Para avaliarmos os sistemas propostos utilizamos quatro aplicações com características distintas em nossos experimentos: *Arithmetic*, *Sudoku*, *N-Queens* e

Parametrizable Binary Constraint Satisfaction Problem (PBCSP). Estas aplicações são as mesmas utilizadas por Andino *et al.* [3] para avaliar o sistema PCSOS. Para as quatro aplicações, a sincronização entre os processadores é realizada através do uso de barreiras e de *locks*. Na tabela 7.1 mostramos para cada aplicação o número de variáveis, o número de restrições existentes e o número de *indexicals* provenientes das restrições.

Características	Aplicações				
	Arithmetic	PBCSP_1	PBCSP_2	*Queens*	*Sudoku*
Variáveis	126	100	200	111	308
Restrições	254	3.713	7.463	6.105	13.942
Indexicals	1.468	7.426	14.925	12.210	27.884

Tabela 7.1: Características das aplicações

Todas as aplicações consomem mais tempo na computação das restrições, ou seja, executando a propagação e a consistência, do que executando o procedimento de *labeling*.

7.3 Análise dos Resultados

Para realizarmos a análise de desempenho, executamos cada uma das quatro aplicações dez vezes para cada experimento variando o número de nós. O desvio padrão foi menor do que 5%. Utilizamos a média dos tempos obtidos com estas execuções. As aplicações *Arithmetic*, PBCSP_1, PBCSP_2 e *Queens* foram executadas para 1, 2, 4 e 8 nós e a aplicação *Sudoku* foi executada para 1, 2 e 4 nós. Todos os tempos foram medidos em microssegundos e convertidos para segundos. Portanto, os números das tabelas e gráficos de tempo são apresentados em segundos.

Para cada aplicação mostramos os tempos de execução, *speedup*, total de *acquires* realizados para acessar os *locks*, total de mensagens, total de barreiras executadas, total de *indexicals* executados por cada processador e total de falhas ocorridas em cada processador. O número total de *acquires*, mensagens e barreiras, assim como o tempo total de *acquires* e barreiras foram obtidos utilizando as estatísticas fornecidas pelo próprio protocolo de TreadMarks. O total de *acquires* corresponde ao número total de chamadas à rotina *Tmk_lock_acquire* e o total de barreiras corresponde ao número total de chamadas à rotina *Tmk_barrier*. O total de mensagens corresponde

à quantidade de vezes que houve troca de *bytes*, referentes aos dados da aplicação ou ao protocolo de TMK. O número total de falhas corresponde ao número de vezes que foi realizado o procedimento de *backtracking* na aplicação.

Todas as aplicações foram executadas na versão PCSOS_TMK. Apenas a aplicação *Arithmetic* foi executada na versão PCSOS_SHM_TMK porque é a que obteve melhor desempenho. Nas próximas seções analisamos as aplicações na ordem decrescente de desempenho.

7.3.1 *Arithmetic*

Na tabela 7.2 mostramos o tempo de execução da aplicação *Arithmetic*. Na figura 7.1 observamos o gráfico do tempo de execução relativo a esta tabela.

Processadores	Tempo de Execução (seg.)
1	4,74
2	0,39
4	0,37
8	0,47

Tabela 7.2: Tempos de execução da aplicação *Arithmetic*

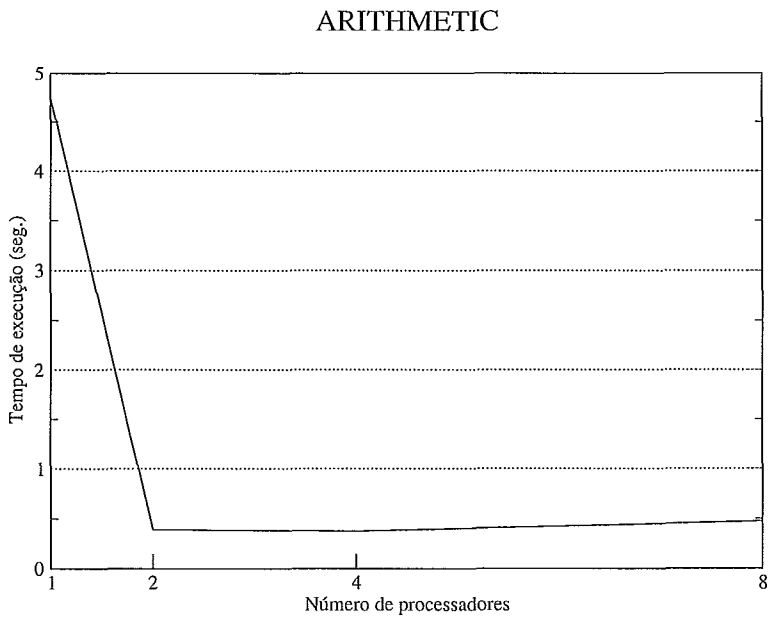

Figura 7.1: Tempo de execução da aplicação *Arithmetic*

Esta aplicação apresenta tempos de execução paralela significativamente inferiores ao tempo obtido na versão seqüencial. Os ganhos superlineares conseguidos nesta

aplicação em relação à execução com 1 processador são devidos ao particionamento dos *indexicals* em blocos, que se mostrou eficaz para esta aplicação, e à distribuição do processamento da fase de *labeling* entre os processadores.

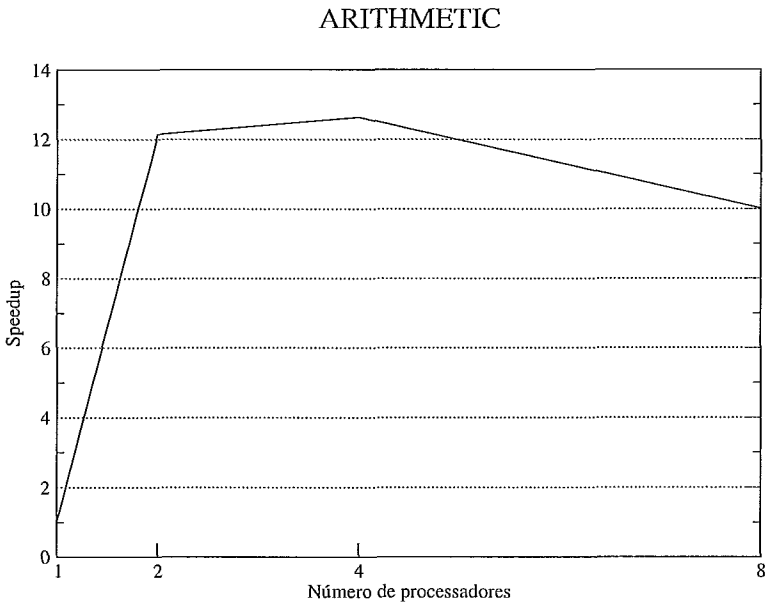

Figura 7.2: Gráfico de *speedup* do *Arithmetic*

Para 2 processadores o tempo de execução diminuiu de 4,74 seg. para 0,39 seg., proporcionando um *speedup* superlinear de 12,14, como pode ser observado no gráfico de *speedup* da figura 7.2. Para 4 processadores o tempo de execução diminuiu para 0,37 seg., em relação a um processador alcançando um *speedup* de 12,63. O tempo de execução com 8 processadores aumenta em relação a 2 e 4 processadores atingindo 0,47 seg., o que proporciona um *speedup* de 10,01, em relação à execução para um processador.

Como a dependência entre os *indexicals* é baixa, a troca de mensagens somente é realizada para as variáveis de ligação entre os blocos. Portanto, seria de se esperar que esta aplicação tivesse desempenhos muito melhores de 2 para 4 processadores e de 4 para 8 processadores. No entanto, isto não ocorreu pelos seguintes fatores:

1. a quantidade de mensagens trocadas cresce entre os processadores à medida que aumentamos a quantidade de processadores devido ao aumento do número de variáveis de ligação;

2. o *store* é protegido por apenas um *lock*;

3. o protocolo de coerência de TMK é baseado em invalidação, o que causa um

overhead excessivo para que um processador obtenha o valor atualizado da variável compartilhada;

4. a quantidade de dados desta aplicação é insuficiente para manter ocupados os vários processadores;

5. desbalanceamento de carga entre os processadores devido ao tipo de procedimento de *labeling* adotado.

Para ilustrar melhor os *overheads* provenientes do protocolo de TMK, apresentamos o número total de *acquires*, mensagens e barreiras, na tabela 7.3.

Estatísticas	Processadores			
	1	2	4	8
Acquires	47.073	39.386	128.250	538.802
Mensagens	0	11.657	80.059	206.205
Barreiras	8	8	8	8

Tabela 7.3: Total de *acquires*, mensagens e barreiras para *Arithmetic*

Para um processador, o protocolo de TreadMarks contabiliza todas as vezes que foram executados os trechos do código referentes a *locks*. Por isso, o número de *acquires* é grande. Entretanto, não há mensagens nem comunicação, pois há somente um processador executando.

As variáveis de ligação entre os blocos de *indexicals* são protegidas por *locks* distintos, mas as filas utilizadas para transmitir a atualização dos domínios aos demais processadores utilizam um único *lock*. A distribuição dos blocos de *indexicals* entre os processadores faz com que as variáveis de ligação estejam presentes em *indexicals* alocados por processadores diferentes. Assim, o total de acessos a *locks* cresce com o aumento do número de processadores, pois são estas variáveis de ligação que são compartilhadas. Conseqüentemente, o número de mensagens cresce com o aumento do número de processadores.

Nesta aplicação o procedimento de *labeling* é realizado por cada processador em paralelo e sobre o seu conjunto de variáveis. Este conjunto é composto por um subconjunto das variáveis da aplicação. O particionamento é realizado de tal forma que atribui a mesma quantidade de variáveis para cada processador.

Como explicado na seção 6.3.1, este particionamento não é o melhor, pois pode gerar desbalanceamento de carga entre os processadores.

Da mesma forma que são particionadas as variáveis, são particionados os *indexicals*. Assim, o seu particionamento é em blocos e formado por um subconjunto dos *indexicals* da aplicação. Além disso, cada processador executa, também, somente sobre o seu conjunto de *indexicals*. A tabela 7.4 apresenta o número de *indexicals* executados por cada processador ao executar esta aplicação.

Proc#	1	2	4	8
0	1.953.660	89.255	23.363	2.392
1	-	44.071	25.457	20.917
2	-	-	4.607	15.001
3	-	-	13.665	10.437
4	-	-	-	3.237
5	-	-	-	2.069
6	-	-	-	19.566
7	-	-	-	3.543
Tot. de ind.	1.953.660	133.326	62.892	77.162

Tabela 7.4: Total de *indexicals* executados por cada processador para *Arithmetic*

Podemos observar que o número de *indexicals* executados por um único processador é bastante maior que a soma dos *indexicals* executados por cada processador, uma ordem de grandeza maior do que para 2 processadores e duas ordens de grandeza maior para 4 e 8 processadores. Este fato ocorre porque na execução em paralelo, o procedimento de *labeling* é realizado pelos processadores sobre seu subconjunto de variáveis, gerando menos falhas que na execução seqüencial, onde o procedimento de *labeling* é realizado sobre o conjunto total das variáveis.

Na versão seqüencial não é possível obter o efeito do procedimento de *labeling* de forma distribuída como na versão paralela, pois os subconjuntos de variáveis possuem dependência, embora pequena. Desta forma, não é possível obter consistência local e global, pois existem variáveis que estão presentes em dois subconjuntos e que só podem ter um valor fixado depois de terem um valor atribuído por outros subconjuntos de variáveis. A consistência local é alcançada se todos os valores das variáveis de um subconjunto estão compatíveis com os valores das variáveis dependentes. A consistência global é alcançada se todos os conjuntos de variáveis estiverem com seus valores consistentes.

Uma falha ocorre quando se detecta alguma inconsistência local ou global. Após a falha, é realizado o procedimento de *backtracking* na tentativa de se atribuir um

novo valor à variável sobre a qual está sendo realizado o *labeling*. Desta forma, todos os *indexicals* relacionados àquela variável são novamente executados. Assim, quanto maior o número de falhas ocorridas em um processador, maior é o número de *indexicals* executados. Este fato pode ser observado nas tabelas 7.4 e 7.5, que mostram o total de *indexicals* e de falhas por processador, respectivamente.

Proc#	1	2	4	8
0	12.438	619	192	9
1	-	306	167	188
2	-	-	23	62
3	-	-	113	105
4	-	-	-	15
5	-	-	-	10
6	-	-	-	122
7	-	-	-	38
Falhas	12.438	925	495	549

Tabela 7.5: Total de falhas por processador para *Arithmetic*

Na tabela 7.5 é mostrado o número total de falhas na execução da aplicação *Arithmetic*. Analogamente ao total de *indexicals* executados, observamos que o total de falhas ocorridas na execução seqüencial é bem maior que o total de falhas das execuções em paralelo.

Podemos observar que o número de *indexicals* executados é proporcional ao número de falhas. Portanto, o processador que tem o maior número de falhas na fase de *labeling* tende a ter o maior número de *indexicals* executados. Este fato gera um desbalanceamento de carga entre os processadores. Quando ocorre uma falha é realizado o procedimento de *backtracking*. O processador precisa, então re-executar alguns *indexicals*, acarretando um aumento do número de *indexicals* executados proporcional ao aumento do número de falhas. Além disso, observa-se que dependendo do número de processadores, a distribuição dos *indexicals* e a distribuição do *labeling* fazem com que a consistência global seja alcançada em tempos diferentes. Isto faz com que o número total de *indexicals* executado tenha variações não proporcionais ao aumento do número de processadores. Na tabela 7.4 observamos que para 4 processadores o total de *indexicals* executados foi menor do que para 2 e para 8 processadores. Esta irregularidade pode causar uma queda de desempenho.

O que se nota nitidamente nesta aplicação é que a fase de *labeling* paralelo

melhorou o desempenho deste procedimento realizado seqüencialmente de forma bastante significativa, ou seja, é possível aumentar a entrada de dados e resolver instâncias muito maiores deste problema em uma máquina paralela.

Acreditamos que o tamanho do problema nesta aplicação não seja suficiente para a quantidade de processadores que utilizamos. Comparando somente os *speedups* em paralelo, obtivemos *speedup* baixo de 2 para 4 processadores e um *slowdown* para 8 processadores. No caso de 8 processadores, o número de mensagens é grande. Quase 2,6 vezes maior do que o número de mensagens para 4 processadores, como pode ser visto na tabela 7.3. Portanto, o tempo gasto em comunicação supera o tempo de computação útil. Além disso, alguns fatores tais como o tipo de protocolo utilizado pelo TMK e a utilização de *lock* centralizado também impedem que o desempenho seja melhor.

Além do benefício alcançado através da distribuição da fase de *labeling*, pode-se ainda modificar esta aplicação de forma a produzir resultados superiores em termos de desempenho do que os obtidos nesta tese, evitando o desbalanceamento de carga. É importante ressaltar que o desbalanceamento de carga é um dos fatores determinantes para o baixo desempenho desta aplicação em 8 processadores em relação a 4 e 2 processadores, como pode ser observado na tabela 7.4.

7.3.1.1 *Arithmetic* para a versão PCSOS_SHM_TMK

Em nossos experimentos, a aplicação que apresentou os melhores resultados foi a *Arithmetic*. Ela é a que apresenta menor comunicação entre os processadores. Portanto, utilizamos esta aplicação para avaliar a versão PCSOS_SHM_TMK.

A versão PCSOS_SHM_TMK (PST) apresenta um tempo maior de execução que a versão PCSOS_TMK (PT), para qualquer quantidade de processadores, como pode ser observado na tabela 7.6 e na figura 7.3.

Processadores	Tempo de Execução (seg.)	
	PT	PST
1	4,74	5,29
2	0,39	0,98
4	0,37	1,02
8	0,47	1,45

Tabela 7.6: Tempos de execução de *Arithmetic* para PT e PST

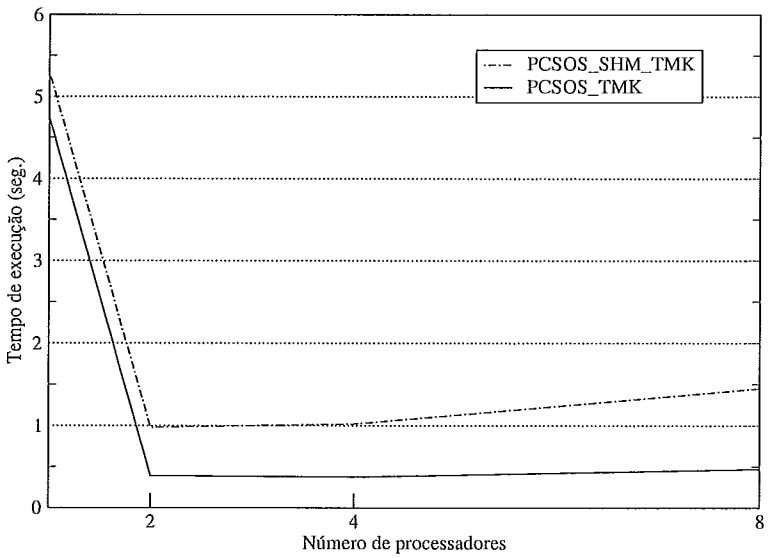

Figura 7.3: Tempo de execução do *Arithmetic* para a versão PCSOS_SHM_TMK

A tabela 7.7 apresenta o total de *acquires*, mensagens e barreiras para as duas versões, PT e PST.

Estatísticas	Processadores							
	1		2		4		8	
	PT	PST	PT	PST	PT	PST	PT	PST
Acquires	47.073	174.602	39.386	43.446	128.250	40.580	538.802	44.773
Mensagens	0	0	11.657	41.017	80.059	114.099	206.205	259.884
Barreiras	8	6	8	6	8	6	8	6

Tabela 7.7: Total de *acquires*, mensagens e barreiras de *Arithmetic* para PT e PST

A aplicação *Arithmetic* executada na versão PT, com 8 processadores, apresenta um número total de *acquires*, acumulado para todos os processadores (538.802), maior que na versão PST (44.473). Esta diferença é aproximadamente 12 vezes maior. Isto ocorre porque a fase de *backtracking* da versão PT é protegida por um *lock* associado a cada variável de ligação para obtenção do valor mais atualizado do domínio. Na versão Porém, o tempo total de *acquire* para a versão PST (2,91 seg.) é 1,5 vezes maior do que para a versão PT (1,93 seg.). Além disso, a quantidade total de *bytes* transmitidos na versão PST (20.536.500) foi quase 4 vezes maior do que na versão PT (5.628.078).

Este efeito causa uma queda de desempenho de 1 para 2 processadores em relação à versão PT como pode ser observado na figura 7.4. Enquanto na versão PT para

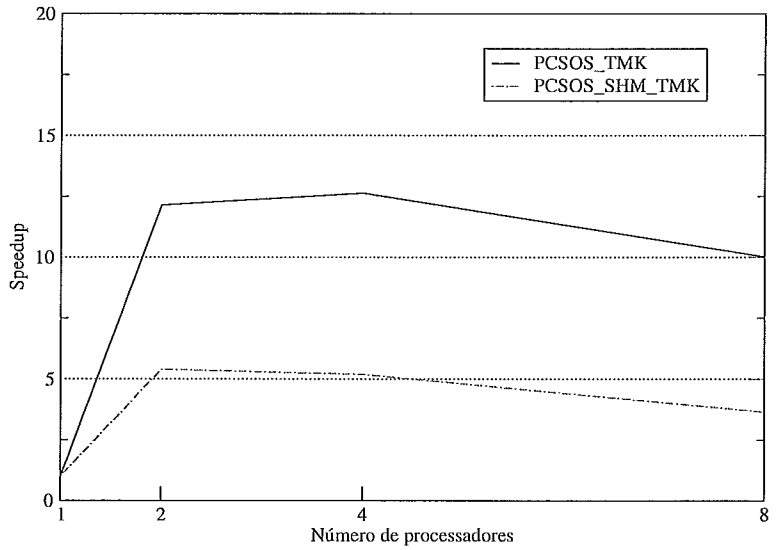

Figura 7.4: Gráfico de *speedup* do *Arithmetic* para as versões PCSOS_TMK e PCSOS_SHM_TMK

2 processadores temos um *speedup* de 12,15, na versão PST temos um *speedup* de 5,4. Ou seja, o *overhead* de comunicação da versão PST impede que tenhamos um *speedup* melhor de 1 para 2 processadores. Note bem que continuamos com um *speedup* superlinear em relação à execução seqüencial. A versão PST sofre mais com o protocolo de invalidação de TreadMarks do que a versão PT. Apesar de utilizarmos um *lock* para cada variável, o *overhead* de invalidação é muito grande, pois o tipo de compartilhamento desta classe de aplicação é verdadeiro. Portanto, as modificações realizadas nas variáveis compartilhadas deveriam ser enviadas em mensagens de *update* se desejássemos obter melhor desempenho.

Nesta aplicação cada variável ocupa 104 *bytes*, tendo então 39 variáveis por página de tamanho igual a 4096 *bytes*. Assim, cada vez que o domínio de uma variável de ligação é alterado, o próximo processador que tocar na variável de ligação recebe todas as notificações de escrita realizadas até o momento do *release* do último processador *releaser*, incluindo a notificação de escrita da página que contém a própria variável de ligação. Desta forma, a quantidade de comunicação aumenta excessivamente. Como TreadMarks utiliza o protocolo de coerência baseado em invalidações, quando o processador for tocar na variável de ligação após um *acquire*, ocorre uma falha de página. Para que a página se torne atualizada são pedidos, recebidos e aplicados todos os *diffs* necessários. Na verdade, como precisaríamos somente das modificações realizadas na variável de ligação, existe um tráfego de

informações desnecessário relativo as demais variáveis que estão alocadas na mesma página. Além disso, como a lista de *indexicals* também é compartilhada, cada vez que o processador tocar em algum *indexical* que contenha uma variável que é modificada por outro processador, também ocorrerá todo o procedimento para atualização do dado compartilhado e que envolve comunicação.

As tabelas 7.8 e 7.9 apresentam o total de *indexicals* e de falhas para as duas versões.

Proc#	1		2		4		8	
	PT	PST	PT	PST	PT	PST	PT	PST
0	1.953.660	1.953.660	89.255	89.255	23.363	23.363	2.392	2.392
1	-	-	44.071	44.071	25.457	25.456	20.917	21.025
2	-	-	-	-	4.607	4.068	15.001	15.002
3	-	-	-	-	13.665	13.150	10.437	10.358
4	-	-	-	-	-	-	3.237	3.237
5	-	-	-	-	-	-	2.069	1.353
6	-	-	-	-	-	-	19.566	18.536
7	-	-	-	-	-	-	3.543	3.193
Tot. de *ind.*	1.953.660	1.953.660	133.326	133.326	67.092	66.037	77.162	75.096

Tabela 7.8: *Indexicals* executados por processador para *Arithmetic* nas duas versões

Proc#	1		2		4		8	
	PT	PST	PT	PST	PT	PST	PT	PST
0	12.438	12.348	619	619	192	192	9	9
1	-	-	306	306	167	167	188	188
2	-	-	-	-	23	23	62	62
3	-	-	-	-	113	112	105	105
4	-	-	-	-	-		15	15
5	-	-	-	-	-		10	10
6	-	-	-	-	-		122	120
7	-	-	-	-	-		38	37
Falhas	12.438	12.438	925	925	495	494	549	546

Tabela 7.9: Total de falhas por processador para *Arithmetic* nas duas versões

Podemos observar que o número total de *indexicals* executados e de falhas é um pouco menor para a versão PST do que para a versão PT para 4 e 8 processadores. A variação entre as duas versões ocorre com o aumento do número de processadores. Tanto na versão PST quanto na versão PT pode ocorrer variação no número de *indexicals* executados por cada processador. Este fato ocorre porque a fase de *labeling*

é realizada em paralelo pelos processadores e a alteração do domínio das variáveis é realizada na estrutura compartilhada. O número de falhas ocorridas durante a execução do *labeling* depende da ordem em que os *indexicals* são executados. Como um processador pode executar mais rápido do que os outros em um experimento e ser mais lento em outro, a ordem de execução dos *indexicals* pode variar, alterando, também, o número de falhas ocorridas em cada processador.

Na versão PT a atualização do domínio de uma variável não é realizada diretamente na variável compartilhada, mas através de uma fila de mensagens (o *store*). O número de barreiras é menor na versão PST.

Nas tabelas 7.8 e 7.9, mais uma vez, podemos observar o desbalanceamento de carga ocorrido nesta aplicação devido à implementação do procedimento de *labeling* distribuído.

Além dos experimentos realizados nesta seção, também testamos a aplicação *Arithmetic*, na versão PCSOS_SHM_TMK, para 2, 4 e 8 processadores, no mesmo *cluster*, porém com uma rede *giganet*. O tempo de execução diminuiu quase pela metade para 4 processadores (1,02 seg. para 0,53 seg.) em relação à rede *Fast-Ethernet*, o que nos leva a concluir que podemos ter resultados ainda melhores.

Outro experimento que realizamos foi a execução da aplicação *Arithmetic* para a versão PCSOS_TMK para um processador utilizando vários processos, ou seja, o particionamento de *indexicals* e *labeling* são feitos de forma concorrente. Comparado com o algoritmo seqüencial, este novo algoritmo comporta-se de forma diferente quando variamos o número de processos. A tabela 7.10 mostra o tempo de execução, em segundos, para 2, 4 e 8 processos para 1 processador e para 2, 4 e 8 processadores, além do *speedup* encontrado comparando a execução para 1 processador com a execução para n processadores. Para que tenhamos uma comparação justa de algoritmos iguais, onde cada processo trabalha sobre o mesmo subconjunto de dados em um processador e em vários processadores, mostramos o ganho em tempo de execução relativo a n processos em um processador com n processos em n processadores. Verificamos que existe um ganho ao executarmos para n processadores em relação à execução em um único processador. Este fato ocorre porque a execução de vários processos em um único processador gera concorrência que provoca atraso no tempo de execução.

Este experimento confirma que nossos *speedups* superlineares são devidos ao *labeling* distribuído. Também demonstra que, apesar de não termos *speedups* signi-

Número de Processos	1 processador	n processadores	*Speedup*
2	0,55	0,39	1,40
4	0,53	0,37	1,42
8	0,59	0,47	1,24

Tabela 7.10: Tempos de execução de *Arithmetic* para 1 proc. e n procs.

ficativos de 2 para 4 e de 4 para 8 processadores, a paralelização contribui de forma positiva para a melhora do tempo de execução, que, no melhor caso, é de 42% para 4 processadores.

7.3.2 PBCSP

A segunda aplicação que utilizamos para avaliar nossos sistemas foi *Parametrizable Binary Constraint Satisfaction Problem* (PBCSP). Ela possui um grau de dependência entre os *indexicals* maior do que a aplicação *Arithmetic* analisada na última seção. Portanto, nesta aplicação não utilizamos um procedimento de *labeling* distribuído. Além disso, utilizamos a política de distribuição *Round-robin* para determinar o particionamento dos *indexicals* na aplicação PBCSP.

Na tabela 7.11 e na figura 7.5 são mostrados os tempos de execução da aplicação PBCSP, para as duas massas de dados, PBCSP_1 (100 variáveis) e PBCSP_2 (200 variáveis).

Processadores	Tempo de Execução (seg.)	
	PBCSP_1	PBCSP_2
1	26,55	39,36
2	20,13	29,54
4	26,38	26,03
8	65,22	50,41

Tabela 7.11: Tempos de execução do PBCSP_1 e PBCSP_2

O menor tempo de execução obtido foi igual a 20,13 seg. para 2 processadores com PBCSP_1. Esta execução gerou um *speedup* de 1,31, mostrado no gráfico de *speedup* da figura 7.6. Ainda considerando a coluna relativa ao PBCSP_1, na execução para 4 processadores há um aumento do tempo de execução em relação à

Figura 7.5: Tempo de execução do PBCSP_1 e PBCSP_2

execução para 2 processadores. E para 8 processadores, o tempo de execução paralelo foi aproximadamente 3 vezes maior que o tempo de execução seqüencial.

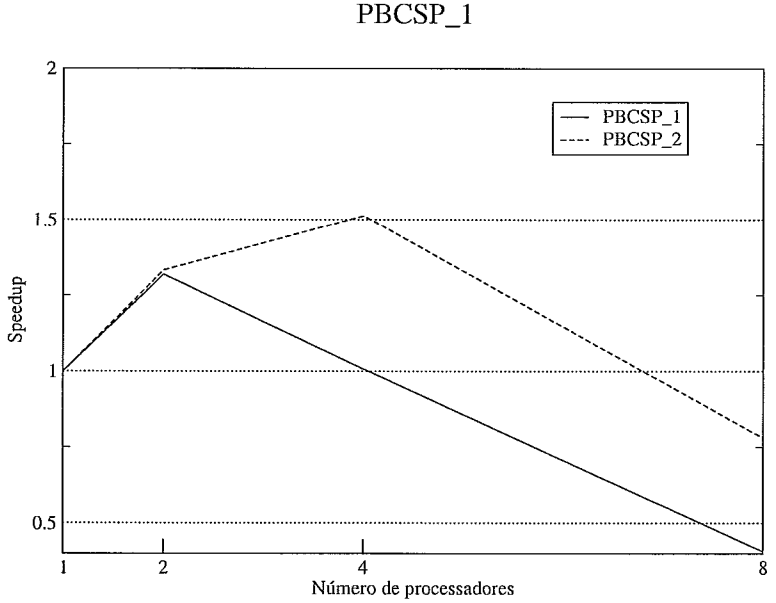

Figura 7.6: Gráfico de *speedup* para PBCSP_1 e PBCSP_2

Para PBCSP_2 o tempo de execução para 2 processadores é menor do que o tempo seqüencial. Ele gera um *speedup* de 1,33, como podemos observar no gráfico de *speedup* da figura 7.6. Para 4 processadores obtivemos o menor tempo de execução (26,03 seg.), gerando um *speedup* de 1,51. Para 8 processadores o tempo de execução supera o tempo de execução seqüencial em 28%.

Comparando os resultados obtidos para PBCSP_1 e PBCSP_2, observamos que ao aumentarmos a massa de dados melhoramos a escalabilidade.

O grafo de restrições desta aplicação é fortemente conexo. Desta forma, a dependência entre os *indexicals* é grande. O resultado desta dependência é que ao particionarmos os *indexicals*, vários deles, que relacionam a mesma variável, são distribuídos entre processadores distintos. Este fato gera muita comunicação entre os processadores, aumentando o tempo de execução da aplicação. Com o aumento do número de processadores a quantidade de comunicação supera a quantidade de execução útil, gerando *speedups* baixos. Com uma massa de dados maior, aumentamos o tempo de execução útil, superando o tempo de comunicação, até 4 processadores. PBCSP_2 sofre uma queda de desempenho para 8 processadores e um dos fatores que contribui para este fenômeno é o desbalanceamento de carga como pode ser observado na tabela 7.12.

Proc#	1		2		4		8	
	PBCSP_1	PBCSP_2	PBCSP_1	PBCSP_2	PBCSP_1	PBCSP_2	PBCSP_1	PBCSP_2
0	542.937	785.551	277.916	403.881	143.907	197.983	73.377	89.371
1	-	-	279.306	424.533	144.645	226.786	73.583	135.042
2	-	-	-	-	137.993	196.468	73.490	83.699
3	-	-	-	-	139.652	193.257	71.782	127.350
4	-	-	-	-	-	-	75.023	86.453
5	-	-	-	-	-	-	75.785	135.846
6	-	-	-	-	-	-	73.469	87.415
7	-	-	-	-	-	-	75.612	128.080
Total	542.937	785.551	557.222	828.414	566.197	814.494	592.121	873.256

Tabela 7.12: Total de *indexicals* para PBCSP_1 e PBCSP_2

Nesta tabela são mostrados os números de *indexicals* executados em cada processador, durante a execução das aplicações PBCSP_1 e PBCSP_2. Podemos observar que o desbalanceamento de carga para 8 processadores no PBCSP_2 é maior do que para o PBCSP_1, ou seja, temos ainda escopo para melhorar o desempenho desta aplicação.

Na tabela 7.13 também mostramos a quantidade de *acquires* e mensagens trocadas durante a execução das aplicações PBCSP_1 e PBCSP_2. Na execução seqüencial o número de *acquires* corresponde ao número de vezes que se passou pelo trecho de código onde há acessos aos *locks*, mas não há geração de mensagens. Podemos observar que o número total de *acquires* aumenta com o número de processadores, pois a concorrência pelo acesso à área de dados compartilhados é maior.

Quando aumentamos o número de processadores, a comunicação entre eles

torna-se grande, por causa da dependência entre os *indexicals*. Toda vez que uma variável é modificada é necessário realizar operações de coerência, que envolvem troca de mensagens entre os processadores, para atualizar os valores dos domínios das variáveis. O crescimento do número de mensagens pode ser verificado na tabela 7.13.

Procs.	*Acquires*		Mensagens		Barreiras		Falhas	
	PBCSP_1	PBCSP_2	PBCSP_1	PBCSP_2	PBCSP_1	PBCSP_2	PBCSP_1	PBCSP_2
1	47.178	182.082	0	0	83	34	28	9
2	166.257	381.131	175.469	413.133	83	34	28	9
4	229.473	599.127	629.666	982.587	83	34	28	9
8	401.925	1.187.166	1.504.176	2.196.837	83	34	28	9

Tabela 7.13: Total de *acquires*, mensagens, barreiras e falhas para PBCSP_1 e PBCSP_2

A quantidade de falhas ocorridas na fase de *labeling* corresponde ao número de vezes em que foi escolhido um valor para uma determinada variável que não resultou na consistência global das restrições. Ao ser detectada a falha é realizado o procedimento de *backtracking* e um novo valor para a variável é escolhido. O número de falhas é o mesmo para qualquer quantidade de processadores, pois o conjunto de variáveis que cada processador executa na fase de *labeling* é o mesmo. O procedimento de *labeling* está sendo executado por todos os processadores, de forma redundante, pois eles estão escolhendo as mesmas variáveis.

Escolhemos esta estratégia para a fase de *labeling*, nesta aplicação, porque o grafo de restrições é complexo e a interdependência entre as restrições não é tão visível quanto a da aplicação *Arithmetic*. Desta forma, torna-se complicado diferenciar uma falha local de uma falha global, durante a realização de um procedimento de *labeling* distribuído. Então, durante a execução, um determinado processador poderia enviar aos demais processadores uma falha local que surtiria o efeito de uma falha global, ou seja, os processadores que recebessem esta falha realizariam *backtracking*. Porém este fato poderia impedir que o procedimento de *labeling* fosse realizado com sucesso e retornaria um domínio sem podas.

Para avaliarmos o efeito de diferentes estratégias de implementação do procedimento de *labeling* e de particionamento de *indexicals* utilizamos a aplicação PBCSP_1 rodando em 2 processadores. Comparamos num experimento, o procedimento de *labeling* seqüencial (LS) e distribuído (LD) combinados com os particionamentos em blocos (PB) e *Round-robin* (PRR). Como podemos observar na tabela 7.14, melho-

ramos o tempo médio de execução de 20,13 seg. para 19,70 seg. ao utilizarmos o *labeling* seqüencial combinado com o particionamento em blocos (LSPB). Uma melhora de 86% foi obtida com a versão de *labeling* distribuído combinado com o particionamento *Round-robin* (LDPRR) em relação à versão de *labeling* seqüencial, utilizando a mesma forma de particionamento de *indexicals* (LSPRR). Além disso, a combinação do *labeling* distribuído com o particionamento em blocos (LDPB) resultou numa melhora de desempenho de 77% em relação ao *labeling* seqüencial com particionamento *Round-robin* (LSPRR). Apesar do procedimento de *labeling* distribuído para esta aplicação necessitar de cuidados especiais (por causa do grafo fortemente conectado) realizamos o experimento com sucesso para 2 processadores.

Combinações	Tempo de Execução (seg.)
LSPRR	20,13
LSPB	19,70
LDPRR	10,81
LDPB	11,31

Tabela 7.14: Diferentes *labelings* e particionamentos para PBCSP_1

7.3.3 *Queens*

A aplicação *Queens*, assim como a aplicação PBCSP, apresenta um grau de interdependência forte entre os *indexicals*. Na tabela 7.15 mostramos os tempos de execução da aplicação *Queens*. O gráfico de execução referente a esta tabela encontra-se na figura 7.7.

Processadores	Tempo de Execução (seg.)
1	9,26
2	104,97
4	381,27
8	1.243,21

Tabela 7.15: Tempos de execução de *Queens*

A aplicação *Queens* apresenta um grafo de dependência de *indexicals* totalmente conectado, isto é, existem restrições entre todas as variáveis. Desta forma, ao particionarmos os *indexicals* entre os processadores, é inevitável alocar *indexicals* referen-

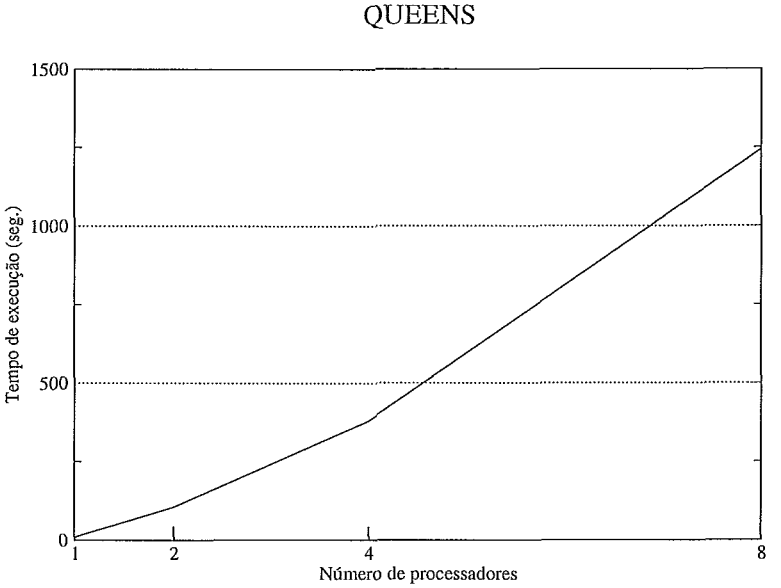

Figura 7.7: Tempo de execução de *Queens*

tes à mesma variável em processadores diferentes. Ao ser realizada alguma alteração no domínio de uma variável, esta modificação deve ser comunicada a todos os outros processadores quando eles necessitarem do domínio atualizado. Assim, o tempo de comunicação afeta o tempo de execução da aplicação. O tempo de execução é bem maior na versão paralela do que na versão seqüencial. Na figura 7.8, observamos o *slowdown* ocorrido para esta aplicação.

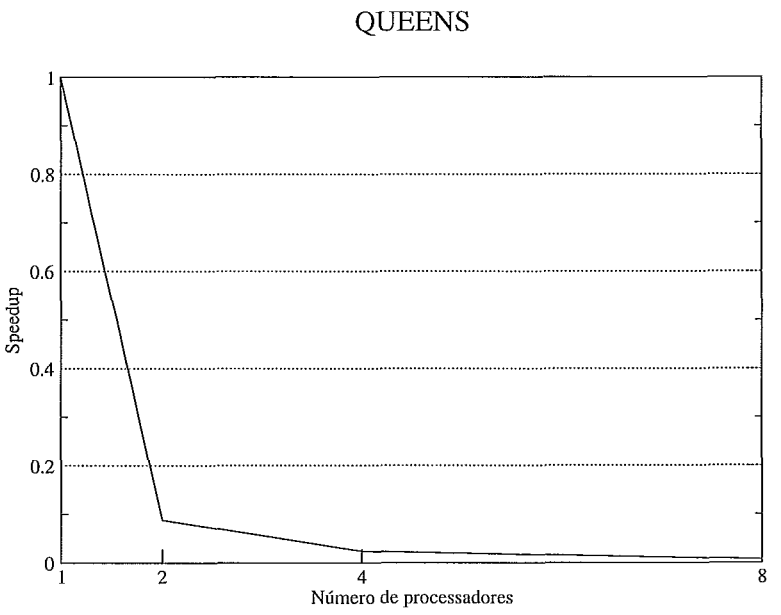

Figura 7.8: Gráfico de *speedup* de *Queens*

A comunicação é um fator limitante no desempenho desta aplicação na execução

em paralelo, porque a quantidade de mensagens aumenta significativamente com o número de processadores. Este fato pode ser observado, também, na tabela 7.16. Esta tabela mostra o número de *acquires*, mensagens, barreiras e falhas, obtidas pela execução da aplicação. Da mesma forma que mencionado nas análises das aplicações anteriores, o número de *acquires* corresponde ao número de vezes que o *lock* foi acessado. E não há mensagens na execução da versão seqüencial. As regiões críticas são acessadas um grande número de vezes quando se realiza a poda. Isto implica na transmissão das alterações ocorridas nos domínios das variáveis aos demais processadores. Por isso, o número de *acquires* e de mensagens é grande. A quantidade de falhas é a mesma para todos os processadores porque a fase de *labeling* é executada sobre o mesmo conjunto de variáveis por todos os processadores.

Estatísticas	Processadores			
	1	2	4	8
Acquires	157.296	1.918.354	5.856.510	7.353.994
Mensagens	0	1.635.691	6.996.274	28.645.600
Barreiras	13.142	13.142	13.142	13.142
Falhas	4.473	4.473	4.473	4.473

Tabela 7.16: Total de *acquires*, mensagens e falhas para *Queens*

Proc#	1	2	4	8
0	718.886	383.308	204.543	114.150
1	-	383.036	199.691	108.984
2	-	-	203.683	97.492
3	-	-	200.580	105.007
4	-	-	-	100.398
5	-	-	-	100.817
6	-	-	-	113.611
7	-	-	-	103.236
Tot. de ind.	718.886	676.344	808.497	843.695

Tabela 7.17: Total de *indexicals* executados por processador para *Queens*

Na tabela 7.17 mostramos o número de *indexicals* executados por cada processador. Esta aplicação não apresenta desbalanceamento acentuado, pois o procedimento de *labeling* é feito seqüencialmente sobre o mesmo conjunto de variáveis por

todos os processadores. Este procedimento não é realizado em paralelo pelo mesmo motivo da aplicação PBCSP. Na aplicação *Queens* o grafo de restrições é totalmente conectado, com muita interdependência entre os *indexicals*. Desta forma, durante a fase de *labeling* podia ocorrer uma inconsistência local que não seria detectada como local e enviada aos outros processadores interferindo no procedimento de *labeling*.

7.3.4 *Sudoku*

A aplicação *Sudoku* é a única aplicação que executamos somente para 1, 2 e 4 processadores. Ela apresenta um padrão sofisticado de interdependência entre os processadores como foi mostrado na seção 6.3.4. Não submetemos esta aplicação para ser executada em 8 processadores, porque a queda de desempenho de 1 para 2 processadores se mostrou excessivamente alta. Avaliamos, então, as razões para a queda de desempenho rodando apenas em 2 e 4 processadores analisando também o comportamento paralelo. A tabela 7.18 apresenta os tempos de execução da aplicação *Sudoku*. O gráfico de tempo de execução referente à tabela encontra-se na figura 7.9.

Processadores	Tempo de Execução (seg.)
1	35,03
2	1.489,83
4	1.670,99

Tabela 7.18: Tempos de execução de *Sudoku*

O grafo de restrições desta aplicação é fortemente conectado. Quando o particionamento dos *indexicals* entre os processadores é realizado, os *indexicals* relativos a uma mesma variável são alocados em processadores diferentes. Assim, toda alteração realizada no domínio de alguma variável deve ser comunicada a todos os outros processadores. O tempo de comunicação das atualizações afeta o tempo de execução da aplicação, implicando em um tempo bem maior para a execução em paralelo do que para a execução seqüencial. A figura 7.10 apresenta o *slowdown* ocorrido para esta aplicação. Os resultados apresentados são para o particionamento de *indexicals* do tipo *Round-robin*. Testamos o mesmo programa com particionamento em blocos e obtivemos uma melhora no tempo de execução de 24%, para 2 processadores, o que sugere que temos oportunidades para obter desempenho superior ao

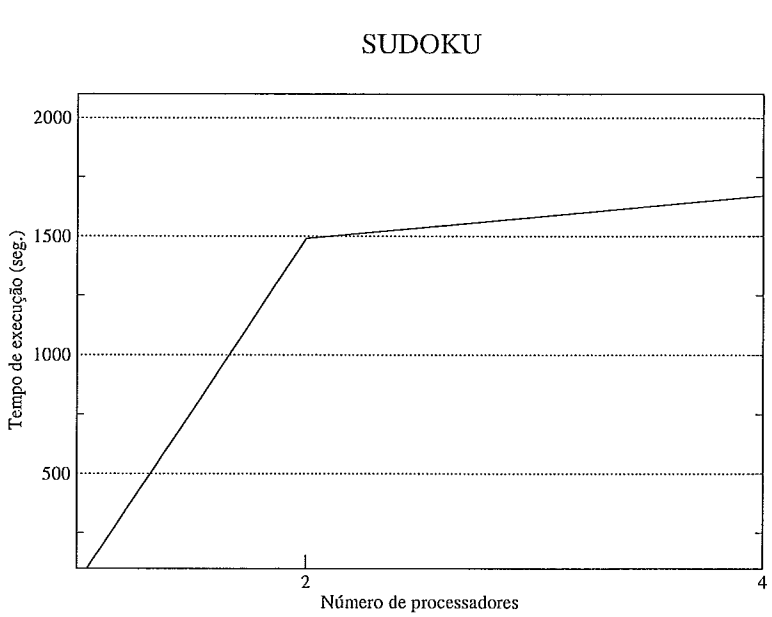

Figura 7.9: Gráfico de tempo de execução de *Sudoku*

alcançado nestes experimentos.

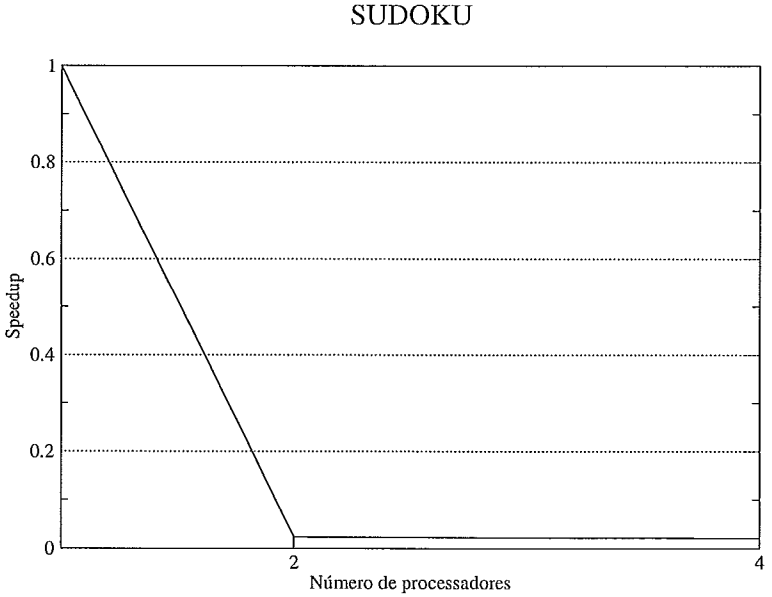

Figura 7.10: Gráfico de *speedup* de *Sudoku*

A tabela 7.19 apresenta o total de *acquires*, mensagens, barreiras e falhas para a aplicação *Sudoku*.

A quantidade total de *acquires* e mensagens cresce com o número de processadores porque os *indexicals* ficam distribuídos e como a dependência entre as variáveis é grande, cada alteração do valor do domínio de uma variável acarreta maior comunicação entre os processadores. O número de barreiras executadas é

Estatísticas	Processadores		
	1	2	4
Acquires	353.216	24.002.551	22.491.934
Mensagens	0	21.810.801	127.997.156
Barreiras	108.297	108.297	108.297
Falhas	36.092	36.097	36.085

Tabela 7.19: Total de *acquires*, mensagens, barreiras e falhas para *Sudoku*

bastante alto, se comparado com as demais aplicações, porque a massa de dados é bem maior do que a massa de dados das outras aplicações. O número de falhas é o mesmo para todos os processadores porque a fase de *labeling* está sendo executada seqüencialmente, ou seja, todos os processadores executam o mesmo conjunto de 308 variáveis. A diferença no total de falhas entre 1, 2 e 4 processadores é causada pela distribuição das variáveis que possuem um valor pré-definido já no início da execução.

O total de *indexicals* executados por cada processador é ilustrado na tabela 7.20.

Proc#	1	2	4
0	9.765.277	5.593.249	2.609.039
1	-	4.776.026	2.551.971
2	-	-	2.673.049
3	-	-	2.354.200
Tot. de ind.	9.765.277	10.369.275	10.188.259

Tabela 7.20: Total de *indexicals* executados por cada processador para *Sudoku*

7.3.5 Discussão

Andino *et al.* atingiram diferentes *speedups* máximos para cada aplicação. Na aplicação *Arithmetic* o *speedup* foi igual a 3,0 em 8 processadores. A aplicação PBCSP_1 apresentou o *speedup* de 18,0 em 34 processadores. A aplicação *Queens* apresentou o *speedup* de 4,0 para 16 processadores e *Sudoku* o *speedup* de 2,3 para 18 processadores. Seus experimentos foram realizados em uma arquitetura de alto desempenho e alto custo financeiro. A rede de interconexão do CRAY T3E tem uma capacidade nominal igual a 480 MBps e 230 MBps na prática. Enquanto a *Fast-Ethernet* tem uma capacidade de apenas 100 Mbps. Além disso, o sistema fornece

suporte eficiente da biblioteca de acesso à memória compartilhada do CRAY T3E, denominada SHMEM. Isto se reflete em uma vantagem para suas aplicações. Acreditamos que a distribuição de *indexicals* realizada em blocos e o particionamento do procedimento de *labeling* poderiam ter produzido melhores resultados no seu ambiente do que aqueles apresentados na literatura. Consideramos nossos resultados bastante positivos, principalmente pelo fato de termos atingido *speedups* superlineares para uma das aplicações mesmo com os *overheads* do sistema de *software* DSM. Além disso, nossa versão do sistema tem muitas oportunidades para otimização.

O *slowdown* de algumas aplicações é agravado pelo fato de que TMK utiliza o protocolo de invalidações para manter as memórias coerentes. Neste tipo de aplicação, o ideal seria a utilização de um protocolo híbrido, onde o protocolo de coerência de atualização fosse utilizado enquanto o processador estivesse utilizando o dado compartilhado e invalidação quando o processador deixasse de acessar o dado compartilhado.

Experimentos realizados com *hardware* DSM utilizando aplicações irregulares do tipo das apresentadas nesta tese mostram que protocolos híbridos podem oferecer melhor desempenho do que protocolo baseado em invalidações [12, 6]. Além disso, experimentos com simulações de TreadMarks mostram que *softwares* DSM adaptativos podem causar uma melhora de desempenho [8].

7.4 Resumo

Neste capítulo foram apresentadas a plataforma, as características das aplicações utilizadas em nossos experimentos e foi realizada a avaliação dos resultados obtidos. Verificamos que para algumas aplicações exploramos melhor o paralelismo do que em outras, onde há maior comunicação.

Vários fatores contribuem para o baixo desempenho de algumas aplicações para algum número de processadores:

1. desbalanceamento de carga causado pelo tipo do particionamento da fase de *labeling* que efetuamos e o falso compartilhamento entre os processadores;
2. protocolo de coerência utilizado por TMK (invalidações);
3. o tamanho da massa de dados;
4. a distribuição de *indexicals* (blocos ou *Round-robin*) entre os processadores.

Capítulo 8

Conclusões e Trabalhos Futuros

Neste trabalho modificamos um sistema de consistência de arcos paralelo para uma plataforma *sotfware* DSM, utilizando o protocolo TreadMarks e um *cluster* de PC's. Nosso sistema é baseado no sistema PCSOS, implementado para realizar consistência de arcos sobre domínios finitos para resolver problemas de satisfação de restrições (CSP). O PCSOS foi anteriormente desenvolvido para uma plataforma de memória logicamente compartilhada, numa máquina CRAY-T3E com 34 processadores.

Realizamos a implementação de duas versões de PCSOS, a primeira utiliza um modelo de programação onde o *constraint store* e os *indexicals* são replicados para cada processador (PCSOS_TMK). Na segunda implementação o *constraint store* e a lista de *indexicals* são completamente compartilhados entre os processadores (PCSOS_SHM_TMK).

Implementamos dois tipos de particionamento de *indexicals*: *Round-robin* e blocos. Além disso, fizemos o *labeling* distribuído para algumas aplicações, cujo padrão de interdependência do grafo de restrições é visível diretamente na aplicação. O particionamento de *indexicals* em blocos e o *labeling* distribuído não foram implementados na versão original PCSOS.

Nosso sistema apresentou um bom desempenho para todas as aplicações que possuem pouca dependência entre as restrições. As aplicações que apresentam maior grau de dependência entre as restrições geram mais comunicação entre os processadores, dificultando a obtenção do mesmo desempenho. A implementação do procedimento de *labeling* paralelo proporcionou *speedups* superlineares.

O *overhead* de comunicação da plataforma de *hardware* e *software* utilizada em nossos experimentos é maior do que na plataforma utilizada pelo sistema em que nos baseamos originalmente. Este fato se deve às diferentes formas de implemen-

tação da comunicação entre os processadores. O sistema original foi executado num sistema de memória compartilhada que possui suporte de *hardware* para escrita em memória remota. Nossos experimentos foram realizados utilizando somente suporte em *software* para coerência de memória.

Consideramos nossos resultados bastante positivos e mostramos várias oportunidades para otimização tanto do sistema PCSOS quanto da biblioteca TreadMarks que utilizamos. Além disso, observamos que o particionamento dos *indexicals* em blocos e o *labeling* distribuído afetam positivamente o desempenho de nossas aplicações.

Para minimizar o *overhead* de comunicação em aplicações que apresentam maior grau de dependência entre as restrições, podemos estudar diferentes particionamentos de *indexicals* e de variáveis de forma a obter maior paralelismo. Este particionamento pode ser realizado em tempo de compilação ou execução observando as características topológicas do grafo de restrições.

Outra característica que pode ser melhor observada é a estrutura de dados que representa as variáveis. Tanto na versão original quanto nas nossas versões, a estrutura de dados de uma determinada variável pode estar armazenada em páginas diferentes. Isto ocasiona maior comunicação entre os processadores que compartilham esta variável cuja estrutura pode estar parte em uma página e parte em outra página.

A utilização de novos particionamentos e modificações na estrutura de dados das variáveis podem contribuir para a redução da contenção no sistema.

Baseado no estudo e na análise de desempenho do sistema proposto, percebemos que existe um bom potencial para se obter resultados eficientes na implementação de algoritmos de consistência de arcos em plataformas com memória compartilhada distribuída desenvolvidos em *software*.

Referências Bibliográficas

[1] A. R. Andino. *CSOS User Manual (Draft Version)*, June 1997.

[2] A. R. Andino, L. Araujo, and J. Ruz. Parallel Solver for Finite Domain Constraints. Technical Report SIP 71.98, Department of Computer Science. Universidad Complutense de Madrid, 1998.

[3] A. R. Andino, L. Araujo, F. Sáenz, and J. Ruz. Parallel Execution Models for Constraint Programming Over Finite Domains. In *Principles and Practice of Declarative Programming*, pages 134–151, 1999.

[4] B. Baudot and Y. Deville. Analysis of Distributed Arc-Consistency Algorithms. Technical Report RR 97-07, University of Louvain, Belgium, 1997.

[5] D. Bessiere. Arc-Consistency and Arc-Consistency Again. *Artificial Intelligence*, 65:179–190, 1994.

[6] V. M. Calegario and I. C. Dutra. Performance Evaluation of Or-Parallel Logic Programming Systems on Distributed Shared Memory Architectures. In *Proceedings of the EUROPAR'99*, pages 1484–1491, Aug-Sep 1999.

[7] Mats Carlsson and Johan Widen. SICStus Prolog User's Manual. Technical report, Swedish Institute of Computer Science, 1997. Release 3#6.

[8] M. C. S. Castro. *Técnicas para Detecção e Exploração de Padrões de Compartilhamento em Sistemas de Memória Compartilhada Distribuída.* PhD thesis, Engenharia de Sistemas e Computação (COPPE/UFRJ), Dezembro 1998.

[9] P. Codognet and D. Diaz. Compiling Constraints in CLP(FD). In *The Journal of Logic Programming*, pages 1–199, 1996.

[10] P. R. Cooper and J. M. Swain. Arc Consistency Algorithm: Parallelism and Domain Dependence. *Artificial Intelligence*, (58):207–235, 1992.

[11] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[12] I. C. Dutra, V. Santos Costa, and R. Bianchini. The Impact of Cache Coherence Protocols on Parallel Logic Programming Systems. In *International Conference on Computational Logic*, volume 1861, pages 1285–1299, 2000. Lecture Notes in Artificial Intelligence.

[13] M. Fabiunke. Parallel Distributed Constraint Satisfaction. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 99) - Las Vegas*, pages 1585–1591, June 1999.

[14] M. C. Ferris and O. L. Mangasarian. Parallel Constraint Distribution. *SIAM Journal on Optmization*, (4):487–500, 1991.

[15] A. Geist, A. Beguelin, J. Dongarra, J. Weicheng, R. Manchek, and V. Sunderam. *PVM - Parallel Virtual Machine: A Users´ Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1996.

[16] C. F. R. Geyer, I. C. Dutra, and P. K. Vargas. Parallelism in Logic Programming. In *Intl. School on Advanced Techniques for Parallel Computation with Application, Natal, RN, Brasil*, page 35, Sept-Oct 1999.

[17] S. Gregory and R. Yang. Parallel constraint solving in andorra-i. In *Proceeding of the Fifth Generation Computer Systems*, pages 843–850, June 1992.

[18] J. Gu and R. Sosic. A Parallel Architeture for Constraint Satisfaction. In *International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, pages 229–237, June, 1991.

[19] P. V. Hentenryck, Y. Deville, and C. M. Teng. A Generic Arc-Consistency Algorithm and its Specifications. *Artificial Intelligence*, 57(2–3):291–321, October 1992.

[20] M. Hermenegildo. Parallelizing Irregular and Pointer-Based Computations Automatically: Perpectives from Logic and Constraint Programming.

[21] J. Jaffar and J. L. Lassez. Constraint Logic Programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, pages 11–119, January 1987.

[22] Joxan Jaffar, editor. *Principles and Practice of Constraint Programming–CP'99: Fifth International Conference*, volume 1713. Springer Verlag, October 1999. Lecture Notes in Computer Science, Alexandria, VA, USA.

[23] S. Kasif. On the Parallel Complexity of Discrete Relaxation in Constraint Satisfaction Networks. *Artificial Inteligence*, (45):275–328, 1990.

[24] S. Kasif and A. L. Delcher. Local Consistency in Parallel Constraint Satisfaction Networks. *Artificial Intelligence*, (69):307–327, 1994.

[25] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix*, pages 115–131, 1994.

[26] V. Kumar. Algorithms for Constraint Satisfaction Problems: A Survey. *Artificial Intelligence Magazine*, 13(1):32–44, 1992.

[27] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

[28] Q. Y. Luo, P. G. Hendry, and J. T. Buchanan. Heuristic search for distributed constraint satisfaction problems. Research Report KEG-6-93, Department of Computer Science. University of Strathclyde, 1993.

[29] A. K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 28:225–233, 1977.

[30] A. K. Mackworth and E. C. Freuder. The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems. *Artificial Intelligence*, 25:65–74, 1985.

[31] K. Marriot and P. J. Stuckey. *Programming with constraints: An Introduction.* MIT Press, 1998.

[32] R. Mohr and T. C. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28:225–233, 1996.

[33] T. Nguyen and Y. Deville. A Distributed Arc-Consistency Algorithm. In *Science of Computer Programming*, pages 227–250, 1998.

[34] P. S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann Publishers, Inc., California, 1997.

[35] ParallelTools. *Concurrent Programming with TreadMarks*, 1994.

[36] S. L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *ASPLOS7*, pages 26–36, October 1996.

[37] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* MIT Press, 1986.

[38] Y. Zhang and A. K. Mackworth. Parallel and distributed algorithms for constraint networks. Technical Report 91.6, Department of Computer Science. The University of British Columbia, Canada, 1991.

Apêndice A

Tela de Execução do CSOS e um Exemplo de Programa Utilizando TMK

Este apêndice apresenta a tela de execução do CSOS e mostra um programa exemplo com chamadas à biblioteca que exprime o paralelismo utilizando o sistema *software* DSM TreadMarks.

A.1 Tela de execução do CSOS

O sistema original apresentado por Andino *et al.* apresenta uma tela de execução com as características que podemos observar na figura A.1. Esta figura mostra a execução da aplicação *8-queens* no sistema CSOS, que é a implementação seqüencial do sistema.

Na figura A.1, o comando "post_bin q8.bin" é usado para acessar o arquivo de entrada que contém *indexicals* referentes ao problema das 8 rainhas. Este CSP consiste em conseguir uma configuração de 8 rainhas em um tabuleiro de xadrez de forma que estas não se ataquem nas linhas, diagonais e colunas. Após a leitura do arquivo de entrada, os *indexicals* são armazenados nas estruturas de dados, que são discutidas na seção 5.2, e a consistência de arcos é realizada. O comando *label first ff(V0..V7)* refere-se à chamada do *labeling*, buscando a primeira solução para o CSP, isto é, a primeira configuração das 8 rainhas no tabuleiro que satisfaça todas as restrições, usando o algoritmo *first fail*. Após encontrar a primeira solução e mostrá-la na tela, ou seja, encontrar os primeiros valores para todas as variáveis que satisfaçam todas as restrições, aparece uma seta (->) que é o *prompt* desta fase da interface. Se digitarmos ponto e vírgula (;) a próxima solução é encontrada e

```
Welcome to Constrained Stochastic Optimization Solver (CSOS).
Version 2.0. April 1997
(c) Alvaro Ruiz-Andino Illera.
Dpto. de Sistemas Informaticos y Programacion.
Universidad Complutense de Madrid. Spain.
Type 'help' for help on commands.
Type 'quit' to leave the system.
Tue Jan 16 22:32:19 2001
./csos, 307650, Tue Jul 11 10:46:46 2000

(0) CSOS > post_bin q8.bin
q8.bin, 6257, Tue Jul 11 10:54:10 2000
Loading and installing indexicals from file 'q8.bin' ... OK
Elapsed wall clock time loading 'q8.bin' = 0.000 sec.
Goal : main([8,[v0 .. v7]])
Posting indexicals ...
yes
There are suspended indexicals
Elapsed CPU user time        = 0.000 sec.
Elapsed CPU user+system time = 0.000 sec.
Elapsed wall clock time      = 0.000 sec.

(1) CSOS > label first ff(V0..V7)
Finding first solution, labeling:
first fail: V0..V7, UP
[1, 5, 8, 6, 3, 7, 2, 4]
yes
Elapsed CPU user time = 0.02 sec.

-> ;
[1, 6, 8, 3, 7, 4, 2, 5]
yes
Elapsed CPU user time = 0.010 sec.

->
```

Figura A.1: Tela de execução do CSOS/PCSOS

return espera por um próximo comando.

A.2 Exemplo de programa paralelo utilizando TMK

Nesta seção apresentamos um exemplo de um programa, implementado utilizando a linguagem C, que demonstra como TreadMarks é utilizado para se desenvolver um programa paralelo [35]. Este programa armazena elementos em um vetor compartilhado por todos os processadores, imprime estes elementos e computa e imprime a soma de todos os elementos que compõem o vetor. Todas as chamadas da biblioteca de TreadMarks são iniciadas com "Tmk_".

```
/* programa app */
#include <stdio.h>
#include "Tmk.h"
struct shared {    /* Estrutura compartilhada entre os processadores */
        int sum;
        int turn;
        int* array;
}*shared;
main(int argc, char **argv)
{
        int start, end, i, p;
        int arrayDim = 100;
        int c;
        extern char* optarg;

        /* Lê tamanho de array da linha de comando */
        while ((c = getopt(argc, argv, "d:")) != -1){
                switch (c){
                     case 'd':
                        arrayDim = atoi(optarg);
                        break;
                }
        }
        Tmk_startup(argc, argv); /* Inicia processos */
        if (Tmk_proc_id == 0){   /* Alocação da memória compartilhada */
                shared = (struct shared *)Tmk_malloc(sizeof(shared));
                if (shared == NULL)
                        Tmk_exit(-1);
                /* Ponteiro comum compartilhado com todos os processos */
                Tmk_distribute(char *)(&shared, sizeof(shared));
                shared->array = (int *) Tmk_malloc(arrayDim*sizeof(int));
                if (shared->array == NULL)
                        Tmk_exit(-1);  /* Terminação com erro */
                shared->turn = 0;
```

```
            shared->sum =0;
        }
        Tmk_barrier(0); /* Sincronização de barreira */
        /* Determina o intervalo do vetor para cada processo */
            int id0 = Tmk_proc_id, id1 = Tmk_proc_id + 1;
            int perProc = arrayDim / Tmk_nprocs;
            int leftOver = arrayDim % Tmk_nprocs;
            start = id0 * perProc + id0 * leftOver / Tmk_nprocs;
            end = id1 * perProc + id1 * leftOver / Tmk_nprocs;
        }
        for (i = start; i < end; i++)
            shared->array[i] = i;
        Tmk_barrier(0);
        /* Imprime vetor de elementos na ordem de saída natural */
        for (p = 0; p < Tmk_nprocs; p++){
            if (shared->turn == Tmk_proc_id){
                for (i = start; i < end; i++)
                    printf("%d: %d\n", i, shared->array[i]);
                shared->turn++;
            }
            Tmk_barrier(0);
        }
        /* Computa a soma local e adiciona à soma global */
            int mySum = 0;
            for (i = start; i < end; i++)
                mySum += shared->array[i];
            Tmk_lock_acquire(0); /* Sincronização com lock*/
            shared->sum += mySum;
            Tmk_lock_release(0);
        }
        if (Tmk_proc_id == 0){
            printf("Sum é %d\n", shared->sum);
            Tmk_free(shared->array);
            Tmk_free(shared);
        }
        Tmk_exit(0);
}
```