UM ESTUDO DOS PROTOCOLOS DE COERÊNCIA PARA SISTEMAS LÓGICOS PARALELOS

Eduardo Pereira Gaspar de Oliveira

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. Anna Dolejsi Santos, D.Sc.

Prof. Cristiana Bentes, D.Sc.

RIO DE JANEIRO, RJ - BRASIL
FEVEREIRO DE 2005

OLIVEIRA, EDUARDO PEREIRA GASPAR DE OLIVEIRA

Um Estudo dos Protocolos de Coerência para Sistemas Lógicos Paralelos [Rio de Janeiro] 2005

XII, 70 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2005)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1 - Protocolos de coerência

2 - Memória compartilhada distribuída

3 - Hardware DSM

4 - DASH

I. COPPE/UFRJ II. Título (série)

À minha querida mãe e meu querido irmão

Agradecimentos

Agradeço a minha mãe Luci, presente nos momentos difíceis a me confortar, aconselhar e auxiliar. Perante os problemas apresentados na vida continuou lutando arduamente para o meu bem estar e sucesso, assim como o do meu irmão. Não estaríamos aqui sem sua mão a nos guiar. Pelo carinho, força e determinação dedicada a nós e que nos inspira só tenho a agradecer de ter nascido seu filho.

Agradeço ao meu irmão Alberto, que primeiro trilhou as estradas e assegurou-me caminhos mais tranquilos. Com a perda de nosso pai se fez responsável por parte de meu desenvolvimento. Só tenho a agradecer de ter um irmão, padrinho e amigo, cada qual de forma singular, em uma só pessoa que está ao meu lado desde o meu nascimento.

Agradeço a minha namorada Luana pelo apoio, inspiração e compreensão às minhas faltas, tanto necessárias ao desenvolvimento deste trabalho.

Agradeço a minha orientadora Inês, que me apresentou novos caminhos na ciência, indicando os passos necessários a percorrê-los. Agradeço por escutar-me e instruir-me diante das idéias, algumas destas mirabolantes, a serem executadas.

Agradeço a minha co-orientadora Maria Clícia que junto a Inês integrou-se na tarefa de orientar-me.

Agradeço ao meu pai, avós e entes queridos hoje presentes em outro plano, mas prontos a nos auxiliar.

Agradeço a Paulo Rezende, Celso Rossi e demais integrantes da equipe Infomarket pelo apoio oferecido ao desenvolvimento deste trabalho.

Agradeço ao Antônio Branco, Doris Ferraz de Aragon (in memorian), Luiz Cláudio da Silva Leão e demais integrantes do ILTC que auxiliaram no início desta conquista.

Agradeço a Deus por deixar participar de meu caminho todas as pessoas citadas anteriormente e pelos auxílios dispensados para efetuar este percurso.

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

UM ESTUDO DOS PROTOCOLOS DE COERÊNCIA PARA SISTEMAS LÓGICOS PARALELOS

Eduardo Pereira Gaspar de Oliveira

Fevereiro/2005

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

Modelos de programação lógica têm características que o tornam a solução mais adequada para determinados problemas. Alguns destes ligados a áreas de grande importância, por exemplo, a de Biotecnologia, que muitas vezes exigem alto poder computacional.

Este modelo também tem a vantagem de permitir que suas aplicações sejam executadas em arquiteturas paralelas de forma implícita ao desenvolvedor, isto é, sem alterações no código fonte, ao contrário de aplicações desenvolvidas em outros modelos como o de programação funcional ou imperativa. Este paralelismo implícito é aproveitado através de *frameworks* que administram os recursos da arquitetura paralela.

Arquiteturas paralelas, entretanto, têm uma série de parâmetros estruturais que influenciam o desempenho das aplicações executadas sobre ela. O protocolo de coerência de memória é um dos fatores mais significativos.

Este trabalho avalia o comportamento de algumas aplicações lógicas em arquiteturas paralelas de acordo com os protocolos de coerência de memória, complementando trabalhos anteriores e sendo mais uma referência no assunto.

Nossos resultados mostram que protocolos híbridos melhoram o desempenho de sistemas paralelos de programação lógica em até 80%, produzindo *speedups* de até 11,6 em 16 processadores, quando comparados com protocolos de invalidação, normalmente utilizados nos processadores atuais.

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

A STUDY OF HYBRID COHERENCE PROTOCOLS FOR PARALLEL LOGIC PROGRAMMING SYSTEMS

Eduardo Pereira Gaspar de Oliveira

February/2005

Advisors: Inês de Castro Dutra

Maria Clicia Stelling de Castro

Department: Computing and Systems Engineering

The logic programming model has characteristics that are suitable to solve problems in some areas of great importance, such as Biotechnology. In these areas, finding a solution requires, in general, a high computational power.

This model brings the advantage of being declarative, which makes the control not completely explicit, and allows for implicit parallelisation. This implicit parallelism is captured by frameworks that extract the available parallelism and manage the resources of a parallel architetcure.

Parallel architectures have various parameters that can affect the performance of the applications. One of the most important, specially in modern architectures, is the memory coherence protocol.

This work evaluates the behaviour of applications in parallel logic programming systems running on a scalable architecture, and study different cache coherence protocols. Our results show that hybrid protocols can improve the performance of parallel logic programming systems by 80%, achieving speedups of 11.6, with 16 processors, over invalidate-based protocols commonly used in current multiprocessors.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Programação em lógica constitui-se em um modelo de programação declarativo, baseado em um subconjunto de lógica de primeira ordem, cláusulas de Horn [31]. Neste tipo de modelo, o programador concentra-se na definição da solução do problema de forma lógica sem se preocupar com o controle ou como a solução vai ser encontrada. Um exemplo simples de programa em lógica é mostrado na Figura 1.1. Este exemplo é um programa escrito na linguagem Prolog [46] (mais popular para escrever programas em lógica) e define a concatenação de duas listas. O primeiro predicado define que o resultado da concatenação da lista vazia (representada como `[]`) com qualquer outra lista (representada pela variável `L`) é a própria lista. O segundo predicado trata do caso em que a primeira lista é não vazia (representada pelo termo `[X|L1]`, onde X é uma variável representando pelo menos um elemento da lista e L1 é o restante da lista, que pode ser lista vazia). Neste segundo caso, o predicado simplesmente diz que se a concatenação de L1 com L2 for L3 então, quando adicionado um elemento (X) a L1, este elemento vai ser adicionado, também, ao resultado L3.

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

Figura 1.1: Exemplo de programa prolog

Note que neste programa não há comandos que dizem "como" computar a concatenação. O programa apenas oferece a sua definição. E esta é uma das principais vantagens de programação em lógica. Como o controle está implícito na linguagem este tipo de modelo oferece maior flexibilidade aos sistemas de execução. Uma grande flexibilidade é oferecer a possibilidade de execução de programas em

ambientes paralelos, de forma que o usuário não precise se preocupar se o ambiente é paralelo. Em outras palavras, o usuário não precisa utilizar construções paralelas, existentes em outras linguagens imperativas, ou bibliotecas para paralelizar o seu programa. E paralelização é importante para programas em lógica porque, geralmente, programas em lógica são escritos para resolver problemas complexos na área de Inteligência Artificial, que consomem muitas horas ou mesmo meses de computação quando executados em uma única CPU.

As formas predominantes de exploração de paralelismo em programação lógica são paralelismo *-OU* e paralelismo *-E*. Paralelismo *-OU* é explorado em sistemas como Aurora [32] ou Muse [1]. Paralelismo *-E* é ainda classificado em paralelismo E-independente, e explorado por sistemas tais como &-Prolog [25] e &-ACE [24]; paralelismo E-dependente em Parlog's JAM [15], KLIC [47], e DDAS [43]; paralelismo de dados, em Reform Prolog [4]; e combinações de paralelismo *-E* e *-OU* como explorado pelos sistemas Andorra-I [3], Penny [35], ACE [23] e PALS [53]. Todos estes sistemas têm sido capazes de obter desempenho razoável em arquiteturas paralelas baseadas em barramento, tais como multiprocessadores Sequent Symmetry ou Sun.

A medida que arquiteturas mais modernas são desenvolvidas e a diferença de velocidades entre a CPU e a memória aumenta (atualmente as CPUs têm avançado em velocidade numa taxa muito maior do que a velocidade das memórias) surge a questão da eficiência destes sistemas paralelos de programação lógica nestas novas arquiteturas. Em multiprocessadores modernos, o desempenho depende principalmente das taxas de falha nas *caches* e pode ser limitado pelo *overhead* de comunicação introduzido pelos protocolos de comunicação utilizados para manter dados compartilhados coerentes.

Compartilhamento em sistemas paralelos de programação lógica tem característica complexa e pode ocorrer em várias circunstâncias dependendo do programa Prolog a ser executado em paralelo. O uso da variável lógica como canal de comunicação em aplicações que possuem paralelismo E-dependente, por exemplo, é um exemplo de compartilhamento de dados baseado num padrão produtor-consumidor. Nesse padrão o processador que instancia a variável (atribui-lhe um valor) escreve na variável e outro processador a lê. Esta comunicação pode ser feita por *streams* se a variável em questão for uma lista com vários elementos.

A segunda maior forma de compartilhamento em programação em lógica é de origem migratória. Esta forma de compartilhamento ocorre quando há sincronização entre processadores. A sincronização ocorre em tarefas tais como busca por trabalho de outros processadores ou quando um processador executa um predicado Prolog que pode causar um efeito colateral relacionado à execução seqüencial. Por exemplo, dois processadores podem querer ler e/ou escrever na tela ou em arquivo em ordens diferentes da original seqüencial. Outro exemplo, que acarreta um custo alto, seria um caso de sincronização onde processadores podem suspender e re-inicializar tarefas. Neste caso, pode acontecer destes processadores ficarem por longo tempo tentando buscar trabalho nas filas de outros processadores, lendo e escrevendo em estruturas compartilhadas. Um processador que produz uma tarefa escreve em uma das estruturas de dados compartilhadas, que mais tarde será modificada por um ou mais processadores que estão ociosos.

O compartilhamento de estruturas de dados de escrita introduz o problema de coerência de *caches*. A maior parte das máquinas paralelas utiliza protocolos de invalidação para manter as *caches* coerentes [22]. Neste protocolo, quando um processador escreve em um dado compartilhado, cópias deste dado que estão em outras *caches* são invalidadas. Se um dos processadores que teve o dado invalidado, posteriormente acessar o mesmo dado, terá que obtê-lo de volta para a sua *cache*. Em multiprocessadores escaláveis, este fato envolve uma comunicação.

Protocolos de atualização [34] são a alternativa principal para os protocolos baseados em invalidação. Em protocolos de atualização, sempre que um dado é escrito, cópias do novo valor são enviadas para outros processadores que compartilham o dado. Mais especificamente, considere o caso de paralelismo E-dependente, onde o padrão é similar a produtor-consumidor. Neste caso, um protocolo de atualização garante que o valor mais atualizado do dado está disponível para o consumidor. Enquanto com um protocolo de invalidação, o consumidor teria solicitar o dado ao outro processador que escreveu. Isso pode levar algumas centenas ou milhares de ciclos. Em casos deste tipo, sistemas paralelos de programação lógica claramente se beneficiariam de protocolos de atualização.

Mas não apenas padrões produtor-consumidor podem se beneficiar de protocolos de atualização. Sincronização envolvendo busca por trabalho, também pode se beneficiar de protocolos de atualização. Por exemplo, assim que um processador produzir trabalho os processadores ociosos teriam acesso à informação

imediatamente, sem ter que buscar a informação no processador que escreveu o dado. Um problema ocorre, porém, quando outros processadores recebem a atualização mas não podem utilizá-la porque outro processador ocioso já obteve aquele trabalho produzido.

Protocolos baseados em invalidação são mais populares do que protocolos baseados em atualização por causa do tráfego excessivo de mensagens produzido pelos protocolos de atualização. Em determinadas clases de aplicações, uma quantidade significativa de atualizações recebidas pelos processadores podem não ser utilizadas. Isto causa tráfego desnecessário, consome banda, e pode degradar o desempenho.

Neste trabalho, estudamos o comportamento de um protocolo híbrido na execução paralela de sistemas de programação lógica. Utilizamos 4 aplicações Prolog que, normalmente, são utilizadas como *benchmarks* nestes sistemas. O sistema de programação lógica utilizado é Andorra-I [3]. Este sistema explora a combinação de paralelismo E-dependente, uma forma restrita de paralelismo E-independente, e paralelismo -*OU*. Nosso objetivo com este estudo é saber se um protocolo híbrido consegue alcançar desempenho melhor do que protocolos baseados somente em invalidação ou em atualizações.

Utilizamos um simulador orientado por execução (*execution-driven*), de uma arquitetura de multiprocessador escalável. A versão de Andorra-I utilizada não contém otimizações e é a mesma que foi implementada originalmente para máquinas que utilizam barramento. Uma versão otimizada de Andorra-I que favorece o protocolo de invalidação foi desenvolvida [38]. Porém nosso objetivo neste trabalho é averiguar como o protocolo de coerência de memória pode afetar uma aplicação originalmente desenvolvida para memória centralizada. Comparamos nossos resultados com protocolos de invalidação e de atualização. Nossos resultados mostram que protocolos híbridos produzem um ganho de desempenho para todos os números de processadores e aplicações, independente do tipo de paralelismo existente na aplicação.

Nosso trabalho difere de outros estudos de desempenho de protocolos de coerência em sistemas paralelos de programação lógica. Tick e Hermenegildo [50] estudaram o comportamento de *caches* em sistemas que exploram apenas paralelismo E-independente em máquinas baseadas em barramento. Outros pesquisadores estudaram o desempenho de sistemas de programação lógica em

arquiteturas escaláveis tais como a DDM [36], porém não avaliaram o impacto de diferentes protocolos de coerência. Nosso trabalho inicial avaliou o impacto de protocolos de coerência de *cache* para um número menor de *benchmarks* e tamanho menor de *cache* [39]. Um protocolo híbrido com *threshold* fixo foi utilizado na avaliação. Também estudamos em detalhes o impacto do protocolo de invalidação em Andorra-I [40], o que nos permitiu implementar otimizações que melhoraram o desempenho [38, 41]. Silva *et al.* estudaram o impacto de diferentes parâmetros arquiteturais tais como: tamanho do bloco de *cache*, tamanho da *cache* (foi estudado até 256 Kbytes), tamanho do *buffer* de escrita e utilização de escritas concatenadas, utilizando protocolos de invalidação e atualização. Neste trabalho estudamos em detalhes o protocolo híbrido com vários valores de *threshold*, incluímos mais uma aplicação, e estudamos o comportamento do protocolo quando utilizamos tamanhos diferentes de *cache*, variando até 1 MByte.*

Este trabalho está organizado da seguinte forma. No Capítulo 2 apresentamos as arquiteturas paralelas existentes e suas diferenças, concentrando-se em multiprocessadores escaláveis e protocolos de coerência de *cache*. No Capítulo 3 apresentamos as características de sistemas paralelos de programação lógica, explicando os tipos de paralelismo explorados e as principais estruturas de dados utilizadas em sistemas paralelos de programação lógica implementados para máquinas baseadas em barramento, com ênfase no sistema Andorra-I. No Capítulo 4 descrevemos a metodologia utilizada neste trabalho e o simulador da arquitetura multiprocessada. No Capítulo 5 apresentamos os resultados e discutimos o impacto do protocolo híbrido nas aplicações Prolog, comparando com os resultados utilizando apenas invalidação e apenas atualização. Finalmente, no Capítulo 6 concluímos o nosso trabalho e sugerimos trabalhos futuros.

*Uma parte resumida deste trabalho foi publicada e apresentada na 16th International IASTED Conference on Parallel and Distributed Computing and Systems [18].

Capítulo 2

Arquiteturas Paralelas

Arquiteturas paralelas têm demonstrado ser uma das soluções mais efetivas para problemas que exigem alto poder computacional. As máquinas de Von Neumman (monoprocessadores) têm chegado aos limites de minituarização conhecidos. Assim é necessário um conceito alternativo para atender a crescente demanda de processamento.

Exemplos de aplicações candidatas à execução em sistemas paralelos são o de processamento de dados da área de Biotecnologia, simulações de grande porte, aplicações maciças da área de Inteligência Artificial, entre outras. Elas exigem um nível maior de processamento e armazenamento de informações.

Arquiteturas paralelas podem atender a estas duas exigências. Pode haver redução do tempo total de processamento a partir da divisão de tarefas entre as diversas unidades de processamento do sistema. Podemos, também, obter o aumento da memória do sistema. A *cache* pode ter sua utilização otimizada, pois pode ocorrer a melhor situação entre alocação de dados na *cache* e processamento de dados, de forma a diminuir a probabilidade de desalocar um dado da *cache* em detrimeno de outro, aumentando o desempenho do sistema.

De forma a compreender a metodologia apresentamos, a seguir, algumas características e noções referentes ao processamento paralelo.

2.1 Arquiteturas de computadores

A categorização definida por Flynn para arquiteturas de computadores é baseada nos fluxos de instruções e fluxos de dados [20]. O fluxo de instruções corresponde às tarefas que devem ser executadas e o fluxo de dados correponde aos dados que devem ser manipuladas durante a execução destas tarefas. As categorias são apresentadas

a seguir.

SISD (Single Instruction Single Data)

Nas arquiteturas SISD um único fluxo de instruções é executado sobre um único fluxo de dados em um mesmo ciclo de relógio. Um processador é suficiente para desempenhar a tarefa. A Figura 2.1 exemplifica esta arquitetura.

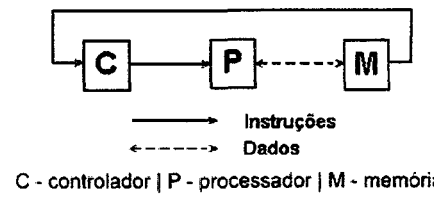

Figura 2.1: Arquitetura SISD

As arquiteturas SISD podem ser representadas pelos computadores monoprocessados, que se encontram em larga escala no mercado, por exemplo, os PCs (computadores pessoais e estações de trabalho).

SIMD (Single Instruction Multiple Data)

Nas arquiteturas SIMD um mesmo fluxo de instruções é executado sobre diferentes fluxos de dados em um mesmo ciclo de relógio. É necessário mais que um processador para executar esta tarefa. Cada um dos processadores estará em sincronia com os demais em relação às instruções, mas cada uma destas unidades as executará sobre um fluxo distinto de dados. A Figura 2.2 exemplifica esta arquitetura.

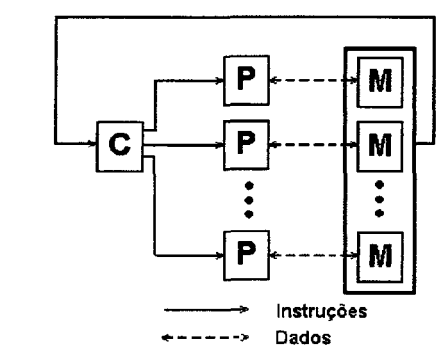

Figura 2.2: Arquitetura SIMD

Como exemplos de arquiteturas SIMD temos os processadores vetoriais e processadores matriciais. Um exemplo de aplicação capaz de se utilizar desta

arquitetura é a previsão meteorológica, onde os dados meteorológicos referentes a diferentes regiões seguem a mesma regra de cálculo.

MISD (Multiple Instruction Single Data)

Nas arquiteturas MISD vários fluxos de instruções são executados sobre um mesmo fluxo de dados em um mesmo ciclo de relógio. É necessário mais que um processador para que os diferentes fluxos de instruções sejam executados. A Figura 2.3 exemplifica esta arquitetura.

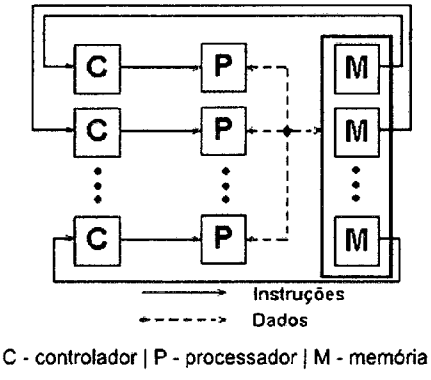

Figura 2.3: Arquitetura MISD

Aplicações capazes de usufruir desta arquitetura são raras. Um possível exemplo é a execução de múltiplos algoritmos de decriptografia sobre um mesmo dado criptografado ou a aplicação de vários filtros eletrônicos sobre um mesmo sinal recebido. A princípio não foram desenvolvidos sistemas que seguem este modelo arquitetural.

MIMD (Multiple Instruction Multiple Data)

Nas arquiteturas MIMD fluxos de instruções diferentes podem ser executados com diferentes fluxos de dados de forma assíncrona. Para este tipo de arquitetura é necessário mais de um processador para que os diferentes fluxos de instruções sejam executados. A Figura 2.4 exemplifica esta arquitetura.

As arquiteturas MIMD podem assumir outras divisões [48]. Estas divisões foram baseadas a partir de modelos de espaços de endereçamento e das arquiteturas de memória, como mostra a Figura 2.5.

Apresentamos a seguir as características de cada uma destas divisões da arquitetura MIMD.

Multiprocessadores (Memória compartilhada centralizada)

Esta subdivisão da arquitetura MIMD é composta por arquiteturas fortemente

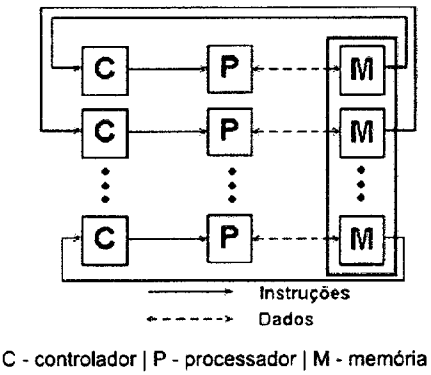

Figura 2.4: Arquitetura MIMD

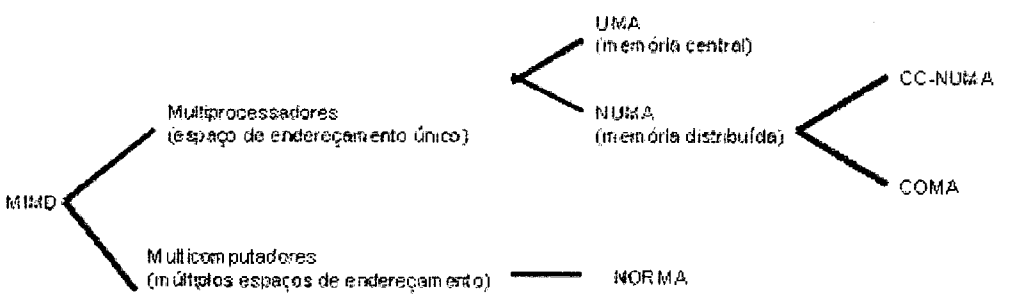

Figura 2.5: Divisões das arquitetura MIMD

acopladas. Um barramento liga as unidades de processamento a uma memória logicamente única, compartilhada e centralizada. O espaço de endereçamento é único.

A seguir, apresentamos as características das arquiteturas UMA e NUMA, pertencentes a esta categoria, e que se diferenciam pela localização da memória.

UMA (*Uniform Memory Access*)

Na arquitetura UMA a memória compartilhada encontra-se a mesma distância de todas as unidades de processamento.

Para amenizar o acesso a memória principal é utilizada uma memória *cache*. Isto diminui a necessidade de acesso a memória principal, que é mais dispendioso que o acesso a *cache*, além de reduzir a contenção de memória.

A Figura 2.6 exemplifica a arquitetura UMA.

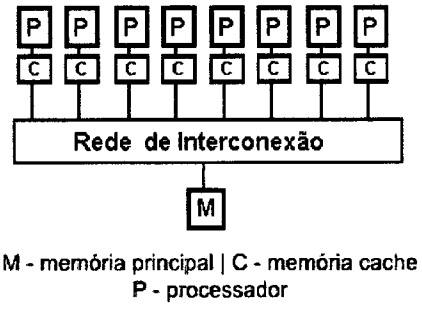

Figura 2.6: UMA - *Uniform Memory Access*

NUMA (*NON - Uniform Memory Access*)

Na arquitetura NUMA a memória é fisicamente distribuída e logicamente compartilhada. Cada processador tem sua memória privativa, existindo um controlador implementado em hardware responsável pela coerência dos dados compartilhados.

Nesta arquitetura o dado é endereçado a uma das memórias do sistema. Uma vez definido o endereçamento, ele não pode ser alterado, sendo considerado, então, como estático. Assim, o desenvolvimento destas arquiteturas é simplificado, mas pode resultar em queda de desempenho caso o dado seja endereçado a uma memória remota e o sistema não esteja otimizado para acessos remotos.

A Figura 2.7 exemplifica a arquitetura NUMA.

COMA (*Cache Only Memory Access*)

A arquitetura COMA é fisicamente semelhante a NUMA (Figura 2.7). A

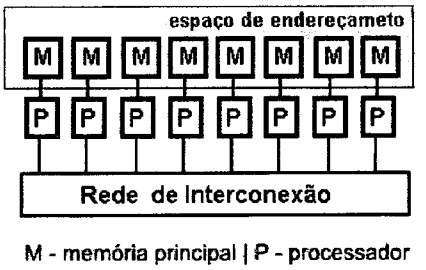

Figura 2.7: NUMA - *NON Uniform Memory Access*

diferença está no processo de endereçamento da memória que é dinâmico. Assim, o dado endereçado inicialmente a uma memória pode ser migrado para uma outra que, de acordo com o algoritmo de coerência utilizado, possa aumentar o desempenho do sistema por proximidade ao processador que fará uso mais eficiente deste dado.

CC-NUMA (*Cache Coherente - NON Uniform Memory Access*)

A arquitetura CC-NUMA segue o mesmo conceito arquitetural de COMA. A diferença está nas *caches* incluídas entre a memória principal e os processadores para aumentar o desempenho da arquitetura.

A Figura 2.8 exemplifica a arquitetura CC-NUMA.

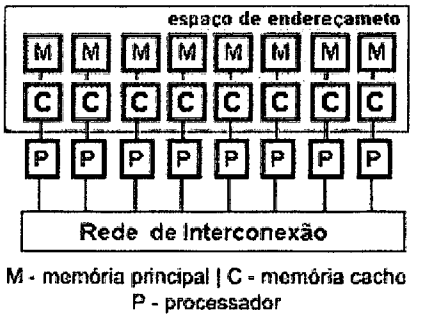

Figura 2.8: CC-NUMA - *Cache Coherente - NON Uniform Memory Access*

NORMA (Multicomputadores)

A arquitetura NORMA (*Non-Remote Memory Access*), também denominada multicomputadores, é uma subdivisão das arquiteturas MIMD (Figura 2.5). Ela é fracamente acoplada. Cada processador tem sua memória privativa. Os dados comuns a estas unidades são trocados a partir de uma rede de interconexão.

Como a memória é privativa é necessário que haja uma forma de obter os dados necessários ao processamento local em cada processador e entregar os dados requisitados por alguma outra unidade. Este mecanismo pode ser implementado em

software [28, 21], hardware ou híbrido [44].

Em geral, o desempenho de NORMA é menor quando comparado aos multiprocessadores. A rede de interconexão utilizada na comunicação nestas arquiteturas, normalmente, é mais lenta que os barramentos utilizados em sistemas multiprocessadores. O módulo controlador implementado em hardware em sistemas multiprocessadores, também, é mais eficiente que os implementados em software em sistemas multicomputadores.

Em contrapartida, a conexão estabelecida por redes torna a inserção de novos processadores bastante flexível, tornando o sistema escalável. Ao contrário dos sistemas multiprocessadores em que a comunicação por barramento não permite interconexão de outros processadores além dos pré estabelecidos. Mas as tecnologias de interconexão sofrem com as contenções da rede, tornando esta escalabilidade limitada.

Em termos de custo de aquisição os multicomputadores são mais acessíveis. O desenvolvimento em grande escala de PCs e estações de trabalho torna o custo menor. Sistemas multiprocessadores são desenvolvidos em uma escala bem menor e, normalmente, utilizam componentes de hardware específicos, tornando o custo de um sistema com o mesmo número de processadores bem maior.

2.2 Modelos de consistência

Os modelos de consistência têm por objetivo determinar quando os dados compartilhados devem estar presentes junto às unidades de processamento do sistema. O modelo de consistência dá subsídios ao protocolo de coerência para determinar em que momento o dado compartilhado deve ser enviado.

O modelo de consistência influi diretamente no desempenho do sistema.

Apresentamos a seguir uma evolução dos modelos de consistência.

Modelo de consistência seqüencial

O modelo de consistência seqüencial estabelece que as memórias do sistema devem operar como em um modelo monoprocessado. Toda alteração executada por um processador do sistema deve ser imediatamente observada pelas demais unidades. Desta forma, todo acesso à memória compartilhada, seja leitura ou escrita, só pode ser executado quando uma instrução de escrita anterior tiver sido visto pelas demais unidades do sistema.

Os sistemas que utilizam este modelo de consistência têm de executar constantemente os protocolos associados ao modelo, tornando o sistema ineficiente e causando um *overhead* nas redes de interconexão.

Modelos de consistência relaxados

O modelo de consistência seqüencial precisava ser otimizado para se obter melhor desempenho. A programação paralela tem características que permitiam executar corretamente os programas sem a necessidade de tornar o acesso aos dados compartilhados conhecidos por todas as unidades de processamento no mesmo instante em que é alterado.

Os modelos de consistência relaxados trabalham com estas características, como a necessidade de sincronização somente em determinados momentos. Existem vários modelos de consistência relaxados que atuam de forma diferente para aumentar o desempenho. Alguns deles são apresentados a seguir.

Processor consistency

O modelo *processor consistency* é semelhante ao modelo seqüencial. A diferença está nas alterações executadas por um determinado processador que não precisam ser imediatamente visualizadas pelas demais unidades do sistema logo após o acesso. Assim, somente a ordem de escrita está restrita à unidade que realizou a operação. As leituras em outras unidades de processamento podem ser executadas sem as restrições estabelecidas no modelo de consistência seqüencial, desde que sejam em regiões de memória compartilhada diferentes.

Desta forma, o sistema fica capacitado a operar com algumas técnicas para o aumento do desempenho, como *write buffers* e *pipelining*.

Release consistency

O modelo *release consistency* considera o fato de que o programador pode determinar em que ponto a sincronização dos dados é necessária. Para tanto são estabelecidas primitivas de sincronização que definem quando deve haver uma sincronização.

As primitivas mais comumente utilizadas são: *lock-acquire*, *lock-release*, e *barriers* (barreiras). As primitivas *lock-acquire* e *lock-release* estabelecem os pontos de sincronização e as seções críticas. Nessas regiões somente um processador está capacitado a executar por vez, protegendo, então, as variáveis compartilhadas associados a esta área.

O *lock-acquire* controla o acesso a dados compartilhados. Antes do início da

execução é verificado se alguma outra unidade de processamento tem a variável de *lock* sob sua custódia. Em caso positivo, a unidade requisitante espera a liberação do *lock*. A partir da primitiva *lock-release* a região protegida é liberada e é realizada a sincronização com os demais processadores. Ela é invocada quando é terminado o processamento dentro da região crítica por parte de um dos processadores.

As primitivas de barreiras são utilizadas quando as unidades de processamento precisam ter seu processamento emparelhado. Geralmente, são utilizadas para definir etapas. Elas impedem que a aplicação continue até que todos os processos tenham atingido a barreira.

As trocas de informações entre os elementos de processamento só ocorrem nas chamadas ao *lock-release* ou na saída de uma barreira. Este fato aumenta o desempenho, uma vez que diminui a quantidade de dados que percorrem a rede de interconexão e que acontecem nestes momentos, e não a cada acesso a memória compartilhada.

Lazy release consistency

O modelo *lazy release consistency* é semelhante ao modelo *release consistency*, porém, ainda mais relaxado.

Quando um dado é alterado não é necessário sincronizar imediatamente. Estes dados são enviados somente se a próxima unidade necessitar utilizá-los dentro de mesma região crítica (*acquirer*). Desta forma, existe redução da propagação de dados entre os processos e a diminuição do *overhead* da rede de interconexão.

Em contrapartida, o desenvolvedor tem a responsabilidade de estabelecer os pontos de sincronização para leitura, diferente dos outros modelos de consistência, porque este modelo garante que os dados estarão coerentes apenas durante a execução de um *lock-acquire*. Conseqüentemente, pontos de sincronização extras devem ser introduzidos no código, o que pode tornar o sistema deficiente.

2.3 Modelos de programação

Apresentamos, a seguir, alguns dos modelos de programação para arquiteturas paralelas. Estes são utilizados pelo programador para desenvolver os programas voltados a estas arquiteturas. Estes modelos definem o nível de complexidade do programa e pode influenciar inclusive no desempenho da aplicação.

Modelo de programação para memória compartilhada

O modelo de programação para memória compartilhada, em princípio, foi desenvolvido para arquiteturas de memória centralizada. Assemelha-se bastante ao modelo de programação seqüencial, tornando simples a tarefa do desenvolvedor. Dependendo do modelo de consistência utilizado é necessária a utilização das primitivas para a definição dos pontos de sincronização.

Os custos de desenvolvimento e manutenção dos programas com o modelo de programação de memória compartilhada são menores.

Entretanto, em relação ao custo de hardware, as máquinas de memória compartilhada são desenvolvidas em menor escala, tornando o custo de aquisição mais alto.

Modelo de programação para memória distribuída

O modelo de programação para memória distribuída é também conhecido por troca de mensagens. Quando necessário acessar dados coerentes, as alterações são enviadas por mensagens, exigindo um protocolo de comunicação controlado pelo programador.

O trabalho de desenvolvimento de programas paralelos pelo modelo de memória distribuída é mais árduo em comparação ao modelo de memória compartilhada. Isto porque é necessário declarar explicitamente os métodos para troca de dados. A manutenção, em geral, é mais complexa tornando os custos de desenvolvimento e de manutenção maiores.

Entretanto, em geral, os custos com aquisição de hardware são inicialmente menores, pois as máquinas são ligadas em rede e podem ser máquinas construídas em larga escala. Também oferecem uma escalabilidade maior, já que as redes de interligação permitem a inserção de novos processadores facilmente.

Discussão sobre o modelo de programação para memória compartilhada distribuída (DSM)

Sistemas de memória compartilhada distribuída tem por objetivo oferecer as facilidades encontradas no modelo de programação para memória compartilhada junto à simplicidade de aquisição de hardware e a escalabilidade encontradas nos sistemas de memória distribuída.

Em contrapartida, determinadas características tornam o modelo de programação ligeiramente difererente do modelo de programação para sistemas de memória compartilhada. Por exemplo, para a leitura de dados compartilhados é necessário que os dados estejam coerentes através da invocação de uma primitiva

de sincronização em sistemas DSM, já que a memória é fisicamente distribuída, ao contrário dos sistemas de memória compartilhada.

O modelo DSM realiza a troca de mensagens baseada em unidades de coerência estabelecidas pelo sistema, normalmente alguma utilizada pelo sistema operacional (páginas) ou pelo hardware (linhas de *cache*).

Devido às características específicas de cada aplicação, o desempenho com este modelo de programação provavelmente não é maior que o obtido no programa desenvolvido com o modelo de programação de memória distribuída da forma mais otimizada. Isto porque a responsabilidade pela migração dos dados é do sistema e não do programador. Como estes modelos utilizam uma unidade de coerência do sistema, a qual pode ser grande, pode ocorrer que nem todos os dados enviados sejam acessados, incrementando o *overhead* na rede de interconexão desnecessariamente.

2.4 Problemas associados a multiprocessadores

No desenvolvimento de sistemas multiprocessadores (Figura 2.5) é necessário avaliar quais os possíveis pontos que podem afetar o desempenho das aplicações. Os principais pontos são a memória e a rede de interconexão. Os problemas associados a estes elementos estão definidos a seguir.

Contenção de memória

A contenção de memória ocorre quando várias unidades de processamento acessam um mesmo *chip* de memória em um mesmo instante. Como este recurso é restrito, é necessária a serialização dos acessos, acarretando assim a contenção de memória.

Em sistemas multiprocessados os dados podem ser compartilhados por vários processos. O protocolo de coerência de memória pode estar restringindo o acesso a memória ou a alguma região dela para proteger os dados, possibilitando a perda de desempenho.

Contenção de comunicação

A contenção de comunicação ocorre por causa do número limitado de canais entre os processadores do sistema multiprocessado. Quando existe um número de requisições maior que a capacidade oferecida pelos canais existentes ocorre a contenção de comunicação.

Em sistemas multiprocessadores o canal de comunicação, seja ele um barramento

ou uma rede de interconexão, é limitado e compartilhado entre os vários processadores do sistema, possibilitando a contenção e perda do desempenho.

Latência de rede

Em um sistema multiprocessado os processadores estão interligados por uma rede de conexão. O tempo decorrente entre o envio de uma requisição de um processador ao outro é denominado latência de rede.

A latência de rede é extremamente influenciada pela tecnologia de rede de interconexão utilizada e pela quantidade de dados que trafegam por ela.

2.5 Classes de coerência de memória

Em sistemas paralelos existe a necessidade de determinar como os dados compartilhados serão acessados pelas diversas unidades de processamento. Para tanto são desenvolvidos protocolos que devem ser seguidos por todos os elementos da arquitetura paralela. Esta tarefa é denominada coerência de *cache*.

Em geral, estes protocolos podem pertencer a duas classes de coerência de *cache*: migração e replicação.

A técnica de migração envia o dado alterado por uma unidade de processamento a uma outra, enviando, também, a custódia da administração do dado. Quando outra unidade de processamento necessitar do dado ela requisitará ao nó que o tem sob custódia.

A técnica de replicação é definida quando um dado não é administrado por uma única unidade de processamento. O dado é replicado nas unidades de processamento que fazem parte do sistema.

Estes dois procedimentos são críticos e a sua utilização afeta o desempenho dos sistemas compartilhados. A classe de migração tem uma taxa de tráfego de dados menor que a encontrada na classe de replicação. O dado migra de um elemento de processamento diretamente para um outro quando requisitado. Na classe de replicação quando ocorre a alteração de um dado, ele deve ser enviado aos demais nós do sistema, aumentando o tráfego de dados pela rede.

Sistemas atuais utilizam a classe de replicação, pois as redes de interconexão são lentas, tornando o desempenho extremamente deficiente quando necessário o uso constante da rede para acesso aos dados compartilhados e espera pela chegada destes, o que ocorre no caso das classe de migração.

Para sistemas multiprocessados, ainda existem duas classes de coerência de memória: *directory-based* [29] e *snooping* [27].

Em *directory-based* o estado do compartilhamento do bloco de memória compartilhado está presente em uma estrutura física denominada *directory*. Em *snooping* toda *cache* tem uma cópia dos blocos de memória compartilhados, tornando o estado de compartilhamento não centralizado.

Snooping diminui os possíveis gargalos existentes no acesso a estrutura *directory* existente na classe *directory-base*. Em contrapartida, aumenta o fluxo de dados na rede de conexão. *Snooping*, também, aumenta a escalabilidade, já que uma estrutura *directory* limita o número das memórias *cache* que estão distribuídas no sistema.

2.6 Protocolos de coerência de memória

Em sistemas paralelos para manter os dados coerentes é necessário um conjutno de regras que mantenham atualizados os dados compartilhados nos elementos de processamento. Para tanto são desenvolvidos os protocolos de coerência de memória.

Estes protocolos influenciam diretamente no desempenho do sistema. As influências dos protocolos dependem das características da aplicação e das tecnologias de rede utilizadas.

A seguir apresentamos os principais protocolos de coerência de memória: invalidação, atualização e híbrido.

2.6.1 Invalidação (*invalidate*)

No protocolo de invalidação [22] quando uma unidade de processamento executa uma operação de *release*, as demais unidades são informadas, através de mensagens, quais elementos foram alterados. Não são enviadas as modificações propriamente ditas.

Em uma operação de *acquire* o processador é responsável por requisitar os dados que estão indicados como alterados às demais unidades do sistema. Este protocolo tem como vantagem a diminuição da transferência de dados na rede de interconexão. Entretanto, quando for necessária a aquisição de dados modificados o processador tem que esperar até que os dados sejam enviados e atualizados localmente, ficando em modo de espera.

2.6.2 Atualização (*update*)

No protocolo de atualização [34] quando uma unidade de processamento executa uma operação de *release* ela repassa às demais unidades todos os dados alterados.

A vantagem deste protocolo é a presença imediata dos dados quando um processador acessa o dado compartilhado. Dessa forma o protocolo de atualização elimina o período de espera pelos dados remotos.

Este protocolo tem como desvantagem o *overhead* na rede de conexão, que aumenta consideravelmente em relação ao protocolo de invalidação. Além disso, várias das alterações recebidas não são acessadas pelos processos, tornando muitas destas atualizações desnecessárias.

2.6.3 Híbrido (*competitive update*)

O protocolo híbrido [27] visa encontrar um equilíbrio das vantagens proporcionadas pelos protocolos de invalidação (menor tráfego de dados pela rede de conexão) e de atualização (menor tempo de espera pelos dados remotos), se adaptando melhor as necessidades da aplicação.

Ele se baseia na heurística de que se um dado não for acessado durante n atualizações, onde n é o valor do *threshold*, então, aquele dado pode ser invalidado. Ou seja, não é mais útil ao processador nos próximos tempos.

Após a atualização de um dado, o protocolo de atualização volta a atuar nele durante os n *thresholds*.

A maior parte dos processadores atuais utiliza protocolo de invalidação, porque a atualização produz uma quantidade excessiva de mensagens que pode causar contenção na rede de interconexão. Multiprocessadores baseados em barramento que usam o processador DEC Alpha AXP21064 [49] utilizam protocolo híbrido.

2.7 Tipos de *miss*

Quando a memória *cache* é acessada, é possível que um dado, que esteve presente anteriormente, não esteja mais presente naquele momento. É necessário, então, requisitá-lo em outro nível de *cache* ou na memória principal. À ausência do dado na *cache* denominamos *cache miss* (falha na *cache*).

Em sistemas onde os dados são compartilhados e as memórias são distribuídas pode haver outros tipos de *miss*, além dos já existentes em sistemas

monoprocessados. A seguir estão definidos os tipos de *miss* e suas características.

2.7.1 *False miss*

Um *false miss* (falha por falso compartilhamento) ocorre quando o sistema entende que uma unidade (página, bloco de memória ou linha de *cache*) da memória não está coerente em relação ao sistema compartilhado. Porém o dado acessado, que é uma parte da unidade, não estava incoerente.

Isto ocorre principalmente quando o sistema utiliza unidades de coerência com um tamanho considerável. Aumentando o tamanho da unidade aumenta também a probabilidade de existirem dados que não foram alterados. Isso porque a unidade possui maior capacidade de armazenar dados.

A existência de *false miss* pode gerar aumento desnecessário do tráfego na rede e o aumento da latência pela espera dos dados remotos.

2.7.2 *True miss*

Um *true miss* ocorre quando um dado local não está coerente em relação ao sistema compartilhado. É necessário requisitar o dado modificado mais recentemente.

Neste tipo de falha é necessário requisitar os dados. Assim, podemos usar o *true miss* como parâmetro para avaliar o quanto o protocolo de coerência está sendo eficiente em diminuir o número destas falhas.

2.7.3 *Eviction miss*

Eviction miss (ou *replacement miss*) ocorre quando um sistema troca linhas na *cache* e posteriormente precisa acessá-las novamente.

Em sistemas hardware DSM, o *eviction miss* pode comprometer o sistema tanto quanto o *false miss*. A princípio, uma solução é aumentar o tamanho da memória *cache* em harware.

2.7.4 *Cold start miss*

Cold start miss ocorre na primeira referência a um bloco da *cache*. Isto é, a falha ocorre porque o bloco nunca esteve na memória, por estar no início da execução.

Este dado é importante para determinar o quanto o sistema está capacitado a melhorar o desempenho a partir do comportamento da *cache*, já que estas falhas são obrigatórias.

2.7.5 *Drop miss*

Drop miss ocorre quando o processador referencia uma palavra que foi invalidada pelo protocolo híbrido quando ele atua como protocolo de invalidação.

Capítulo 3

Programação Lógica-Paralela

Neste capítulo apresentamos conceitos básicos de sistemas paralelos de programação lógica, mecanismos de extração de paralelismo implícito, tipos de paralelismo implícito e modelos de implementação. Apresentamos, também, alguns sistemas que exploram paralelismo em programação lógica, enfocando no sistema Andorra-I, base para o estudo realizado nesta tese.

3.1 Conceitos Básicos

Computação lógica segue o paradigma da programação declarativa, onde o algoritmo é definido por um conjunto de regras [33, 31]. Estas regras expressam relações condicionais a partir de uma ou mais cláusulas. A partir de uma base de dados que contém fatos - cláusulas sempre verdadeiras - o processamento é realizado sendo possível gerar as conclusões desejadas.

Nesta metodologia o programador se concentra na descrição do problema, diminuindo consideravelmente o tempo devotado aos mecanismos de computação, presente em outros modelos de programação, como o funcional ou imperativo.

Os programas gerados por este modelo são mais simples de desenvolver e compreender. Estas são características importantes em sistemas com alto nível de complexidade, presentes em áreas como Inteligência Artificial e Biotecnologia. A linguagem de programação mais conhecida que segue este modelo é o Prolog [11, 13, 6, 9].

Uma característica deste modelo de programação é a capacidade de paralelizar seus programas de forma implícita [37], não exigindo intervenção no código por parte do programador. Outros modelos de programação, como o funcional ou imperativo, exigem a participação do desenvolvedor na definição do paralelismo.

A paralelização implícita é possível já que as regras que constituem o programa lógico são constituídas por cláusulas e estas são constituídas por termos que se relacionam. Cada qual pode ter seu processamento distribuído em processadores de uma arquitetura paralela. Dentro destas características é possível dividir a metodologia de paralelização em dois tipos: paralelismo -*OU* e paralelismo -*E*.

O paralelismo do tipo -*OU* está associado ao processamento de diferentes cláusulas que correspondem à definição do problema. Estas cláusulas podem ser processadas de forma independente. Por exemplo:

```
(1) tio(A,B) :- pai(A,X), irmao(X,B).
(2) tio(A,B) :- mae(A,X), irmao(X,B).
```

Figura 3.1: Exemplo de algoritmo de programação lógica - definição de tio

Para avaliar a regra *tio(X,Y)* o processamento da cláusula (1) do algoritmo apresentado na Figura 3.1 pode ser delegado a um processador diferente daquele que executará a cláusula (2) deste mesmo algoritmo.

O paralelismo do tipo -*E* visa executar os diferentes termos de uma mesma cláusula paralelamente. Por exemplo:

```
(3) irmao(A,B) :-
      pai(A,X), pai(B,X),
      mae(A,Y), mae(B,Y).
```

Figura 3.2: Exemplo de algoritmo de programação lógica - definição de irmão

Na cláusula (3) do algoritmo descrito na Figura 3.2 a regra *irmao* é composta por quatro termos que inicialmente podem ser avaliados em processadores diferentes.

O paralelismo do tipo -*E* pode, ainda, ser subdividido em outras duas categorias: o paralelismo -*E* dependente (DAP) e o independente (IAP).

Em DAP os termos que compõem a cláusula compartilham variáveis comuns, de tal forma que é necessário um pré-processamento de um dos termos para capacitar a execução do próximo. Esta situação ocorre quando a resposta obtida de um dos termos é utilizada como base de avaliação ao outro termo na mesma cláusula. Exemplificamos o paralelismo DAP pela cláusula (1) do algoritmo descrito na Figura 3.1, onde para executar o termo *irmao* é necessário que a variável X tenha o resultado obtido da cláusula *pai*. Este tipo de execução caracteriza um padrão produtor-consumidor.

Em IAP os termos que compõem a cláusula podem ser excecutados independentemente, pois ainda que os termos compartilhem as variáveis estas não servem de base para a execução dos demais. Um exemplo de paralelismo *-E* independente é mostrado no exemplo da Figura 3.3.

```
quicksort([X|L],S) :-
        partition(X,L,L1,L2),
        quicksort(L1,S1),
        quicksort(L2,S2),
        append(S1,[X|S2],S).
```

Figura 3.3: Exemplo de algoritmo de programação lógica - *quicksort*

Esta figura mostra uma parte da solução que ordena elementos em uma lista utilizando o algoritmo *quicksort*. Neste pequeno programa, particionamos a lista de entrada `L`, utilizando o primeiro elemento da lista `X` como pivô, e geramos duas novas listas, `L1` (contém todos os elementos de `L` menores ou iguais que `X`) e `L2` (contém todos os elementos maiores do que `X`). Em seguida, as ordenações de `L1` e `L2` podem ser feitas em paralelo, dando origem ao paralelismo *-E* independente.

Sistemas foram desenvolvidos para aproveitar o paralelismo *-OU* como o Aurora [8, 10] e MUSE [1]. Outros sistemas aproveitam o paralelismo do tipo *-E* independente, como o &-Prolog [25], e outros o paralelismo do tipo *-E* dependente, como o Parlog [16]. Existem ainda sistemas que estão capacitados a aproveitar os dois tipos de paralelismo como o Andorra-I [3].

Para executar um programa lógico em uma arquitetura paralela precisamos de um *framework*, sendo este composto por dois módulos principais: a máquina virtual Prolog e o escalonador de tarefas.

A máquina virtual Prolog tem a função de executar as tarefas lógicas. A função do escalonador é distribuir eficientemente as tarefas entre os vários processadores da arquitetura paralela, de forma que a granulosidade das tarefas não seja tão pequena, o que aumentaria a comunicação na rede, e nem tão grande, o que possiblitaria que alguma máquina se tornasse ociosa. O escalonador, também, é capaz de otimizar o processo de execução, fazendo com que tarefas associadas às condições que não tenham sido satisfeitas não sejam executadas.

3.2 Andorra-I

Andorra-I [42] é um sistema capaz de executar programas lógicos desenvolvidos em Prolog e distribuir as tarefas entre diversos processadores de uma arquitetura paralela. A principal diferença quando comparado a outros sistemas já desenvolvidos é a capacidade de explorar os dois tipos de paralelismo, presentes em aplicações reais e conseqüentemente obtendo boas taxas de desempenho.

Andorra-I estrutura o programa lógico como uma árvore. Cada nó desta árvore corresponde a uma tarefa a ser executada, sendo esta uma cláusula ou um termo de uma cláusula. Os nós que se seguem a este são as tarefas derivadas do nó que o gerou. Quando os resultados são gerados estes são repassados ao nó que os gerou, sendo esta tarefa denominada *backtracking*. A Figura 3.4 apresenta um exemplo de árvore de tarefas.

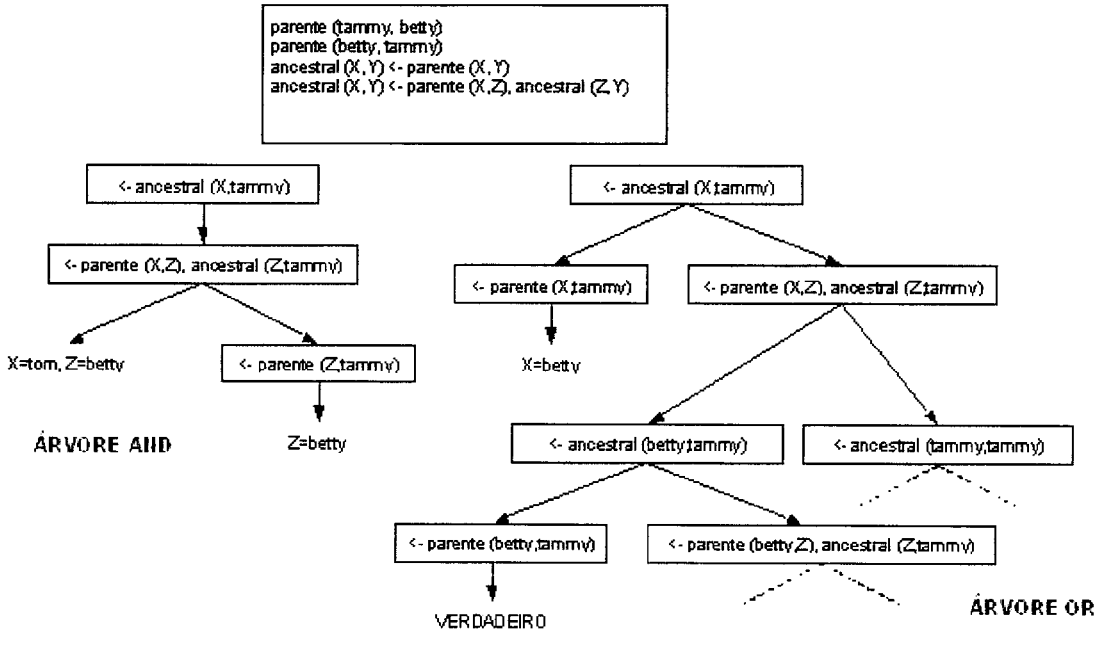

Figura 3.4: Exemplo de árvores de tarefas *-E* e *-OU*

A partir desta estrutura o sistema é capaz de determinar quais são as tarefas *-OU* e *-E*. Na árvore *-E* da Figura 3.4 os termos *parente* e *ancestral* da regra *ancestral* são separados. Já na árvore *-OU* são separadas as duas cláusulas referentes a regra *ancestral*.

A partir desta estrutura os escalonadores são capazes de determinar quais são as tarefas a serem distribuídas pelos processadores e como devem ser executados de acordo com o tipo de paralelismo.

Em Andorra-I cada processador é denominado *worker*, sendo este capaz de executar as tarefas lógicas delegadas pelo escalonador. Os *workers* podem ser agrupados em *teams*, onde cada *team* tem um processador *master*. Dentro de um *team* os *workers* cooperam de forma a explorar o paralelismo do tipo *-E*, enquanto os *teams* cooperam entre si de forma a explorar o paralelismo do tipo *-OU*.

O escalonador administra uma área de memória comum a todos os processos, onde devem estar presentes os resultados obtidos das tarefas delegadas a outros processadores e as tarefas que ainda devem ser executadas.

Associado a cada processador existe, também, uma área de memória privativa. Ela contém os dados referentes às tarefas atribuídas ao processador.

A Figura 3.5 apresenta a estrutura seguida pelo sistema Andorra-I.

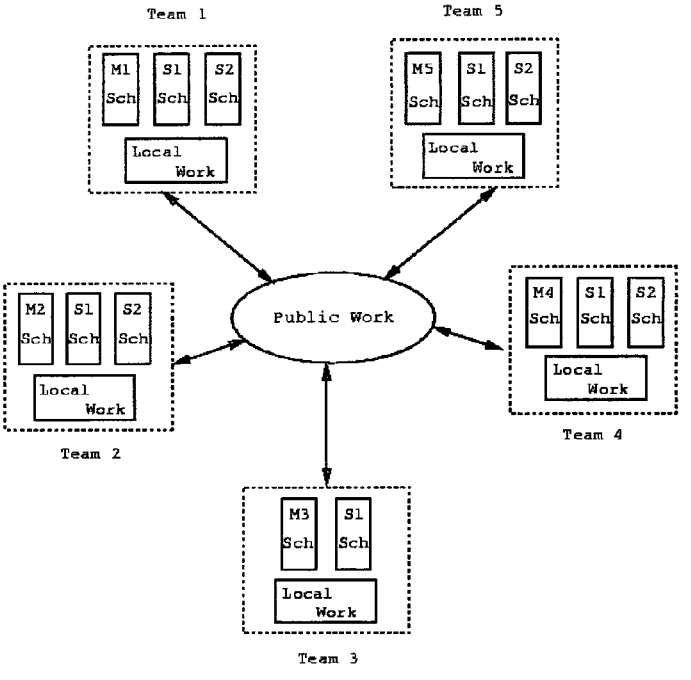

Figura 3.5: Estrutura do sistema Andorra-I

O modelo estabelecido por Andorra-I para aproveitar o paralelismo *-OU* é o SRI [55], onde cada *team* coopera com o escalonador na tarefa de procura de tarefas a serem processadas, estas associadas a uma sub-árvore da estrutura citada anteriormente. Este escalonador foi desenvolvido na universidade de Bristol [2].

Andorra-I divide a memória compartilhada em vários segmentos com diferentes funcionalidades. Esta divisão permite a melhor administração dos dados de acordo com as tarefas do *framework*, como a sincronização das tarefas. Descrevemos estas

áreas a seguir, incluindo suas funcionalidades.

Estrutura de dados do escalonador-OU. O escalonador *-OU* de Bristol utiliza três estruturas compartilhadas: um conjunto de campos para cada ponto de escolha e duas estruturas de dados com variáveis globais. Esta região de memória engloba estas estruturas.

Estrutura de dados das tarefas (workers). Estruturas necessárias para controlar as variáveis associadas ao controle das tarefas a serem executadas. O acesso a esta estrutura está normalmente relacionado ao escalonamento *-E*.

Vetor de locks. Usado para estabelecer a relação entre uma determinada região de memória e uma variável de *lock*. O acesso é realizado sempre que há a necessidade de um *lock*. Esta área é necessária nas simulações realizadas neste trabalho, já que o simulador não realiza as instruções de sincronização do MIPS. A estrutura utiliza o mecanismo de *hashing* para diminuir o efeito de perda de desempenho.

Espaço de código. Espaço para o código compilado. Durante a execução são realizados acessos somente de leitura.

Espaço do heap. Utilizado de forma comum em sistemas Prolog. Armazena estruturas e variáveis. Ele aumenta durante o avanço da execução da tarefa e diminui no *backtracking* referente ao término de um objetivo.

Espaço de objetivos (goal frame). Este espaço armazena argumentos para desenvolver os objetivos, variáveis para ligar os vários objetivos e campos de controle críticos. O sistema tenta reutilizar os objetivos conforme a execução.

Pilha do ponto de escolha (choicepoint). Armazena os ponteiros para o topo das pilhas e *flags* que são alterados de acordo com a execução do sistema.

Pilha de trail. Região acessada somente por aplicações que tenham tarefas do tipo *-OU*. Contêm variáveis com múltiplas atribuições e condições referentes ao *binding*.

Binding array. Região acessada somente pelas aplicações que tenham tarefas do tipo *OU*.

Outras variáveis compartilhadas. Armazena outras variáveis compartilhadas, como descritores de arquivos.

Capítulo 4

Metodologia de Avaliação

Este trabalho visa avaliar o comportamento de sistemas de programação lógica paralela variando os protocolos de coerência de memórias compartilhadas, complementando trabalhos já existentes como [45] e [14]. A maioria dos trabalhos apresentados até então tem avaliado o comportamento de aplicações científicas.

Procuramos observar se o protocolo híbrido pode gerar os melhores desempenhos em sistemas lógicos paralelos, já que ele procura oferecer o melhor dos outros dois protocolos: invalidação e atualização.

A avaliação é feita em sistemas multiprocessadores. Isso porque as características de desempenho dos multiprocessadores são mais adequadas para sistemas de programação lógica paralela. Trabalhos anteriores indicam que arquiteturas de sistemas multicomputadores ainda precisam ser aprimoradas para comportar sistemas de programação lógica paralela [19], pois existem perdas de desempenho constatadas.

Outros trabalhos mostram que o desempenho em sistemas multicomputadores podem ser otimizados por novas técnicas, mas não obtiveram uma resposta mais significativa que as encontradas em sistemas multiprocessadores. Principalmente devido à tecnologia de rede [12].

Por estes motivos concentramos os esforços em arquiteturas multiprocessadas, apesar da grande flexibilidade encontrada em sistemas de memória distribuída.

4.1 Simulador MINT

Utilizamos um simulador orientado à execução, que viabiliza a execução de aplicativos compilados para uma arquitetura MIPS R3000 multiprocessada, DASH-like [30], com 24 processadores, diretamente conectados. Cada nó da máquina

simulada contém um único processador, um *buffer* de escrita, uma *cache*, memória local, um diretório *full-map* e uma rede de interconexão. Este simulador foi desenvolvido na universidade de Rochester e utiliza como *front-end* o MINT (Mips INTerpreter), desenvolvido por Veenstra e Fowler [52, 51]. Ele simula a arquitetura MIPS e gera referências à memória, às quais são repassadas ao *back-end*, desenvolvido por Bianchini, Kontothanassis e Veenstra [5] para simular os sistemas de memória e de interconexão. Quando o módulo *back-end* termina a tarefa ele envia um sinal ao *front-end*, indicando o resultado e que pode continuar a tarefa.

As aplicações a serem executadas sobre o simulador devem ser compiladas para a arquitetura MIPS R3000 e os endereços devem ser gerados estaticamente.

Nas simulações realizadas cada elemento de processamento foi definido como tendo uma *cache* de 512 kbytes mapeada diretamente, com blocos de 64 *bytes*. Todas as instruções e acessos a *cache* realizados com sucesso duram 1 ciclo de relógio. Falhas de leitura suspendem o processador até que a leitura seja completada. Escritas são armazenadas em um *buffer* de escrita com 16 entradas e duram 1 ciclo de relógio. A exceção encontra-se quando o *buffer* está cheio. Neste caso, o processador espera até que exista uma entrada livre. Leituras podem passar a frente de escritas no *buffer*. Os dados compartilhados são distribuídos pelas memórias de forma intercalada no nível de bloco (64 bytes).

Um barramento de memória com metade da velocidade de processamento conecta os principais componentes de cada nó da máquina. Uma nova operação no barramento pode começar a cada 34 ciclos de processador. Um módulo de memória pode prover a primeira palavra de uma linha de *cache* 20 ciclos após o pedido ser feito. Cada palavra subseqüente é enviada a cada 2 ciclos.

A rede de interconexão é uma grade bidirecional *wormhole-routed*, com roteamento ordenado por dimensão. A velocidade da rede é a mesma do processador. Nós de chaveamento introduzem um atraso de 4 ciclos ao cabeçalho de cada mensagem. A largura da rede é de 16 bits, equivalente a largura de banda de memória.

Estes parâmetros são considerados coerentes para arquiteturas paralelas atuais. Os Apêndices A e B apresentam os parâmetros que podem ser simulados pelo MINT.

Nossas simulações foram realizadas com 1, 2, 4, 8 e 16 processadores.

Os protocolos utilizados nas simulações são os de invalidação, atualização e híbrido. O protocolo híbrido é executado com os seguintes valores de *threshold*:

1, 2, 4 e 8. Nosso protocolo de invalidação utiliza o protocolo DASH com *release consistency* [29].

Nas figuras mostradas neste capítulo, o protocolo de invalidação apresenta taxas de falhas para as categorias *true*, *false*, *cold* e *eviction*. Esta categorização utiliza o algoritmo descrito em [17] e estendido por Bianchini [5]. O protocolo de atualização apresenta taxas de falha apenas para as categorias *cold* e *eviction*. O falso compartilhamento no protocolo de atualização é capturado através de uma outra categorização que classifica mensagens de atualização em *useful* ou *useless*. As mensagens de atualização têm a seguinte classificação:

- ***True sharing updates.*** O processador que recebe a atualização acessa a palavra atualizada antes que outra mensagem de atualização para a mesma palavra chegue ao processador;
- ***False sharing updates.*** O processador que recebe a atualização não acessa a palavra atualizada antes desta ser sobrescrita por outra atualização, mas acessa outra palavra no mesmo bloco de *cache*;
- ***Proliferation updates.*** O processador que recebe a atualização não acessa a palavra atualizada antes dela ser sobrescrita por outra atualização, e, também, não acessa outra palavra do mesmo bloco;
- ***Replacement updates.*** O processador que recebe a atualização não acessa a palavra atualizada até que o bloco é substituído de sua *cache*;
- ***Termination updates.*** Esta é uma mensagem de proliferação (*proliferation update*) que ocorre no término do programa.

Esta classificação utiliza o algoritmo descrito em [5]. É uma categorização simples, exceto pela classe de *false updates*. Sucessivas atualizações para a mesma palavra na *cache* são classificadas como proliferação, se o processador não estiver acessando outras palavras no mesmo bloco. Se o falso compartilhamento estiver realmente acontencendo, ou seja, o processador recebe sucessivas atualizações para o mesmo bloco, mas acessa outra palavra no bloco, estes *useless updates* são classificados como falsos.

O protocolo híbrido utiliza as mesmas classes do protocolo de atualização, porém inclui a classe *drop misses*, como descrito na Seção 2.7.

4.2 Aplicações e suas características

As aplicações utilizadas neste trabalho possuem somente paralelismo *-E*, somente paralelismo *-OU*, ou uma combinação das duas formas. Portanto, seus padrões de comportamento são variados. Estas são aplicações normalmente utilizadas para medir desempenho de sistemas paralelos de programação lógica.

4.2.1 Problema do caixeiro viajante (tsp)

A aplicação TSP (*Traveller Salesperson Problem*) possui paralelismo do tipo *-E*. É um problema clássico, onde é calculada a rota com menor distância entre vários pontos, sem que um ponto esteja presente duas vezes em uma mesma rota.

Esta aplicação encontra a solução aproximada do problema e é baseada numa solução escrita em Reform Prolog [4].

Nos experimentos utilizamos um grafo com 24 pontos, onde cada ponto representa uma cidade e as interconexões entre estes indicam as possíveis rotas.

4.2.2 Gerenciamento de redes da British Telecom (bt)

A aplicação BT possui paralelismo do tipo *-E*. Essa aplicação implementa um algoritmo de agrupamento (*clustering*) da Britsh Telecom. Seu objetivo é agrupar os pontos cuja distância entre si seja menor que um determinado limite. Nos experimentos são utilizados 400 pontos.

4.2.3 Sistema de pergunta-resposta usando linguagem natural (chat)

A aplicação CHAT apresenta somente paralelismo do tipo *-OU*. Utilizamos a versão implementada por Pereira e Warren [54] que executa uma consulta para acesso à base de dados geográfica chat-80.

Esta aplicação foi utilizada inicialmente na avaliação dos sistemas Aurora [7] e Muse [1].

4.2.4 Sistema de alocação de recursos Pandora (pan2)

A aplicação PAN2 possui paralelismos do tipo *-E* e do tipo *-OU*. É uma aplicação real de controle de recursos aéreos. Por exemplo, o controle de porta-aviões associado a um conjunto de aeronaves.

Paralelismo do tipo *-OU* é explorado quando é feita a busca pelas várias possibilidades de alocação que possam existir. Paralelismo do tipo *-E* é explorado quando se executa várias restrições em paralelo.

O grau de paralelismo *-E* e *-OU* desta aplicação depende da consulta. Na consulta utilizada nos experimentos, o paralelismo *-E* é predominante. A consulta feita ao programa requer a alocação de 11 aeronaves, 36 membros da tripulação e 10 vôos.

A Tabela 4.1 apresenta as características de paralelismo e o *speedup* esperado pelas aplicações utilizadas neste estudo.

Aplicação	Tipo de paralelismo	*Speedup* esperado em 16 processadores
BT	*-E*	entre 14 e 15
TSP	*-E*	aproximadamente 12
PAN2	*-E* e *-OU*	entre 8 e 10
CHAT	*-OU*	aproximadamente 10

Tabela 4.1: *Speedup* esperado para cada aplicação simulada

Estas aplicações foram escolhidas baseadas na variedade e quantidade de paralelismo. A terceira coluna da Tabela 4.1 indica o *speedup* esperado para cada aplicação baseado no seepdup obtido quando rodando o simulador MINT configurado como uma máquina onde o custo de acesso à memória é desprezível.

Em um trabalho anterior [41], mostramos o padrão de acesso à memória exibido por três destas aplicações no que diz respeito às áreas de dados de Andorra-I (mencionadas na Seção 3.2). A Tabela 4.2 mostra a porcentagem de referências às áreas compartilhadas de Andorra-I, com relação ao total de referências à memória compartilhada, de aplicações que contém paralelismo *-E* (`BT`), paralelismo *-OU* (`CHAT`) e combinação de paralelismo *-E* e paralelismo *-OU* (`PAN2`).

Este estudo foi realizado com a intenção de estabelecer a importância relativa de cada seção de dados, durante a execução das aplicações em 16 processadores, e, consequentemente, caracterizar mais detalhadamente o padrão de acesso à memória dependendo do tipo de paralelismo explorado. Este número de processadores foi escolhido para possibilitar o estudo do tráfego em uma situação onde o sistema está iniciando a saturação. Os resultados na tabela variam significativamente para as diferentes áreas de dados dependendo do tipo e da quantidade de paralelismo na aplicacação. A maior parte das referências em BT estão concentradas nas áreas de

Área	BT	CHAT	PAN2
OrSched	1.7	58.8	3.0
Worker	19.3	7.4	33.4
Locks	3.1	1.8	3.6
Code	39.7	10.2	18.0
Heap	7.6	1.7	4.0
Goals	12.6	3.6	6.8
ChoicePt	0.0	7.8	0.1
Trail	0.1	2.3	0.2
BA	6.2	3.3	20.0
Misc	8.8	3.5	9.4

Tabela 4.2: Referências por área compartilhada (%)

dados `Code`, `Worker`, `Goals`, e `Misc`. O alto número de referências para áreas de dados da máquina abstrata, como `Code` e `Goals`, indica que os processadores estão executando código relacionado à máquina abstrata durante a maior parte do tempo de execução. Isto sugere que existe paralelismo suficiente para 16 processadores. Em CHAT, as áreas mais referenciadas são `OrSched`, `Code`, `ChoicePt`, e `Worker`. Este comportamento indica que não existe trabalho suficiente para ocupar os 16 trabalhadores. Portanto, a maior parte do tempo é gasto no escalonador *-OU*. O *benchmark* PAN2 mostra pouco paralelismo *-OU* (quase nenhum acesso às áreas `ChoicePt` ou `Trail`) e uma porcentagem significativa de acessos às áreas relacionadas ao escalonador *-E*: `Worker` e `Misc`. Este fato confirma que PAN2 é um *benchmark* que contém predominantemente paralelismo *-E*. Porém, o paralelismo não é suficiente para 16 trabalhadores. Da mesma forma que em BT e CHAT, a área `Code` é, também, altamente referenciada em PAN2.

Capítulo 5

Resultados

Apresentamos neste capítulo os resultados obtidos a partir de várias simulações.

A avaliação é realizada através de medidas de *speedup*, taxas de falha nas *caches* e taxas de tráfego na rede de interconexão. Para cada aplicação apresentamos gráficos de *speedup* para os protocolos de invalidação, atualização e híbrido com *thresholds* variáveis.

A Tabela 5.1 mostra os tempos de execução, expresso em número de ciclos de execução, para um processador.

Applicação	Tempo de execução (ciclos)
BT	355.118.105
TSP	312.304.759
PAN2	49.677.419
CHAT	12.042.968

Tabela 5.1: Tempo de execução das aplicações simuladas com um processador

O primeiro gráfico apresentado para cada aplicação avalia o *speedup*. É apresentado de acordo com o número de processadores e o protocolo utilizado. No caso do protocolo híbrido, apresentamos resultados para *thresholds* variáveis.

O segundo gráfico apresenta os ganhos percentuais de um protocolo sobre aquele que teve menor desempenho nas simulações com um mesmo número de processadores. Tem por objetivo quantificar o desempenho dos protocolos avaliados.

O terceiro gráfico apresenta o percentual de falhas de acesso de cada aplicação. As variáveis de avaliação são o número de processadores e os protocolos de coerência de *cache* com seus respectivos *thresholds*. A avaliação é categorizada pelos tipos de *miss* (descritos na Seção 2.7). Quando necessário apresentamos, também, gráficos relativos à quantidade de *useless updates*, para os protocolos de atualização e híbrido.

O quarto gráfico apresenta o percentual de *useless update*, com as leituras não utilizadas pelas aplicações no protocolo escolhido na simulação.

O quinto gráfico apresenta o comportamento do uso da rede. Dentro de cada avaliação, ainda utiliza-se a seguinte categorização: requisição de coerência, mensagens de coerência e dados.

5.1 Aplicação BT

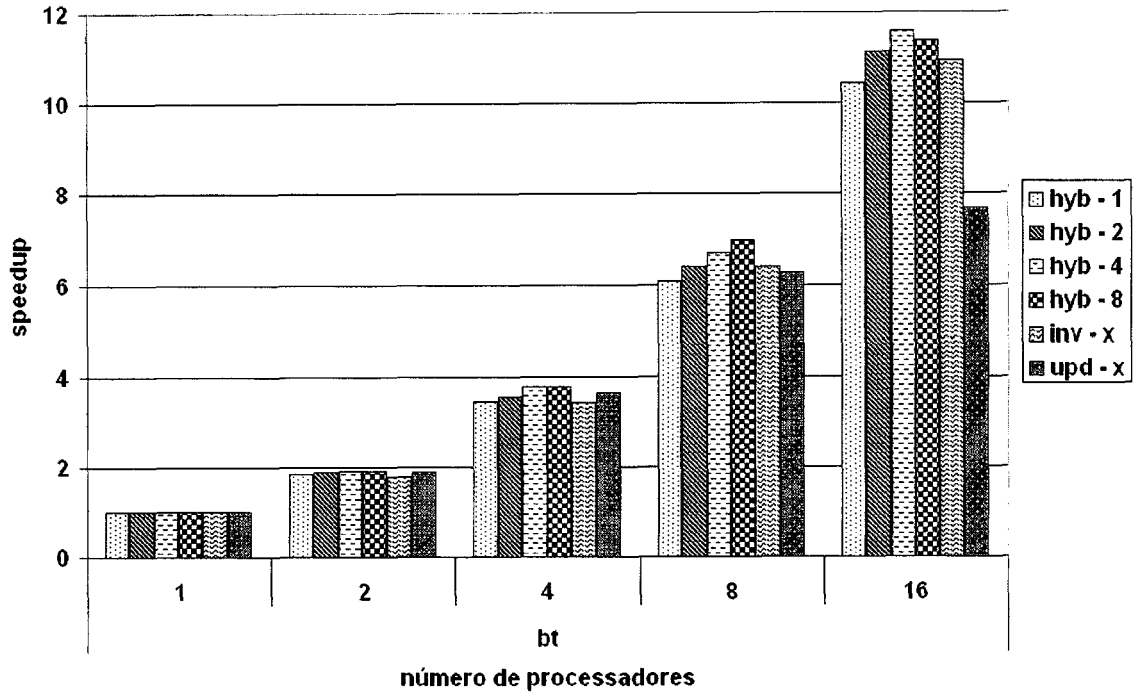

Figura 5.1: *Speedup* da aplicação BT, *cache* de 512 Kbytes

A aplicação BT tem uma quantidade significativa de paralelismo -*E* capturada pelo sistema paralelo de programação lógica que, para a arquitetura simulada, produz *speedups* entre 7,8 (protocolo de atualização) e 11,8 (protocolo híbrido com *threshold* igual a 4) para 16 processadores. A diferença entre estes dois valores é superior a 50%.

Observa-se na Figura 5.1 que o desempenho do protocolo de atualização supera o de invalidação nas simulações executadas com 2 e 4 processadores. Esta situação se inverte nas simulações com 8 e 16 processadores, por causa do aumento excessivo de tráfego introduzido pelo protocolo de atualização, com o aumento do número de processadores. Figura 5.5 confirma este fato e mostra que o tráfego na

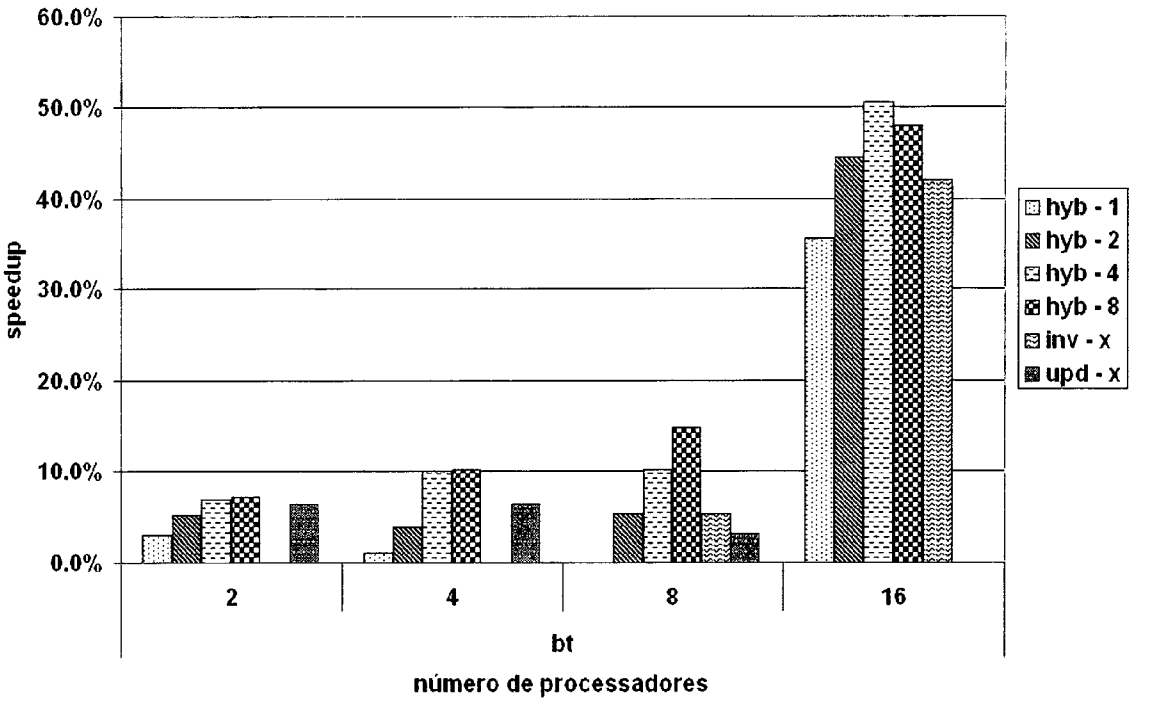

Figura 5.2: Comparativo de ganhos percentuais da aplicação BT em relação ao *speedup* de menor desempenho, *cache* de 512 Kbytes

rede de interconexão aumenta exponencialmente com o aumento do número de processadores, para o protocolo de atualização. O protocolo híbrido, por sua vez, causou um aumento moderado do número de mensagens com o aumento do número de processadores, mantendo uma margem de desempenho muito boa quando comparada com o protocolo de atualização.

Para todos os números de processadores, o protocolo híbrido comporta-se melhor do que o protocolo de invalidação ou de atualização. Por exemplo, para 4 processadores o melhor protocolo foi o híbrido com *threshold* igual a 4. Enquanto para 8 processadores o melhor protocolo foi novamente o híbrido, mas com *threshold* igual a 8. Para esta aplicação, que apresenta paralelismo *-E*, o protocolo híbrido mostrou o melhor resultado, embora o protocolo de invalidação, também, tenha produzido bons resultados.

A Figura 5.2 mostra que a escolha certa do protocolo pode influenciar em mais de 50% o desempenho em 16 processadores.

As simulações realizadas com 2 e 4 processadores têm as maiores taxas de falhas de acesso a memória com o protocolo de invalidação. Já nas simulações com 8 e

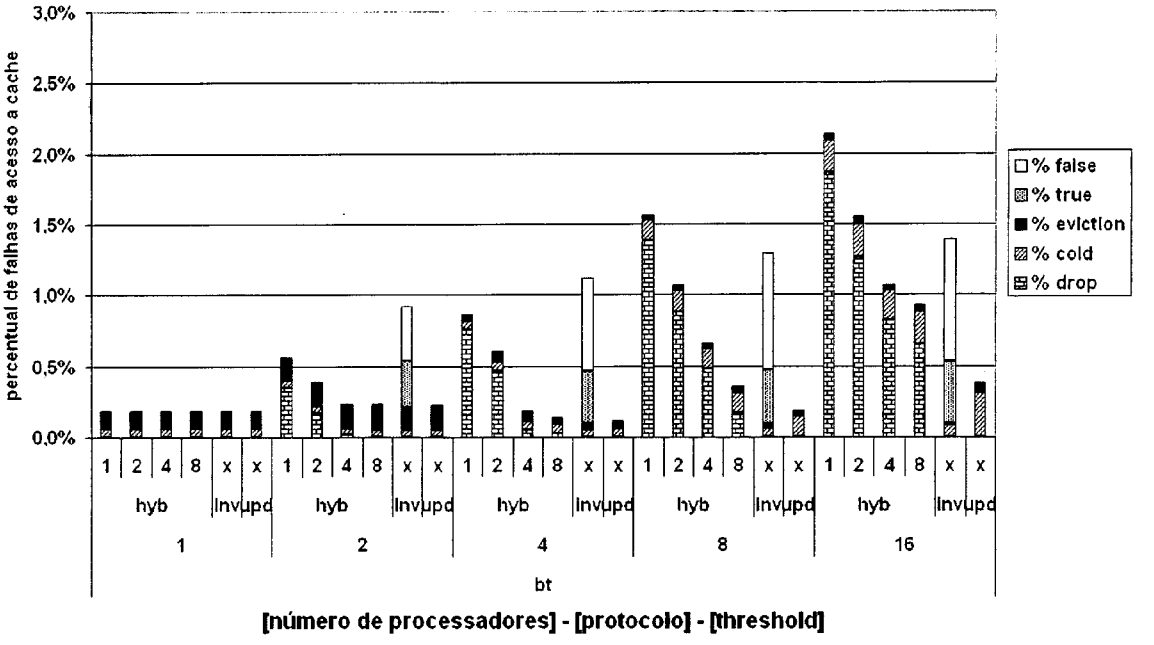

Figura 5.3: Número de *misses* da aplicação BT por número de processadores

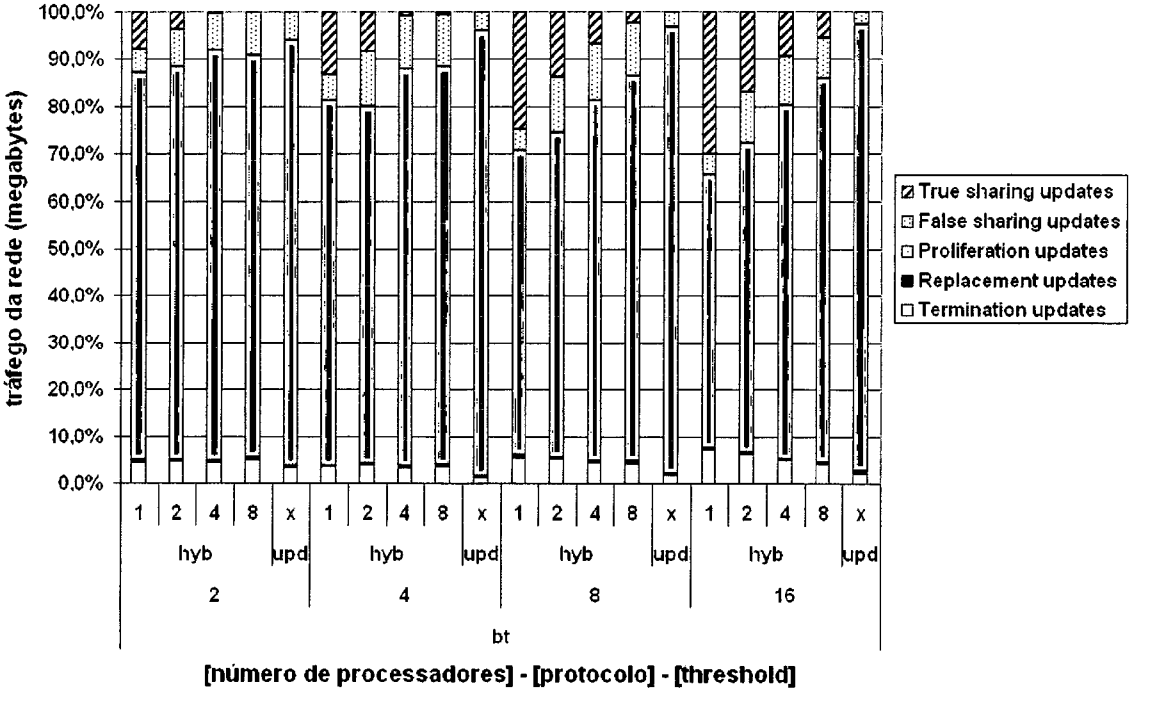

Figura 5.4: Número de *useless updates* da aplicação BT por número de processadores

16 processadores, o protocolo mais influenciado pelas falhas à *cache* foi o híbrido com *threshold* 1. Isto explica porque este protocolo apresenta desempenho pior do que o produzido pelo protocolo de invalidação para esses números de processadores. Observa-se claramente que o aumento do *threshold* diminui a influência das falhas na *cache*. Este fato indica que o dado compartilhado, ou pelo menos o bloco atualizado, está sendo re-utilizado pelo mesmo processador que recebeu a atualização. É importante notar que o número de *drop misses* diminui significativamente com a variação do *threshold*, o que indica que esta aplicação se beneficia do protocolo híbrido com valores altos de *threshold*.

Embora o protocolo de atualização tenha a menor taxa de *misses*, o número de *useless updates* e o tráfego causado por estas mensagens na rede não compensam sua utilização para números de processadores maiores do que 4.

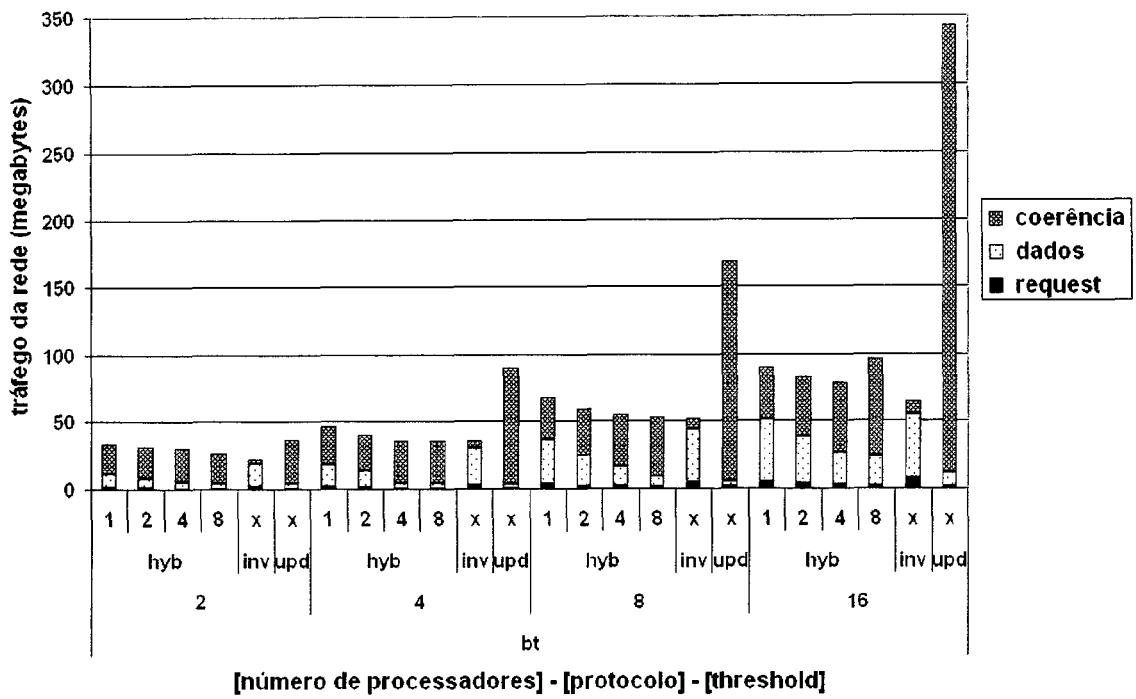

Figura 5.5: Comportamento da rede da aplicação BT por número de processadores

Dentro do protocolo híbrido as taxas de uso da rede crescem de acordo com o aumento do *threshold* e as maiores taxas são encontradas no protocolo de atualização, como esperado, porque transmite maiores quantidades de dados.

A taxa de uso da rede pelo protocolo híbrido é bem inferior a encontrada no protocolo de atualização, sendo mais próxima às apresentadas pelo protocolo de

invalidação.

Nesta aplicação o balanceamento entre uso da rede e as falhas de acesso a memória não segue um padrão pré-determinado. Depende do número de processadores e dos protocolos utilizados. O fato do aumento do número de processadores aumentar consideravelmente as taxas de uso da rede explica o melhor desempenho, com 16 processadores, ter sido obtido com o protocolo híbrido com um *threshold* menor que o encontrado nas simulações com 8 processadores. A tendência do protocolo híbrido é diminuir o número de mensagens enviadas pela rede.

5.2 Aplicação CHAT

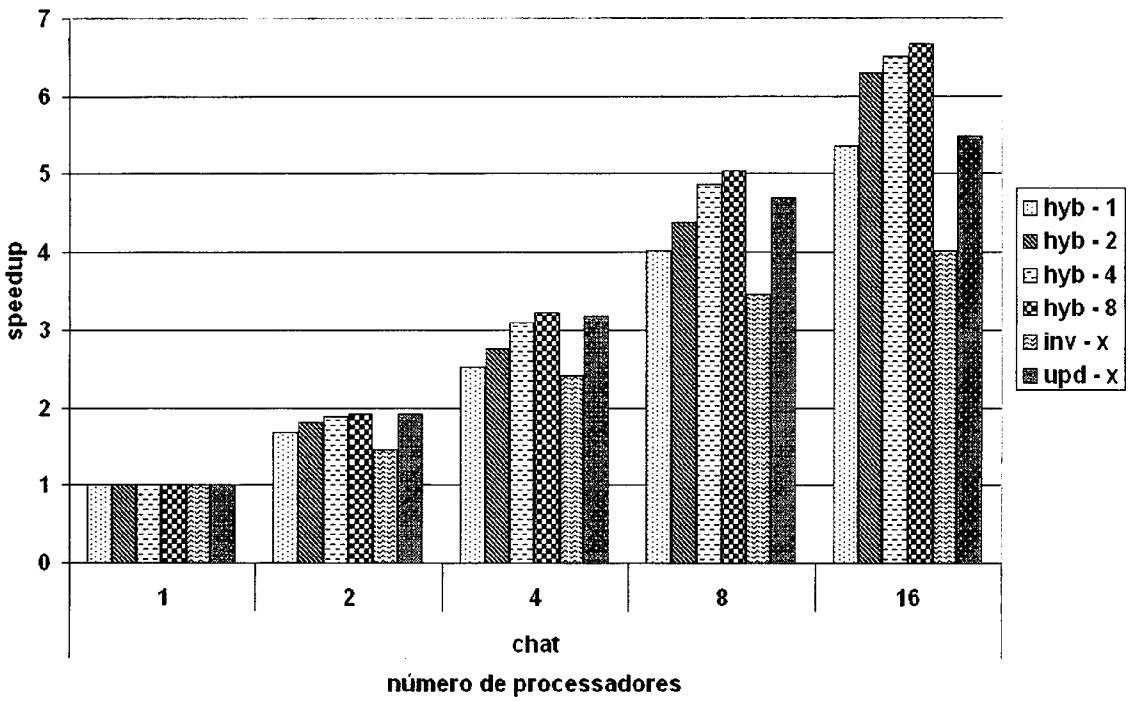

Figura 5.6: *Speedup* da aplicação CHAT, *cache* de 512 Kbytes

A aplicação CHAT é composta por paralelismo do tipo -*OU*. Para a arquitetura simulada os *speedups* produzidos estão entre 4 (protocolo de invalidação) e 6,7 (protocolo híbrido com *threshold* igual a 8) para 16 processadores. A diferença entre estes dois valores é superior a 65%.

Observa-se na Figura 5.6 que o protocolo híbrido é o que oferece os melhores desempenhos. O aumento do *threshold*, dentre os simulados, aumenta o desempenho da aplicação.

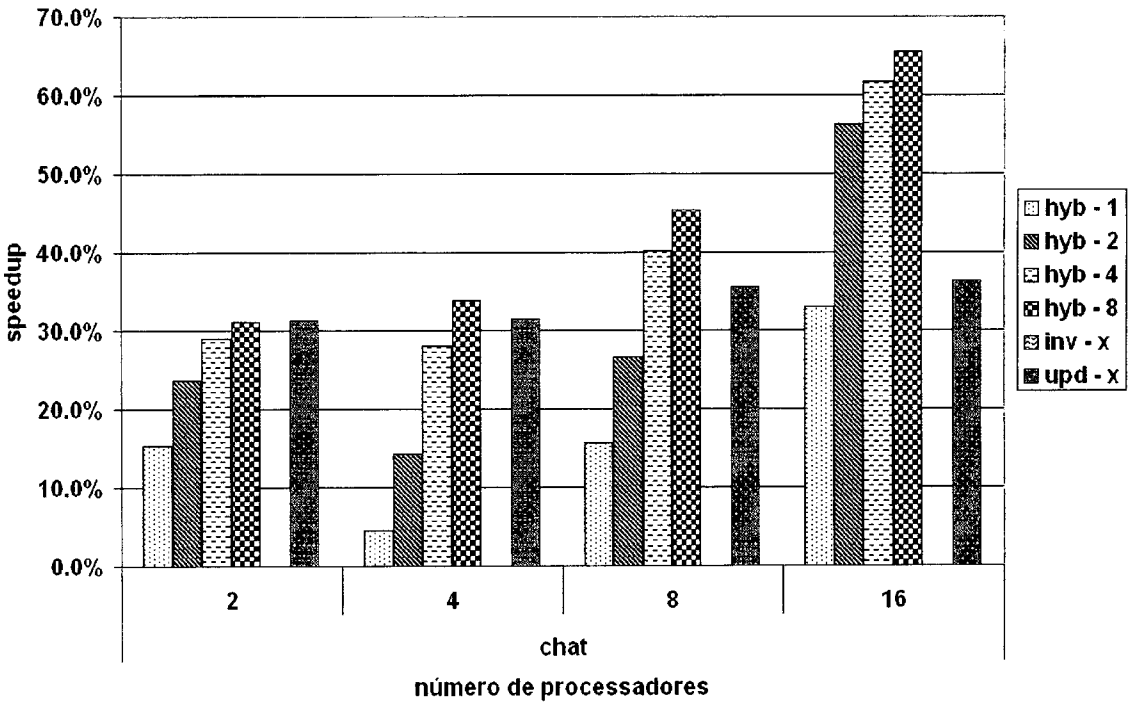

Figura 5.7: Comparativo de ganhos percentuais da aplicação CHAT em relação ao *speedup* de menor desempenho, *cache* de 512 Kbytes

Pela Figura 5.6 é possível observar que o protocolo híbrido proporciona maiores ganhos de desempenho com o aumento do número de processadores. A Figura 5.7 mostra que os ganhos com o protocolo híbrido com *threshold* 8 são de aproximadamente 30% em relação ao protocolo de invalidação para 2 processadores, sendo elevado para mais de 65% em relação ao protocolo de invalidação para 16 processadores.

O baixo desempenho encontrado com o uso do protocolo de invalidação na aplicação CHAT deve-se a ausência do dado durante o processamento, sendo necessário a espera para coerência de *cache*, como mostrado na Figura 5.8. Note a forte influência das falhas na *cache* com o uso do protocolo de invalidação. A Figura 5.10 mostra altas taxas de transferência de dados quando usados os protocolos híbrido e de invalidação, confirmando o citado anteriormente.

O aumento do *threshold* no protocolo híbrido traz melhorias consideráveis na redução de falhas de acesso a *cache*. Mesmo em simulações com o protocolo híbrido com *threshold* 1 observa-se melhorias significativas no *speedup*.

Em todos os grupos de simulações com o mesmo número de processadores as

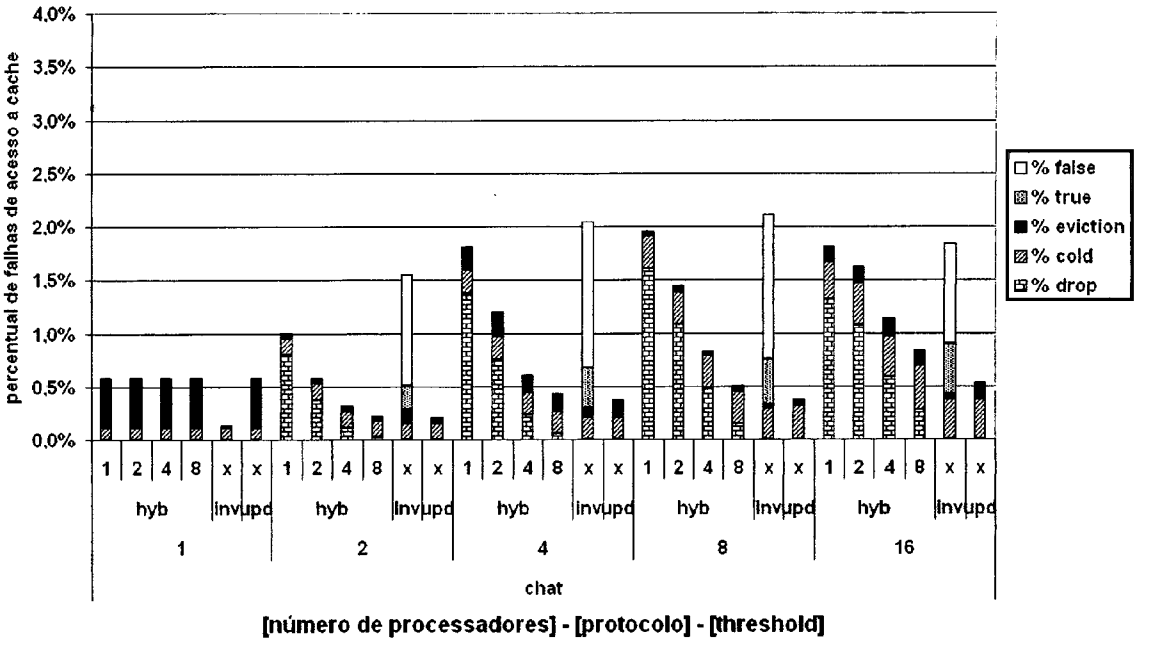

Figura 5.8: Número de *misses* da aplicação CHAT por número de processadores

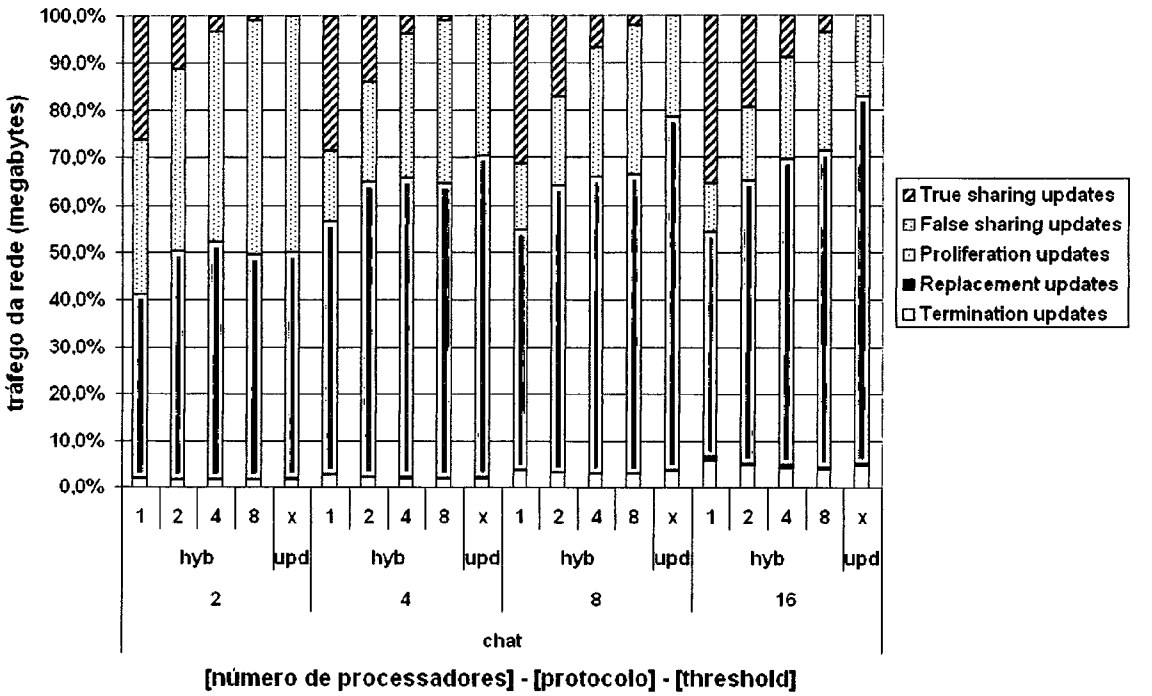

Figura 5.9: Número de *useless updates* da aplicação CHAT por número de processadores

taxas de falhas a *cache* do protocolo híbrido com *threshold* 8 (o maior avaliado) são semelhantes às taxas encontradas no protocolo de atualização.

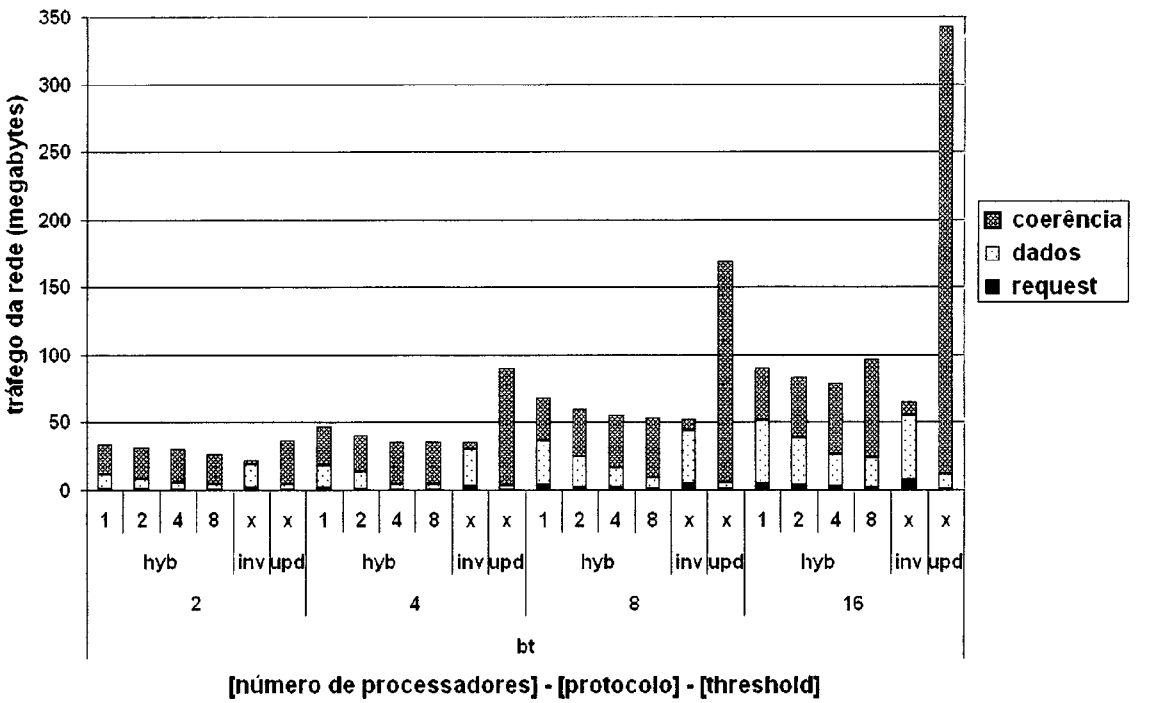

Figura 5.10: Comportamento da rede da aplicação CHAT por número de processadores

A aplicação tem um comportamento que tende a ser beneficiado pelo protocolo de atualização ao invés do protocolo de invalidação. O híbrido oferece ganhos maiores que o de invalidação, pois a rede não fica tão congestionada e os acertos na *cache* são consideravelmente altos, como pode ser observado na Figura 5.9.

5.3 Aplicação PAN2

A aplicação PAN2 oferece paralelismos do tipo *-E* e do tipo *-OU*. Os *speedups* gerados variam de 4,5 (protocolo de atualização) a 5,2 (protocolo híbrido com *threshold* igual a 8) para 16 processadores. Uma diferença de aproximadamente 16%.

Observa-se na Figura 5.11 que os melhores desempenhos são encontrados no protocolo híbrido. Embora o desempenho varie de acordo com o número de processadores utilizados e os diferentes *thresholds*. Por exemplo, nas simulações executadas com 2 processadores o protocolo com melhor desempenho foi o híbrido

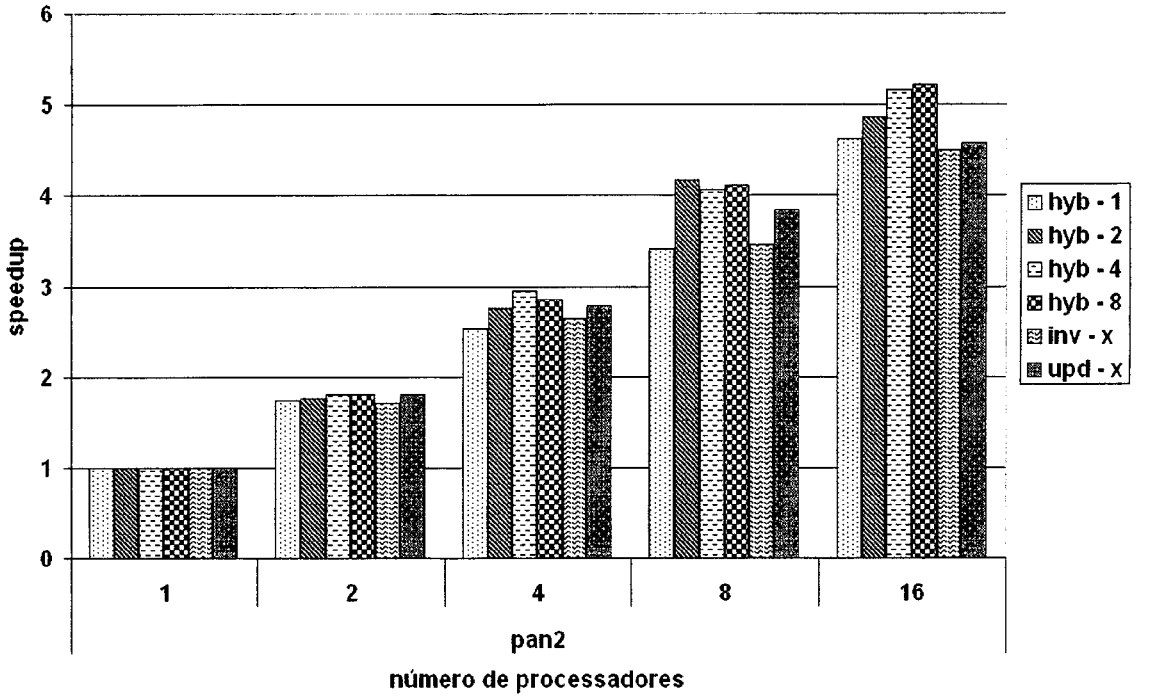

Figura 5.11: *Speedup* da aplicação PAN2, *cache* de 512 KBytes

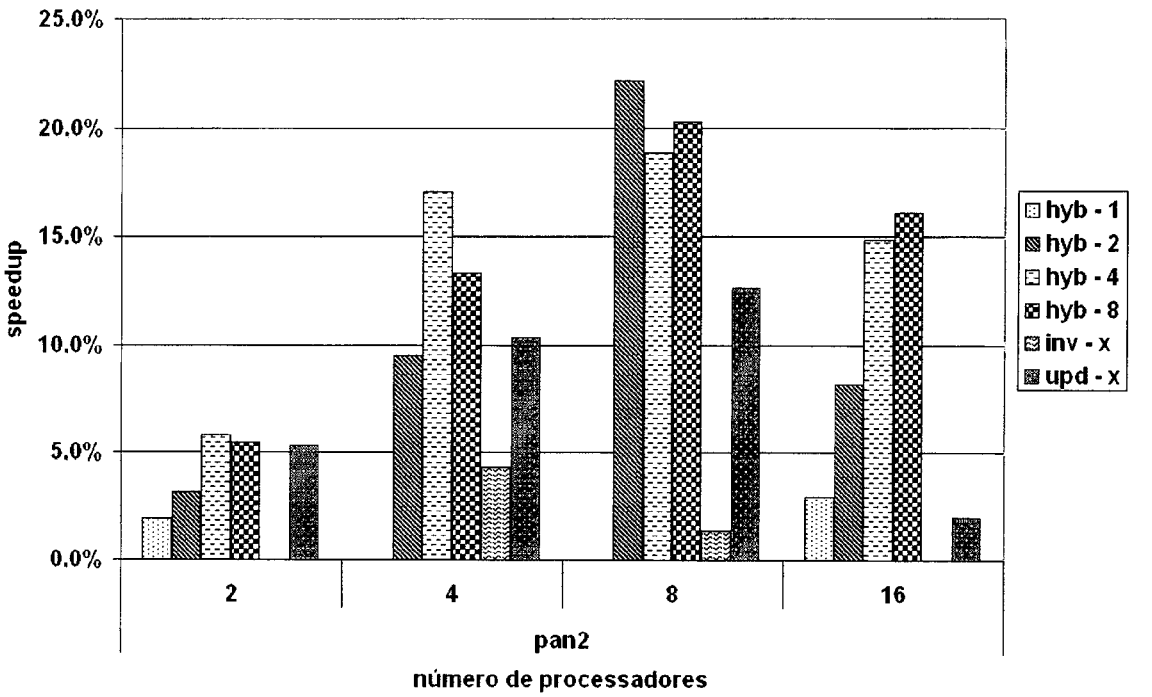

Figura 5.12: Comparativo de ganhos percentuais da aplicação PAN2 em relação ao *speedup* de menor desempenho, *cache* de 512 Kbytes

com *threshold* 4. Nas simulações com 8 processadores o melhor desempenho foi encontrado no protocolo híbrido com *threshold* 2.

Com o aumento do número de processadores, o protocolo híbrido mostra-se o mais adequado à aplicação. As Figuras 5.11 e 5.12 indicam isto, onde o protocolo de atualização é uma escolha melhor que em outras 2 simulações realizadas com o protocolo híbrido. Para 16 processadores o protocolo de atualização perde para as outras 4 simulações desenvolvidas com o protocolo híbrido.

Esta aplicação tem como característica que a diferença de ganho de desempenho entre os protocolos não é tão abrupta. Em contrapartida, os melhores desempenhos são encontrados em diferentes protocolos e com diferentes *thresholds*, de acordo com o número de processadores.

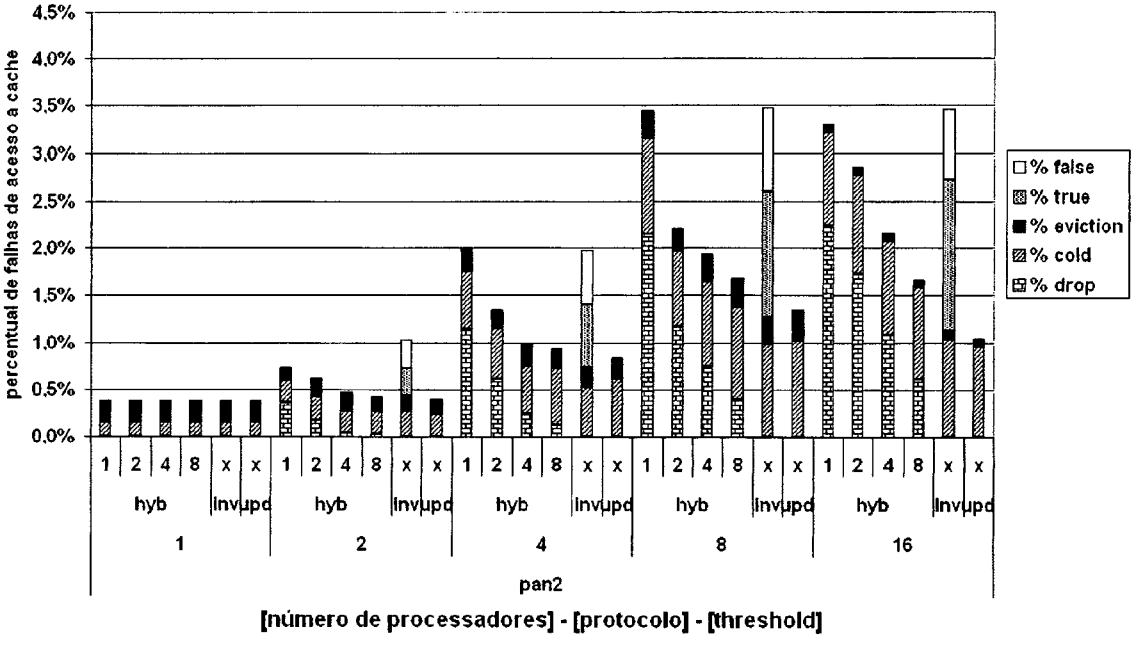

Figura 5.13: Número de *misses* da aplicação PAN2 por número de processadores

A Figura 5.13 mostra que o protocolo de atualização e o protocolo híbrido com *threshold* 1 têm comportamentos semelhantes quanto a falhas de acesso a *cache*.

Observa-se na Figura 5.13 uma redução considerável das taxas de falha de acesso a *cache* com o protocolo híbrido com *threshold* 2, em relação ao mesmo protocolo com *threshold* 1. Isto explica as diferenças percentuais de ganhos com este número de processadores serem tão diferenciadas, como apresentado na Figura 5.12.

O protocolo de atualização tem aumentos significativos de uso da rede a partir

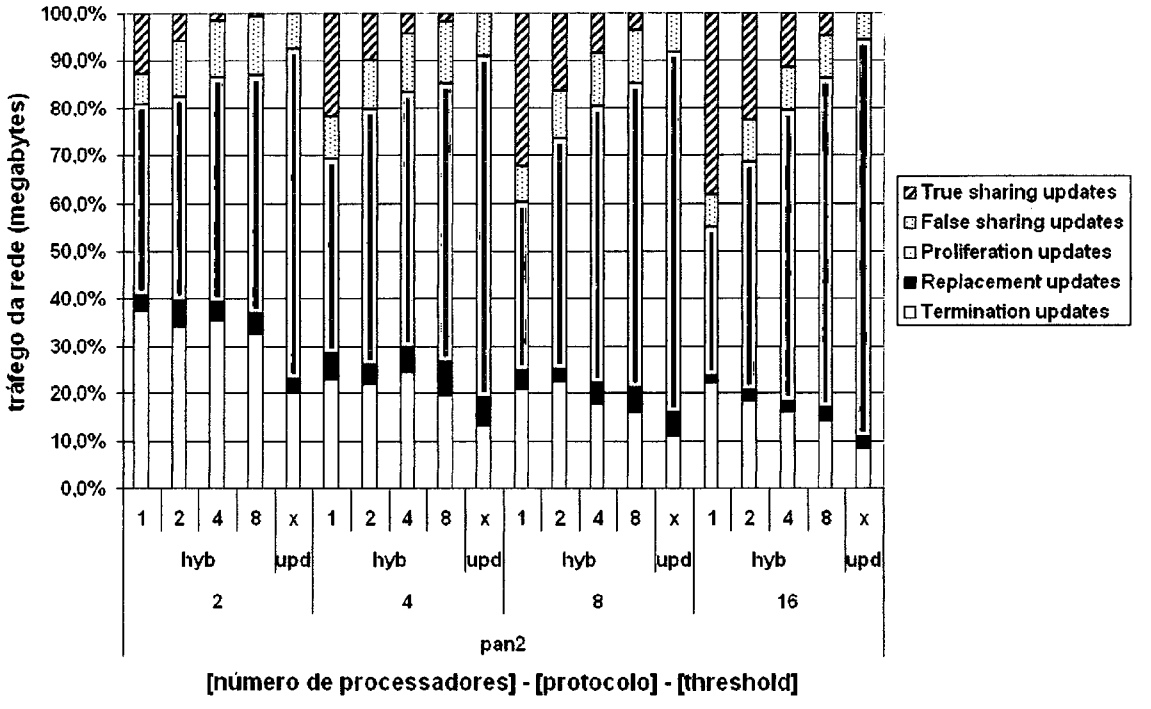

Figura 5.14: Número de *useless updates* da aplicação PAN2 por número de processadores

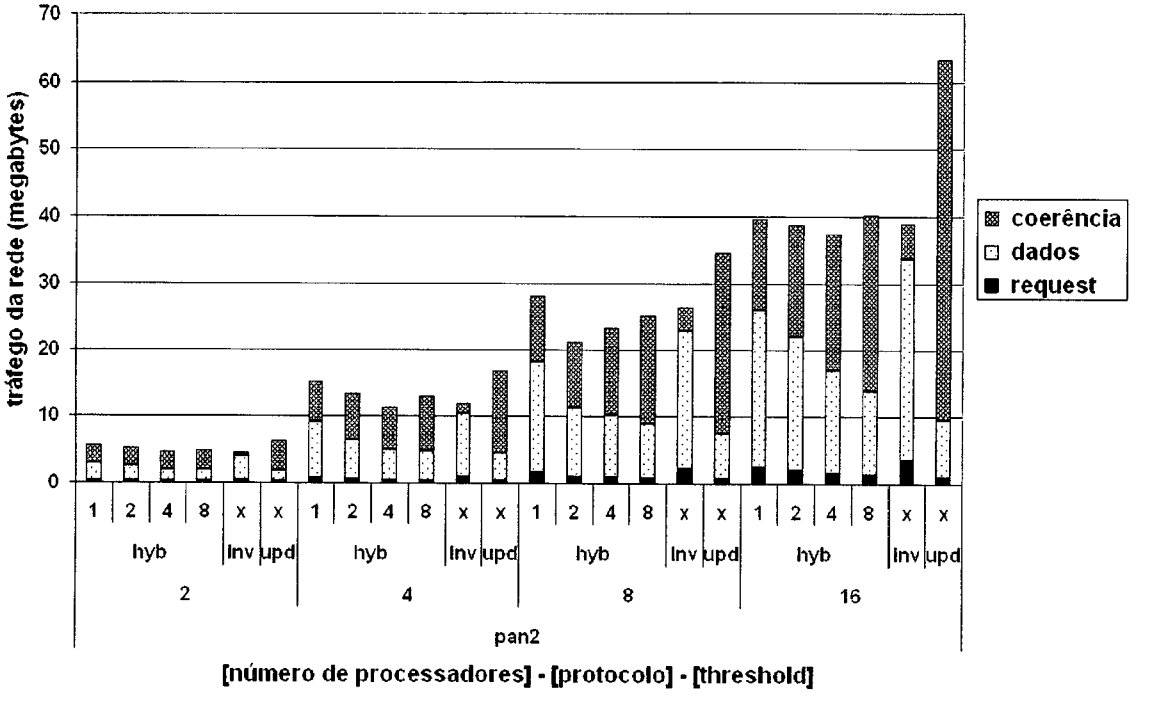

Figura 5.15: Comportamento da rede da aplicação PAN2 por número de processadores

de 4 processadores, como pode ser visto na Figura 5.15. Isto indica a causa dos protocolos híbridos com os maiores valores de *threshold* apresentarem ganhos percentuais maiores em relação aos demais a partir deste número de processadores.

5.4 Aplicação TSP

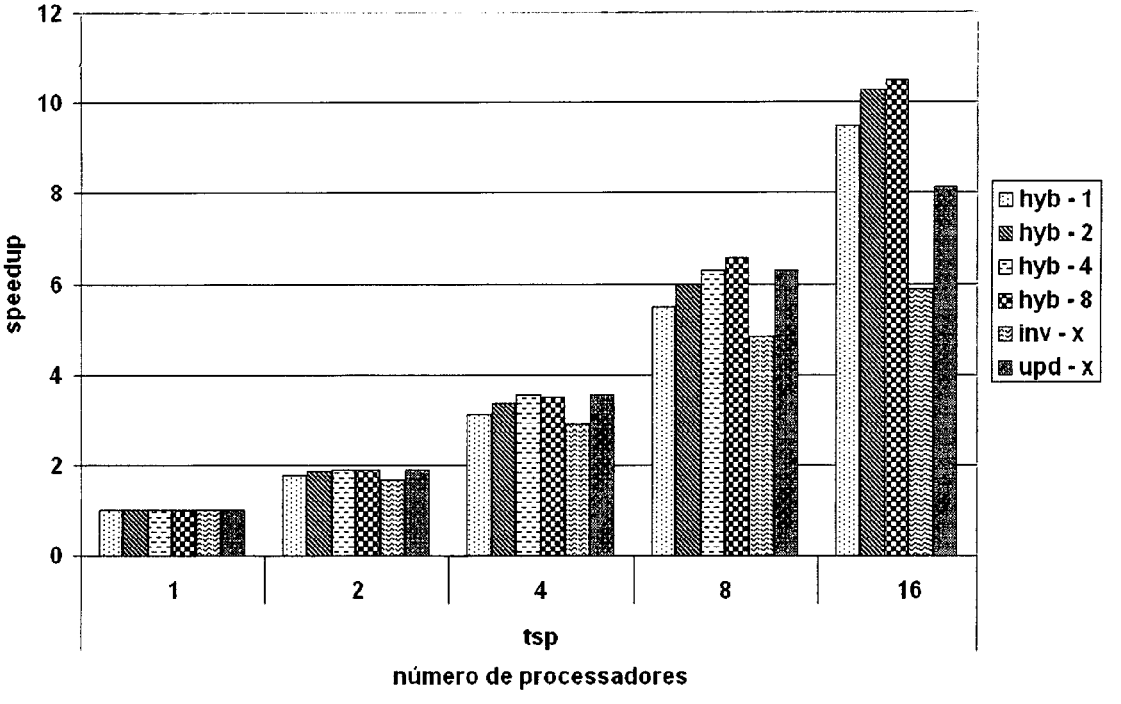

Figura 5.16: *Speedup* da aplicação TSP, *cache* de 512 Kbytes

A aplicação TSP tem uma quantidade significativa de paralelismo -*E*, produzindo *speedups* entre 5,9 (protocolo de atualização) e 10,5 (protocolo híbrido com *threshold* igual a 8) para 16 processadores. A diferença entre estes dois valores é de aproximadamente 80%.

Observa-se na Figura 5.16 que o desempenho do protocolo de atualização supera o de invalidação em todas as simulações executadas. Nas simulações com até 8 processadores o desempenho do protocolo de atualização é equivalente ou superior ao protocolo híbrido. A exceção é a simulação com 8 processadores usando o protocolo híbrido e *threshold* igual a 8, que supera os demais com o mesmo número de processadores. Esta situação se inverte nas simulações com 16 processadores, onde todas as simulações com o protocolo híbrido superam o protocolo de atualização. A

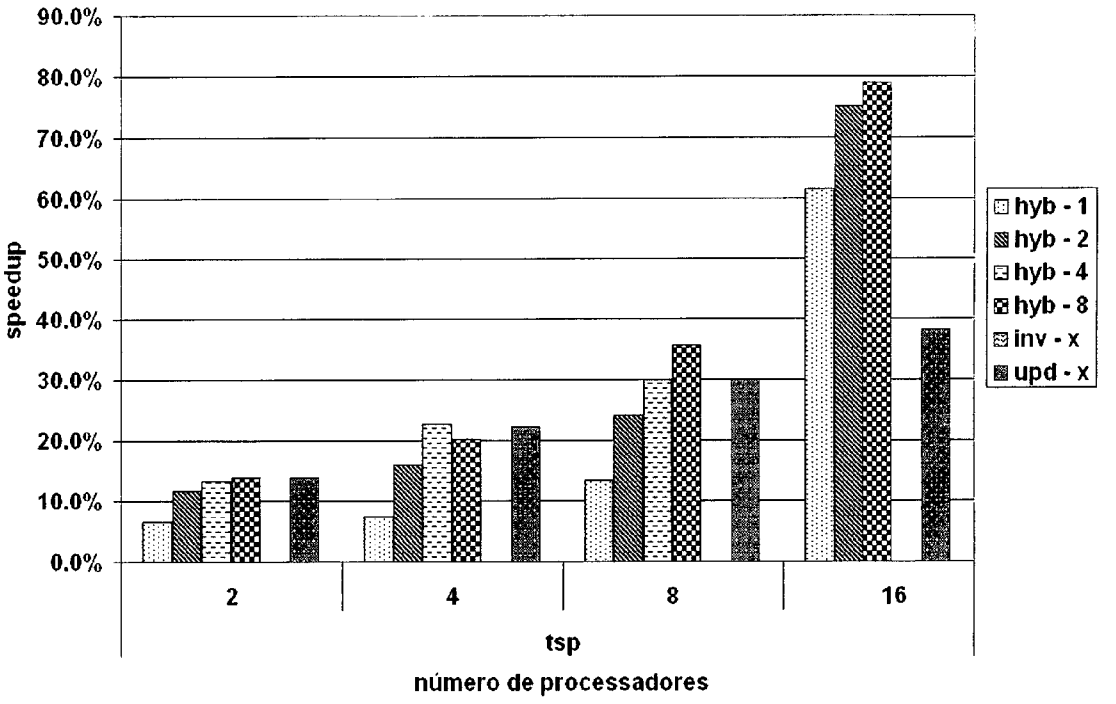

Figura 5.17: Comparativo de ganhos percentuais da aplicação TSP em relação ao *speedup* de menor desempenho, *cache* de 512 Kbytes

tendência apresentada é que com o aumento do número de processadores o uso do protocolo híbrido seja mais adequado.

As taxas de falha de acesso a *cache* com o protocolo de invalidação para 16 processadores apresentadas na Figura 5.18 são bem próximas dos valores encontrados com 8 processadores. Já a utilização da rede com o protocolo de atualização com 16 processadores é bem maior que a utilizada com 8 processadores, como apresentado na Figura 5.20. As atualizações são melhor aproveitadas com o protocolo híbrido com 16 processadores, como apresentado na Figura 5.19, indicam que o protocolo híbrido com valores mais altos de *threshold* tendem a ganhos maiores para simulações com um número maior de processadores.

5.5 Discussão

Estes resultados mostram que uma alternativa razoável ao protocolo de atualização é alguma forma de protocolo híbrido. Nesta tese confirmamos que protocolos híbridos baseados na implementação do protocolo utilizado em multiprocessadores baseados em barramento que usam o processador DEC Alpha AXP21064 [49], com *thresholds*

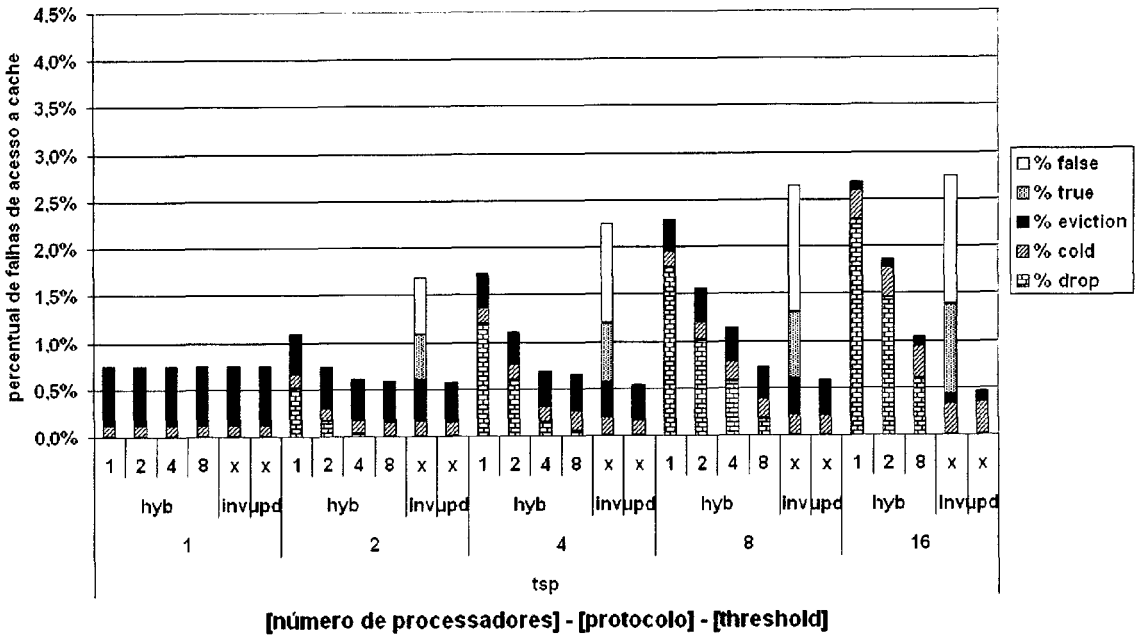

Figura 5.18: Número de *misses* da aplicação TSP por número de processadores

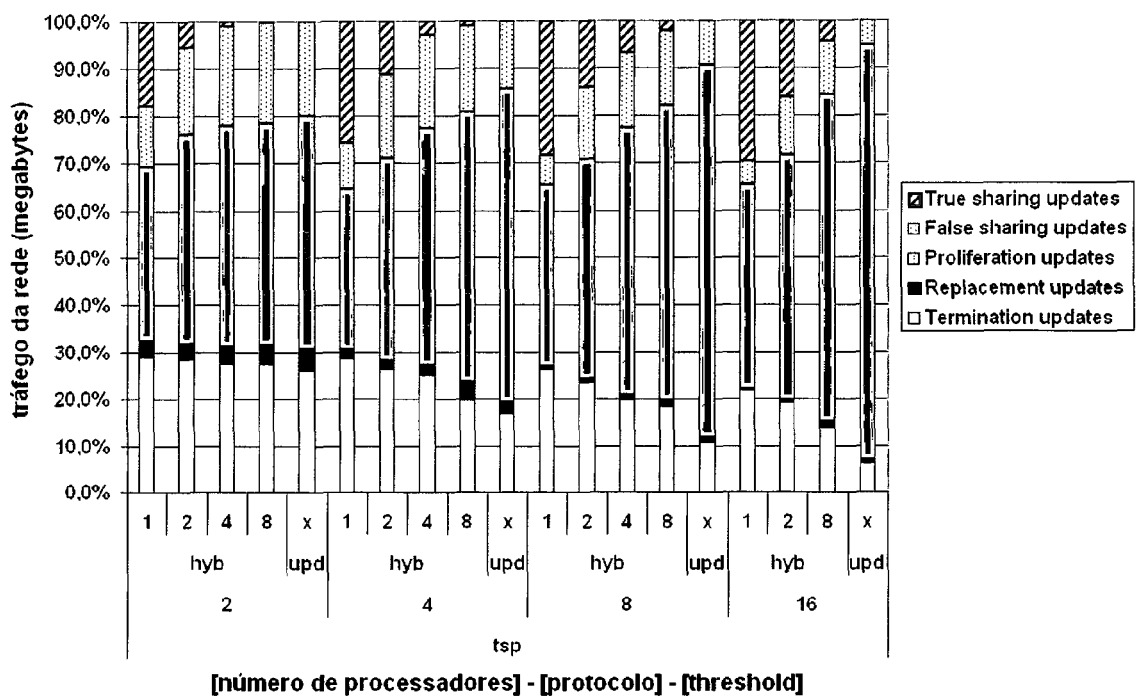

Figura 5.19: Número de *useless updates* da aplicação TSP por número de processadores

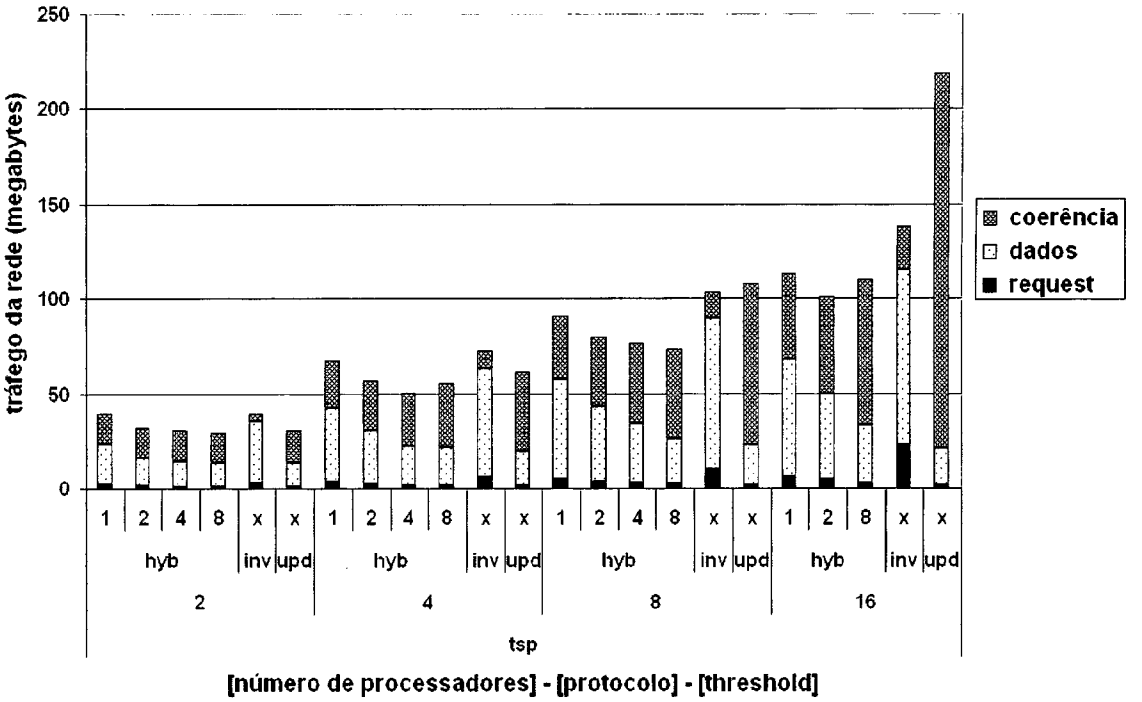

Figura 5.20: Comportamento da rede da aplicação TSP por número de processadores

altos, podem alcançar um desempenho próximo do desempenho ideal (Tabela 4.1) para cada uma das aplicações estudadas.

O desempenho obtido com todas as aplicações ficou próximo da implementação ideal, com 16 processadores. A aplicação BT alcançou *speedup* de quase 12, enquanto CHAT alcançou aproximadamente 7. Já a aplicação PAN2 obteve *speedup* pouco superior a 5 e TSP pouco superior a 10.

A aplicação que perdeu mais em desempenho quando comparada com a simulação ideal foi PAN2, devido à complexidade de combinação entre paralelismo *-E* e paralelismo *-OU*.

Estas aplicações, principalmente CHAT, consomem muita memória durante sua execução. Com o objetivo de investigar se o tamanho escolhido da *cache* estava afetando os resultados de forma significativa, realizamos experimentos para uma *cache* maior, de 1024 Kbytes, e para uma *cache* menor, de 128 Kbytes. Os resultados são mostrados nas Figuras 5.21 até 5.36.

A aplicação BT, até 8 processadores, não teve mudança de comportamento com o aumento da *cache* de 128 (Figura 5.21) para 512 (Figura 5.1) e 1024 (Figura 5.22) Kbytes. Em 16 processadores, o desempenho variou, sendo a maior taxa entre 7% e

8%, para 512 Kbytes, para o protocolo híbrido com *threshold* 4. O aumento de 512 Kbytes (Figura 5.1) para 1024 Kbytes (Figura 5.22) não foi significativo. O protocolo de atualização, em 16 processadores, teve uma queda de desempenho considerável (aproximadamente 9,5%) com o incremento de memória.

O ganho percentual dos protocolos, em 16 processadores, com o aumento de 128 Kbytes (Figura 5.23) para 512 Kbytes (Figura 5.2) foi significativo, com uma diferença de aproximadamente 23%, para o protocolo híbrido, com *threshold* 4.

A aplicação CHAT obteve melhor desempenho em todas as simulações quando a memória é incrementada de 128 Kbytes (Figura 5.25) para 512 Kbytes (Figura 5.6), atingindo até 16% aproximadamente, em 16 processadores, com o protocolo híbrido, com *threshold* 2. A exceção encontra-se na simulação com 16 processadores utilizando o protocolo de invalidação, apresentando queda de desempenho de até 8% aproximadamente.

Mais uma vez, podemos notar que comparativamente o protocolo híbrido tem ganhos superiores aos demais. Figuras 5.27, 5.7, e 5.28 ilustram melhor este fato mostrando os ganhos percentuais em relação ao protocolo de menor desempenho. O maior ganho, para esta aplicação, foi obtido com a *cache* de 512 Kbytes.

A aplicação PAN2 apresenta queda de desempenho na maioria das simulações, chegando até aproximadamente 8% para o protocolo de atualização com 16 processadores. Em contrapartida, obteve ganhos de aproximadamente 11% para o protocolo híbrido com *threshold* 2 com 8 processadores.

A aplicação TSP, do tipo *-E*, obteve ganhos para as simulações com até 8 processadores, atingindo aproximadamente 10% de ganho em desempenho para o protocolo híbrido com *threshold* 8, com 8 processadores. Todas as simulações com 16 processadores tiveram queda de desempenho, chegando até 15% para o protocolo de atualização.

Observa-se que as aplicações que têm paralelismo *-E* são as que se beneficiam menos do aumento do tamanho de *cache* para alguns protocolos. O pior desempenho é alcançado com o protocolo de atualização. CHAT, que é uma aplicação *-OU* é a única que se beneficia do aumento do tamanho de *cache*, com o protocolo de invalidação tendo um desempenho baixo para 16 processadoress, por causa do falso compartilhamento. PAN2 é a aplicação que apresenta comportamento mais irregular das aplicações *-E*. Isto se deve à complexidade da combinação de paralelismo *-E* com paralelismo *-OU* existente nesta aplicação, que não é adequadamente capturada pelo

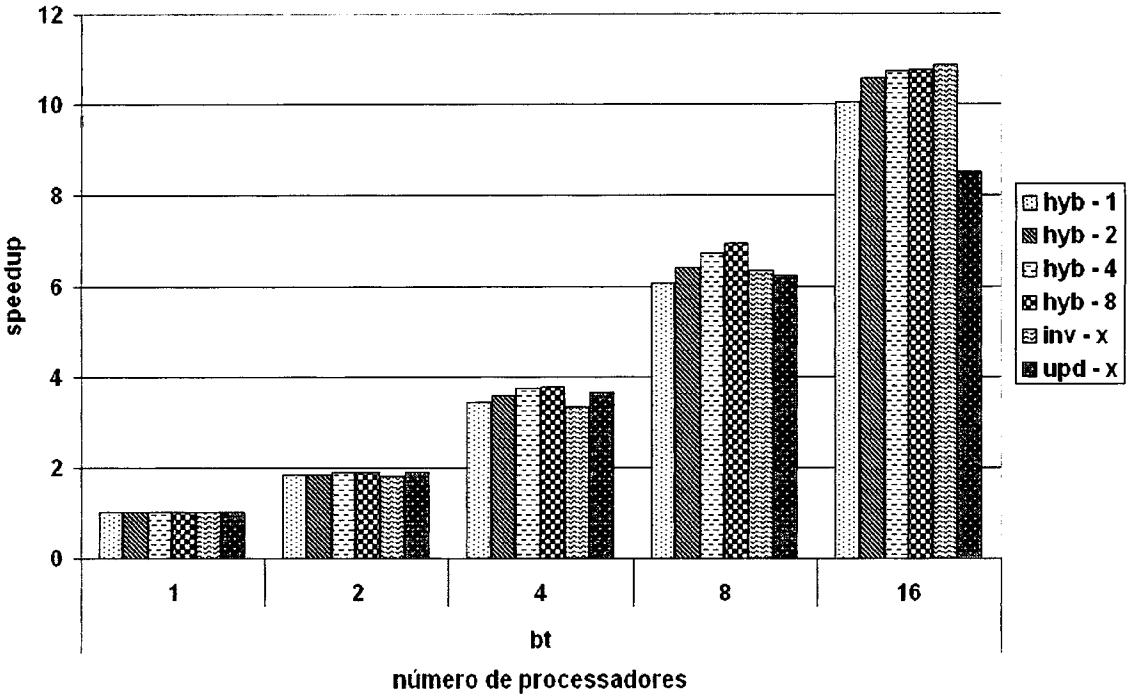

Figura 5.21: *Speedup* da aplicação BT, *cache* de 128 Kbytes

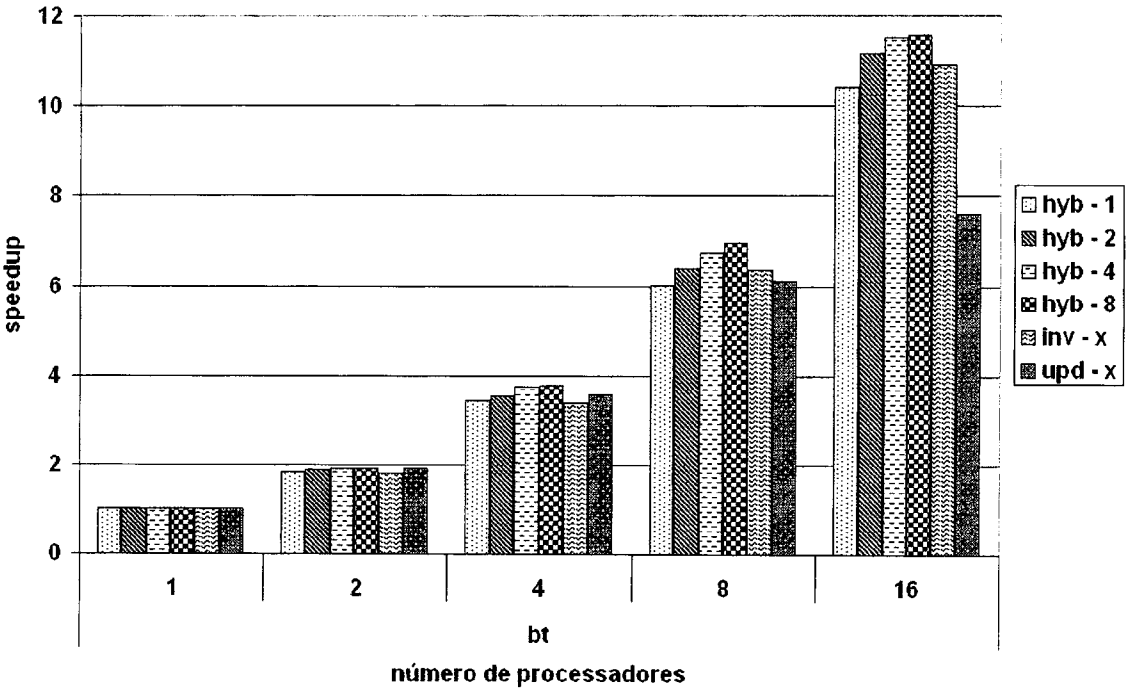

Figura 5.22: *Speedup* da aplicação BT, *cache* de 1024 Kbytes

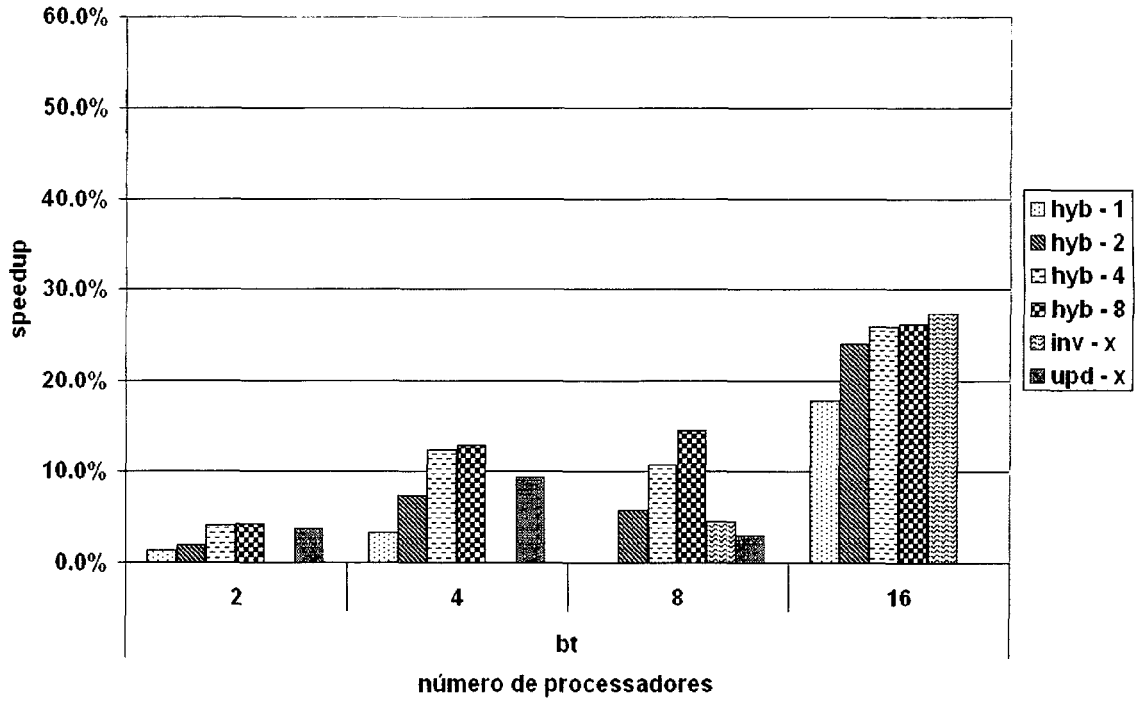

Figura 5.23: Comparativo de ganhos percentuais da aplicação BT em relação ao *speedup* de menor desempenho, *cache* de 128 Kbytes

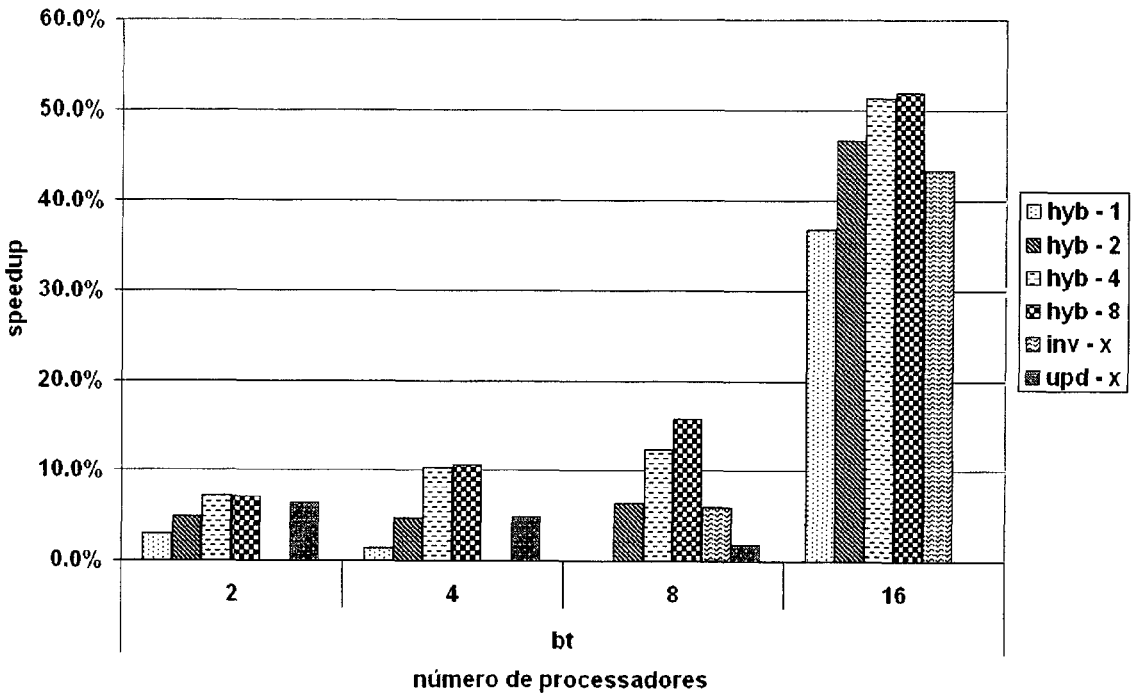

Figura 5.24: Comparativo de ganhos percentuais da aplicação BT em relação ao *speedup* de menor desempenho, *cache* de 1024 Kbytes

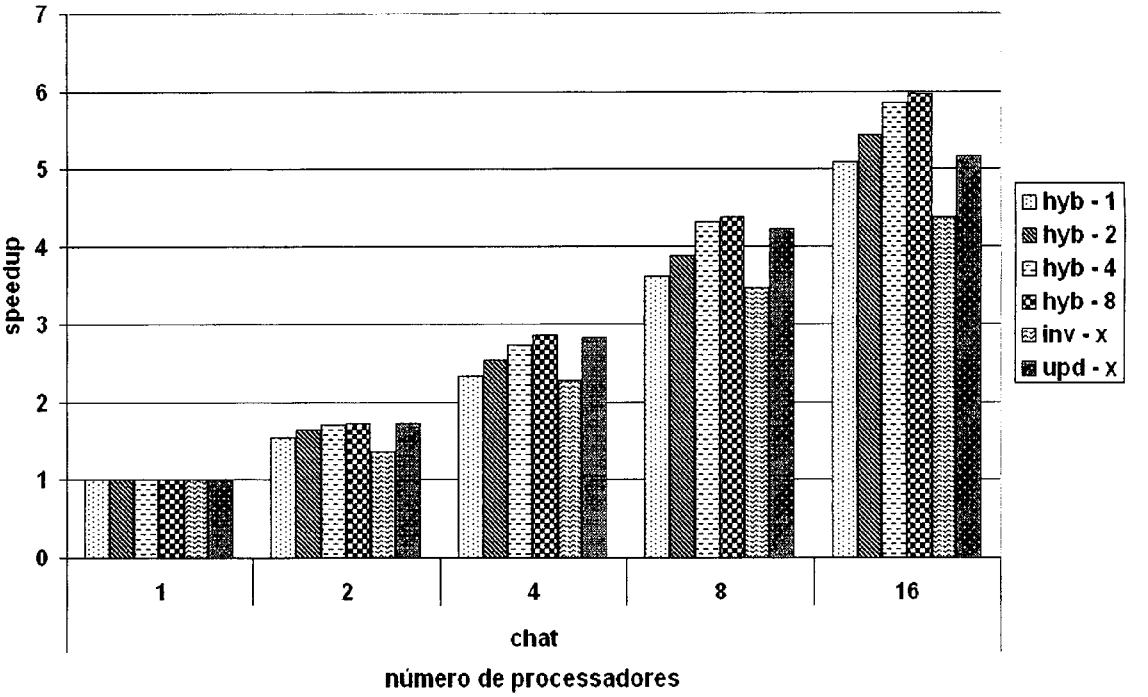

Figura 5.25: *Speedup* da aplicação CHAT, *cache* de 128 Kbytes

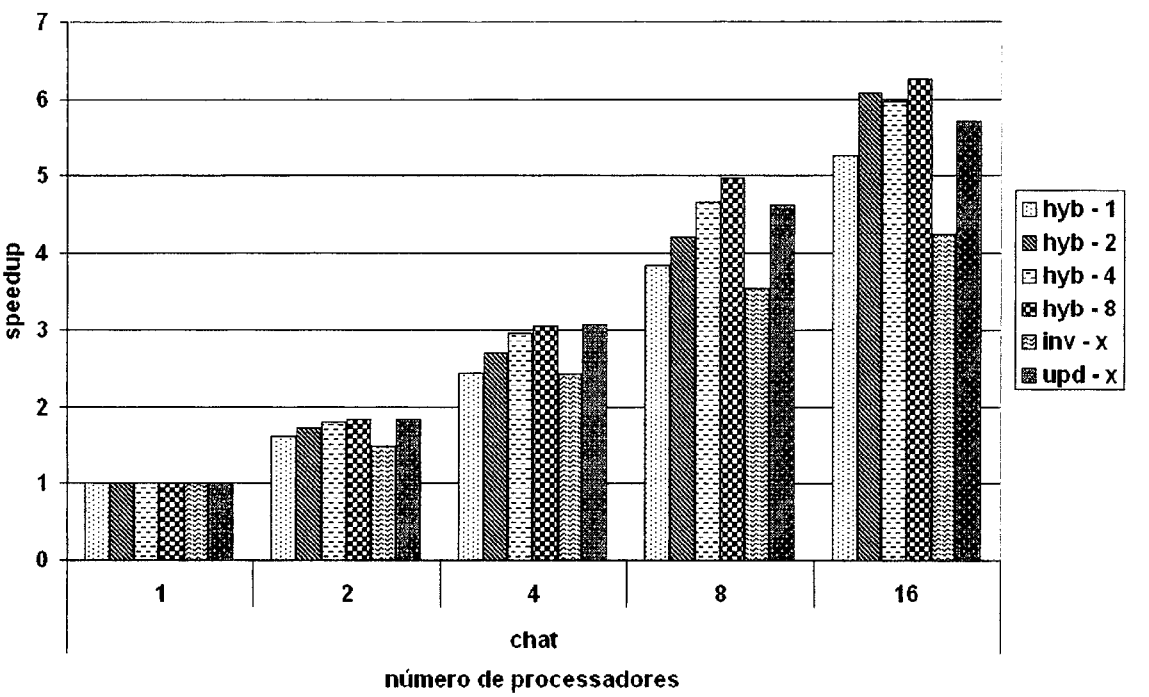

Figura 5.26: *Speedup* da aplicação CHAT, *cache* de 1024 Kbytes

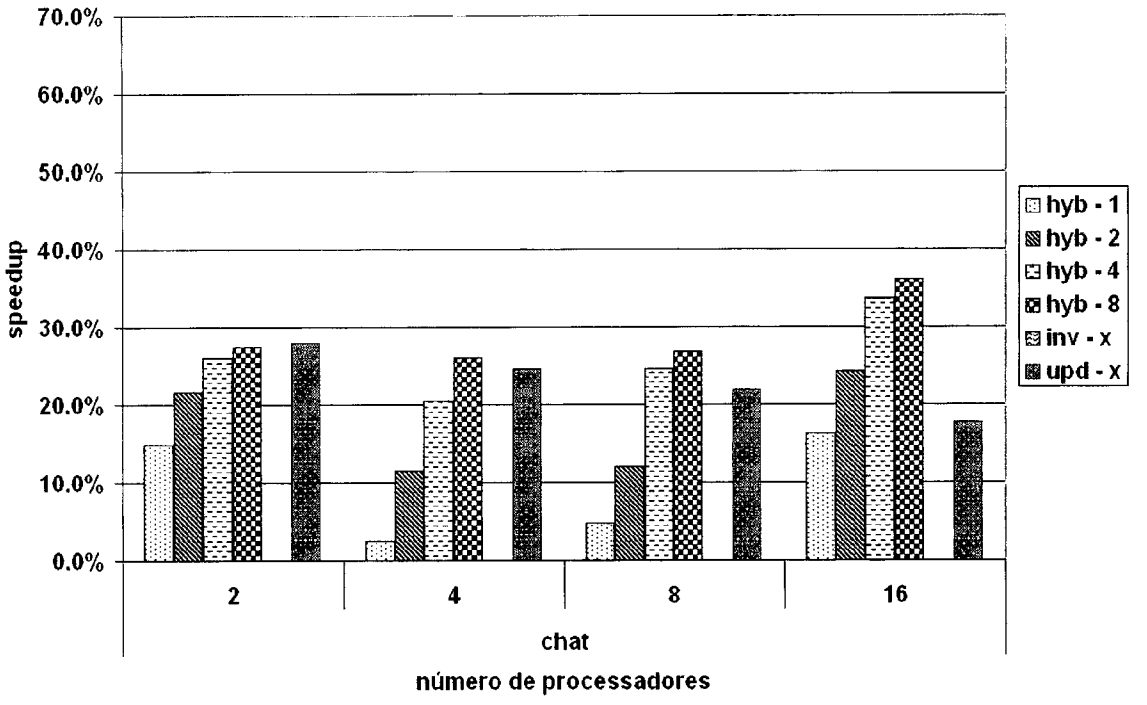

Figura 5.27: Comparativo de ganhos percentuais da aplicação CHAT em relação ao *speedup* de menor desempenho, *cache* de 128 Kbytes

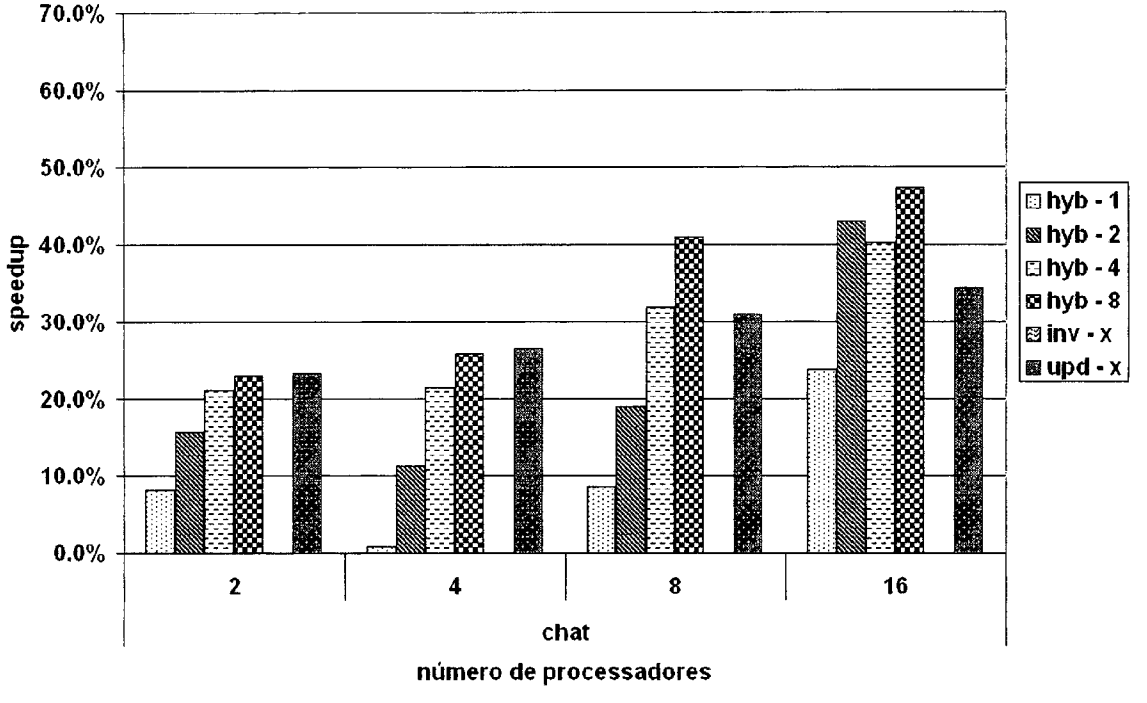

Figura 5.28: Comparativo de ganhos percentuais da aplicação CHAT em relação ao *speedup* de menor desempenho, *cache* de 1024 Kbytes

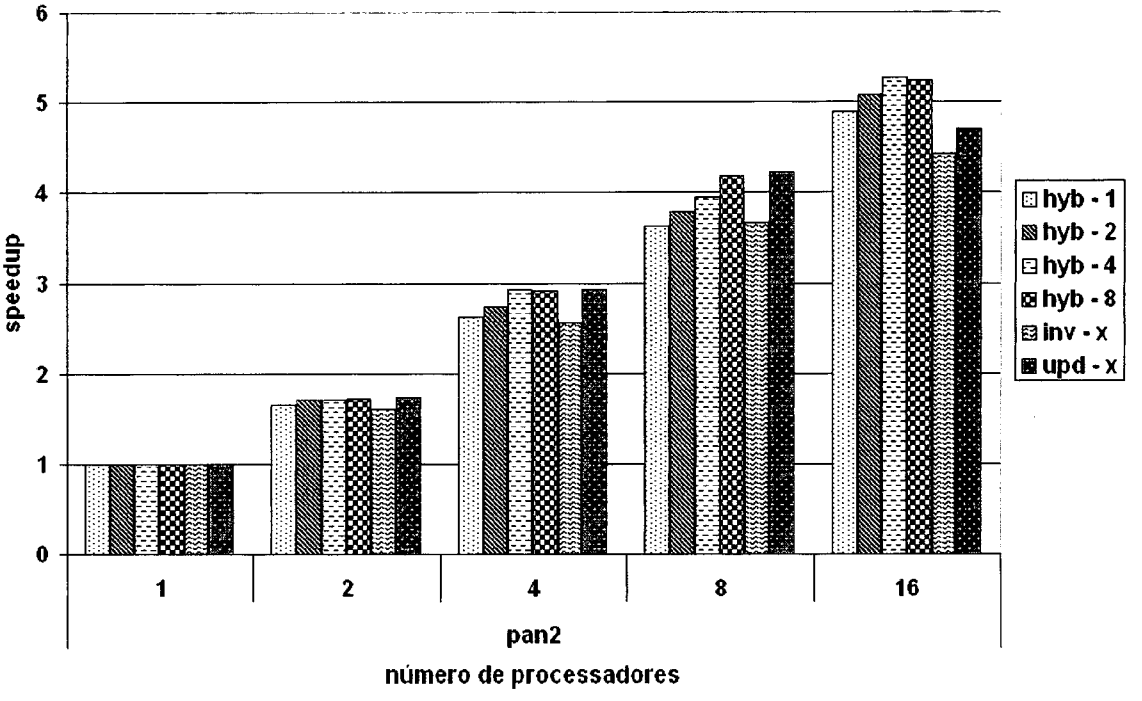

Figura 5.29: *Speedup* da aplicação PAN2, *cache* de 128 Kbytes

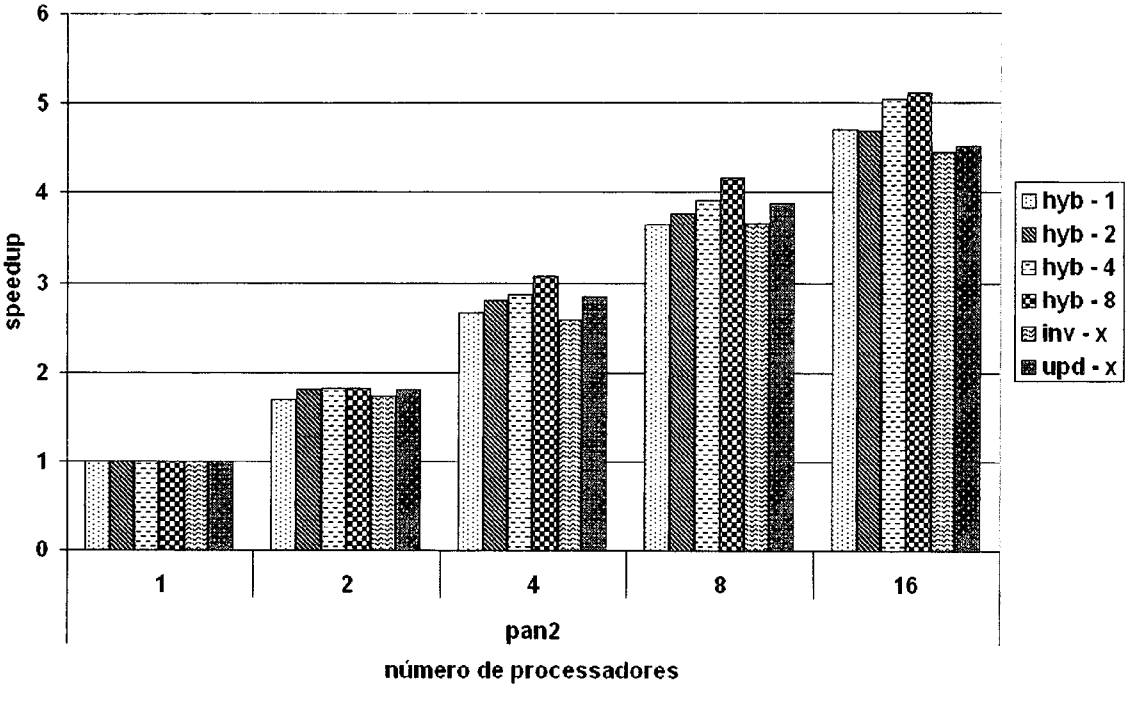

Figura 5.30: *Speedup* da aplicação PAN2, *cache* de 1024 Kbytes

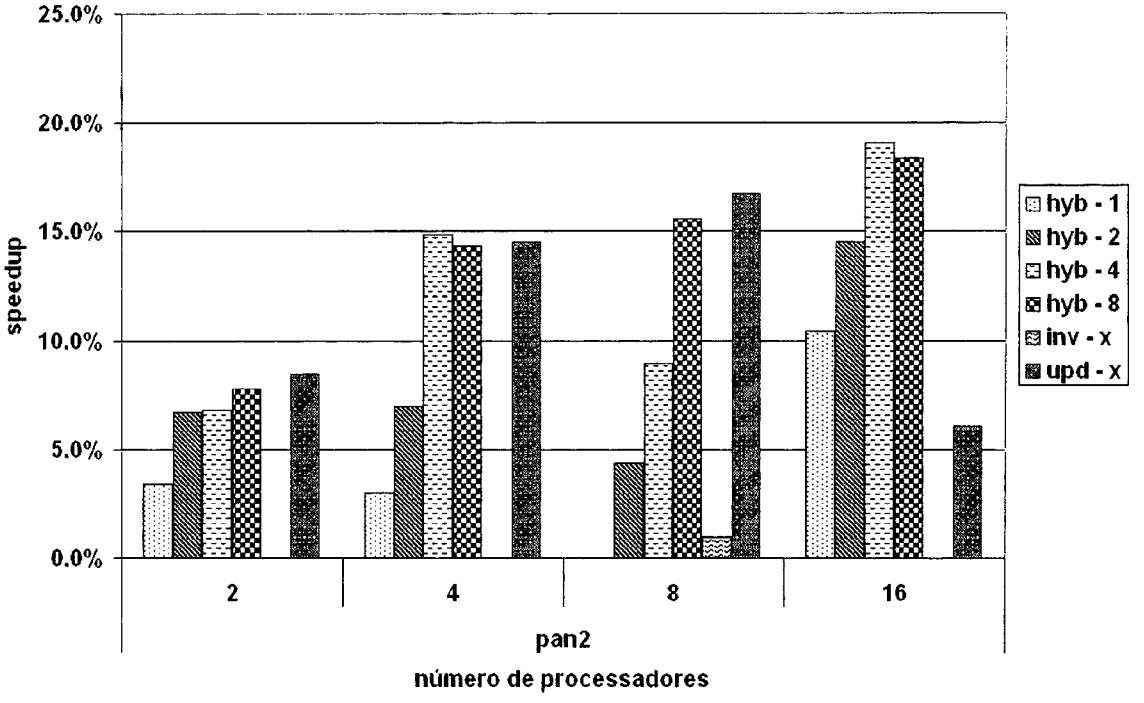

Figura 5.31: Comparativo de ganhos percentuais da aplicação PAN2 em relação ao *speedup* de menor desempenho, *cache* de 128 Kbytes

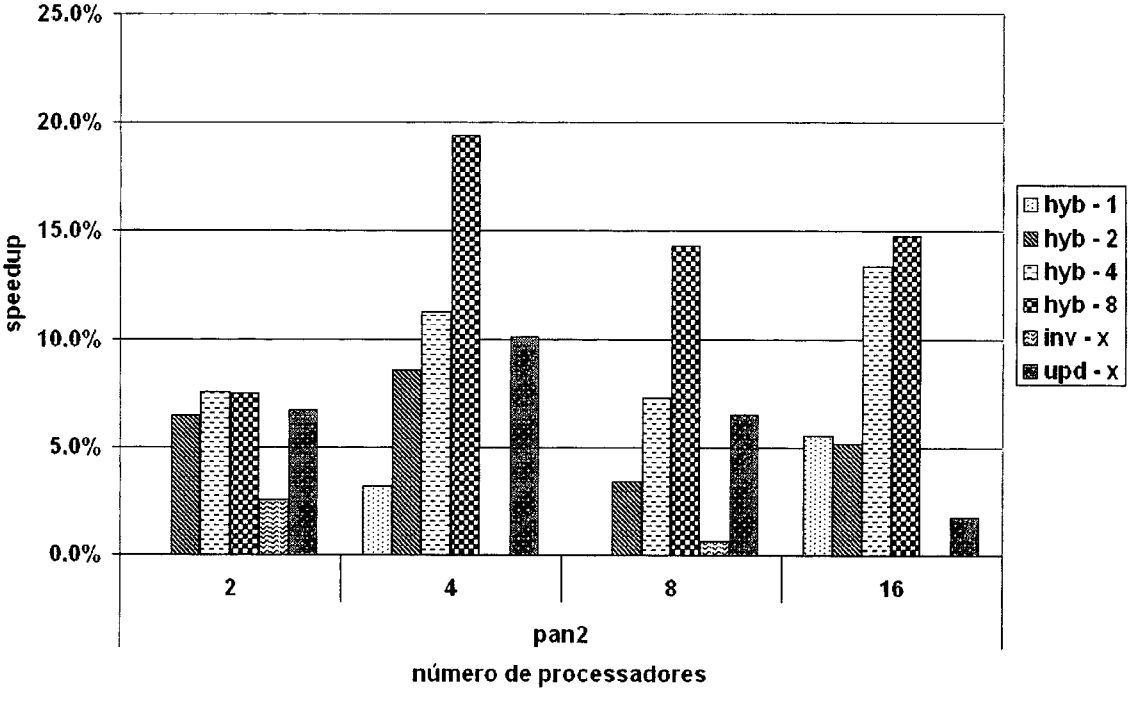

Figura 5.32: Comparativo de ganhos percentuais da aplicação PAN2 em relação ao *speedup* de menor desempenho, *cache* de 1024 Kbytes

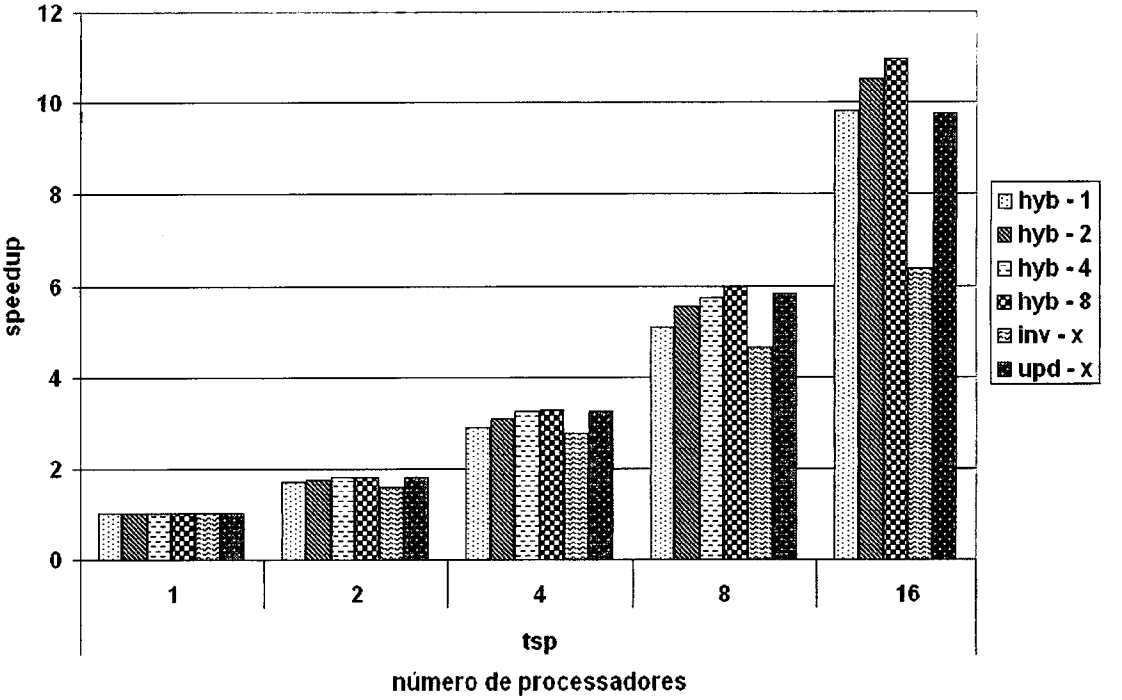

Figura 5.33: *Speedup* da aplicação TSP, *cache* de 128 Kbytes

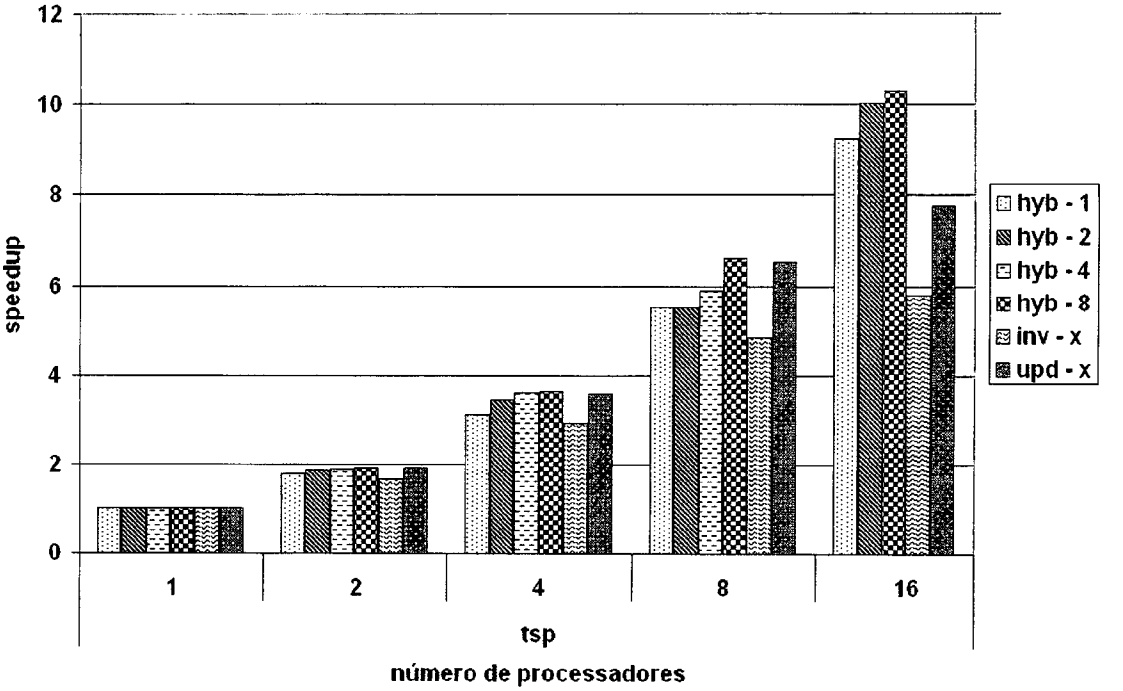

Figura 5.34: *Speedup* da aplicação TSP, *cache* de 1024 Kbytes

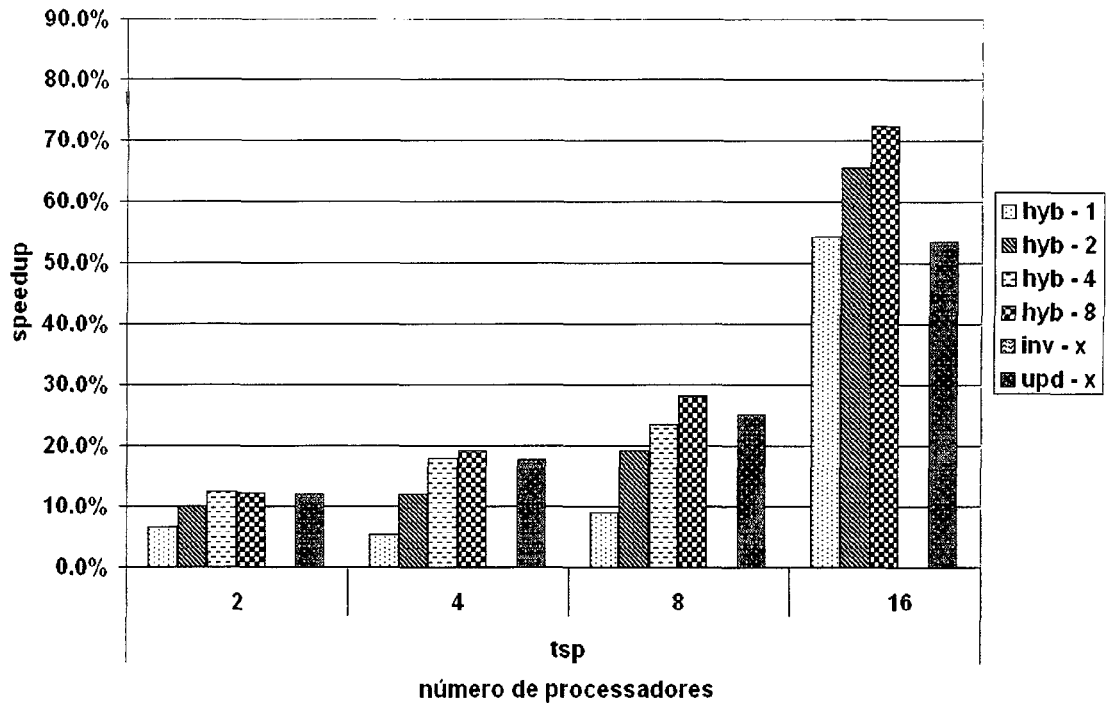

Figura 5.35: Comparativo de ganhos percentuais da aplicação TSP em relação ao *speedup* de menor desempenho, *cache* de 128 Kbytes

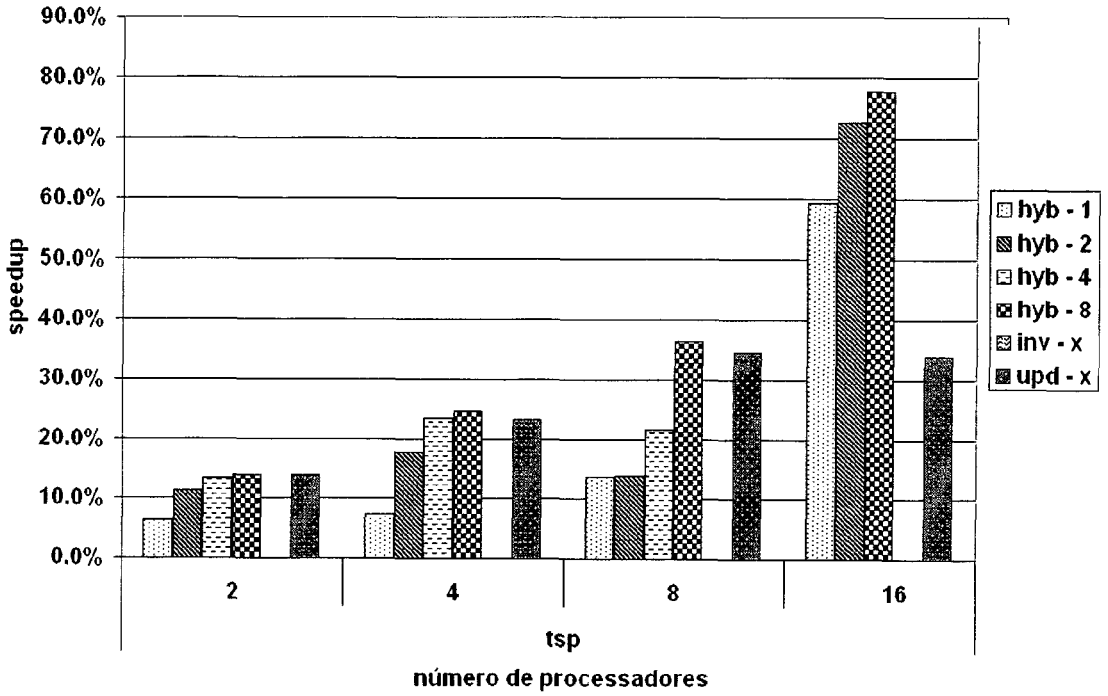

Figura 5.36: Comparativo de ganhos percentuais da aplicação TSP em relação ao *speedup* de menor desempenho, *cache* de 1024 Kbytes

hardware.

É importante ressaltar que, embora alguns protocolos apresentem queda de desempenho com o aumento da *cache*, o protocolo híbrido, com qualquer tamanho de *cache*, se apresenta sempre como a melhor alternativa.

Assim, para todas as aplicações, com exceção de PAN2, a melhor relação custo-benefício foi a utilização de uma *cache* de 512 Kbytes.

Estes resultados indicam que ainda há espaço para melhoramentos do ponto de vista do hardware da máquina DSM, sem o usuário ter que modificar sua aplicação original escrita para máquinas de memória centralizada com barramento.

Capítulo 6

Conclusões e Trabalhos Futuros

Neste trabalho estudamos o impacto de protocolos de coerência de *cache* em hardware DSM, em um sistema paralelo de programação lógica, escrito originalmente para arquiteturas paralelas baseadas em barramento. Sistemas paralelos de programação lógica podem explorar paralelismo implícito de aplicações escritas em Prolog. Cada aplicação pode ter um tipo diferente de paralelismo a ser explorado, que pode gerar diferentes padrões de computação e acessos à memória. Nosso objetivo com este trabalho foi estudar se estes sistemas poderiam alcançar bom desempenho em arquiteturas de hardware DSM, sem modificações estruturais e algorítmicas no código fonte.

Para a realização deste trabalho, utilizamos um simulador de uma máquina *DASH-like*, com até 24 nós, mas realizamos experimentos com 1, 2, 4, 8 e 16 nós. Os protocolos simulados foram os de invalidação, de atualização e híbrido. No protocolo híbrido o processador mantém o dado atualizado durante um número máximo de vezes em que recebe atualização para o dado. É utilizado um contador. Quando o contador atinge o limite, o dado é invalidado na *cache* do processador receptor. Utilizamos quatro aplicações Prolog normalmente utilizadas para avaliação de desempenho de sistemas paralelos de programação lógica. Estas aplicações variam em tamanho, quantidade de computação e tipo de paralelismo a ser explorado.

A análise dos resultados indica que o sistema Andorra-I, originalmente escrito para máquinas baseadas em barramento, apresenta boas taxas de desempenho em sistemas hardware DSM. Porém a escolha do protocolo pode afetar significativamente o desempenho do sistema. Em particular, em várias aplicações para vários números de processadores, o protocolo de atualização produz desempenho melhor do que o protocolo de invalidação, contrário aos estudos

encontrados na literatura, que avaliam aplicações científicas. Porém os ganhos maiores foram obtidos com o protocolo híbrido, que se mostrou uma alternativa eficiente e adaptável aos padrões diferentes de computação encontrados nas quatro aplicações.

Os ganhos obtidos no protocolo híbrido se tornam mais evidentes com o aumento do número de processadores do sistema. Este fato indica que o protocolo híbrido pode ser mais atuante com um número maior de processadores, permitindo o aumento da escalabilidade do sistema.

Os percentuais de ganho do protocolo híbrido sobre os demais aumentam proporcionalmente ao número de processadores. A exceção se encontra na aplicação PAN2, do tipo *-E*, onde em 16 processadores as taxas de ganhos foram percentualmente menores que as encontradas para aplicações com 8 processadores (Fgura 5.12), onde ainda foram obtidos ganhos. Este é outro indicador de que este protocolo aumenta a escalabilidade do sistema.

Em grande parte das simulações com 16 processadores o protocolo híbrido tem obtido ganhos percentuais maiores sobre os demais com o aumento do threshold. É possível que esta tendência continue até um determinado valor de threshold maior que os simulados neste trabalho.

As simulações apresentam comportamentos diferentes no que se refere ao desempenho dos protocolos de acordo com a aplicação e o número de processadores. Os melhores desempenhos podem ser encontrados em protocolos diferentes, conforme a aplicação e o número de processadores utilizados. No caso do protocolo híbrido esta é uma informação importante, pois a escolha do threshold pode determinar o melhor desempenho da aplicação. E esta é uma de suas limitações. Valores diferentes de threshold*s* podem determinar melhor desempenho para diferentes aplicações, com diferentes números de processadores.

A partir destes dados concluímos que o desenvolvimento de um protocolo híbrido que possa determinar dinamicamente o threshold a ser utilizado pelo sistema venha a oferecer ganhos significativos às aplicações lógicas. A escolha do valor do threshold para o sistema pode se adaptar conforme as diferentes etapas de processamento da aplicação permitindo, assim, taxas de ganhos mais significativas. Por exemplo, a aplicação pode estar propensa ao protocolo de atualização durante um determinado momento na execução, alternando para o protocolo de invalidação em outros momentos.

Para o desenvolvimento do *threshold* dinâmico em hardware DSM propomos alterações que permitam ao sistema determinar quantas das alterações enviadas aos demais processadores foram bem sucedidas. A partir destes dados estima-se um valor de *threshold* a ser utilizado nos nós do sistema. Este cálculo pode ser efetuado em períodos. Pois a troca de dados e processamentos estatísticos constantes pode causar perda de desempenho se forem realizadas constantemente. Estes cálculos podem ser feitos por um único processador para, então, enviar o novo valor do *threshold* aos demais processadores.

Uma tarefa a ser avaliada é determinar se um *threshold* específico a cada nó do sistema pode aumentar o desempenho da aplicação.

A unidade de coerência utilizada nestas simulações tem tamanho pequeno e a rede de interconexão tem um desempenho melhor quando comparada aos sistemas software DSM. Desta forma, a execução de programas lógicos paralelos em sistemas software DSM a princípio apresentariam um comportamento diferente. Por exemplo, nas simulações desenvolvidas observamos aplicações que oferecem desempenhos maiores com o protocolo de atualização do que com o protocolo de invalidação, o que provavelmente causaria queda de desempenho em um sistema software DSM devido as características da rede e da unidade de coerência. Em [19] foi apresentada uma migração do sistema Aurora, um outro sistema paralelo de programação lógica que explora apenas paralelismo -OU, para sistemas software DSM, não obtendo ganhos consideráveis.

Uma proposta a ser verificada para aumentar o desempenho de programas lógicos paralelos em sistemas software DSM é o envio antecipado de tarefas associadas a uma mesma cláusula para um mesmo processador. Desta forma, o processador possuiria um conjunto de tarefas, tornando desnecessária a requisição ao escalonador e a respectiva espera para a execução de outra tarefa. A escolha de tarefas associadas a uma mesma cláusula é devido a proximidade dos dados na *cache*, aumentando a probabilidade do dado utilizado nesta tarefa ser o mais recente naquele processador, já que este processador é o que está atuando mais incisivamente sobre este dado. O número de tarefas enviadas ao processador poderia ser determinado de acordo com dados estatísticos, que poderiam ser gerados pelo software DSM.

Porém esta solução está fora do escopo deste trabalho, pois nosso objetivo era migrar o software que foi originalmente escrito para arquiteturas baseadas em barramento, para arquiteturas mais escaláveis, sem modificar o código fonte.

Apêndice A

Parâmetros do backend do simulador

Tabela A.1: parâmetros do simulador - backend

mint options	
[-C interval]	checkpoint interval
[-c file]	file of cycles for operations
[-e file]	use file to generate events
[-h heap_size]	heap size in bytes, default: 65536 (0x10000)
[-i item_space_size]	use separate item size
[-I]	use hardware interlocking. Not supported yet
[-K keyfile]	file containing key for socket connections
[-k stack_size]	stack size in bytes, default: 32768 (0x8000)
[-l page=size,blk=size]	lock allocation policy, default: page=4096,blk=0
[-m mem_size]	align private regions on mem_size boundary
[-O0]	no optimization
[-O1]	execute instruction blocks, default
[-P port]	port for socket connections, default: 19030
[-p procs]	number of per-process regions, default: 32
[-q]	quiet mode – suppress all warning messages
[-r]	back-end uses release consistency
[-R]	do not recycle process ids
[-s shmem_size]	shared memory size, default: 8388608 (0x800000)
[-S]	spin, don't block, on a failed lock acquire
[-t i]	trace instructions
[-t n]	trace nothing
[-t r]	trace memory references, default
[-t s]	trace shared memory references
[-u file]	log events to file for Upshot
[-V]	verify protocol
[-v]	print version number
[-x directory]	use directory for item space
[-y directory]	use directory for shared memory
[-W]	check load-linked addr on every write

Apêndice B

Parâmetros do frontend do simulador

Tabela B.1: parâmetros do simulador - frontend

simulator options	
[-A]	(anticipate write: read-excl if read miss but present in wbuf)
[-a file]	(output file for address info (inv protocol only))
[-b sgi|dec|fast|var]	(type of bus connection to use, default = dec)
[-C 0|1]	(1=use coalescing write buffers, 0=do not coalesce)
[-c file]	(output file for malloc info)
[-E #]	(wait for # write buffer entries before sending head, default=half)
[-F 0|1]	(1=flush cache on first fork, 0=no action, default=1 for upd)
[-h #]	(for dynamic DASH hybrid, WU threshold to stop updates)
[-i file]	(output file for instruction stats)
[-L 0|1]	(1=lock cache lines, 0=lock caches, default=0)
[-l]	(locks are free, instead of causing memory references)
[-m]	(migrate dirty block on read miss for DASH)
[-M WC|RC1|RC2]	(memory model: WC=stall on acq and release, RC1=stall on release, RC2=stall only if full)
[-n #]	(number of processors for DASH)
[-O 0|1]	(1=use prmcache_read optimization, default=1)
[-o file]	(output file for general statistics)
[-P protocol]	(protocol name)
[-p blk=#,siz=#]	(primary blocksize (bytes) and size of cache (bytes))
[-r options...]	(Ricardo's options)
[-R]	(for write-update, do not read block on a write miss)
[-s blk=#,siz=#]	(secondary blocksize (bytes) and size of cache (bytes))
[-S]	(static selection should use read latency)
[-T file]	(record trace to file)
[-t testfile]	(input file for tests)
[-U]	(for Alpha, all processors accept an update if any one does)
[-u file]	(input file specifying which cache blocks should use WU)
[-W #]	(use # number of write buffer entries)

Referências Bibliográficas

[1] K. A. M. Ali and R. Karlsson. The Muse Or-parallel Prolog Model and its Performance. In *Proceedings of the 1990 North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.

[2] A. Beaumont, S. Muthu Raman, and P. Szeredi. Flexible Scheduling of Or-Parallelism in Aurora: The Bristol Scheduler. In Aarts, E. H. L. and van Leeuwen, J. and Rem, M., editor, *PARLE91: Conference on Parallel Architectures and Languages Europe*, volume 2, pages 403–420. Springer Verlag, June 1991. Lecture Notes in Computer Science 506.

[3] Anthony Beaumont, S. Muthu Raman, Vítor Santos Costa, Péter Szeredi, David H. D. Warren, and Rong Yang. Andorra-I:an implementation of the Basic Andorra Model. Technical Report TR-90-21, University of Bristol, Computer Science Department, September 1990. Presented at the Workshop on Parallel Implementation of Languages for Symbolic Computation, July 1990, University of Oregon.

[4] J. Bevemyr, T. Lindgren, and H. Millroth. Reform Prolog: The Language and its Implementation. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 283–298. MIT Press, June 1993.

[5] R. Bianchini and L. I. Kontothanassis. Algorithms for Categorizing Multiprocessor Communication Under Invalidate and Update-Based Coherence Protocols. In *Proceedings of the 28th Annual Simulation Symposium*, April 1995.

[6] I. Bratko. *Prolog Programming for Artificial Intelligence.* Addison-Wesley Ltd., 1986.

[7] Alan Calderwood. Aurora—Description of Scheduler Interfaces. Internal Report, Gigalips Project, January 1988.

[8] Alan Calderwood. Aurora—The Manchester Scheduler. Internal Report, Gigalips Project, January 1988.

[9] J. Campbell, editor. *Implementations of Prolog.* Ellis Horwood, 1984.

[10] Mats Carlsson and Péter Szeredi. The Aurora Abstract Machine and its Emulator. SICS Research Report R90005, Swedish Institute of Computer Science, 1990.

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

[12] Maria Clicia Stelling de Castro. *Técnicas para Detecção e Exploração de Padrões de Compartilhamento em Sistemas de Memória Compartilhada Distribuída.* PhD thesis, COPPE/Sistemas, UFRJ, Dezembro 1998.

[13] William Clocksin and Chris Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[14] Vítor Santos Costa, Ricardo Bianchini, and Inês de Castro Dutra. Evaluating parallel logic programming systems on scalable multiprocessors. In *PASCO '97: Proceedings of the second international symposium on Parallel symbolic computation*, pages 58–67. ACM Press, 1997.

[15] J. A. Crammond. *Implementation of Committed Choice Logic Languages on Shared Memory Multiprocessors.* PhD thesis, Heriot-Watt University, Edinburgh, May 1988. Research Report PAR 88/4, Dept. of Computing, Imperial College, London.

[16] J. A. Crammond. The Abstract Machine and Implementation of Parallel Parlog. Technical report, Dept. of Computing, Imperial College, London, June 1990.

[17] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenstrom. The detection and elimination of useless misses in multiprocessors. In *Proceedings of the 20th ISCA*, pages 88–97, May 1993.

[18] I. C. Dutra E. P. G. de Oliveira, F. Ramos and M. C. S. de Castro. Study of hybrid coherence protocols for parallel logic programming systems. In *The 16th IASTED International Conference on Parallel and Distributed Computing and Systems*, Cambridge, MA, USA, November 2004.

[19] Tatiana Cavalcanti Fernandes. Migração de um sistema de grande porte baseado em memória-compartilhada para um ambientre de memória-compartilhada distribuída. Master's thesis, COPPE/Sistemas, UFRJ, Março 2004.

[20] M. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21:948–960, 1972.

[21] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, University of Tennesse, 1994.

[22] J. R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proceedings of the 10th International Symposium on Computer Architecture*, pages 124–131, 1983.

[23] Gopal Gupta, M. V. Hermenegildo, E. Pontelli, and V. Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *Proceedings of the Eleventh International Conference on Logic Programming*, Italy, June 1994.

[24] Gopal Gupta, Enrico Pontelli, and Manuel Hermenegildo. &ACE: A High Performance Parallel Prolog System. In *Proceedings of the First International Symposium on Parallel Symbolic Computation, PASCO'94*, 1994.

[25] M. V. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.

[26] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. *Algorithmica*, 3:79–119, 1988.

[27] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *USENIX Winter'94 Technical Conference*, pages 17–21, Jan 1994.

[28] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. *Proceedings of the 17th ISCA*, pages 148–159, May 1990.

[29] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The dash prototype: Logic overhead and performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, Jan 1993.

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

[31] E. Lusk, D. H. D. Warren, S. Haridi, et al. The Aurora Or-parallel Prolog System. *New Generation Computing*, 7(2,3):243–271, 1990.

[32] Zohar Manna. *Mathematical Theory of Computation.* McGraw-Hill, 1974.

[33] E. M. McCreight. The Dragon Computer System, an Early Overview. In *NATO Advanced Study Institute on Microarchitecture of VLSI Computers*, July 1984.

[34] Johan Montelius. Penny, A Parallel Implementation of AKL. In *ILPS'94 Post-Conference Workshop in Design and Implementation of Parallel Logic Programming Systems, Ithaca, NY, USA*, November 1994.

[35] S. Raina, D. H. D. Warren, and J. Cownie. Parallel Prolog on a Scalable Multiprocessor. In Peter Kacsuk and Michael J. Wise, editors, *Implementations of Distributed Prolog*, pages 27–44. Wiley, 1992.

[36] V. Santos Costa, R. Bianchini, , and I. Dutra. Evaluating the Impact of Coherence Protocols on Parallel Logic Programming Systems. ES-389/96, COPPE/Sistemas, Universidade Federal do Rio de Janeiro, May 1996.

[37] V. Santos Costa and R. Bianchini. Optimising Parallel Logic Programming Systems for Scalable Machines. In *Proceedings of the EUROPAR'98*, pages 831–841, Sep 1998.

[38] V. Santos Costa, R. Bianchini, and I. C. Dutra. Evaluating the Impact of Coherence Protocols on Parallel Logic Programming Systems. In *Proceedings of the 5th EUROMICRO Workshop on Parallel and Distributed Processing*, pages 376–381, 1997. Also available as technical report ES-389/96, COPPE/Systems Engineering, May, 1996.

[39] V. Santos Costa, R. Bianchini, and I. C. Dutra. Parallel Logic Programming Systems on Scalable Multiprocessors. In *Proceedings of the 2nd International Symposium on Parallel Symbolic Computation, PASCO'97*, pages 58–67, July 1997.

[40] V. Santos Costa, R. Bianchini, and I. C. Dutra. Parallel Logic Programming Systems on Scalable Architectures. *Journal of Parallel and Distributed Computing*, 60(7):835–852, July 2000. http://www.idealibrary.com/links/toc/jpdc/60/7/0.

[41] V. Santos Costa, D. H. D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 83–93. ACM press, April 1991. SIGPLAN Notices vol 26(7), July 1991.

[42] Kish Shen. *Studies of And/Or Parallelism in Prolog.* PhD thesis, Computer Laboratory, University of Cambridge, 1992.

[43] G. Silva, R. Pinto, M. HorMeyll, M. de Maria, L. Whately, J. Barros Jr., Ricardo Bianchini, and C. L. Amorim. O hardware do computador paralelo ncp2 da coppe/ufrj. Technical report, COPPE/Sistemas, UFRJ, June 1996.

[44] Marcio G. Silva, Inês C. Dutra, Ricardo Bianchini, and Vítor Santos Costa. The influence of architectural parameters on the performance of parallel logic programming systems. *Lecture Notes in Computer Science*, 1551:122–??, 1999.

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

[46] T. Chikayama, T. and Fujise, and H. Yashiro. A Portable and Reasonably Efficient Implementation of KL1. In *Proceedings of the Eleventh International Conference on Logic Programming*, June 1993.

[47] A. Tanenbaum and L. Henessy. *Computer Architecture - A Quantitative Approach (Second Edition).* Prentice Hall, 1996.

[48] C. P. Thacker, D. G. Conroy, and L. C. Stewart. The alpha demonstration unit: A high-performance multiprocessor for software and chip development. *Digital Technical Journal*, 4(4):51–65, 1992.

[49] E. Tick. *Memory Performance of Prolog Architectures.* Kluwer Academic Publishers, Norwell, MA 02061, 1987.

[50] J. E. Veenstra and R. J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the 2nd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '94)*, 1994.

[51] J. E. Veenstra and R. J. Fowler. Mint tutorial and user manual. Technical report, University of Rochester, Computer Science Department, 1994.

[52] Karen Villaverde, Enrico Pontelli, Hai-Feng Guo, and Gopal Gupta. Pals: An or-parallel implementation of prolog on beowulf architectures. In *Proceedings of the 17th International Conference on Logic Programming*, pages 27–42. Springer-Verlag, 2001.

[53] D. H. D. Warren and F. C. N. Pereira. An Efficient Easily Adaptable System for Interpreting Natural Language Queries. Technical Note, Dept of AI, University of Edinburgh, 1981.

[54] David H. D. Warren. The SRI Model for Or-Parallel Execution of Prolog—Abstract Design and Implementation Issues. In *Proceedings of the 1987 International Logic Programming Symposium*, pages 92–102, 1987.