ESTUDO DO EFEITO DA SINCRONIZAÇÃO DE BARREIRA IMPLEMENTADA EM SOFTWARE NO DESEMPENHO DE MÁQUINAS PARALELAS

Martha X. Torres D.[1], Edward D. Moreno O.[2], Celso A. S. Santos[3],Sergio T. Kofuji[4]
Laboratório de Sistemas Integráveis
Departamento de Engenharia Eletrônica
Escola Politécnica da Universidade de São Paulo

RESUMO

Este artigo estuda o efeito do "overhead" da sincronização de barreira implementada em software durante a execução de um programa em uma máquina paralela. São estudados os efeitos qualitativos e quantitativos de sincronização de barreira implementada em software para máquinas de memória compartilhada e descrito um algoritmo de sincronização de barreira para uma máquina de memória distribuida com rede mesh. São desenvolvidos dois modelos para obter quantitativamente o efeito do "overhead" da sincronização de barreira implementada em software no desempenho de laços paralelos: um modelo analítico, para máquinas de memória compartilhada baseadas em barramento, e um modelo de simulação, para máquinas de memória distribuída com rede mesh.

ABSTRACT

This paper examines the effect of overhead of the coordination operations implemented in software during the execution of programs in a parallel machine. The qualitative and quantitative effects of the synchronization barrier implemented in software for machines of shared memory are studied. Two models for obtaining the effect of overhead of the synchronization barrier implemented in software on the performance of parallell loops are also developed. An analytical model for bus-based machines and a simulation model for machines of distributed memory with mesh network.

[1] Pesquisador do LSI da USP. E-mail: edmoreno@lsi.usp.br
[2] Pesquisador do LSI da USP. E-mail: mxtd@lsi.usp.br
[3] Pesquisador do LSI da USP. E-mail: mxtd@lsi.usp.b
[4] Professor e Pesquisador do LSI da USP. E-mail: kofuji@lsi.usp.br

1. Introdução

A execução de programas em máquinas paralelas exige a coordenação entre os processadores para que possam compartilhar recursos e trocar dados corretamente. Operações como escalonamento, comunicação entre os processadores e sincronização realizam essa coordenação.

Estas operações constituem um "trabalho extra" na execução do programa e provocam um "overhead" que é somado ao tempo de execução paralela. A maioria destas operações é inerentemente seqüencial ou necessita do uso exclusivo dos recursos compartilhados, não podendo ser totalmente executadas de forma paralela em uma máquina real. Seu tempo de execução constitui, portanto, um verdadeiro fator limitante no processamento paralelo.

Sincronização de Barreira: É uma operação comum de coordenação de processadores que estabelece um ponto específico (barreira) do programa onde todos os processadores devem chegar antes de continuar a execução.

Os laços são uma fonte muito importante de paralelismo [1] [2] e seu tempo de execução afeta consideravelmente o tempo de execução total de um programa paralelo. A sincronização de barreira é geralmente utilizada na execução de um laço paralelo para sincronizar os processadores dentro do corpo do laço ou em seu término. Como sua execução provoca um "overhead" que pode afetar muito negativamente o desempenho na execução do laço, é fundamental que se obtenha soluções para sincronização de barreira que sejam eficientes. Procura-se obter uma solução que seja escalável, ou seja, que seu tempo de execução não aumente quando o número de processadores participando na sincronização de barreira aumenta.

Estabelecer modelos que mostrem quantitativamente o efeito da execução de uma determinada solução para os problemas de coodenação no desempenho de uma tarefa paralela, ajuda a avaliar as vantagens e desvantagens das soluções propostas, além de contribuir no entendimento de seu comportamento dentro de uma arquitetura determinada.

Neste artigo vai-se mostrar, na seção 2, algumas das melhores soluções existentes para sincronização de barreira em software para máquinas de memória compartilhada e seus diferentes efeitos sobre o desempenho. Também se mostra quantitativamente, na secão 3, o efeito do "overhead" de um algoritmo de sincronização de barreira em software para uma máquina paralela de memória compartilhada baseada em barramento . Com base nas soluções usadas para sincronização de barreira em software em máquinas fortemente acopladas, propõe-se, na seção 4, um algoritmo de sincronização de barreira em software para uma máquina multicomputadora com arquitetura mesh bidimensional e se avalia seu desempenho usando um modelo de simulação desenvolvido especificamente para essa arquitetura e esse algoritmo.

Existem soluções em hardware que tentam eliminar o "overhead" causado pelos algoritmos de sincronização de barreira em software; alguns deles serão mencionados na seção 5. Soluções simples, dedicadas especialmente para realizar esta operação, constituem uma boa opção para melhorar o desempenho.

2. Soluções de Sincronização de Barreira por Software

Existem muitas soluções a nível de software para a sincronização de barreira. Vamos descrever algumas das soluções mais relevantes que tentam ser eficientes e escaláveis. O bom desempenho de um algoritmo de sincronização de barreira em software, depende principalmente da arquitetura da máquina paralela (se usa interconexão por barramento ou uma rede, se tem coerência de cache, se fornece primitivas de sincronização, etc.) e da colocação física das variáveis envolvidas no algoritmo. Por exemplo, um algoritmo que tem um desempenho razoável em uma máquina com rede de interconexão multiestágio pode ter um desempenho ruim em uma máquina baseada em barramento.

2.1. Barreira Centralizada ou Contador Central

Esta é a solução tradicional [3] [4]: consiste em usar um contador central compartilhado, onde cada processador que chega à barreira incrementa o contador na memória compartilhada (o contador inicialmente tem o valor zero), e fica esperando em um laço ("spin waiting") até que o contador central chegue ao valor de N (número total de processadores participando na sincronização de barreira).

Claramente se observa que cada processador deve realizar uma operação de exclusão mútua para acessar o contador, tomando $\Omega(N)$ (mínimo 2N) unidades de tempo para executar esta operação [5][4]. São também chamadas barreiras lineares[3] pois o tempo de execução aumenta proporcionalmente ao número de processadores, sendo em geral não escaláveis. Os algoritmos de barreira centralizada apresentam vários inconvenientes: 1. Não são escaláveis. 2. Com um número grande de processadores produzem acessos do tipo "hot spot"(quando um número grande de processadores tenta acessar ao mesmo tempo uma mesma variável) gerando saturação do barramento, contenção de memória e saturação de árvore na rede de interconexão. 3. Todos os processadores realizam o "spin waiting" sobre a mesma variável compartilhada.

2.2. Barreira em Árvore [6] [7]

Este algoritmo converte a variável compartilhada comum (do algoritmo de barreira centralizada) em uma árvore de variáveis, com cada nó da árvore representando um módulo de memória diferente. Os processadores são divididos em grupos e os processadores em cada grupo sincronizam-se internamente. Um processador de cada grupo vai se sincronizar com os demais grupos no nível seguinte da árvore e assim por diante até conseguer a sincronização total. A sincronização em cada grupo é generalmente feita como na barreira centralizada. O tamanho da barreira em árvore depende do

número de processadores em cada grupo. Fazendo uma comparação deste algoritmo com o algoritmo anterior, temos que para executar a barreira em árvore são utilizadas mais variáveis, mas com menos acessos sobre elas, ou seja, o "hot spot" diminuiu, mas o número de variáveis de sincronização aumentou. A barreira em árvore produz mais operações de sincronização (cada processador produz $O(\log_{fan\text{-}in}(P))$ operações), mas se o algoritmo esta sendo executado em uma máquina que tenha rede de interconexão, estas operações podem ser executadas concorrentemente (em máquinas baseadas em barramento os acessos são serializados) produzindo um "overhead" total logarítmico com o número de processadores. Este algoritmo também é chamado de barreira logarítmica.[3], e foi baseado na solução em hardware para redes de interconexão desenvolvido em [8], o qual implementa fisicamente a árvore na rede.

O principal objetivo deste algoritmo é diminuir o "hot spot", sendo importante que cada variável compartilhada fique em um módulo de memória diferente. Isto é sempre possível: a maior árvore é aquela que tem o maior número de nós, ou seja, a árvore que tem o mínimo fan-in (2). O número total de nós em uma árvore com N processadores é N/2 + N/4 + ... + 2 + 1 = N - 1. Em geral, em uma máquina paralela o número de processadores é igual ao número de módulos de memória [6].

2.3. Nova Barreira em Árvore[4]

Este algoritmo é feito com o objetivo de eliminar o "hot spot" causado pelo "spin waiting". Na barreira centralizada o "spin waiting" é feito somente sobre uma variável, enquanto na barreira em árvore é realizado sobre variáveis que são compartilhadas por um número fixo de processadores (tamanho do grupo). Neste novo algoritmo cada processador realiza o "spin waiting" sobre uma variável que não é comum a outro processador, podendo ser uma variável local, se houver coerência de cache, ou simplesmente uma que esteja em um módulo de memória diferente, no qual seu acesso não gera contenção, nem na memória, nem na rede de interconexão. Este algoritmo pode ser classificado como barreira logarítmica e seu comportamento apresenta um melhor desempenho em máquinas com redes de interconexão, onde as operações de sincronização podem ser feitas concorrentemente.

2.4. Barreiras não Centralizadas

Existem outras soluções nas quais se aproveita a comunicação entre os processadores com o objetivo de aumentar a concorrência; cada processador precisa de uma identificação virtual para executar cada um destes algoritmos. Uma solução é a barreira Butterfly [9.] na qual a sincronização de P processadores é efetuada em $\log_2 P$ estágios. No estágio k o processador i comunica-se com o processador $i \oplus 2k$: o processador i leva a 1 uma variável do processador $i \oplus 2k$ e fica esperando até que o processador $i \oplus 2k$ leve a 1 sua variável. Como em cada estágio cada processador usa um número de variáveis diferentes, precisa-se portanto de um espaço da ordem de $O(P\log_2 P)$ para as variáveis compartilhadas. Quando o número de processadores participando na sincronização de barreira não é uma potência de 2, o número de operações no algoritmo aumenta e sua eficiência diminui.

O algoritmo de disseminação de barreira [10] também é efetuado em $\log_2 P$ estágios, onde no estágio k o processador i efetua a sincronização com o processador $(i-2^k) \bmod P$. A comunicação é feita da mesma maneira como se faz na barreira butterfly. A vantagem sobre a barreira butterfly é que neste algoritmo pode-se fazer a sincronização com qualquer número de processadores.

Existe também o algoritmo torneio de barreiras[10][5], que também é executado em $\log_2 P$ estágios e precisa de um espaço de $O(P \log_2 P)$, mas a diferença é que o número de operações em cada estágio diminui.

Nestes algoritmos o "hot spot" não é crítico, pois em cada estágio o processador acessa uma variável diferente. Estes algoritmos são mais eficientes em máquinas com rede de interconexão porque em cada estágio as comunicações entre os diferentes processadores podem ser feitas concorrentemente. Cada processador produz somente $\log_2 P$ sincronizações, mas o número de operações necessárias aumenta consideravelmente quando o número de processadores aumenta.

3. Efeito Quantitativo do "Overhead" de Sincronização de Barreira em uma Máquina de Memória Compartilhada Baseada em Barramento

Como temos visto, o desempenho de um algoritmo de sincronização depende da arquitetura utilizada. No caso de uma máquina baseada em barramento, os algoritmos que acessam variáveis diferentes em módulos de memória diferentes (como na barreira em árvore), ou que aproveitam uma comunicação concorrente entre os processadores (como as barreiras não centralizadas) não apresentam vantagem, pois os acessos simultâneos aos diferentes módulos de memória são serializados no barramento. Portanto, os algoritmos de barreira centralizada oferecem um melhor comportamento, além de produzir um número menor de operações. Para se obter quantitativamente o efeito do "overhead" da sincronização de barreira, desenvolveu-se um modelo analítico, tomando-se por base o modelo proposto por Beckmann [3], o qual tem em conta as seguintes considerações:

- A máquina tem memória local, mas não tem coerência de cache.
- O mecanismo de arbitração do barramento usa uma política "round robin".
- O hardware fornece uma primitiva de sincronização Fetch&Add(A,R) que se executa rapidamente sobre o barramento. Esta operação consiste em ler o valor A, acessando o barramento, e escrever A+R nessa posição de memória sem que outra operação intervenha nessa operação.
- A máquina de P processadores está executando um laço paralelo com N iterações e cada iteração demora *b* unidades de tempo constantes para ser executado. Sua execução não precisa de acesso ao barramento, ou seja, as operações que precisam do uso do barramento são as operações de coordenação.
- O algoritmo de sincronização de barreira escolhido é um algoritmo de barreira centralizada, pelas razões descritas anteriomente. Um código possível pode ser [3]:

```
IF Fetch&Add(BARRIER,1) = NPROCS -1
   BARRIER = 0
   BROADCAST = POLARITY
ELSE
   WHILE BROADCAST != POLARITY DO
   CONTINUE
ENDIF
POLARITY = NOT POLARITY
```

Neste algoritmo BARRIER e BROADCAST são variáveis compartilhadas o "spin waiting" é feito sobre BROADCAST e os incrementos são feitos sobre BARRIER.

- O escalonamento das iterações é feito de maneira dinâmica usando "self-scheduling"[11][12]. Cada processador executa um algoritmo para pegar uma determinada iteração em tempo de execução. Um algoritmo para "self-scheduling" pode ser[3]:

```
LOOP
   MYITER = Fetch&Add(I,1)
   IF MYITER > N THEN
      CALL BARRIER
      EXIT LOOP
   ELSE
      EXECUTE INSTRUCT(MYITER)
   ENDIF
GOTO LOOP
```

MYITER é uma variável local que tem o número da iteração que cada processador tem que executar, I é a variável compartilhada que tem que ser atualizada em cada iteração. INSTRUCT(MYITER) é o laço na iteração MYITER.

- A contribução, no tempo de execução do laço paralelo, feita pelo mecanismo de escalonamento é *a* unidades de tempo, onde *a* corresponde ao o tempo que cada processador gasta para pegar o valor da iteração que vai executar (isto é feito em cada iteração) e incrementar o valor da variavél compartilhada I.

- No modelo desenvolvido por Beckmann [3] não se tem em conta a contenção do barramento causado pelo algoritmo de sincronização nem o "overhead" causado pelo " spin waiting". Fazendo algumas considerações, pode-se obter um modelo que o considere. Primeiro descreve-se o modelo de Beckmann, depois o modelo modificado e finalmente nossos resultados.

3.1. Modelo de Beckmann [3]:

Este modelo leva basicamente em conta dois casos:

1. Barramento não saturado: Quando $(P-1)a<b$, o barramento não é usado totalmente.
   Se N é múltiplo inteiro de P o tempo de execução paralela é:

$$T[P]=\frac{N}{P}(a+b)+(a+c+d) \tag{3.1}$$

   Se N não é multiplo de P o tempo de execução paralela é

$$T[P]=\left\lfloor\frac{N}{P}\right\rfloor(a+b)+(N \bmod P)(a+b)+(2P-N \bmod P)d \tag{3.2}$$

   Considerando que o tempo de execução serial é : $Ts=Nb$ então o "speedup" é:

$$S[P]=\frac{Nb}{T[P]} \tag{3.3}$$

2. Barramento saturado: Quando $(P-1)a \geq b$, o barramento é saturado pelos pedidos gerados pelo escalonamento; portanto é mais significativo a contribução do "overhead" do escalonamento do que a contribução do "overhead" de sincronização de barreira. O tempo de execução paralela é (Sem levar em conta se N é ou não divísivel por P):

$$T[P]=(N+P)a+P(c+d) \tag{3.4}$$

Então o "speedup" é:como (3.3) usando (3.4) como T[P].

3.2. Modelo de Beckmann Modificado

Temos considerado que o barramento sempre está saturado, ou seja, sempre está totalmente ocupado (devido à natureza do algoritmo de sincronização):

1. Quando $(P-1)a<b$ e N é múltiplo de P, o tempo de execução paralela é (3.1).
2. Quando $(P-1)a<b$ e N não é de múltiplo de P, temos:

   2.1. Se $b-(P-1)a<(P-N \bmod P)c$, como na figura 3.1, com P=5, N=12,b=8a, d=a=1, c=3a. Neste caso, tem-se saturação do barramento causada pelo incremento à variável compartilhada no algoritmo de sincronização de barreira e aparece a contribução do "spin waiting".

$$T[P]=\left\lfloor\frac{N}{P}\right\rfloor(a+b)+P(a+c)+d(2P-N \bmod P)+a(N \bmod P) \quad (3.5)$$

2.2. Se $b-(P-1)a \geq (P-N \bmod P)c$, Neste caso todos processadores que acabaram de executar suas iterações tem incrementado a variável compartilhada; então os processadores podem tentar ler a variável compartilhada através do barramento ("spin waiting"), como na figura 3.2, com P=4,N=14,b=13a, d=a=1, c=3a.

Como $b-(P-1)a-(P-N \bmod P) \geq 0$ então $b=(P-1)a+(P-N \bmod P)+Xd$ e com d=1 temos $X=b-(P-1)a-(P-N \bmod P)$

Dependendo do valor de X pode-se saber a contribuição do overhead do spin-waiting no cálculo do tempo de execução paralela.

2.2.1 Se X é múltiplo de (P-NmodP) e $X \geq$ (P-NmodP) , como a figura 3.2,então:

$$T[P]=\left\lfloor\frac{N}{P}\right\rfloor(a+b)+N \bmod P(a+c)+Pd+(P-N \bmod P)d \quad (3.6)$$

2.2.2 Se X não é múltiplo de (P-NmodP) ou $X \leq$ (P-NmodP), como a figura 3.3 com, P=4, N=14, b=10a, d=a=1, c=3a então:

$$T[P]=\left\lfloor\frac{N}{P}\right\rfloor(a+b)+N \bmod P(a+c)+Pd+(P-N \bmod P)d+((P-N \bmod P)-X \bmod (P-N \bmod P))d \quad (3.7)$$

3. Barramento saturado pelo escalonamento: Quando $(P-1)a \geq b$ o tempo de execução paralela fica como em (3.4).

Para avaliar o desempenho usa-se (3.3) com os novos valores de T[P]. Na figura 3.4 mostra-se a influençia do "overhead" do algoritmo de sincronização de barreira. Quando a máquina paralela está executando N=64 iterações com b=200, a=3, d=1, c=3,6 e 16.

A figura 3.5 mostra a comparação entre os dois modelos para a=3, c=6,b=200,N=64,d=1, Claramente se observa que existe uma degradação no desempenho da máquina causado pelo algoritmo de sincronização. No modelo melhorado o desempenho começa a diminuir com um número menor de processadores, pois no modelo é levado em conta o efeito do "overhead" para mais situações.

O modelo também serve para avaliar o efeito do "overhead" causado pelo algoritmo de escalonamento, como se mostra na figura 3.6, para N=64, b=200, c=3, d=1 e a=1,2,3,4,6,16. Estes resultados são praticamente os mesmos obtidos com o modelo de Beckmann.

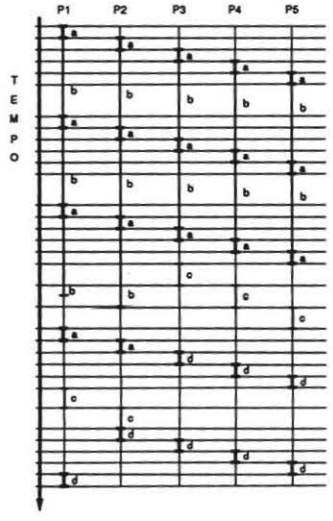

Figura 3.1 Execução de um laço numa máquina com barramento caso 2.1

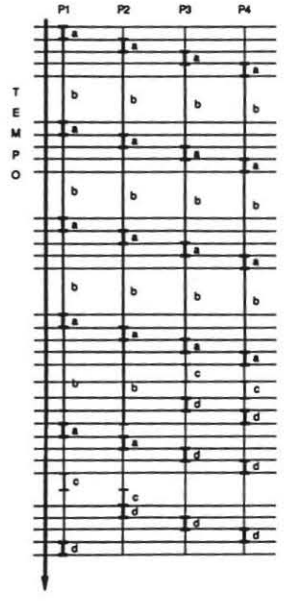

Figura 3.3 Execução de um laço numa máquina com barramento caso 2.2.2

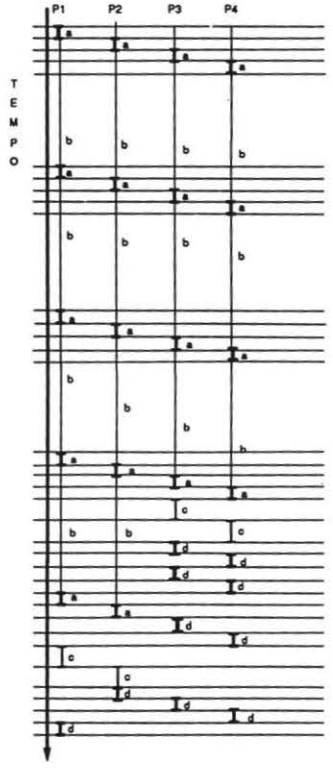

Figura 3.2 Execução de um laço numa máquina com barramento caso 2.2.1

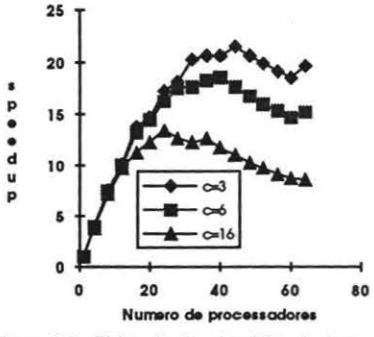

Figura 3.4. Efeito do "overhead"do algoritmo de sincronização de barreira no "speedup".

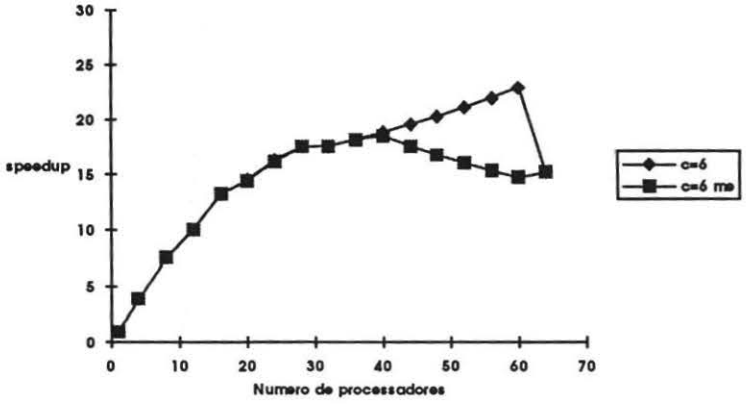

Figura 3.5. Comparação entre o modelo de Beckmann[3] e o melhorado.

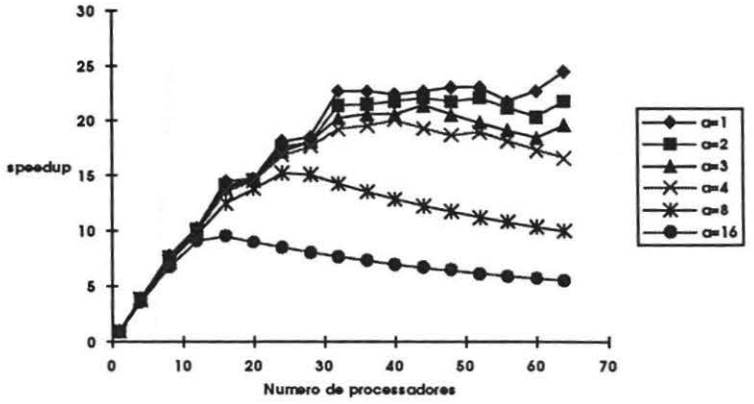

Figura 3.6. Efeito do "overhead" causado pelo mecanismo de escalonamento usado

4. Efeito Quantitativo do "Overhead" de Sincronização de Barreira em uma Máquina de Memória Distribuída com Rede Mesh Bidimensional

Primeiro vai-se explicar o algoritmo proposto para a sincronização de barreira nesta arquitetura, onde se a memória está distribuída entre os nós, e o acesso remoto se faz através de mensagens ("Distributed Shared Memory"). A seguir se faz o modelo de simulação para estudar quantitativamente o seu efeito no desempenho de sistema. Este algoritmo está baseado principalmente nos algoritmos de barreira não

centralizada pois estes aproveitam mais a comunicação entre os processadores e não realizam uma sincronização de barreira centralizada.

Na realização deste algoritmo se fizeram as seguintes suposições:

- Cada elemento de processamento EP tem a capacidade de receber simultaneamente vários acessos de escrita na sua memória , mas estes acessos devem estar localizados em diferentes posições da memória
- Cada elemento de processamento EP tem a capacidade de enviar concorrentemente várias petições de escrita a EPs diferentes.
- Cada EP permite acesso à sua memória através de mensagens pela rede de interconexão e os EPs se comunicam mediante o acesso às diferentes memórias.
- O mecanismo de escalonamento usado é "pre-scheduling".

4.1. Princípio de Funcionamento

O algoritmo é executado por etapas, e cada PE se comunica com todos seus vizinhos em cada etapa; o objetivo em cada etapa é entregar uma informação a mais entre os PEs até que cada PE fique sabendo da chegada de todos os demais. Quando um PE precisa fazer a sincronização de barreira ele entra na primeira etapa do algoritmo: envia sinais a seus vizinhos e depois fica esperando pelos sinais de seus vizinhos e quando todos os seus vizinhos tiverem enviado os sinais passa à seguinte etapa, e assim sucessivamente, até saber que todos os PEs chegaram na barreira. Por exemplo, com NEP (número de elementos de processamento) igual a 4 temos

Primeira etapa (figura 4.1): Quando EP1 acaba de executar sua tarefa (por exemplo suas iterações em um loop paralelo) ele envia dois sinais para EP2 e EP3 falando de sua chegada e espera que cheguem sinais de EP2 e EP3. Quando os dois sinais chegam, EP1 passa à etapa seguinte

Segunda etapa (figura 4.1): EP1 envia sinal a EP2 falando que EP3 já chegou e também envia sinal para EP3 falando que EP2 já chegou. Fica então esperando as mensagens de EP2 e EP3 que dirão que EP4 chegou. Dessa maneira, em duas etapas EP1 tem a informação de que todos os EPs envolvidos na sincronização de barreira chegaram, concorrentemente acontece o mesmo para EP2, EP3 e EP4, como se apresenta na figura 4.1.

O número de etapas depende do número de elementos de processamento e da maneira como são alocados pelo compilador:

- No caso em que a geometria usada pelo compilador para alocar os elementos de processamento na arquitetura mesh seja de dimensões $2*N$ com $N= 1,2,3,...$ Como na figura 4.2. Temos que com NEP=2 ou N=1 precisa-se de somente uma etapa; no caso de NEP=4 ou N=2 como se falou anteriormente, precisa-se de 2 etapas; para NEP=6 ou N=3 precisa-se de 3 etapas (figura 4.2);

para NEP=8 precisa-se de 4 etapas; para NEP=10 de 5 etapas; para NEP=12 de 6; para NEP=14 de 7; para NEP=16 de 8, e em geral para NEP= 2*N precisa-se de NEP / 2 etapas

- No caso em que a geometria usada pelo compilador para alocar os elementos de processamento na arquitetura mesh seja de dimensões 3*N com N=1,2,3,..., temos que com NEP=3 precisa-se de 2 etapas; com NEP=6 de 3 etapas; com NEP=9 de 4 etapas; com NEP=12 de 5 etapas, e assim por diante.
- No caso em que a geometria usada pelo compilador para alocar os elementos de processamento na arquitectura mesh seja de dimensões 4*N com N=1,2,3,..., temos que com NEP=4 precisa-se de 3 etapas; com NEP=8 de 4; com NEP=12 de 5; com NEP=16 de 6, e assim por diante.

Em geral, para qualquer geometria usada pelo compilador de dimensões M*N com N=1,2,3,.., o número de etapas é M-1, M , M+1,M+2,...respectivamente, com M=1,2,3,....

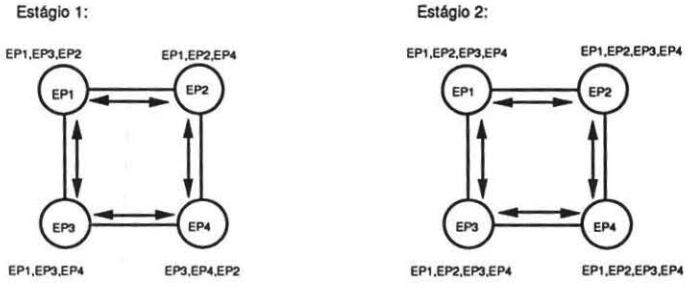

Figura 4.1 Execução do algoritmo de barreira de sincronização para NEP=4 usando uma geometria 2*2

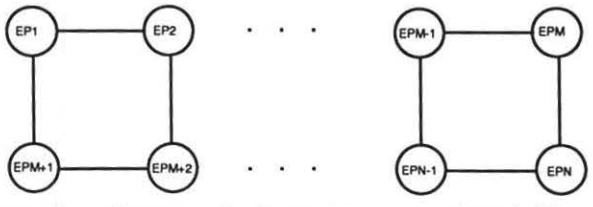

Figura 4.2 Geometria 2*N alocada pelo compilador em uma rede mesh bidimensional

4.2. Implementação do Algoritmo

Assumindo que qualquer vizinho de PE possa acessar sua memória, temos: cada PE reserva um número de variáveis igual ao número de vizinhos vezes o número de etapas (com "pre-scheduling" pode se saber isso). Por exemplo, na figura 4.3, com NEP=6, para uma geometria 2*N, os elementos de processamento 1,4,3,6 alocam 6 variáveis, usando 2 em cada etapa; quando EP1 chega na barreira ele leva a 1 uma variável em EP2 e em EP4, via barramento, e fica esperando que 2 variáveis em sua

memória sejam estabelecidas por EP2 e EP4; na segunda etapa novamente leva a 1 uma variável em EP2 e em EP4 (diferente à da primeira etapa) Na terceira etapa se faz o mesmo , mas em variáveis diferentes. Portanto EP1,EP4,EP3,EP6 precisam de 6 variáveis diferentes e EP2 e EP5 de 9 variáveis diferentes.

Este algoritmo precisa que os processadores tenham uma posição fixa antes de executar o algoritmo. Dependendo da geometria, o número de vizinhos muda, e portanto também o número de variáveis reservadas.

4.3. Modelo de Simulação para Avaliar o Desempenho de Algorimo de Sincronização

A simulação é feita com as seguintes suposições :

- O mecanismo de escalonamento usa uma alocação de geometria 2*N.
- O "overhead" é tomado como um valor fixo, que é tomado como sendo o custo para usar o barramento ao fazer as escritas às variáveis dos vizinhos, medido em ciclos de relógio.
- A máquina está executando um loop paralelo de N iterações e cada EP executa N/NPE iterações alocadas em tempo de compilação
- O tempo de execução de cada iteração é variável e segue uma distribuição Normal, com valor médio= b e desvio padrão=d, medidas em ciclos de relógio.
- O tempo de execução serial é $Ts = N \times Ti$ onde Ti é o tempo de execução de cada iteração, o qual segue uma distribuição Normal.

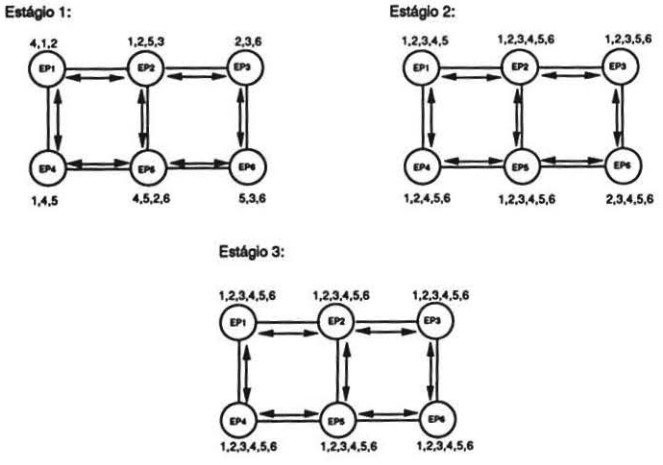

Figura 4.3 Execução do algoritmo de barreira de sincronização para EP=6 usando uma geometria 2*3.

O efeito do algoritmo quando a máquina está executando um laço é mostrado na figura 4.4, quando N(número de iteracões) =64; o valor médio das iteracões é b=10, o desvio padrão d=0.1. Observa-se uma diminuição do speedup quando se aumenta o valor do "overhead", que neste caso representa a latência do barramento e o custo da escrita na memória do EP vizinho na execução sincronização de barreira. Mostra também que quando o número de processadores que executam o laço aumenta o desempenho diminui ,ou seja, apresenta um comportamento não escalável.

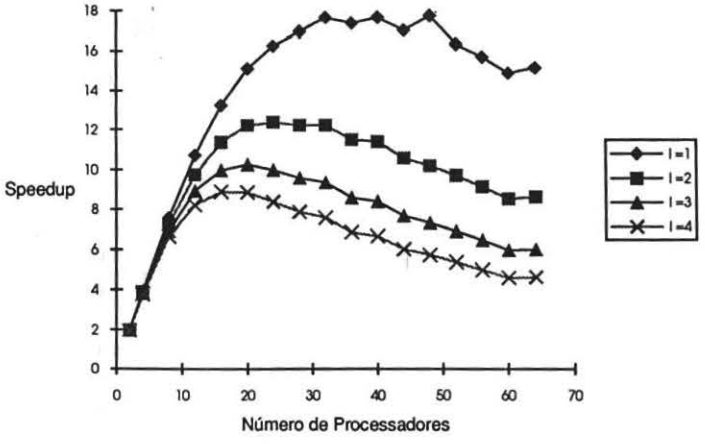

Figura 4.4. Efeito do "overhead" do algoritmo de sincronização no desempenho da máquina.

5. Outras Soluções para a Sincronização de Barreira

Embora as soluções em software sejam mais baratas, seu desempenho não é muito bom, especialmente em máquinas de memória compartilhada baseadas em barramento. Com o objetivo de eliminar o "overhead" causado pela sincronização de barreira, tem sidos propostas soluções em hardware. Uma solução para diminuir o "overhead", especialmente aquele causado pelo "spin waiting", é o armazenamento em cache das variáveis de sincronização. Se estas estiverem no cache de cada processador, o "spin waiting" vai ser feito localmente e somente terá influência no tráfego do elemento de interconexão quando as variáveis trocarem de valor. Apesar disso, existe um problema de coerência cache cuja solução obriga que o hardware das caches e do elemento de interconexão.(barramento ou rede) possuam algumas características especiais[13] e contribua com um novo "overhead".

Existem outras soluções, como a do Alliant FX-8 [3], no qual existe um barramento dedicado somente às operações de sincronização, que apresenta um bom desempenho, mas com a utilização de apenas 8 processadores

Uma solução para evitar o problema de "hot spot" foi desenvolvida por [8] para redes de interconexão, mediante o chamado "hardware combining". Essa solução consiste em combinar, nos elementos de chaveamento da rede de interconexão,. os pedidos dos processadores para acessar (leitura ou escrita) uma mesma posição de memória e produzir apenas um pedido. Como os acessos são combinados na rede de interconexão, a contenção de memória pode ser eliminada[6]. As desvantagens do " hardware combining" são sua complexidade e seu custo, pois os elementos de chaveamento devem ter um hardware adicional (o custo de uma malha "combining" pode ser de 6 a 30 vezes maior do que uma malha não "combining"[6]).

Atualmente, tem-se proposto soluções de hardware dedicadas muito simples, como a proposta em [3] ou a de [14], com as a quais consegue-se obter "overhead" muito reduzido. No entanto, estas soluções ainda não são escaláveis, além de que possuem limitações físicas ou exigem hardware mais complexo quando se precisam usar varias barreiras de sincronização concorrentemente

6. Conclusões

Mediante os modelos desenvolvidos, verificou-se que o efeito das operações de coordenação no desempenho da máquina é bastante considerável. Soluções de software são muito mais econômicas e podem chegar a ter um desempenho razoável dependo da arquitetura e das políticas de alocação de variáveis, mas as soluções de hardware são muito mais eficientes, especialmente para máquinas de memória compartilhada baseadas em barramento. Soluções simples de hardware dedicado especialmente para resolver o problema de sincronização de barreira, podem ser a solução para eliminar o "overhead", mas as soluções atuais ainda não conseguem ser escaláveis. A solução de software proposta para máquinas de memória distribuida com rede mesh foi importante para avaliar o comportamento de máquina. As soluções em hardware seguramente representarão soluções mais eficientes do que as soluções em software.

Com base na pesquisa realizada, os objetivos futuros são: estabelecer uma solução de hardware para uma máquina de memória partilhada distribuida (virtualmente partilhada) que usa rede mesh, que apresente um bom desempenho e desenvolver um modelo de simulação para quantificar seu comportamento.

7. Referências

[1] C. D. Polychronopoulos, "The impact of Runtime Overhead on Usable Parallelism", Proc. Int. Conf. of Parallel Processing, 1988

[2] C. Beckmann, .C. D. Polychronopoulos, "The Effect of Barrier Synchronization and Scheduling Overhead on Parallel Loops", .Proc. Int. Conf. of Parallel Processing, 1989

[3] C. Beckmann, "Reducing Synchronization and Scheduling Overhead in Parallel Loops", M.S. thesis, University of Illinois at Urbana-Champaign, August. 1989

[4] J. M. Mellor-Crummey, M. L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory multiprocessors", ACM Trans. on Computer Systems, Vol. 9, No. 1, Fev. 1991

[5] B. D. Lubachevsky, "Synchronization Barrier and Related Tools for Shared Memory Parallel Programing", Proc. Int. Conf. of Parallel Processing, 1989

[6] P. C. Yew, N. F.Tzeng, D. H. Lawrie, "Distributing Hot-Spot Addressing in Large-Scale Multiprocessors", IEEE Trans. Computers, April 1987

[7] D. N. Jayasimha, "Distributed Synchronizers", Proc. Int. Conf. of Parallel Processing, 1988.

[8] G. F. Pfister, V.A. Norton,"Hot Spot Contention and Combining in Multistage Interconnection Networks ", Proc. Int. Conf. of Parallel Processing, 1985

[9] E. D. Brooks, "The Butterfly Barrier", Int. Journal of Parallel Programming, Vol. 15, No. 4, 1987

[10] D. Hensgen, R. Finkel, U. Manber, "Two Algorithms for Barrier Synchronizatio", Int. Journal of Parallel Programming, Vol. 17, No. 1, 1988

[11] C. D. Polychronopoulos, D. J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers", IEEE Trans. Computers, Dec. 1987

[12] P. Tang, P-C. Yew, "Processor Self-Scheduling for Multiple-Nested Parallel Loops", Proc. Int. Conf. of Parallel Processing, 1986

[13] P. Bitar, A. M. Despain, "Multiprocessors Cache Synchronization. Issues, Innovations, Evolution", 13th Int. Symposium on Computer Architecture, Tokyo,Japan, June 1986

[14] K. Ghose, D-C Cheng, "Efficient Synchronization Schemes for Large-Scale Shared-Memory Multiprocessors", Proc. Int. Conf. of Parallel Processing,1991