Um Sistema de Programação Paralela com Variáveis Compartilhadas para Arquiteturas Distribuídas

Luciana Bezerra Arantes, Liria Matsumoto Sato

Laboratório de Sistemas Integráveis
Escola Politécnica da Universidade de São Paulo.
Av. Prof. Gualberto, trav.3, 158, CEP 05508-900, São Paulo.
e-mail: {arantes, liria}@lsi.usp.br

Resumo

Sistemas com Memória Compartilhada Distribuída (DSM) implementam a abstração da memória compartilhada para arquiteturas distribuídas, mas nem sempre oferecem uma linguagem de programação que facilite o desenvolvimento das aplicações. O sistema de programação e processamento **CPAR-DSM**, aqui apresentado, não só provê o modelo de programação com variáveis compartilhadas, mas também construções de linguagem para expressar o paralelismo, balancear o processamento de laços paralelos e organizar a memória compartilhada de forma hierárquica. Para executar as aplicações numa arquitetura distribuída e simular a existência da memória compartilhada, o suporte de processamento do sistema CPAR-DSM utiliza os mecanismos oferecidos por um sistema DSM, explorando ao máximo o princípio da localidade dos dados para obter um melhor desempenho das aplicações.

Abstract

Distributed Shared Memory systems (DSM) implement the shared-memory abstraction on distributed architectures, but they do not always offer a programming language to ease the development of application. The programming and processing **CPAR-DSM** system, presented in this article, provides not only the shared-memory programming model, but also language constructions for expressing parallelism, load balancing of parallel loops and organizing shared memory in a hierarchic manner. To run the programs in a distributed architecture and simulate the shared memory, the CPAR-DSM run-time library uses the mechanisms offered by a DSM, exploring the data locality at the utmost in order to achieve a better performance of its applications.

1 Introdução

O crescente aumento da capacidade computacional dos microprocessadores e o avanço das redes de alta velocidade vêm contribuindo para que os sistemas distribuídos sejam cada vez mais adequados para as aplicações que demandam alto desempenho. Por outro lado, o desenvolvimento de programas distribuídos não é uma tarefa trivial, principalmente quando baseados no paradigma de passagem de mensagem, onde os processos precisam tratar explicitamente o envio e o recebimento de dados remotos.

Essa complexidade não existiria se os processadores pudessem compartilhar, como nas máquinas multiprocessadoras, uma memória comum a que todos eles tivessem acesso. Este é o principal objetivo dos **sistemas com Memória Compartilhada Distribuída** (*DSM - Distributed Shared Memory*). Eles implementam um espaço de endereçamento de dados lógico, global, a que todos os processadores têm acesso [3].

Entretanto, a maioria dos sistemas DSM não estão integrados a seus ambientes de programação [5]. Atualmente, existem algumas exceções a esta afirmação como por exemplo o Midway [2] e o Munin [4]. Porém, estes fornecem ferramentas ou extensões de linguagem para melhorar o desempenho do sistema, mas não primitivas de programação que facilitem o desenvolvimento de aplicações distribuídas.

Várias linguagens para ambientes distribuídos, consideradas como sistemas DSM por alguns autores [13], também oferecem o compartilhamento de dados. Todavia, na maioria dos casos, este é acompanhado de restrições ou novos conceitos. Por exemplo, a linguagem Orca [1], que adota o modelo de objeto de dados, proporciona acesso transparente aos dados compartilhados, contanto que eles estejam encapsulados em objetos. A Linda [1,13], por sua vez, introduz o conceito de unidades lógicas, as tuplas, mas que só podem ser referenciadas de forma associativa e nunca por endereçamento.

O sistema de programação e processamento CPAR foi originalmente definido para arquiteturas multiprocessadoras com memória compartilhada [11,12]. Permite que aplicações explorem o paralelismo em diferentes níveis e que unidades lógicas de processamento paralelo se comuniquem através de variáveis compartilhadas. Como um sistema DSM emula este tipo de variável, as rotinas da biblioteca CPAR poderiam ser adaptadas de forma a referenciá-lo, fazendo com que a linguagem CPAR se tornasse, deste modo, uma ferramenta para o desenvolvimento de aplicações distribuídas. Com base nesta idéia e no fato de que existem vários sistemas DSM já disponíveis, surgiu a proposta do **sistema de programação e processamento CPAR-DSM.** Além de oferecer o paradigma, sem restrições, de variáveis compartilhadas e primitivas que possibilitam expressar o paralelismo mais facilmente, o sistema **CPAR-DSM** é voltado para arquiteturas distribuídas, onde a localidade dos dados deve ser explorada, a troca de mensagens, minimizada e a carga de processamento, balanceada. Vale ressaltar que, como há várias implementações de DSMs para aglomerados de estações, economicamente mais acessíveis do que as máquinas multiprocessadores, o sistema CPAR-DSM é uma alternativa de baixo custo para a execução de aplicações paralelas.

2 Sistemas com Memória Compartilhada Distribuída (DSM)

Um sistema DSM oferece a abstração de um espaço de endereçamento lógico, a que todos os processadores de uma arquitetura distribuída têm acesso, apesar de, fisicamente, a memória ser local a cada um deles. Implementados tanto em *software* como em *hardware*, os sistemas DSM controlam a distribuição física dos dados, oferecendo ao programador acesso transparente à memória virtual compartilhada [3,8].

Para diminuir a troca de mensagens entre os processadores, os dados são agrupados em unidades básicas de transferência (páginas, segmentos, objetos). Estas, então, são duplicadas e armazenadas localmente, pois o acesso a um dado local é significativamente mais rápido do que se estivesse na memória de um processador remoto [10]. Entretanto, manter várias cópias de uma mesma unidade básica demanda um controle adicional por parte do sistema para que nenhum processador fique com a sua inconsistente. Em outras palavras, oferecer uma **consistência "perfeita"** (*strict consistency*) pode ser muito custoso e até mesmo inviável em termos de desempenho. Por este motivo, alguns sistemas DSM propõem relaxar o seu modelo de consistência de memória, garantindo a coerência da memória apenas em determinados pontos do programa [13].

O modelo de **consistência relaxada** (*release consistency*) permite a um processador adiar, até a próxima operação de sincronização ou liberação de acesso, a efetivação das atualizações que fez na memória compartilhada [4,13]. DSMs como Munin [4] e Quarks

[7] adotam este modelo. Existem vários outros modelos de consistência de memória, cujas descrições podem ser encontradas em [8,13].

A existência de várias cópias de um mesmo dado exige do sistema DSM um protocolo de coerência mais complexo para o controle das duplicações. Os mais utilizados são: o de **invalidação de escrita** (*write-invalidate*) e o de **atualização de escrita** (*write-update*). No primeiro, a atualização de um dado é enviada a todos os nós que possuem uma cópia do mesmo. No segundo, todas as cópias de um dados são invalidadas, exceto uma, antes que uma operação de escrita possa prosseguir [8].

3 Por Que Utilizar o Sistema CPAR em Ambientes Distribuídos

O sistema de programação e processamento CPAR-DSM foi definido com base no modelo de memória com **consistência relaxada**. Ele apresenta as seguintes vantagens quando comparado com as de um sistema DSM puro:

- *Facilidade de programação:* sistemas DSM, em geral, oferecem apenas algumas primitivas básicas. A linguagem CPAR, por outro lado, possui construções mais poderosas (semáforos, monitor, comandos paralelos) que facilitam a definição, estruturação e programação de aplicações distribuídas. Utilizar um sistema DSM, principalmente aquele com consistência relaxada de memória, demanda do programador alguns cuidados extras, como por exemplo, a distribuição das iterações de laços paralelos entre os processadores da arquitetura, além da chamada a primitivas de sincronização ou liberação de *locks* [10] do DSM para que a atualização das variáveis compartilhadas seja efetivada após a execução desses laços. No sistema CPAR-DSM, estas mesmas operações são automáticas: o comando referente a um laço paralelo (*forall*) já divide as iterações entre os vários nós processadores e imediatamente antes ou após a execução de um comando paralelo garante-se que as variáveis estarão consistentes.

- *Suporte de processamento voltado para arquiteturas distribuídas:* por ter ciência de que a memória compartilhada é lógica, o suporte de processamento CPAR-DSM define uma política, baseada na localidade das variáveis do programa do usuário, para selecionar qual processador irá executar uma determinada tarefa.

- *Construções para melhorar o desempenho de aplicações*: diretivas adicionadas à linguagem CPAR possibilitam explicitar como as variáveis compartilhadas de um programa devem ser agrupadas, para que o suporte de processamento explore, mais uma vez, a localidade dos dados e amenize o problema do falso compartilhamento [8].

- *Independência em relação ao sistema DSM:* as rotinas da biblioteca CPAR-DSM podem facilmente se adequar a qualquer pacote DSM que implemente o modelo de memória com consistência relaxada e que forneça um conjunto básico de primitivas e recursos.

4 O Sistema de Programação e Processamento CPAR-DSM

O Sistema CPAR-DSM é uma adaptação, para arquiteturas distribuídas, do sistema de programação e processamento CPAR [11]. Este foi definido para arquiteturas multiprocessadoras com memória compartilhada. É composto pela linguagem CPAR, o pré-compilador CCPAR e um suporte de processamento (biblioteca de rotinas).

4.1 A linguagem CPAR

Baseada no paradigma da memória compartilhada, a CPAR é uma extensão da linguagem C [12]. O programador pode expressar o paralelismo de sua aplicação nos seguintes níveis:

- *blocos paralelos*: no corpo principal do programa é possível definir blocos de código que irão executar concorrentemente, utilizando o *multitasking* que o sistema operacional oferece.

- *macrotarefas*: macrotarefa é a unidade lógica de paralelismo do sistema CPAR. Definida como uma subrotina, executa em paralelo com as demais.

- *microtarefas*: ao se ativar uma macrotarefa, é possível subdividi-la em *p* microtarefas. Cada uma delas será atribuída a um processador diferente. São responsáveis pela execução do paralelismo mais fino. Este pode ser de dois tipos: dados ou controle. O primeiro é o resultado da aplicação da mesma operação em múltiplos elementos de uma estrutura de dados, enquanto que o segundo é decorrente da execução simultânea de códigos distintos [6]. Na CPAR, os comandos *forall* e *parbegin* possibilitam expressar, respectivamente, cada um destes tipos de paralelismo.

As iterações consecutivas de um laço paralelo podem ser agrupadas em blocos. Através do comandó *forall*, a CPAR permite ao programador selecionar, dentre duas estratégias, aquela que melhor distribui estes blocos entre os processadores. São elas: o escalonamento estático por bloco (*chunk static scheduling*) e o escalonamento dinâmico por bloco (*chunk dynamic scheduling*). No primeiro, a atribuição das iterações aos processadores é pré-definida. Ela é feita de forma homogênea, ou seja, se uma macrotarefa for ativada com *p* microtarefas e possuir um *forall* com *N* iterações, cada uma das microtarefas executa $\lceil N/p \rceil$ iterações . No segundo, processadores ociosos, em tempo de execução, atribuem iterações a si próprios. Um fator de blocagem define quantas iterações cada processador irá processar a cada vez e um controle centralizado indica quais ainda não foram executadas [9]. O escalonamento dinâmico por bloco é interessante para aplicações executadas em arquiteturas distribuídas, cujos nós processadores não possuem a mesma capacidade computacional ou taxa de utilização.

Para a sincronização das macrotarefas, duas primitivas são oferecidas: a *wait_task (nome_macrotarefa)*, que espera pelo término da execução de uma macrotarefa específica e a *wait_all* ( ), que aguarda a finalização de todas.

A linguagem CPAR admite também uma hierarquia a nível de variáveis compartilhadas: as **globais** e as **locais** a uma macrotarefa. Precedidas pelo identificador *shared*, as primeiras devem ser declaradas na área de dados global do programa, enquanto que as segundas no escopo de uma macrotarefa.

O programa exemplo em CPAR da Figura 1 faz a multiplicação de duas matrizes globais, compartilhadas de dimensão 500X500. Uma macrotarefa inicia as matrizes e uma outra faz efetivamente a multiplicação. Ambas são executadas por 8 processadores (8 microtarefas). O comando *forall* com escalonamento estático por bloco foi o adotado.

As demais construções da CPAR como, semáforos, eventos e monitores não serão apresentadas neste artigo, mas podem ser encontradas em [11,12].

4.2 Compilador CCPAR

Para se gerar um código executável, o programa fonte em CPAR tem que ser submetido ao pré-compilador **CCPAR**, que produz como saída um outro fonte em C.

Este então é compilado e ligado com a biblioteca específica para cada solução CPAR (arquitetura multiprocessadora ou distribuída). Para o sistema CPAR-DSM, o CCPAR gera o programa fonte em C segundo o modelo de programação SPMD (*Single Program Multiple Data*): cada processador executa o mesmo programa, mas opera diferentemente dependendo da identificação do processador ou valores de variáveis [6].

```
task spec inic_mat ( );
task spec mult_mat ( ) ;

shared int a [500][500];
shared int b [500][500];
shared int c [500][500];

task body mult_mat ( ) {
  int i,j,k;
    forall i=0 to 499 {
      for j=0 to 499
        for k=0 to 499
          c[i][j]=c[i][j]+a[i][k]*b[k][j];
    }
}

task body inic_mat ( ) {
  int i,j;
  forall i=0 to 499 {
    for j=0 to 499 {
          a[i][j] = b[i][j] = 1;
          c[i][j] = 0;
        }
    }
}

main ( ){
  printf ("alocar 8 processadores \n");
  alloc_proc (8);

  printf ("iniciar matrizes \n");
  create 8, inic_mat ( );

  /*esperar finalização da macrotarefa*/
  wait_task (inic_mat);

  printf ("multiplicar matrizes \n");
  create 8, mult_mat ( );

  /*esperar finalização da macrotarefa*/
  wait_all ( );
  printf ("fim \n");
}
```

Figura 1: multiplicação de matrizes

4.3 Características do sistema CPAR-DSM

A proposta do Sistema CPAR-DSM não é apenas oferecer a facilidade de programação com o paradigma de variáveis compartilhadas para arquiteturas distribuídas, mas também amenizar alguns dos problemas intrínsecos aos sistemas distribuídos e DSMs, na tentativa de melhorar o desempenho das aplicações.

O comando *forall* com escalonamento dinâmico surgiu em decorrência da heterogeneidade computacional das arquiteturas multicomputadoras. Entretanto, o balanceamento do processamento não é o único ponto explorado pelo sistema CPAR-DSM. O princípio da localidade dos dados, tão importante nas arquiteturas distribuídas, também é fortemente enfatizado. Com este objetivo, uma política para atribuir microtarefas a processadores foi definida e diretivas adicionadas à linguagem CPAR.

O suporte de processamento (biblioteca de rotinas) do sistema CPAR-DSM difere significativamente do seu equivalente para arquiteturas multiprocessadoras. Não só devido às implementações acima citadas, mas também porque as rotinas passaram a alocar e ter acesso à memória compartilhada através de um sistema DSM com modelo de consistência relaxada de memória, onde a coerência do conteúdo das variáveis compartilhadas não é garantida em qualquer ponto do programa.

Com base na literatura sobre Memória Compartilhada Distribuída, pôde-se verificar que alguns DSMs ofereciam o mesmo conjunto básico de primitivas. Por esta razão, a biblioteca CPAR-DSM não chama diretamente as primitivas de um DSM. Estas, reunidas em um único módulo, são encapsuladas em funções, cujas interfaces independem do DSM. Criando mais esta camada, o suporte de processamento não fica específico para

nenhum DSM, mas sim facilmente adaptável a qualquer outro com modelo de memória com consistência relaxada e que forneça determinadas funcionalidades.

- Localidade das variáveis

Em decorrência da forma como operam, os sistemas DSM podem apresentar um fenômeno conhecido como **falso compartilhamento**. Ele ocorre quando, por exemplo, duas variáveis não relacionadas, referenciadas por processos diferentes, residem na mesma unidade de consistência (bloco, página, etc.) [3,8]. Se cada processador possuir uma cópia local da página, o desempenho do sistema será prejudicado, pois o protocolo de coerência precisará atualizar ambas.

Este problema poderia ser evitado se as variáveis a que cada processador faz acesso estivessem em páginas distintas. O sistema DSM ou o compilador não tem condições de estabelecer tal divisão, mas o programador sim. Partindo desta premissa, criou-se o conceito de **macrobloco**: através de diretivas de linguagem é permitido ao programador agrupar as variáveis de seu programa, isto é, estabelecer blocos de variáveis independentes. Desta forma, duas macrotarefas não relacionadas poderiam isolar as suas variáveis em macroblocos diferentes.

Uma outra função do macrobloco é permitir que a localidade dos dados seja explorada. Ao ativar uma macrotarefa, o programador pode informar quais macroblocos irá utilizar. Com a política de escalonamento, descrita a seguir, o sistema CPAR-DSM irá atribuir as microtarefas desta macrotarefa aos processadores que porventura possuam, nas suas memórias locais, cópias atualizadas desses macroblocos.

- Execução de macrotarefas

A solução CPAR-DSM adota o modelo de programação SPMD [6], ou seja, o mesmo programa é carregado em todos os nós, mas devido à lógica condicional gerada pelo pré-compilador CCPAR, um dos processos, o mestre, executa trechos de código diferentes dos demais, os escravos.

Quando da ativação do programa, o processo mestre é criado. Ele, então, gera um processo escravo para cada um dos demais processadores. Estes, por esta razão, são denominados de processadores escravos. O papel do mestre é basicamente ativar e sincronizar as macrotarefas, selecionando os processos escravos que irão efetivamente executar as microtarefas correspondentes. Ele segue, essencialmente, o definido no corpo principal do programa. Todo escravo, por sua vez, fica bloqueado, aguardando ser acordado pelo mestre para executar uma microtarefa. Depois, volta a "dormir".

No sistema CPAR-DSM, uma microtarefa não é atribuída aleatoriamente a um processo escravo. Para evitar que o sistema DSM faça acessos remotos, o processo mestre adota uma política que aproveita, sempre que possível, a cópia atualizada das variáveis compartilhadas que já se encontram na memória local de um processador escravo. O mestre mantém atualizadas as informações sobre a última microtarefa que cada processador escravo executou e os respectivos macroblocos referenciados. Com base nelas e seguindo uma prioridade de critérios, o mestre procura pelo processador escravo, que ele supõe ser o mais adequado para executar a microtarefa em questão. São os seguintes os referidos critérios:

- *processador escravo possui, na sua memória local, cópias atualizadas dos mesmos macroblocos que serão utilizados pela microtarefa a ser executada*: como mencionado anteriormente, ao ativar uma macrotarefa, é dada a opção ao programador de informar quais os macroblocos a que as microtarefas farão acesso.

- *processador escravo já executou a mesma microtarefa e ainda possui cópia atualizada das variáveis compartilhadas referenciadas*: é grande a probabilidade da nova execução da microtarefa vir a fazer acesso a essas mesmas variáveis compartilhadas.

- *processador escravo já executou outra microtarefa da mesma microtarefa e ainda possui cópia atualizada das variáveis locais compartilhadas referenciadas*: as variáveis locais compartilhadas são comuns às microtarefas de uma mesma macrotarefa.

- *processador escravo nunca executou uma microtarefa*: este critério visa preservar as informações já armazenadas na memória local dos demais processadores escravos. Uma terceira microtarefa poderá se beneficiar das regras anteriores.

- *qualquer processador disponível.*

Essa política é seguida somente se houver processadores livres e não existir outras microtarefas à espera para serem executadas. Caso contrário, a microtarefa será colocada no final de uma fila única de microtarefas "prontas para executar". Assim que chegar no início da fila e um dos processos escravos estiver disponível, ela será executada.

5 Desempenho

A implementação atual do Sistema CPAR-DSM utiliza o sistema DSM **Quarks** (versão 0.8 release 6), de domínio público, que adota o modelo de memória com consistência relaxada [7]. Os testes foram feitos em uma rede local de estações SPARC (Ethernet 10Mbit/s), utilizando o Quarks com protocolo de **invalidação de escrita**. Foram alocadas até 7 estações (Sun OS 4.1.3) com capacidades computacionais (processadores e memória) distintas, conforme a Tabela 1 abaixo. Por se tratar de estações multi-usuários, os testes foram aplicados muitas vezes, quando a taxa de utilização da rede era baixa.

Estação	Modelo/CPU	Memória
mestre	Sparc Station 1+, Sun 4/65	12 Mb
escravos 1-4	Sparc Station IPC, Sun 4/40	12 Mb
escravo 5	Sparc Station 1+, Sun 4/65	24 Mb
escravo 6	Axil-220	32 Mb

Tabela 1: configuração das estações da rede

Dois programas em CPAR foram desenvolvidos para os testes de desempenho: multiplicação de matrizes e *Successive Over-relaxation* (SOR). O primeiro é um exemplo de "paralelismo fácil" [1], pois as microtarefas pouco trocam mensagens. O segundo provoca um maior tráfego na rede, principalmente por que a cada iteração, os processadores precisam se sincronizar para a atualização dos dados.

Em ambos os casos, foram desenvolvidos programas seqüenciais em C para o cálculo do *speedup* (relação entre o tempo de execução do programa seqüencial e do programa CPAR executado por *p* CPUs). O *speedup* máximo se refere ao tempo de execução do programa seqüencial numa das CPU mais lenta (escravos 1-4) e o *speedup* mínimo, ao tempo na mais rápida (escravo 6).

5.1 Multiplicação de matrizes

Os resultados foram obtidos a partir da execução do programa exemplo da Figura 1, variando-se o número de processadores escravos. Foi medido o tempo médio total de execução do programa. Os tempos mínimo (*minbar*) e máximo (*maxbar*) de espera nas barreiras também foram monitorados. Eles indicam, respectivamente, quanto os processadores estavam sobrecarregados ou ociosos.

- Escalonamento estático de laços paralelos

O programa seqüencial foi executado na CPU mais rápida (escravo 6) num tempo médio de 383.88 segundos e numa mais lenta (escravo 1) em 626.90 segundos.

qtd. escravos	id_escravo	tempo	maxbar	minbar	speedup max.	speedup min.
1	6	484.20	0.58	0.58		0.79
2	5,6	489.77	191.79	0.74	1.28	0.78
4	1,2,5,6	282.30	96.74	2.65	2.22	1.36
6	1-6	208.05	61.19	4.79	3.01	1.85

Tabela 2: multiplicação de matrizes c/ escalonamento estático por bloco (seg.)

Apesar de terem sido realizados testes com diferentes configurações de estações, a Tabela 2 mostra somente aquelas que apresentaram os melhores resultados. Pode-se verificar que, com exceção de 1 CPU, o programa em CPAR sempre apresentou melhor desempenho se comparado com o seqüencial executando na CPU mais lenta. Com relação à CPU mais rápida, passou-se a ter ganhos somente a partir de 4 processadores.

- Escalonamento dinâmico de laços paralelos

Outra conclusão importante a que se chega a partir dos resultados acima é que alguns processadores ficaram estacionados na barreira muito mais tempo do que outros. Por exemplo, no caso de 2 processadores, o tempo de 191.79 seg. se justifica pois a capacidade computacional da estação 6 é consideravelmente maior do que a da estação 5. Assim, para aproveitar a ociosidade de estações como aquela, novos testes foram aplicados utilizando o escalonamento dinâmico por bloco.

qtd. escravos	id_escravo	bloco	tempo	maxbar	minbar	speedup max.	speedup min.
2	5,6	80	318.22	9.22	2.08	1.97	1.21
4	1,2,5,6	25	236.22	8.28	1.95	2.65	1.63
6	1-6	10	205.29	19.30	2.17	3.05	1.87

Tabela 3: multiplicação de matrizes com escalonamento dinâmico por bloco (seg.)

A Tabela 3 indica que, em arquiteturas cujas estações não possuem a mesma capacidade computacional ou taxa de utilização, a distribuição não homogênea do processamento de laços paralelos é uma estratégia a ser adotada. Comparando com os valores da Tabela 2, o ganho significativo foi sentido naquelas configurações, onde a diferença entre o tempo máximo e mínimo de espera na barreira pelos processadores era expressiva. Na Tabela 3, estes valores ficaram mais próximos e passou-se a ter ganho a partir de 2 CPUs (*speedup* sempre maior que 1). Vale observar também que quanto maior o número de processadores, menor deve ser o tamanho dos blocos de iterações, para assim diminuir a probabilidade de um processador executar um bloco muito grande,

ficando os demais à espera de sua finalização. O gráfico da figura 3 compara os *speedups* mínimo e máximo para os 2 tipos de escalonamento.

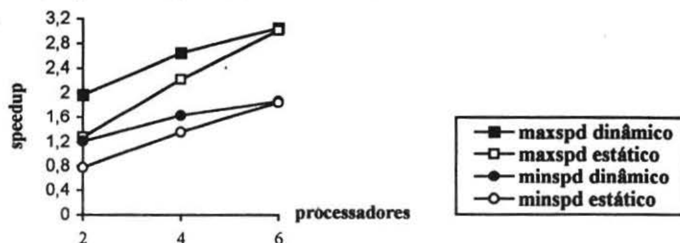

Figura 3: speedup multiplicação de matrizes

5.2 Successive Over-relaxation

O *Successive Over-relaxation* (SOR) é um método iterativo, que efetua sucessivas operações sobre os elementos de matrizes [1]. O algoritmo implementado foi o "red-black" SOR, que utiliza duas matrizes, cujas posições são iniciadas com 0 ou 1 (bordas da matrizes). A cada iteração, elas são atualizadas com a média dos elementos vizinhos [3]. O programa teste SOR em CPAR utilizou duas matrizes de 2400X1000, executando 30 iterações. A Tabela 4 contém, além dos tempos totais de execução do programa, aqueles referentes apenas à execução do algoritmo SOR, sem a iniciação das matrizes. O *speedup* foi calculado somente em relação à execução do programa seqüencial na estação mais rápida (escravo 6). Este apresentou um tempo médio de 769.61 segundos. O gráfico da figura 4 representa, em tempos absolutos, o ganho total do programa e do algoritmo SOR, quando se variou o número de CPUs.

qtd. escravos	escravos	tempo total	tempo SOR	speedup
1	6	842.90	690.95	0.91
2	5,6	919.70	772.49	0.84
4	1,2,5,6	658.77	409.74	1.17
6	1-6	583.07	342.81	1.32

Tabela 4: resultados execução do SOR (seg.)

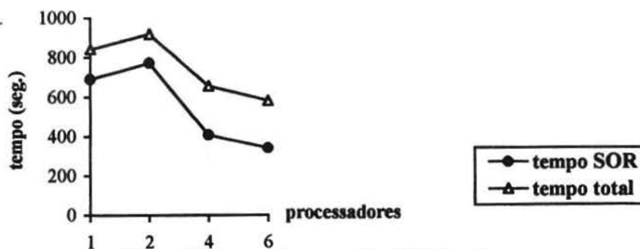

Figura 4: tempo de execução SOR (seg.)

O ganho de desempenho não foi tão significativo, devido principalmente à sobrecarga de comunicação decorrente da necessidade de sincronização entre as tarefas e atualização dos valores das matrizes a cada iteração.

Conclusão

A abstração da memória compartilhada por si só não simplifica o desenvolvimento de aplicações paralelas para sistemas distribuídos. É necessário um modelo de programação que permita ao programador expressar o paralelismo de sua aplicação mais facilmente e um suporte de processamento voltado para as característica das arquiteturas distribuídas. O sistema CPAR-DSM oferece este modelo através da linguagem CPAR e o seu suporte de processamento, apesar de utilizar um sistema DSM, foi projetado para explorar a localidade das variáveis e minimizar acessos remotos. Construções para explicitar o paralelismo em vários níveis, mecanismos para evitar o problema do falso compartilhamento, política de escalonamento de tarefas, balanceamento do processamento de laços paralelos, independência em relação ao DSM são algumas das vantagens em se utilizar o sistema CPAR-DSM. Os resultados apresentados na seção anterior comprovam a viabilidade da proposta do sistema CPAR-DSM, principalmente considerando que os testes foram realizados numa rede local de estações multi-usuários.

Referências

[1] BAL, H. E. **Programming distributed systems**. New York, Prentice Hall, 1990.

[2] BERSHAD, B. N. et al. The Midway distributed shared memory system. In: IEEE INTERNATIONAL COMPUTER CONFERENCE, 38°, San Francisco, 1993. **Proceedings**. Los Alamitos, IEEE Computer Society Press, 1993. p. 528-37.

[3] CARTER, J; BENNETT, J. K.; ZWAENEPOEL, W. Implementation and performance of Munin. **Operating Systems Review**, v. 28, n.3, p.165-82, 1991.

[4] CARTER, J. B. Design of the Munin shared memory system. **Journal of Parallel and Distributed Computing**, v.29, n.2, p. 219-27, Sept. 1995.

[5] CARTER, J. B; KHANDEKAR, D.; KAMB,L. Distributed shared memory: where we are and where should be headed?. In: WORKSHOP ON HOT TOPICS IN OPERATING SYSTEMS, 5°, Orcas Island, 1995. **Proceedings.** Los Alamitos, IEEE Computer Society Press, 1995. p.119-22.

[6] HATCHER, P. J.; QUINN, M. J. **Data-Parallel programming on MIMD computers**. Cambridge, The MIT Press, 1991.

[7] KHANDEKAR, D. **Quarks: portable DSM on Unix**. Salt Lake City, Department of Computer Science, University of Utah, 1995.(Technical Report).

[8] NITZBERG, B.; LO, V. Distributed shared memory: a survey of issues and algorithms. **IEEE Computer**, v. 24, p. 52-60, Aug., 1991.

[9] POLYCHRONOPOULOS, C. D. **Parallel programming and compilers**. Boston, Kluwer Academic Publishers, 1988.

[10] RAMACHANDRAN, M.; SINGHAL, M. **On the synchronization mechanisms in distributed shared memory systems**. Columbus, Department of Computer and Information Science, Ohio University, Oct. 1994. (OSU-CISRC-10/94-TR54).

[11] SATO, L. M. Um Sistema de Programação Paralela para Sistemas Multiprocessadores. In: SIMPÓSIO BRASILEIRO DE ARQUITETURAS DE COMPUTADORES - Processamento de Alto Desempenho, 4°, São Paulo, 1992. **Anais**. São Paulo, ed. por Volnys B. Bernal et al., 1992. p. 95-107.

[12] SATO, L. M. **Ambientes de programação para sistemas paralelos e distribuídos**. São Paulo, 1995. 115p. Tese (Livre Docência)-Escola Politécnica, Universidade de São Paulo.

[13] TANENBAUM, A. S. **Distributed Operating Systems**, New Jersey, Prentice Hall, 1995.