Implementação e Avaliação Preliminar de um novo Sistema Software DSM para Cluster de Computadores*

Rafael Mendes
COPPE-UFRJ
rmendes@cos.ufrj.br

Lauro Whately
COPPE-UFRJ
whately@cos.ufrj.br

Cristiana Bentes
DESC-UERJ
cris@eng.uerj.br

Claudio Luis Amorim
COPPE-UFRJ
amorim@cos.ufrj.br

Resumo

Neste trabalho apresentamos resultados preliminares da implementação e de desempenho de Clik, um novo sistema software DSM (SDSM) multithread para clusters de computadores. Em contraste com sistemas SDSM tradicionais que suportam a programação SPMD, Clik implementa o modelo de programação multithread proposto pela linguagem Cilk, o qual permite explorar paralelismo assíncrono e dinâmico freqüentemente encontrado em aplicações paralelas, mas que o modelo SPMD não consegue expressar de maneira eficiente. Como resultado, além de oferecer um sistema SDSM multithread, Clik também introduz um mecanismo eficiente e transparente de distribuição de tarefas entre os nós de um cluster, facilitando o balanceamento de carga. Nossos resultados preliminares de desempenho de Clik para três aplicações paralelas executadas em um cluster de 8 processadores mostraram que Clik é potencialmente capaz de superar o desempenho de sistemas SDSM estado-da-arte para essa classe importante de aplicações.

1. Introdução

Sistemas software DSM (SDSM) permitem combinar a simplicidade do modelo de programação de memória compartilhada com o baixo custo dos *clusters* de computadores. Na última década, vários projetos SDSM [6, 8, 14, 15, 17, 10] propuseram plataformas de programação de memória compartilhada competitivas com a de passagem de mensagens. O foco destes avanços, entretanto, tem sido a melhora do desempenho, seja com o protocolo de coerência empregado [6, 17], modelos mais relaxados de consistência [8, 14, 15], técnicas de atualização antecipada [5] ou sistemas adaptativos [10, 16]. Muito pouca atenção tem sido dada ao modelo de programação implementado. A grande maioria dos sistemas SDSM suporta apenas o modelo SPMD (*Single Programa Multiple Data*) de programação, onde as tarefas a serem executadas em paralelo são criadas estaticamente, no início da computação. Cada processador, então, executa o mesmo código, mas sobre porções distintas da massa de dados. O modelo SPMD implementa paralelismo de dados e é bastante apropriado para aplicações numéricas e científicas bem estruturadas. Aplicações dinâmicas, em que a execução é altamente dependente dos dados de entrada e dos dados computados durante o processamento, por sua vez, não podem ser implementadas eficientemente no modelo SPMD. Compiladores, simuladores, pacotes gráficos ou de otimização, ou mesmo aplicações web de alto desempenho se incluem nessa classe de aplicações e têm uma grande potencial de uso em sistemas paralelos como *clusters* de computadores.

O objetivo desse trabalho é prover um ambiente SDSM eficiente que permita explorar paralelismo assíncrono e dinâmico que normalmente não pode ser corretamente expressado pelo modelo SPMD. Para atingir esse objetivo, desenvolvemos um novo sistema SDSM *multithread* para *cluster* de computadores, chamado Clik, que implementa o modelo *multithread* proposto pela linguagem Cilk [4]. A linguagem paralela Cilk foi desenvolvida em conjunto com o sistema de mesmo nome pelos pesquisadores do MIT. Ela permite que a criação de *threads* pela aplicação seja totalmente independente do número de processadores do sistema. A idéia é que o sistema de tempo de execução seja o responsável pelo escalonamento das *threads* nos processadores e com isso forneça alocação dinâmica de tarefas e balanceamento de carga.

Nossa proposta é não só oferecer um sistema SDSM *multithread* que suporte aplicações dinâmicas e assíncronas, mas, também, introduzir um mecanismo distribuído, eficiente e transparente de escalonamento de tarefas entre os nós do *cluster*, facilitando o balanceamento de carga. Nossos resultados preliminares de desempenho de Clik para três aplicações paralelas executadas em um *cluster* de 8 processadores mostraram que Clik é potencialmente capaz de superar o desempenho de sistemas SDSM estado-da-arte para essa classe importante de aplicações, além de prover um

* Este trabalho foi apoiado parcialmente pelo MCT/FINEP, HP Brasil R&D e CAPES.

bom balanceamento de carga.

O restante do trabalho está organizado da seguinte forma. Na Seção 2 descrevemos brevemente a linguagem Cilk. Na Seção 3 apresentamos o sistema SDSM *multithread* Clik. Na Seção 4 mostramos alguns resultados preliminares da execução de Clik em um *cluster* de 8 computadores. Na Seção 5 apresentamos os trabalhos relacionados. e na Seção 6 concluímos nosso trabalho, apresentando propostas de trabalhos futuros.

2. A Linguagem Cilk

A linguagem Cilk foi proposta em [1] como uma extensão *multithread* da linguagem C em termos semânticos e de desempenho. Um pré-processador Cilk converte um programa em Cilk para um programa em C. Esse deve ser compilado e ligado ao sistema de tempo de execução de Cilk. A seguir apresentamos uma breve descrição da linguagem Cilk e suas primitivas. As tarefas em Cilk são criadas dinamicamente de acordo com o processamento e os dados de entrada. Na verdade, a computação em Cilk pode ser considerada como uma árvore de espalhamento que cresce de acordo com a execução da aplicação. Uma aplicação Cilk é representada por um grafo acíclico direcionado em que cada *thread* é um nó do grafo. Quando uma *thread* é executada, ela pode criar novas *threads* e assim fazer o grafo se expandir. As arestas do grafo denotam a dependência entre duas *threads*. Isto é, se há uma aresta ligando T_1 a T_2, então T_1 depende do resultado de T_2 para executar. *Threads* independentes podem executar em paralelo.

As primitivas de Cilk descrevem a especificação do paralelismo e a sincronização. São elas: `spawn` - criação dinâmica de *threads*; `lock/unlock` - exclusão mútua de seções críticas; e `sync` - sincronização das *threads*. As primitivas `spawn` e `sync` têm funcionalidade semelhante as primitivas `fork` e `join` de Unix. A primitiva `spawn` cria dinamicamente uma nova *thread*, enquanto a primitiva `sync` funciona como uma barreira local. Na barreira uma *thread* espera que todas as suas *threads* filhas tenham terminado para continuar a computação. As primitivas `lock/unlock` são utilizadas da maneira tradicional para proteger o acesso a seções críticas.

A Figura 1 mostra um exemplo de um algoritmo de ordenação MergeSort, escrito na linguagem Cilk. Como podemos observar, Cilk não emprega o modelo SPMD e permite expressar de forma natural o paralelismo de funções recursivas. Este algoritmo é baseado no paradigma de divisão e conquista. O vetor de n posições é dividido em dois vetores de $n/2$ posições. Os dois vetores são ordenados recursivamente utilizando MergeSort até que atinjam a um tamanho trivial. A partir desse ponto são ordenados pela função tt sort, utilizando o algoritmo *bubble sort* seqüencial. A combinação é realizada pela função `merge`

```
cilk void mergesort(int* main_v,int* vector,
                    int size)
{
  int size1, size2, *v1, *v2;
  if (size > SORTSIZE){
    size1 = size / 2;
    size2 = size - size1;
    v1 = vector;
    v2 = vector + size1;
    spawn mergesort(main_v,v1,size1);
    spawn mergesort(main_v,v2,size2);
    sync;
    spawn merge(v1,v2,size1,size2,
              (v1-main_v),(v2-main_v));
  }
  else
    spawn sort(vector, size);
  sync;
  return;
}
```

Figura 1. MergeSort na linguagem Cilk

que intercala os dois vetores ordenados para obter um vetor também ordenado. Note que cada função ou chamada recursiva da aplicação é executada por uma *thread*. Cada vez que o vetor é dividido ao meio, novas *threads* são criadas, gerando uma árvore binária que cresce durante a computação.

3. O Sistema Clik

O sistema Clik permite a execução de aplicações *multithread* em um *cluster* provendo: um sistema de memória compartilhada distribuída entre os nós do *cluster* e um mecanismo eficiente de distribuição de tarefas nestes nós. A seguir descrevemos a estrutura do sistema de tempo de execução do Clik, assim como as técnicas implementadas para possibilitar a execução eficiente de aplicações *multithread* no *cluster*.

3.1. Criação de Tarefas

Em cada nó do *cluster* são criados: uma *thread* servidora e um conjunto de processos trabalhadores[1]. A *thread* servidora é responsável por atender requisições remotas. Os trabalhadores, por sua vez, são responsáveis pela execução das *threads* da aplicação. Cada trabalhador tem uma fila de tarefas própria. A primeira tarefa criada pela aplicação corresponde à função `main`. Ela é inserida na fila do trabalhador *TR0* quando o sistema é inicializado. Em seguida, cada vez que uma operação `spawn` é executada uma nova tarefa

1 O número de trabalhadores é especificado pelo usuário.

é criada e inserida na fila do trabalhador que a criou. Enquanto a fila de tarefas de um trabalhador não está vazia, ele computa suas tarefas. Quando a fila torna-se vazia, há a distribuição de tarefas através do algoritmo de roubo de trabalho (*work-stealing*), explicado em mais detalhes em 3.3.

3.2. Memória Compartilhada Distribuída

O sistema de memória compartilhada distribuída implementado por Clik utiliza um protocolo de coerência baseado na residência de páginas e no modelo relaxado de consistência LRC (*Lazy Release Consistency*) [7]. Para cada página da aplicação é associado um processador do*cluster* que será o *home* da página. O processador *home* de uma página deve receber todas as modificações realizadas na página. Como resultado, as páginas estarão sempre atualizadas em seus *homes*. A proposta Clik é semelhante a do sistema HLRC [17]. Porém, a distribuição de tarefas em Clik requer a implementação de novos mecanismos para manter a consistência dos dados compartilhados. Uma tarefa pode estar com os dados válidos no nó k, mas ao migrar para o nó j, novas operações de coerência são necessárias. A seguir apresentamos o protocolo de coerência utilizado em Clik.

3.2.1. Protocolo de Invalidação O modelo de consistência relaxado LRC define uma ordem parcial dos acessos aos dados compartilhados. Ela é baseada na ordem seqüencial dos acessos locais e do encadeamento de operações de sincronização do tipo *acquire* e *release* realizadas em processadores diferentes. A execução de uma aplicação é dividida em intervalos. Um novo intervalo é iniciado cada vez que uma operação de sincronização é executada. As modificações realizadas nos dados compartilhados são associadas aos intervalos em que elas ocorreram. Assim, numa operação de *acquire* o processador deve ser notificado sobre todas as modificações associadas a intervalos anteriores (segundo a ordem parcial estabelecida por LRC). A notificação das modificações é realizada através de avisos de escrita (*write-notices*). O recebimento de um *write-notice* para uma página p causa a invalidação de p localmente.

3.2.2. Envio das Escritas Clik provê suporte para múltiplos escritores em uma página, permitindo que processadores escrevam concorrentemente em dados diferentes pertencentes à mesma página. As escritas nos dados compartilhados são propagadas para os processadores *home* de cada página na forma de *diffs*, que descrevem as diferenças da página escrita para a página original. Os *diffs* de uma página são enviados para o *home* quando ocorre: i) a criação de uma tarefa - `spawn`; ii) uma operação de sincronização - `sync`; e iii) o retorno de uma tarefa.

As atualizações devem ser enviadas ao *home* numa operação `spawn` porque a tarefa criada pode migrar para outro processador. O retorno de uma tarefa, também, envolve a atualização do *home* porque quando uma tarefa T_1 termina, pode ocorrer a liberação de uma tarefa T_2. Isso significa que T_2 precisa dos dados gerados por T_1 para continuar executando. Como T_2 pode ser migrada para outro nó pelo algoritmo de distribuição de tarefas, as escritas realizadas por T_1 na memória compartilhada precisam ser propagadas.

3.2.3. Falhas de Acesso No protocolo baseado em *homes* o tratamento de uma falha de acesso é bem simples. Uma cópia atualizada da página deve ser requisitada ao processador *home* da página. Como os trabalhadores são implementados como processos distintos, eles possuem estruturas relacionadas a proteção de memória distintas. Quando a proteção de uma página é alterada para permitir o acesso, esta alteração é válida apenas para aquele trabalhador. Portanto, um trabalhador pode ter acesso a uma determinada página de memória e outro no mesmo nó não possuir esta permissão de acesso.

É possível que um trabalhador receba uma falha de página, mas não necessite buscar a página no *home*. Isso ocorre quando um outro trabalhador, do mesmo nó, já requisitou a página anteriormente, e ela não foi invalidada. Neste caso, o único *overhead* gerado pela falha de página é o da troca das permissões de acesso.

Por fim, ao receber um pedido de página, é possível que o *home* não possua a página na versão solicitada, porque os *diffs* para esta página ainda estão sendo transmitidos. Neste caso, o *home* armazena o pedido de página em uma fila para que possa ser atendido quando a versão de página solicitada estiver disponível.

3.3. Distribuição de Tarefas

O algoritmo de *work-stealing* distribuído implementado em Clik é utilizado para a distribuição inicial das tarefas nos nós do *cluster* e principalmente para o balanceamento da carga. A idéia é que um trabalhador ocioso roube tarefas de um trabalhador ocupado. Primeiramente, o trabalhador tenta obter uma tarefa de um outro trabalhador localizado no mesmo nó. Evitando trocas de mensagens e operações de consistência de dados. Caso não haja tarefa disponível no nó, o trabalhador faz requisições remotas. A implementação de *work-stealing* em Clik se baseia no algoritmo descrito em [4], que obteve com essa estratégia de balanceamento excelentes resultados.

Na requisição remota, o trabalhador **ladrão**, escolhe aleatoriamente um nó[2], chamado de **vítima**, e envia uma men-

2 A escolha aleatória da vítima se baseia nos resultados de [4], pretendemos futuramente utilizar outros algoritmos de escolha.

sagem de *work-steal*. O nó vítima ao receber esta mensagem, envia uma reposta. Se na resposta enviada pelo nó vítima, não houver nenhuma tarefa, o algoritmo de *work-stealing* é reiniciado. Caso contrário, a tarefa é inserida na fila de tarefas do trabalhador ladrão.

Na criação de uma tarefa, Clik aloca um *frame* para a tarefa. O *frame* contém várias estruturas de dados relacionadas à tarefa, como por exemplo, variáveis de escopo local, parâmetros e outros. Para que a tarefa possa migrar e prosseguir a execução em outro nó, sem perdas de dados ou necessidade de *checkpoints*, Clik aloca as áreas de *frames* na memória compartilhada distribuída. Dessa forma, o sistema SDSM garante que os dados do *frame* de uma tarefa cuja execução estava no nó k sejam vistos também pelo nó j para onde a tarefa migrou.

Quando o nó vítima possui uma tarefa para enviar ao nó ladrão, ele envia, também, os *write-notices* relativos às alterações realizadas nos dados compartilhados ainda não vistas pelo nó ladrão. Na verdade, o envio de uma tarefa para um outro nó corresponde a uma operação *release* na vítima e a uma operação *acquire* no ladrão. As operações de consistência são, então, executadas de modo que a mensagem de *work-steal reply* só chegue no ladrão, depois que todas as atualizações realizadas pela vítima tenham chegado aos seus respectivos *homes*.

3.4. Operações de Sincronização

Em sua primeira versão, Clik implementa somente a primitiva `sync` para sincronização das tarefas. Em Clik, a operação `sync` executa primeiramente uma operação *release*. Após este procedimento, existem três casos que devem ser identificados. No primeiro caso, se a tarefa que chamou o `sync` possui filhos, ela é suspensa. No segundo caso, se a tarefa foi obtida de outro nó através de um *work-stealing*, ela é retornada para o nó de origem. No terceiro caso, se a tarefa está no nó onde foi criada originalmente e todos os seus filhos já terminaram, não há nenhum impedimento que ela prossiga a sua execução.

Como todas as atualizações feitas por tarefas criadas dentro do escopo do `sync` devem ser conhecidas pelo trabalhador que executa o `sync`, este deve enviar uma mensagem para cada nó do *cluster*. Esta mensagem recolhe os *write-notices* relativos a todas as modificações realizadas nos dados compartilhados antes do `sync`, e que podem ter sido migradas para outros nós do *cluster*.

4. Resultados Experimentais

Nesta seção apresentamos alguns resultados preliminares da avaliação do desempenho de Clik em um *cluster* de computadores. Primeiramente, comparamos Clik com o sistema HLRC [17] que é o estado-da-arte em termos de sistemas SDSM baseado em residência (*home-based*). HLRC, contudo, utiliza modelo de programação distinto para suas aplicações. Portanto, esta comparação visa apenas mostrar o potencial de desempenho Clik. A seguir, apresentamos uma breve descrição de HLRC e das aplicações utilizadas em nossos experimentos, mostrando as diferentes implementações no modelo da linguagem Cilk e no modelo SPMD. Por fim, apresentamos a avaliação de Clik.

4.1. HLRC

HLRC é um SDSM que utiliza a página como unidade de coerência e implementa protocolo de múltiplos escritores baseado em residência. Tal qual em Clik, HLRC possui para cada página um processador *home* que mantém a versão mais atualizada da mesma. Se um processador acessa a página p que não está válida localmente, ele apenas requisita a cópia mais atualizada de p ao seu *home*.

O processador *home* de uma página p recebe informações sobre as modificações realizadas em p nos pontos de sincronização da aplicação, segundo o modelo de consistência LRC. No início de um intervalo, o processador recebe as invalidações relativas às modificações realizadas em intervalos anteriores (segundo a ordem parcial de LRC). No final de um intervalo, um processador gera os *diffs* relativos às modificações realizadas em páginas onde ele não é *home* e envia esses *diffs* aos processadores *home* das páginas. Os *diffs* têm curta duração em HLRC, o processador que os gerou, os envia e em seguida descarta-os. O processador *home* recebe os *diffs* e os aplica na página, descartando-os em seguida também.

4.2. Aplicações

Utilizamos três aplicações com características diferentes de computação e E/S para a avaliação de desempenho preliminar de Clik: MergeSort, LU e PZSweep. As aplicações **MergeSort** e **LU** são núcleos conhecidos de sistemas de memória compartilhada. Elas foram implementadas no modelo de programação da linguagem Cilk e no modelo de programação SPMD (para a execução em HLRC). Estas aplicações ressaltam a diferença de implementação dos modelos Cilk e SPMD e permitem comparar o desempenho de Clik com HLRC. **PZSweep** é uma aplicação de renderização volumétrica e é avaliada somente no âmbito do sistema Clik. A avaliação desta aplicação mostra o desempenho de Clik para uma aplicação real, cujo desempenho é altamente dependente do balanceamento de carga empregado. A seguir, apresentamos uma breve descrição das aplicações e de suas implementações.

MergeSort - algoritmo de ordenação de um vetor de N chaves utilizando paradigma de divisão e conquista. A versão Cilk de MergeSort é recursiva e idêntica à apresen-

tada na Figura 1. Na versão SPMD, MergeSort divide o vetor a ser ordenado em n partes, onde n é o número de processadores do sistema, e cada processador ordena localmente a sua parte do vetor. Estando os vetores locais ordenados, $n/2$ processadores fazem a combinação dos vetores ordenados para cada dupla de processadores. Na fase seguinte, o mesmo processamento é realizado com $n/4$ processadores combinando o resultado de cada dupla dos $n/2$ processadores da fase anterior. O procedimento se repete até que um único processador realiza a combinação de dois vetores, obtendo o resultado final. Cada uma das fases de combinação é intercalada com barreiras. A entrada utilizada foi de 10M chaves.

LU - método para a solução de um sistema linear que realiza a decomposição de uma matriz densa A em matrizes triangulares. Na versão Cilk, a matriz é decomposta em 4 blocos, B(0,0), B(0,1), B(1,0) e B(1,1). Inicialmente calcula-se LU recursivamente para o bloco B(0,0). Em seguida, há uma sincronização e, com os valores calculados para B(0,0), calcula-se recursivamente a decomposição parcial para os blocos B(1,0) e B(0,1) em paralelo. Com o resultado obtido calcula-se novamente LU recursivamente, mas para o bloco B(1,1). Na versão SPMD, a matriz é decomposta em blocos não-contíguos e distribuída para os processadores. A cada iteração o pivot é computado na primeira fase e será utilizado nas duas fases seguintes para atualização dos blocos. As barreiras asseguram a sincronização a geração do pivot e a utilização do mesmo na fase seguinte. A entrada utilizada foi de 4096 $\times$ 4096.

PZSweep - sistema paralelo de renderização volumétrica *out-of-core*. Esta aplicação possui apenas implementação para o modelo da linguagem Cilk. Nesta implementação, o espaço da imagem é dividido em *tiles* e a computação de cada *tile* é realizada através da varredura dos dados (células tetraedrais) por um plano na direção da coordenada z. A computação de cada *tile* da imagem é feita por uma *thread* e a sincronização ocorre apenas no final da computação de todos os *tiles*. A versão *out-of-core* de PZSweep permite a renderização de grandes massas de dados, porém realiza E/S frequentes para trazer os dados volumétricos para a memória. A massa de dados utilizada representa parte de um reator nuclear da NASA, chamada SPX1 com 103K células. A imagem gerada tem precisão de 512 $\times$ 512 pixels e é dividida em 16 $\times$ 16 *tiles*. Mais detalhes podem ser encontrados em [3].

4.3. Avaliação de Desempenho

Nossos experimentos foram conduzidos em um *cluster* com 8 processadores, onde cada processador é um Pentium 4, 2.80GHz, com 1 GB de memória principal. Os processadores estão interconectados por uma rede Ethernet de 1 Gbps e executam o kernel 2.6.7 de Linux.

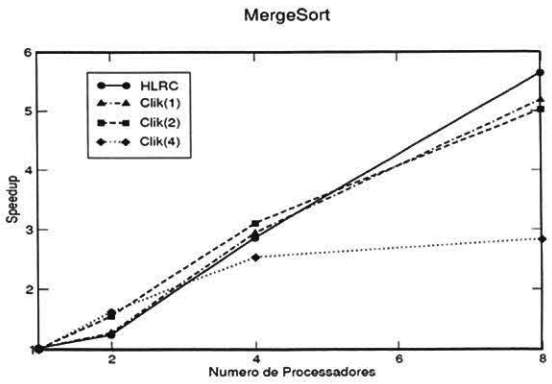

Figura 2. Speedups para MergeSort.

4.3.1. MergeSort A Figura 2 mostra os *speedups* obtidos pela aplicação MergeSort em Clik e em HLRC para 2, 4 e 8 nós. Sendo que as curvas de Clik(1), Clik(2) e Clik(4) mostram a execução de Clik para 1, 2 e 4 trabalhadores em cada nó, respectivamente. Comparando primeiramente as execuções de Clik com 1, 2 e 4 trabalhadores em cada nó, podemos observar que todas as execuções apresentam resultados semelhantes, exceto para o caso de 4 trabalhadores, devido ao excesso de competição em 1 CPU.

Comparando a execução de Clik e HLRC, podemos dizer que Clik é competitivo com HLRC. Observamos que o o desempenho de HLRC é apenas ligeiramente superior ao desempenho de Clik para 8 nós. Este resultado se deve a dois fatores diferentes: (i) à maior quantidade de mensagens transmitidas por Clik e (ii) à maior localidade dos dados obtida pela divisão estática de tarefas da versão MergeSort de HLRC. A maior quantidade de mensagens enviadas por Clik é explicada pela maior quantidade de pontos de sincronização na aplicação, visto que a granulosidade da tarefa no modelo Cilk é bem menor que a do modelo SPMD. Além disso, para prover balanceamento de carga dinâmico e transparente, Clik necessita trocar mensagens para garantir o mecanismo de *work-stealing*.

Com relação à boa localidade dos dados de MergeSort em HLRC, ela foi obtida através de um esforço adicional de programação. O programa foi ajustado para que os nós que fazem a combinação (*merge*) sejam exatamente aqueles que têm coerentes localmente a metade dos dados. Em Clik, a distribuição de tarefas é realizada em tempo de execução, de forma transparente ao programador. Mesmo não tendo uma distribuição de tarefas ajustada pelo programador, Clik obtém um bom balanceamento da carga do sistema, conforme podemos observar na Figura 3. Esta figura mostra o tempo de execução de MergeSort em Clik para cada um dos 8 nós do *cluster*, executando apenas 1 trabalhador em cada nó. O tempo de execução apresentado está dividido em: tempo de computação (gasto com a computação da aplicação propriamente dita); tempo de roubo de traba-

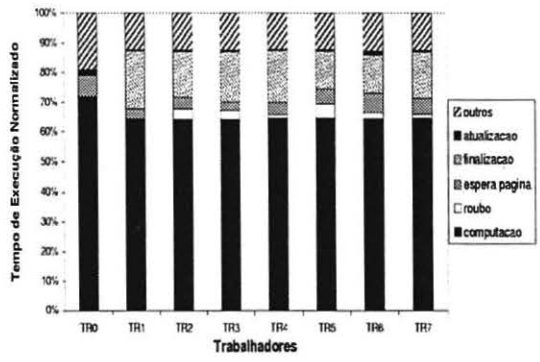

Figura 3. Tempos de execução de MergeSort.

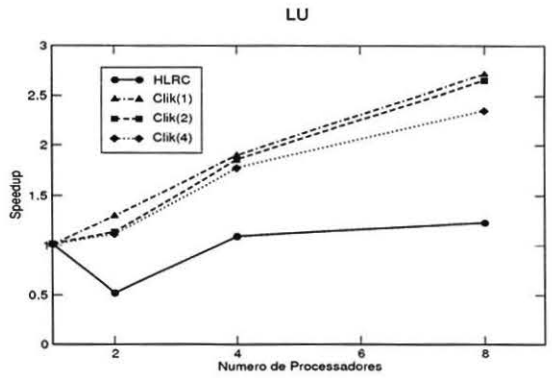

Figura 4. Speedups para LU.

lho (gasto com o algoritmo de *work-stealing* distribuído de Cilk); tempo de espera por páginas; tempo de finalização (quando o processador terminou e não há mais tarefas para roubar); tempo de atualização de páginas no *home*; e outros (basicamente tempos de troca de permissão de acesso de páginas e chaveamento de contexto para o tratador de sinal). Como podemos observar nesta figura, Clik consegue um bom balanceamento de carga em MergeSort. Isso porque o tempo de computação dos 8 nós é bastante próximo (exceto para o nó 0 que é o responsável pela inicialização e finalização da aplicação). Além disso, esta figura mostra que a escolha aleatória do nó vítima tem bom desempenho e o algoritmo de roubo de trabalho tem influência muito pequena no tempo total de execução da aplicação.

4.3.2. LU A Figura 4 mostra os *speedups* obtidos pela execução de LU em Clik e HLRC para 2, 4 e 8 nós, com 1, 2 e 4 trabalhadores em cada nó. Comparando-se as três execuções de Clik, esta aplicação mostra uma ligeira desvantagem na execução com 4 trabalhadores. A explicação para esse resultado, contudo, é diferente da fornecida para MergeSort. O limite para a divisão recursiva em blocos utilizado em nossos experimentos é de até 16 blocos. Dessa forma, com 4 trabalhadores para cada um dos 8 nós, não há tarefas suficientes para todos os trabalhadores. A pequena queda no desempenho se deve ao fato de que alguns trabalhadores ficam apenas tentando roubar trabalho sem realizar computação útil.

Na comparação de Clik com HLRC, observamos que Clik obteve desempenho superior. A diferença se deve ao modelo de programação empregado. Nesse caso, na versão SPMD de LU, nas fases do cálculo do pivot, os processadores que não são responsáveis pelo pivot, ficam ociosos esperando na barreira até que o pivot seja calculado. Este desbalanceamento não ocorre na versão Cilk. Nesta versão, um bloco é subdividido recursivamente em 4 partes até chegar a um tamanho considerado trivial para cálculo seqüencial. A divisão do bloco gera uma *quadtree*, que a partir da computação das folhas (caso trivial) permite que os resultados sejam combinados em paralelo. Além disso, o cálculo de LU parcial nos blocos B(1,0) e B(0,1) pode, também, ser feito em paralelo. Diminuindo bastante o desbalanceamento de carga gerado.

4.3.3. PZSweep A Figura 5 mostra os *speedups* obtidos pela execução de PZSweep em Clik para 2, 4 e 8 nós, com 1, 2 e 4 trabalhadores em cada nó. Podemos observar por essa figura que PZSweep apresenta bom desempenho em Clik, obtendo *speedup* de até 7.2 com 8 processadores para 2 trabalhadores por nó. Esta aplicação apresenta alta carga de E/S, portanto a execução de um número maior de trabalhadores por nó é justificável. Conforme podemos observar no gráfico, a execução com 4 trabalhadores por nó apresenta apenas ligeira queda de desempenho, o que mostra que a computação e a E/S estão sendo bem intercaladas.

O alto desempenho de PZSweep em Clik se deve ao bom balanceamento de carga alcançado. Algoritmos paralelos de renderização volumétrica sofrem com o desbalanceamento da carga, principalmente por causa da natureza irregular da massa de dados de entrada. Usualmente, o balanceamento da carga é realizado pelo próprio algoritmo paralelo. Nossos resultados mostram que é possível manter a carga balanceada de forma eficiente e totalmente transparente ao programador.

A porcentagem de desbalanceamento de carga gerada para a execução de PZSweep em Clik com 2 trabalhadores, é de: 6% para 2 nós, 15% em 4 nós e 12% em 8 nós. O desbalanceamento gerado é baixo (no máximo 15%), porque PZSweep implementa renderização *out-of-core*. A leitura da massa de dados é feita por demanda, portanto, a execução *multithread* em cada nó permite melhor sobreposição de E/S com computação.

4.3.4. Discussão Nossos resultados preliminares de Clik são bastante promissores. Entretanto, detectamos alguns problemas de desempenho que podem ser solucionados. Em primeiro lugar, os baixos *speedups* encontrados, tanto para

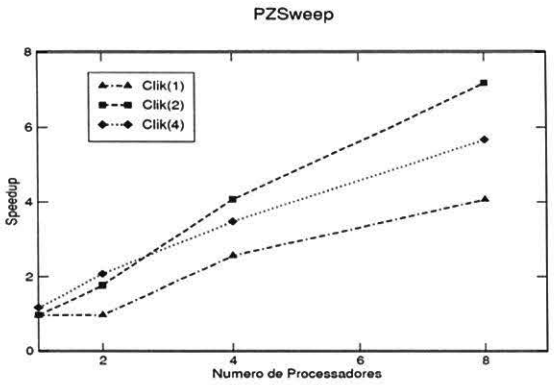

Figura 5. Speedups para PZSweep.

Clik como para HLRC, podem ser explicados pelo fato de que a primeira versão de Clik foi desenvolvida utilizando-se o protocolo TCP para troca de mensagens. Para manter uma comparação justa com HLRC, utilizamos HLRC com o mesmo protocolo TCP. Neste caso, mesmo abrindo-se apenas uma única conexão TCP por trabalhador, o peso das troca de mensagens é bem mais alto do que se estivéssemos utilizando um protocolo UDP [9]. Algumas outras aplicações que executamos com os dois sistemas, não apresentaram *speedup* significativos. Deixamos, portanto, a análise dos seus resultados para a outra versão de Clik, baseada em UDP. Esperamos que a implementação de Clik com UDP forneça *speedups* superiores.

Uma outra questão importante que deve ser mencionada está relacionada ao modelo de programação. Click mostrou-se bastante vantajoso para aplicações dinâmicas que utilizam paradigma divisão e conquista. O balanceamento de carga de Clik permite que o grande número de *threads* criadas mantenham os processadores ocupados de forma transparente ao programador. Entretanto, Clik insere mais pontos de sincronização e consistência de dados na aplicação. Isso ocorre por dois motivos: i) cada *thread* cria seus filhos e fica esperando em um `sync`; e ii) cada vez que uma *thread* migra para outro processador, é necessário realizar operações de consistência de dados, conforme apresentado na Seção 3. Portanto, o número de mensagens enviadas por Clik é superior ao número de mensagens enviadas por HLRC. Além disso, Clik cria mais *diffs* do que HLRC, porém *diffs* menores. A granulosidade das *threads* criadas em Clik pode ser alterada (aumentando-se o limite para execução seqüencial), de modo a diminuir a quantidade de mensagens trocadas e *diffs* criados.

Um fator que favorece Clik com relação à sincronização é a implementação da primitiva `sync`. Ao contrário da implementação de barreiras do modelo SPMD, um `sync` não requer que a próxima etapa do algoritmo se inicie após todos os *diffs* e *write-notices* terem sido enviados e processados por todos os nós. Em Clik, este bloqueio é estabelecido apenas no nó que está executando a primitiva `sync`.

5. Trabalhos Relacionados

Diversos sistemas SDSM foram desenvolvidos com o objetivo de prover modelo de programação de memória compartilhada em um *cluster* de computadores. Dentre esses sistemas podemos destacar TreadMarks [8], HLRC [17], ADSM [10] e AEC [14]. Nosso trabalho se distingue desses sistemas porque implementa um modelo de programação diferente. Enquanto, TreadMarks, HLRC, ADSM e AEC implementam apenas o modelo SPMD de programação, Clik permite a execução de aplicações dinâmicas e assíncronas suportadas pelo modelo da linguagem Cilk.

Clik não é a primeira implementação do modelo da linguagem Cilk em um *cluster* de computadores. Randall [13] propõe uma implementação distribuída de Cilk. Essa implementação se difere da nossa proposta porque se baseia em um modelo de consistência diferente, *dag-consistency* [2]. A consistência dos dados é mantida apenas na seqüência de dependências de *threads*. Embora tenham sido mostradas evidências teóricas do desempenho desse modelo, muito poucos resultados experimentais foram apresentados. Uma outra implementação da linguagem Cilk num sistema distribuído foi proposta em [11] e aprimorada em [12]. Nestes trabalhos, é apresentado o sistema SilkRoad que provê implementação tanto do modelo de consistência *dag-consistency* como do modelo LRC. Clik difere deste sistema em alguns aspectos de implementação importantes, dado que Clik se baseia nos conceitos de HLRC e SilkRoad em TreadMarks. Clik implementa protocolo de residência, que permite que *diffs* sejam imediatamente aplicados a uma página. Ao contrário de SilkRoad que se baseia puramente em protocolo de invalidação. Além disso, SilkRoad mantém a consistência apenas em operações do tipo `lock/unlock`. Enquanto que Clik mantém a consistência em operações `sync` e em migrações de tarefas. Vale ressaltar que ambas as implementações distribuídas de Cilk não estão disponíveis para serem comparadas com Clik[3].

6. Conclusões

Neste trabalho apresentamos o sistema Clik, um novo SDSM para *cluster* de computadores. Em contraste com sistemas SDSM tradicionais que suportam a programação SPMD, Clik implementa o modelo de programação *multithread* proposto pela linguagem Cilk, o qual permite ex-

3 Embora a implementação de Cilk para *cluster* tenha sido disponibilizada na rede, a versão apresentada não está funcionando corretamente.

plorar paralelismo assíncrono e dinâmico freqüentemente encontrado em aplicações paralelas, mas que o modelo SPMD não consegue expressar de maneira eficiente. Além da implementação de um novo modelo de programação, Clik fornece também um sistema de distribuição de tarefas e balanceamento de carga transparente para o usuário.

Resultados preliminares mostraram que Clik é competitivo em relação ao sistema HLRC, que é considerado o estado-da-arte em termos de SDSM. O modelo de programação utilizado em Clik permitiu expressar de forma mais simples o paralelismo de aplicações recursivas no paradigma de divisão e conquista. Além disso, sua proposta *multithread* fornece ambiente propício para uma boa distribuição de tarefas entre os processadores. Embora o balanceamento de carga transparente seja um grande desafio no projeto de sistemas distribuídos, o algoritmo de *work-stealing* mostrou que é capaz de realizar uma boa distribuição dinâmica de carga com um baixo *overhead*. Para a aplicação de renderização volumétrica, em particular, onde a quantidade de computação de cada *thread* é totalmente dependente dos dados de entrada, Clik apresentou um desbalanceamento de carga de no máximo 15%.

Ainda há uma série de melhorias e otimizações a serem implementadas em Clik. Pretendemos implementar Clik com protocolo UDP para diminuir o custo da troca de mensagens. Pretendemos também: avaliar Clik para uma classe maior de aplicações, com um número maior de processadores; reduzir a quantidade de mensagens de Clik, mesmo para *threads* com granulosidade pequena; e avaliar outras estratégias de balanceamento de carga.

Agradecimentos

Gostaríamos de agradecer ao Prof. Ricardo Farias da COPPE/UFRJ pelo código da aplicação PZSweep e pela sua implementação no modelo de programação de Clik. Gostaríamos também de agradecer à Profa. Maria Clícia Stelling de Castro da UERJ pelos comentários e sugestões.

Referências

[1] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, 1995.

[2] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. Dag-consistent distributed shared memory. In *Proc. of the 10th Int'l Parallel Processing Symp. (IPPS'96)*, pages 132–141, 1996.

[3] R. Farias, C. Bentes, A. Coelho, S. Guedes, and L. G. calves. Work distribution for parallel zsweep algorithm. In *XI Brazilian Symposium on Computer Graphics and Image Processing*, pages 107 – 114, October 2003.

[4] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Canada, June 1998.

[5] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving Release-Consistent Shared Virtual Memory Using Automatic Update. In *Proc. of the Symposium on High-Performance Computer Architecture*, pages 14–25, February 1996.

[6] P. Keleher. The Relative Importance of Concurrent Writers and Weak Consistency Models. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, May 1996.

[7] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[8] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, January 1994.

[9] R. Mendes, L. Whately, M. Lobosco, and C. L. Amorim. Memória compartilhada distribuída para redes udp/ip: Implementação e avaliação. In *Workshop em Sistemas Computacionais de Alto Desempenho*, October 2003.

[10] L. R. Monnerat and R. Bianchini. Efficiently Adapting to Sharing Patterns in Software DSMs. In *Proceedings of the 4th IEEE Symposium on High-Performance Computer Architecture*, pages 289–299, February 1998.

[11] L. Peng, W. F. Wong, M. D. Feng, and C. K. Yuen. SilkRoad: A multithreaded runtime system with software distributed shared memory for SMP clusters. In *IEEE International Conferrence on Cluster Computing*, pages 243–249, Nov. 2000.

[12] L. Peng, W.-F. Wong, and C.-K. Yuen. Silkroad ii: mixed paradigm cluster computing with rc-dag consistency. *Parallel Comput.*, 29(8):1091–1115, 2003.

[13] K. H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.

[14] C. B. Seidel, R. Bianchini, and C. L. Amorim. The Affinity Entry Consistency Protocol. In *Proceedings of the 1997 International Conference on Parallel Processing*, pages 65–78, August 1997.

[15] W. E. Speight and J. K. Bennett. Brazos: A Third Generation DSM System. In *Proceedings of the 1997 USENIX Windows/NT Workshop*, pages 95–106, August 1997.

[16] L. Whately, R. Pinto, M. Ragarjan, L. Iftode, R. Bianchini, and C. L. Amorim. Adaptative Techniques for Home-Based Software DSMs. In *Symposium on Computer Archtecture and High-Performance Computing*, pages 164–171, September 2001.

[17] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Memory Virtual Memory Systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 75–88, Oct. 1996.