Tradução e Avaliação de Aplicações OpenMP para a Linguagem Cilk em Software DSM*

Patrícia Sampaio
COPPE Sistems-UFRJ
psampaio@cos.ufrj.br

Maria Clicia Castro
IME-UERJ
clicia@ime.ufrj.br

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

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

Resumo

Este trabalho propõe a tradução e avaliação de aplicações OpenMP para um cluster de SMPs, utilizando sistema software DSM (SDMS). O SDSM empregado, Clik [9], é um sistema multithread que implementa a linguagem Cilk e provê um mecanismo eficiente e totalmente transparente de distribuição de tarefas nos nós de processamento. Nosso objetivo é de avaliar uma metodologia que permita a tradução eficiente de aplicações OpenMP para a linguagem Cilk, dado que a proposta de Cilk é bem diferente da proposta OpenMP. Em nossos resultados preliminares, executamos um conjunto de 3 aplicações em um cluster de SMPs e obtivemos bom desempenho para a tradução quando a aplicação OpenMP não possui grande quantidade de barreiras implícitas.

1. Introdução

A programação paralela é baseada em dois modelos de comunicação: troca de mensagens e memória compartilhada. No modelo de troca de mensagens a comunicação é realizada através do envio e recebimento explícito de mensagens entre os processos. No modelo de memória compartilhada a comunicação é implícita. Ela é realizada através de acessos a memória em dados compartilhados. Embora esse modelo de comunicação seja considerado mais simples de compreender e de programar, ele esteve durante algum tempo restrito a arquiteturas de memória centralizada. Atualmente, entretanto, sistemas de memória compartilhada distribuída (DSM - *Distributed Shared Memory*) mostram que é possível utilizá-lo de uma forma mais abrangente, em arquiteturas distribuídas.

O modelo de memória compartilhada pode ser implementado com diferentes primitivas, diversas formas de se expressar o paralelismo e sincronizar os processos. A iniciativa OpenMP[3] surgiu como um esforço de padronização deste modelo e tem despertado interesse crescente na comunidade científica no desenvolvimento de aplicações paralelas. A idéia é de que o programador possa facilmente desenvolver um programa paralelo inserindo diretivas de compilação em um código puramente seqüencial. As diretivas OpenMP especificam *loops* cujas iterações podem ser executadas em paralelo, funções que podem ser paralelizadas e o escopo de variáveis privativas e compartilhadas. Apesar de OpenMP ter sido desenvolvido para multiprocessadores com memória compartilhada, ele pode ser aplicado em sistemas distribuídos com suporte de sistemas software DSM (SDSM). O grande obstáculo da implementação OpenMP em um sistema SDSM é o desempenho. Um SDSM tem um *overhead* muito maior para acesso a dados remotos e sincronização.

Trabalhos anteriores implementaram OpenMP em sistemas distribuídos através de SDSM [2, 7]. Nestes trabalhos, entretanto, a tradução das diretivas OpenMP para as chamadas do SDSM envolve a conversão para o modelo SPMD (*Single Program Multiple Data*) de programação. No modelo SPMD, é criado um processo para cada processador da arquitetura, que executa o mesmo código para diferentes trechos da massa de dados. De uma forma simplificada, a tradução de uma aplicação OpenMP para execução no SDSM se resume a modificações nos limites dos *loops* a serem paralelizados. Esses limites dependem da porção dos dados atribuída a cada processo. As regiões paralelas são intercaladas com barreiras.

Neste trabalho propomos a utilização de SDSM na implementação e avaliação de aplicações OpenMP. Contudo, a tradução das diretivas OpenMP é realizada para o modelo de programação implementado pelo sistema SDSM Clik [9]. Clik implementa modelo de programação baseado na linguagem *multithread* Cilk [5] e provê distribuição de carga eficiente e totalmente transparente para o programador. Em Clik, a criação das *threads* é independente da quantidade de processadores do sistema. O sistema de tempo de execução é o responsável pelo es-

* Este trabalho é parcialmente financiado pelo MCT/FINEP, HP Brasil R&D

calonamento das *threads* e pelo balanceamento de carga. O modelo de sincronização e a criação de *threads* da linguagem Cilk, entretanto, diferem do modelo tradicional SPMD. Em SPMD as todas as *threads* são criadas estaticamente, uma única vez, no início da computação e sincronizam através de barreiras globais. Em Cilk as *threads* são criadas dinamicamente durante a execução da aplicação e a sincronização ocorre apenas entre a *thread*-pai e suas *threads*-filhas e não entre todas as *threads* criadas.

Propomos aqui uma metodologia para a tradução de aplicações OpenMP para a linguagem Cilk. Com essa metodologia, analisamos e avaliamos o desempenho de aplicações OpenMP executando em um *cluster* de SMPS, utilizando o sistema Clik. Executamos um conjunto de 3 aplicações e, em nossos resultados preliminares, obtivemos bom desempenho para a tradução quando a aplicação OpenMP não possui grande quantidade de barreiras implícitas.

O restante do trabalho está organizado da seguinte forma. Na próxima seção descrevemos o modelo de programação OpenMP com suas principais diretivas e o modelo proposto por Cilk. Na Seção 3 apresentamos nossa metodologia de tradução de OpenMP para Cilk. Em seguida, apresentamos os nossos experimentos, mostrando o desempenho de 3 aplicações OpenMP, executando em um *cluster* de SMPs com o sistema SDSM Clik. Na Seção 5 abordamos alguns trabalhos relacionados. Finalizamos, apresentando nossas conclusões e propostas de trabalhos futuros.

2. Programação no Modelo de Memória Compartilhada

Uma linguagem de programação paralela deve fornecer suporte para três aspectos básicos: execução paralela, comunicação e sincronização entre processos. Muitas linguagens paralelas fornecem este suporte através de extensões da linguagem seqüencial, porém, com diferentes abordagens, [8, 5]. Uma outra forma de expressar paralelismo pode ser através de chamadas a rotinas de bibliotecas de tempo de execução como em OpenMP[3] e Pthread[10].

2.1. Modelo OpenMP

OpenMP foi projetado para facilitar a portabilidade de programas paralelos baseados no modelo de memória compartilhada. OpenMP consiste de um conjunto de diretivas de compilador e rotinas de biblioteca de tempo de execução que podem ser inseridas dentro de um programa escrito em linguagem Fortran, C ou C++. Os principais benefícios obtidos com essa abordagem são: i) o mesmo código pode ser usado em plataformas com um único processador ou múltiplos processadores, onde simplesmente as diretivas podem ser tratadas como comentários e ignoradas numa execução seqüencial, e ii) permite que uma aplicação seja desenvolvida de forma incremental, iniciando com uma versão seqüencial e inserindo diretivas para gerar a versão paralela.

OpenMP segue o modelo de execução *fork-and-join*. Um programa *fork-and-join* inicializa como um único processo leve, denominado *master thread*. A *master thread* executa seqüencialmete até que seja encontrado o primeiro construtor paralelo (`omp parallel`). Nesse ponto, a *master thread* gera um conjunto de *threads*, incluindo ela mesma como um membro do conjunto, para executar concorrentemente comandos com construtores paralelos. Quando um construtor que compartilha trabalho é encontrado, por exemplo, um *loop* paralelo (`omp for`), a carga de trabalho é distribuída entre os membros do conjunto. Uma sincronização implícita ocorre no final do *loop*, a menos que um seja especificado um comando `nowait`. As variáveis compartilhadas são especificadas no início dos construtores paralelos ou de trabalho compartilhado, usando as cláusulas `shared` e `private`. Além disso, as operações de redução (tal como somatório) podem ser especificadas pela cláusula `reduction`. Após o término do construtor paralelo, as *threads* do conjunto são sincronizadas. Somente a *master thread* continua a execução. Os processos *fork-and-join* podem ser repetidos várias vezes durante a execução do programa. Entretanto, para todo *fork* (região paralela) existe um custo de ativação associado que é dependente da máquina.

A Figura 1 mostra um exemplo de um algoritmo simples, para o cálculo da multplicação de duas matrizes, expresso em OpenMP.

2.2. A Linguagem Cilk

A linguagem Cilk é uma extensão *multithread* da linguagem C que provê abstração de *threads* e mecanismos para sincronização. Ela foi desenvolvida para explorar paralelismo dinâmico e assíncrono que muitas vezes não pode ser expresso facilmente pelo modelo SPMD, como, por exemplo, a paralelização de funções recursivas. Uma aplicação Cilk pode ser representada por um grafo acíclico direcionado que cresce dinamicamente. Cada *thread* é um nó de um grafo acíclico e corresponde a uma função C. Quando uma *thread* é executada, ela pode criar novas *threads*, expandindo o grafo durante a execução. Quando uma *thread* T_1 depende do resultado de outra *thread* T_2 para executar, existe uma aresta de ligação entre T_1 e T_2 indicando essa dependência. A computação em Cilk, portanto, pode gerar uma árvore de *threads* que são executadas de acordo com as dependências descritas pelas arestas. *Threads* independentes podem executar em paralelo.

```
void m_mat(int *c, int *a, int *b)
{
   int i, j, k;

#pragma omp parallel
#pragma omp for private(i,j,tmp)
   for (i=0;i<N;i++) {
      c[i][j] = 0.0;
      for (j=0;j<M;j++) {
        for (k=0;k<M;k++)
         c[i][j]=c[i][j]
                 + a[i][k]*b[k][j];
      }
   }
}
int main ()
{
    int a[N][M],b[M][N],c[N][N];

    m_mat(c, a, b);
    printf("Finalizou multiplicação\n");

  }
}
```

Figura 1. Geração da multiplcação de duas matrizes em OpenMP

```
cilk int fib (int n)
{
  int x, y;

  if (n < 2) return n;
  else {
    int x, y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return(x + y);
  }
}
cilk int main ()
{
  int n, result;

  result = spawn fib(n);
  sync;
  printf(''Resultado: \%d \n'', result);
}
```

Figura 2. Geração do n-ésimo número de Fibonacci segundo a linguagem Cilk

A extensão *multithread* proposta por Cilk é muito simples. Ao contrário de OpenMP que provê várias formas de expressar o paralelismo e sincronizar *threads*, Cilk contém apenas três conjuntos de primitivas básicas:

- **spawn** - criação dinâmica de *threads*;
- **sync** - sincronização das *threads*;
- **lock/unlock** - acesso a dados com exclusão mútua.

As primitivas `spawn` e `sync` têm funcionalidades semelhantes às primitivas `fork` e `join` do Unix. Um `spawn` cria dinamicamente uma *thread*-filha que executa ao mesmo tempo que a *thread*-pai. Quando uma *thread* executa uma operação `sync`, ela espera que todos os seus filhos terminem para continuar a sua execução. O `sync` implementa uma barreira local e não uma barreira global. As primitivas `lock` e `unlock` garantem a execução com exclusão mútua de uma determinada seção crítica do código.

A Figura 2 mostra um exemplo de um algoritmo simples, que calcula o n-ésimo número de Fibonacci, escrito na linguagem Cilk. Como podemos observar, Cilk não emprega o modelo SPMD e permite expressar de forma natural o paralelismo de funções recursivas. Neste algoritmo, uma *thread*-pai é criada pelo procedimento `main`. Ao ser executada, ela cria duas *threads*-filhas, responsáveis pelo cálculo dos valores de `fib(n-1)` e `fib(n-2)`, respectivamente. A *thread*-pai, então, permanece aguardando num `sync` que suas filhas terminem para retornar o resultado. Cada *thread*-filha, por sua vez, cria outras duas *threads*-filhas e assim sucessivamente. Dessa forma, a árvore binária que descreve a computação cresce, até que nas folhas da árvore a computação de cada *thread*-filha seja o caso trivial ($n < 2$). Nesse caso, as filhas vão retornando os valores para os pais, até que o valor final seja retornado a raiz da árvore e impresso.

3. Tradução

Nesta seção apresentamos a metodologia de tradução de aplicações OpenMP para a linguagem Cilk. Esta metodologia busca identificar construções da linguagem Cilk que sejam equivalente semanticamente às diretivas e às cláusulas empregadas pelo padrão OpenMP. Ela inclui, ainda, o mapeamento entre o escopo de variáveis do programa original OpenMP para Cilk.

O paralelismo em aplicações OpenMP pode ser expresso através de diversas diretivas com diferentes cláusulas. As cláusulas estão relacionadas com o escalonamento e comportamento das *threads* que executam o trecho de código paralelo. Em função da diversidade de possibilidades, optamos, inicialmente, pela tradução das diretivas e cláusulas que ocorrem com mais freqüência, analisando o *benchmark*

NAS[4]. A seguir, abordamos a tradução dessas diretivas e, posteriormente, o mapeamento em relação ao escopo das variáveis.

3.1. Tradução das diretivas e cláusulas

Observamos que as diretivas mais freqüentes nas 8 aplicações do NAS, foram: `omp parallel`, `omp parallel for`, `omp for`, `omp master`, `omp single`, `omp barrier` e `omp critical`. A partir desta observação foi definida uma construção em Cilk para cada uma das diretivas.

A tradução da diretiva `omp parallel` pode ser observada na Figura 3. Observe que essa diretiva ocorre duas vezes dentro de *rotina A*, na versão OpenMP. Na tradução de cada uma delas é necessário a criação de uma rotina Cilk que contenha o código paralelo correspondente. A quantidade de *threads* Cilk disparadas para execução da rotina (`TOT_THREADS`), equivale à quantidade total de *threads* da versão OpenMP[1].

A Figura 4 mostra que a tradução da diretiva `omp parallel for` possui os mesmos critérios da tradução da diretiva `omp parallel`. Porém, para o *work sharing*, foi necessário o uso de duas variáveis, *btask* e *etask* para a atribuição de trabalho referente a cada *thread* participante.

A diretiva `omp for`, específica de *work sharing*, é sempre usada em OpenMP dentro do contexto de uma diretiva `omp parallel`. Ela é equivalente à diretiva `omp parallel for`. Assim, a tradução é realizada exatamente da mesma forma.

Nas aplicações OpenMP é possível encontrarmos uma combinação entre as diretivas `omp parallel` e `omp parallel for`. Quando combinadas, a tradução da diretiva `omp parallel` é realizada de forma ligeiramente diferente, como pode ser observado na Figura 5.

Foi possível mapear as diretivas `omp master` e `omp single` em uma única construção Cilk (Figura 6).

As cláusulas mais freqüentes consideradas durante a tradução foram `nowait`, `reduction`, `shared` e `private`. A cláusula `nowait`, em geral, é inserida numa região paralela ou de *work sharing*. Ela requer apenas que se crie outra rotina desmembrada a partir da rotina onde existe o paralelismo. A cláusula `reduction` necessita que seja criado um vetor compartilhado. Cada *thread* da região paralela, armazena o seu cálculo em uma posição distinta do vetor. Ao final da operação, uma única *thread* é responsável pela operação de redução sobre os valores previamente armazenados no vetor. A tradução em relação à modificação de escopo das variáveis é descrita a seguir.

Versão OpenMP:

```
int rotinaA()
{
#pragma omp parallel
{
   /* código A */
}

#pragma omp parallel
{
   /* código B */
}
}
```

Construção Cilk:

```
cilk ompclik_rotinaA1(int id_task)
{
   /* código A */
}

cilk ompclik_rotinaA2(int id_task)
{
   /* código B */
}

cilk int rotinaA()
{
  for (tk=0; tk < TOT_THREADS; tk++)
     spawn ompclik_rotinaA1(tk);
  sync;

  for (tk=0; tk<TOT_THREADS; tk++)
     spawn ompclik_rotinaA2(tk);
  sync;
}
```

Figura 3. Tradução da diretiva `omp parallel` OpenMP para Cilk

3.2. Mapeamento do escopo das variáveis

Em programas paralelos é necessário a distinção entre variáveis compartilhadas e privativas. Uma variável é privativa em OpenMP se for declarada dentro de uma região paralela. Porém, ela pode se tornar compartilhada se for declarado um modificador de escopo `shared` na diretiva da região paralela. A princípio, todas as variáveis declaradas fora de uma região paralela são compartilhadas. Elas só podem se tornar privativas se houver um modificador do tipo `private`. Dessa forma, nas aplicações OpenMP, todas as variáveis privativas devem ser declaradas dentro das respec-

1 Em OpenMP a quantidade total de *threads* é definida estaticamente pela variável de ambiente OMP_NUM_THREADS

Versão OpenMP:

```
int rotinaB()
{
#pragma omp parallel for
{
   for(i=0; i<MAX; i++){
    /* código A */
   }
}
}
```

Construção Cilk:

```
cilk ompclik_rotinaB1(int id_task)
{
   btask=(id_task * MAX)/TOT_THREADS;
   etask=((id_task+1)* MAX)/TOT_THREADS;

   for (i=btask; i<etask; i++){
        /* código A */
   }
}

cilk int rotinaB()
{
   for (tk=0; tk<TOT_THREADS; tk++)
        spawn ompclik_rotinaB1(tk);
  sync;
}
```

Figura 4. Tradução da diretiva `omp parallel for` OpenMP para Cilk

Versão OpenMP:

```
int rotinaB()
{
#pragma omp parallel
  #pragma omp parallel for
  {
     for(i=0; i<MAX; i++){
        /* código A */
     }
  }
}
}
```

Construção Cilk:

```
cilk ompclik_rotinaB1(int id_task)
{
  for (tk=0; tk<TOT_THREADS; tk++)
     spawn ompclik_rotinaB2(tk);
  sync;
}

cilk ompclik_rotinaB2(int id_task)
{
   btask=(id_task * MAX)/TOT_THREADS;
   etask=((id_task+1)* MAX)/TOT_THREADS;

   for (i=btask; i<etask; i++) {
        /* código A */
    }
}

cilk int rotinaB()
{
  for (tk=0; tk<TOT_THREADS; tk++)
     spawn ompclik_rotinaB1(tk);
  sync;
}
```

Figura 5. Tradução de diretivas combinadas OpenMP para Cilk

tivas rotinas Cilk. Além disso, caso essa rotina tenha sido desmembrada em várias outras rotinas, as variáveis devem ser alocadas como compartilhadas, para que cada *thread* tenha a sua própria cópia da variável em cada rotina desmembrada. As variáveis que controlam as iterações de um *work sharing* (privativas no OpenMP) possuem este mesmo mapeamento.

As variáveis compartilhadas e com escopo global em OpenMP devem ser declaradas na rotina `main` em Cilk (função `Cilk_dsm_malloc`) e passadas por parâmetro entre as rotinas que a utilizem.

4. Resultados Experimentais

Realizamos nossos experimentos em um *cluster* de SMPs do Laboratório de Computação Paralela da COPPE/UFRJ com 2 nós de processamento. Cada nó de processamento contém 2 processadores Pentium IV 2GHz, compartilhando 1 GB de memória. Os nós estão interconectados por uma rede Fast Ethernet. O sistema DSM utilizado para prover o modelo de memória compartilhada é o sistema Clik [9]. Clik implementa a linguagem Cilk em um *cluster*. A seguir apresentamos uma breve descrição do sistema Clik e, em seguida, apresentamos nossa avaliação de desempenho.

4.1. Sistema Clik

O sistema Clik é um sistema SDSM para *clusters* que implementa a linguagem Cilk. A proposta do sistema Clik é de prover um mecanismo eficiente de distribuição de tarefas nos nós de um *cluster* e um sistema de memória compartilhada distribuída entre estes nós.

A distribuição de tarefas entre os nós de processamento

Versão OpenMP:

```
int rotinaC()
{
#pragma omp parallel
{
  #pragma omp master
  {
        /* código A */
  }
  #pragma omp single
  {
        /* código B */
  }
}
}
```

Construção Cilk:

```
cilk ompclik_rotinaC1(int id_task)
{
   if (id_task == 0) {
        /* código A */
   }
   if (id_task == 0) {
        /* código B */
   }
}

cilk int rotinaC()
{
  for (tk=0; tk<TOT_THREADS; tk++)
     spawn ompclik_rotinaC1(tk);
  sync;
}
```

Figura 6. Tradução de diretivas `omp master` e `omp single` OpenMP para Cilk

inclui um algoritmo de roubo de trabalho (*work-stealing*). O sistema Clik inicia um conjunto de processos trabalhadores em cada nó. Cada trabalhador tem sua fila de tarefas independente. Todas as filas de tarefas estão inicialmente vazias, exceto a do trabalhador 0, que é responsável pela execução da primeira *thread*-pai (raiz da árvore de execução). À medida que a *thread*-pai cria *threads*-filhas, novas tarefas são inseridas na fila. Estas tarefas podem ser "roubadas" por outros trabalhadores. O algoritmo de *work-stealing* procede da seguinte forma. Primeiramente, o trabalhador tenta obter uma tarefa de um outro trabalhador localizado no mesmo nó. Caso não consiga obter uma tarefa, ele, então, inicia o *work-stealing* remoto. O *work-stealing* remoto envia uma mensagem de roubo de trabalho para um outro nó (escolhido aleatoriamente) requisitando trabalho. Ao receber uma resposta de que o nó, também, não tem trabalho, ele inicia o *work-stealing* remoto novamente. Ao receber como resposta de que há trabalho, ele insere a *thread* a ser executada na sua fila de tarefas local. O *work-stealing* só é ativado novamente quando a fila de tarefas local do trabalhador estiver vazia.

A área de memória compartilhada necessária para a execução de aplicações *multithread*, é provida por um protocolo de consistência relaxada baseado em residência (*home-based*), semelhante ao sistema HLRC [12]. O protocolo utiliza invalidações para garantir a coerência de uma página, baseado em intervalos para estabelecer uma ordenação parcial das escritas feitas na memória. Um trabalhador recebe invalidações para uma página quando recebe uma tarefa de outro nó (*work-stealing*) e ao executar uma operação `sync`. O sistema Clik implementa um protocolo de múltiplos escritores, permitindo que escritas em áreas de dados diferentes de uma mesma página sejam realizadas concorrentemente. Estas escritas são enviadas para o nó *home* da página na forma de *diffs* em três momentos: quando uma nova tarefa é criada (`spawn`); na execução de um `sync`; e no retorno de uma tarefa. Quando um trabalhador sofre uma falha de página ele requisita uma cópia atualizada da página ao nó *home* daquela página.

4.2. Avaliação de Desempenho

Implementamos a metodologia de tradução descrita na Seção 3 para 3 aplicações diferentes: multiplicação de matrizes, CG e IS, as duas últimas do conjunto NAS de benchmarks [4]. A aplicação de multiplicação de matrizes, em especial, foi utilizada para comparar a versão traduzida de OpenMP com a versão recursiva escrita para a linguagem Cilk. CG e IS não possuem implementação própria para a linguagem Cilk, portanto, avaliamos apenas os *speedups* obtidos após a tradução efetuada com nossa metodologia.

4.2.1. Multiplicação de Matrizes As Tabelas 1 e 2 mostram os *speedups* obtidos para a execução da multiplicação de matrizes 512×512 e 1024×1024, respectivamente, em dois códigos diferentes: i) versão traduzida de OpenMP para Cilk (**Omp_Cilk**) com nossa metodologia; ii) versão recursiva escrita originalmente para Cilk (**Cilk**), baseada no paradigma de divisão e conquista, facilmente expresso pela linguagem Cilk[2]. Com esses experimentos, estamos interessados em avaliar o desempenho da nossa tradução com relação à melhor implementação da multiplicação de matrizes para Cilk em um *cluster* de SMPs. Além da variação das versões do código executado, variamos também a quantidade de processos

2 A versão recursiva da multiplicação de matrizes em Cilk, divide a matriz em submatrizes de tamanhos idênticos, recursivamente, até que o tamanho da submatriz gerada seja trivial, a multiplicação de matrizes seqüencial é aplicada nas submatrizes triviais e os resultados combinados

		Omp_Cilk			Cilk
		# Threads			
		4	8	16	
1 nó SMP		1.6	1.7	2.1	2.0
2 nós SMP	2 trab.	1.5	2.5	3.0	2.1
	4 trab.	2.1	2.0	2.2	2.2

Tabela 1. Multiplicação de Matrizes 512×512

		Omp_Cilk			Cilk
		# Threads			
		4	8	16	
1 nó SMP		1.7	1.6	2.0	2.5
2 nós SMP	2 trab.	3.2	3.2	3.2	2.7
	4 trab.	3.5	3.2	2.5	2.2

Tabela 2. Multiplicação de Matrizes 1024×1024

trabalhadores por nó do sistema SDSM Clik quando utilizamos 2 nós de processamento (na execução com apenas 1 nó, Clik executou com 2 trabalhadores). Para Omp_Cilk estabelecemos diferentes quantidades de *threads* criadas: 4, 8 e 16.

Comparando-se o desempenho de Omp_Cilk com Cilk, podemos observar nas Tabelas 1 e 2, que a versão traduzida executa tão bem ou até um pouco melhor que a versão original recursiva de Cilk, principalmente com 16 *threads*. A aplicação de multiplicação de matrizes apresenta pouca sincronização implícita em OpenMP, por esse motivo, sua tradução para Cilk ficou bastante eficiente. Neste caso, a ligeira vantagem da tradução sobre a versão recursiva se deve ao fato de que a versão recursiva precisa da fase de combinação dos resultados. As *threads*-filhas devem passar seus valores para as respectivas *threads*-pai. Quando pai e filha executam em nós distintos, por conta do *work-stealing*, esta operação envolve trocas de mensagens para a manutenção da coerência dos dados. Para a massa de dados menor, o aumento do número de *threads* geradas para Omp_Cilk se traduz em melhora no *speedup*. Para a massa de dados maior, entretanto, observamos que o aumento da quantidade de *threads* de Omp_Cilk não resulta em aumento de *speedup*. Isso ocorre devido à grande quantidade de dados que devem ser transferidos pelo sistema SDSM a cada roubo de trabalho.

4.2.2. IS e CG A aplicação IS é um *benchmark* de ordenação que classifica um vetor de N inteiros, utilizando chaves no intervalo [0, B_{max}], através da técnica *bucket sort*. A entrada utilizada foi $N = 2^{23}$, $B_{max} = 2^{15}$ e 10 iterações. A aplicação CG utiliza um método *Conjugate Gradient* para computar uma aproximação para os menores autovalores de uma grande matriz não estruturada e esparsa. A entrada utilizada pertence à classe B (matriz com 75000 elementos e 75 iterações).

		IS			CG		
		# Threads			# Threads		
		4	8	16	4	8	16
1 nó SMP		0.8	0.9	1.0	1.3	1.3	1.9
2 nós SMP	2 trab.	0.4	0.3	0.2	0.4	0.4	0.4
	4 trab.	0.3	0.2	0.1	0.3	0.3	0.4

Tabela 3. Resultados de IS e CG

A Tabela 3 mostra os *speedups* obtidos para a execução da versão traduzida de IS e CG em Clik para 1 e 2 nós de processamento (2 e 4 processadores), com a criação de 4, 8 e 16 *threads*. Conforme podemos observar nesta tabela, os desempenhos das traduções de IS e CG são muito ruins. Os *speedups* são muito baixos nesta arquitetura, quando não há *slowdown*. Este resultado se deve à grande quantidade de operações de sincronização (`sync`) inseridas no código como uma forma de implementar as barreiras implícitas de OpenMP. Essas sincronizações degradam o desempenho do sistema principalmente quando mais de 1 nó é utilizado. Neste caso, devido à necessidade de operações de coerência de dados, muitas trocas de mensagens são necessárias. O problema do excesso de barreiras implícitas em OpenMP em sistemas distribuídos já foi abordado em outros trabalhos [7, 2, 6]. Ele continua sendo o grande obstáculo para a implementação de OpenMP em sistemas distribuídos, conforme mostra a seção de trabalhos relacionados, a seguir.

Este resultado negativo mostra que a tradução direta de OpenMP para Cilk não deve ser feita quando muitas regiões paralelas (e suas respectivas barreiras implícitas) existirem. Neste caso, é necessário um mecanismo de tradução que aproveite a sincronização mais "leve" provida pela primitiva `sync` de Cilk e que combine algumas regiões paralelas em procedimentos únicos Cilk.

5. Trabalhos Relacionados

A implementação de OpenMP em sistemas de memória distribuída não é trivial. Devido ao seu potencial como um modelo de programação de alto nível para aplicações paralelas, a sua implementação em sistemas de memória distribuída tem sido estudada em uma série de trabalhos [1, 7, 2, 6, 11].

Kee *et al*[7] propõem um ambiente de programação baseado em OpenMP, denominado ParADE (*Parallel Application Development Environment*). ParADE consiste de um tradutor e um sistema de tempo de execução. A tradução das aplicações OpenMP para ParADE é automática, mas ela é

feita, entretanto, para um modelo híbrido de passagem de mensagens e memória compartilhada. A proposta de utilizar o modelo híbrido em ParADE é de evitar o excesso de sincronização de operações de *locks* e barreiras, impostos implicitamente pelo padrão OpenMP.

Outra implementação de OpenMP para SDSM foi proposta por Hu *et al.*[6]. Eles estenderam as primitivas de TreadMarks para suportar aplicações OpenMP. A tradução de um programa OpenMP em C para TreadMarks é realizada por um compilador baseado no pré-processador SUIF. Sua principal contribuição foi ser o primeiro a desenvolver um sistema SDSM *multithreaded* para OpenMP. Este estudo mostra, tal qual nossos resultados, que aplicações desenvolvidas em OpenMP possuem mais barreiras e um alto *overhead* com tráfego na rede. A diferença é que estamos nos baseando em um modelo de programação distinto e utilizando outro SDSM como alvo.

Basumallik *et al.*[2] discutiram as questões de desempenho de OpenMP em sistemas SDSM. Eles apresentaram diversas idéias de técnicas de otimização para melhorar o desempenho. Além disso, ressaltaram as sincronizações freqüentes, especificamente barreiras, como o principal obstáculo para o alto desempenho, mas não apresentaram qualquer sistema real para demonstrar suas idéias.

6. Conclusões

Neste trabalho apresentamos uma metodologia de tradução de aplicações OpenMP para a linguagem Cilk. Avaliamos o potencial de nossa metodologia em um *cluster* de SMPs, executando um conjunto de 3 aplicações OpenMP no sistema SDSM Clik. Clik implementa a linguagem Cilk e fornece um mecanismo eficiente de distribuição de tarefas nos nós do *cluster*.

Em nossos resultados preliminares, utilizando a tradução da forma mais direta possível, obtivemos bom desempenho para a tradução quando a aplicação OpenMP não possui grande quantidade de barreiras implícitas. Concluímos que este problema deve ser melhor avaliado em versões futuras da metodologia de tradução onde pretendemos avaliar a possibilidade da combinação diferentes regiões paralelas em um único procedimento Cilk.

Referências

[1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, 1996.

[2] A. Basumallik, S.-J. Min, and R. Eigenmann. Towards openmp execution on software distributed shared memory systems. In *ISHPC - LNCS 2327*, pages 457–468, 2002.

[3] R. Chandra, L. Dagun, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publisher, 2001.

[4] D. B. et al. The NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[5] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, New York, NY, USA, 1998. ACM Press.

[6] Y. C. Hu, H. Lu, A. L. Cox, and W. Zwaenepoel. OpenMP for networks of SMPs. *Journal of Parallel and Distributed Computing*, 60(12):1512–1530, December 2000.

[7] Y.-S. Kee, J.-S. Kim, and S. Ha. ParADE: An OpenMP programming environment for SMP cluster systems. In *SC*, page 6, 2003.

[8] D. B. Loveman. High performance Fortran. *j-IEEE-PAR-DIST-TECH*, 1(1):25–42, Febrary 1993.

[9] R. Mendes, L. Whately, C. Bentes, and C. Amorim. Implementação e avaliação preliminar de um novo sistema software dsm para cluster de computadores, 2005. artigo submetido para avaliação.

[10] B. Nichols, D. Buttlar, and J. Farrell. *Pthreads programming*. Nutshell handbook. O'Reilly & Associates, Inc, 1998.

[11] M. Sato, H. Harada, A. Hasegawa, and Y. Ishikawa. Cluster-enabled OpenMP: An OpenMP compiler for the SCASH software distributed shared memory system. *j-SCI-PROG*, 9(2–3):123–130, Spring–Summer 2001.

[12] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *OSDI '96: Proceedings of the second USENIX symposium on Operating systems design and implementation*, pages 75–88, New York, NY, USA, 1996. ACM Press.