Implementação de suporte ao paralelismo geométrico em Anahy*

Lucas Adams Seewald†, Gerson Geraldo H. Cavalheiro

Programa Interdisciplinar de Pós-Graduação em Computação Aplicada
Universidade do Vale do Rio dos Sinos
São Leopoldo - RS - Brasil
ladams@turing.unisinos.br, gersonc@unisinos.br

Introdução

Atualmente, verifica-se que aglomerados de computadores, comumente denominados de *clusters*, são intensamente utilizados como solução de arquitetura de hardware para o processamento de alto desempenho (PAD). Programadores fazem uso de ferramentas de programação concorrente, paralela e distribuída para explorar a capacidade de processamento destas arquiteturas na implementação de suas aplicações. Dentre estas ferramentas, as mais utilizadas oferecem modelos de programação concebidos com outros fins que não o desenvolvimento de aplicações para PAD. Estas ferramentas, ditas tradicionais, se caracterizam por oferecer primitivas elementares de programação para a exploração dos recursos de processamento paralelo da arquitetura, tais como ativação de uma nova execução ou ocupação do meio de comunicação pelo envio de uma mensagem. Como exemplos destas ferramentas, podem ser citadas aquelas baseadas em multiprogramação leve e em troca de mensagens.

Ferramentas baseadas em multiprogramação leve exploram uma arquitetura de hardware dotada de um espaço de endereçamento compartilhado entre diversos processadores. Este modelo de programação encontra-se disponível em diversas bibliotecas de programação, por exemplo bibliotecas baseadas nos padrões POSIX para threads e OpenMP. Ferramentas baseadas em troca de mensagem exploram uma arquitetura de hardware onde a comunicação entre processadores somente é possível através do uso de um meio de comunicação externo (rede). Este modelo de programação se encontra disponível em bibliotecas de programação baseadas no padrão sockets, ou ainda, MPI e PVM.

No entanto, com o crescente uso de aglomerados com suporte de hardware para aplicações de PAD, novos modelos de programação concorrente [ELR 97, SKI 98], oferecendo maiores níveis de abstração, se fazem necessários. Assim, diversos ambientes de programação tem sido propostos, entre eles Anahy [COR 2005, CAV 2003], o qual implementa um modelo de programação com decomposição explícita de atividades concorrentes com mapeamento implícito destas atividades nos recursos de hardware [SKI 98]. A interface de programação (API) de Anahy foi desenvolvida segundo o modelo de memória compartilhada, sendo implementada segundo o padrão POSIX para threads.

*Apoio: FAPERGS
†PIBIC/CNPq

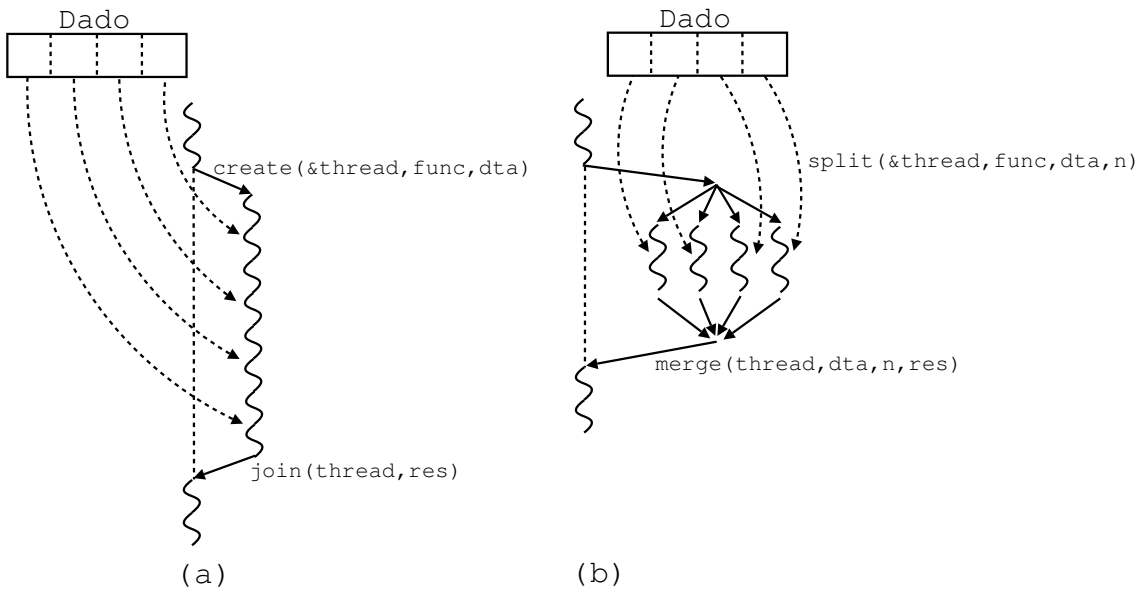

Figura 1: Comparação entre o modelo de criação de threads tradicional e SCM

Uma restrição da implementação realizada para Anahy é limitar a descrição da concorrência em termos de atividades concorrentes. Este trabalho visa extender a API de Anahy para contemplar também a possibilidade de descrição de concorrência em termos de paralelismo de dados (ou paralelismo geométrico). Este trabalho apresenta o modelo Split-Compute-Merge (SCM) e a extensão proposta à API de Anahy.

Paralelismo geométrico em Anahy

O modelo Split-Compute-Merge (SCM) é adaptado a problemas onde o paralelismo geométrico pode ser aplicado. Ele emprega três operadores: *split*, *compute* e *merge*. A funcionalidade oferecida pelo modelo é a de possibilitar a execução de múltiplas réplicas de uma mesma função (*compute*), uma vez tendo sido informado um conjunto de dados de entrada para esta. A propriedade a ser obedecida para o problema é de que os dados de entrada possam ser divididos (*split*) em diferentes subdomínios e computados de forma independente. O resultado final é a composição (*merge*) dos resultados parciais obtidos por cada uma das réplicas através de uma operação cumulativa com semântica equivalente ao operador += da linguagem C.

A Figura 1 compara a utilização do modelo SCM (b) com o modelo tradicional de criação de thread (a). Nesta figura se encontra representada a exploração da concorrência da aplicação em seus diferentes domínios de dados. Para que este modelo seja suportado em uma implementação real, o programador deve ser capaz de prover as funções *split* e *merge* considerando os dados reais manipulados.

Para Anahy suportar SCM, sua API tem que ser extendida de forma a possibilitar a execução paralela de réplicas de um mesmo trecho de código atuando sobre diferentes domínios de um mesmo conjunto de dados. O número *n* réplicas é fornecido pelo pro-

gramador, indicando o número de *threads* independentes que executarão o trecho mencionado, bem como o número de subconjuntos nos quais os dados serão divididos. *Split* deve ter caráter seletivo, provendo as distintas entradas para cada *thread*. Quando todas tiverem concluído sua execução, *n* saídas terão sido geradas, sendo, para cada saída, realizado o *merge* dos resultados. Ao término de todas as *threads* o resultado é disponibilizado no endereço enviado para a rotina de sincronização.

Implementação

Para que seja mantido o caráter genérico que as funções *split* e *merge* exigem, e em respeito ao padrão POSIX, seguido por Anahy, a entrada e a saída das *threads* devem ser do tipo `void*`. Tais ponteiros podem referenciar um único dado, ou uma estrutura contendo quantos campos forem necessários. *Split*, *compute* e *merge* são providenciadas pelo programador, visto que ele determina os dados, bem como sua manipulação. As funções seguem os seguintes protótipos:

```
void *split(void *dta, int n, int r)
void *compute(void *dta)
void *merge(void *dta, int n, int r, void *res)
```

O parâmetro `n` representa o número de réplicas efetivamente criadas e `r` o identificador da réplica no grupo criado. Ambas são necessárias para realizar cálculos seletivos com respeito ao tratamento que os dados receberão antes e após a execução de cada *thread*. Estas, por sua vez, têm seu código de execução definido em `compute`.

Quando utilizadas as funções `split` e `merge`, os dados apontados pelo primeiro argumento da rotina de criação de *threads* Anahy (`athread_create`) já não representam mais o `dta` de `compute`, mas o de `split`. De forma análoga, os ponteiros retornados pelas *threads* não apontam para os dados que devem ser disponibilizados pela função de sincronização (`athread_join`), porém para os dados equivalentes aos argumentos `dta` de `merge`. Apenas após esta ser executada *n* vezes o resultado estará formado em `res`, passando então a ser apontado pelo segundo argumento de `athread_join`. No caso de `compute` o parâmetro `dta` se refere ao retorno de `split`. Uma particularidade de `merge`, devida a sua atuação incremental, é a necessidade do ponteiro `res`, também do tipo `void*`, que estabelece comunicação entre o resultado da união das saídas das *threads* já executadas e as novas saídas que serão geradas pelas demais cuja execução ainda não foi concluída.

Para implementar esta funcionalidade em Anahy, foram adicionados três novos campos aos atributos das Anahy *threads*: o número de *threads* a serem executadas, um ponteiro para a função `split` e outro para a função `merge`. Foram desenvolvidas três funções para manipular estes atributos:

```
int athread_attr_set_splitnum(athread_attr_t *attr, int n)
int athread_attr_set_splitfunc(athread_attr_t *attr,
                               void *(* split)(void *, int, int))
int athread_attr_set_mergefunc(athread_attr_t *attr,
                               void *(* merge)(void *,
                               int, int, void *))
```

Um dado do tipo `athread_t`, após devidamente inicializado, deve ter o seu endereço enviado como segundo argumento de `athread_create`. Durante a criação das *n threads* Anahy especificadas, cada uma receberá uma identificação única. Para manter um elo entre elas, todas apontam para um vetor de identificações comum. Através deste a rotina `merge` é capaz de identificar a relação entre as *threads*, selecionando o tratamento adequado para os dados independentemente da ordem de término das mesmas. Antes da conclusão de `athread_join`, o vetor de identificação tem sua área de memória liberada e as *threads* têm sua identificação original mantida.

Até o momento foram desenvolvidas duas aplicações utilizando o modelo SCM. A primeira consiste em um acumulador que soma todos os valores contidos em um vetor de inteiros. A segunda se trata de um multiplicador de matrizes quadradas.

O acumulador faz uso da função *split* para dividir o vetor em *n* subvetores. Uma rotina que soma (*compute*) todos os inteiros de um vetor foi executada sobre cada um deles por uma *thread* diferente. Por fim, através do *merge*, as somas dos subvetores são somadas, e este resultado é disponibilizado na área de memória apontada pelo segundo argumento de `athread_join`.

O multiplicador de matrizes quadradas faz uso da propriedade associativa das matrizes para particioná-las em blocos de mesmo tamanho. A partir de duas matrizes quadradas iguais ele gera duas vezes *n* blocos. Novamente observa-se que *split* não se limita a dividir a área de dados, pois além de gerar blocos também faz a seleção de quais devem ser enviados para a função de multiplicação (*compute*). Esta se resume a multiplicar os dois blocos linha por coluna. Como se espera, haverão *n* submatrizes de tamanho idêntico ao dos blocos. Desta vez *merge* dispõe cada uma destas submatrizes na posição adequada dentro da matriz resultante.

Conclusão

Neste artigo foi exposto o princípio da expansão de Anahy para incluir o modelo SCM. Através deste se visa maximizar o desempenho proporcionando paralelismo de dados e aumentando a granulosidade. Como ainda são necessários aprimoramentos e testes antes do término da implementação, ainda não foram obtidos índices de desempenho ou testes definitivos. Próximos passos nesta linha de trabalho extenderão a API de Anahy com outros modelos de expressão de concorrência.

Referências

[CAV 2003] CAVALHEIRO, G. G. H.; REAL, L. C. V.; DALL'AGNOL, E. C. Uma biblioteca de processos leves para a implementação de aplicações altamente paralelas. **IV Workshop em Sistemas Computacionais de Alto Desempenho**, p.117–124, Novembro 2003.

[COR 2005] CORDEIRO, O. C. et al. Exploiting multithreaded programming on cluster architectures. **The 19th International Symposium on High Performance Computing Systems and Applications - University of Guelph, Guelph, Ontario, Canada**, p.90–96, Maio 2005.

[ELR 97] EL-REWINI, H.; LEWIS, T. **Distributed and parallel computing**. [S.l.]: Manning Publications, 1997.

[SKI 98] SKILLICORN, D. B.; TALIA, D. Models and languages for parallel computation. **ACM Computing Surveys**, v.30, n.2, p.123–169, Junho 1998.