Melhorando a Geração Automática de Código Paralelo em Arquiteturas Multi-core na SPar

Júnior Löff[1], **Dalvan Griebler**[1], **Luiz Gustavo Fernandes**[1]

[1] Escola Politécnica, Grupo de Modelagem de Aplicações Paralelas (GMAP), Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), Porto Alegre, Brasil

junior.loff@acad.pucrs.br, {dalvan.griebler,luiz.fernandes}@pucrs.br

Resumo. *Neste trabalho, a fim de melhorar a eficiência do código paralelo gerado em arquiteturas multi-core, foi estendida a linguagem e o compilador da SPar para permitir a geração automática de padrões paralelos pertencentes aos dois principais domínios de paralelismo, o de stream e de dados. Experimentos mostram que a nova versão da SPar obteve resultados similares, ou até mesmo melhores, que as versões implementadas manualmente.*

1. Introdução

Processadores *multi-core* têm se popularizado a medida que limites físicos impediram continuar aumentando a frequência em processadores sequenciais. Assim, é imprescindível usar o paralelismo para extrair o máximo desempenho das arquiteturas *multi-core*. Por outro lado, a programabilidade paralela ainda é um desafio, visto que o desenvolvedor precisa lidar com detalhes específicos da arquitetura e de sistemas operacionais. Adicionalmente, o programador está suscetível à introduzir erros no código, pois o desenvolvimento mistura a lógica da aplicação com os detalhes de baixo nível da programação paralela. Recentemente, introduziu-se o paradigma de programação paralela estruturada [McCool et al. 2012], onde bibliotecas oferecem padrões paralelos através de *templates* ou funções do C++. A principal vantagem em utilizar os padrões paralelos é que os detalhes de baixo nível do paralelismo são isolados da lógica da aplicação.

Entretanto, os padrões paralelos existentes podem ser insuficientes para modelar aplicações mais complexas. Nessas aplicações, ainda é um desafio paralelizá-las, pois é necessário implementar manualmente toda a modelagem e sincronização dos dados. Assim, bibliotecas de maior nível de abstração vem sendo propostas. Neste cenário, encontram-se as bibliotecas com propósito específico e linguagens de domínio específico (DSL). Estas soluções implementam uma camada adicional de abstração sobre bibliotecas de propósito geral, ou seja, utilizam as bibliotecas já consolidadas para intermediar a geração de código paralelo. De acordo com Griebler [Griebler et al. 2017], as bibliotecas de propósito geral, como OpenMP, Intel TBB e FastFlow, ainda expõem demasiados detalhes específicos do paralelismo ao programador. Propondo uma solução, Griebler introduz a SPar, uma DSL do C++ para expressar paralelismo através de atributos do C++11.

Os dois principais trabalhos relacionados são descritos a seguir. O trabalho de [Thies et al. 2002] introduz uma DSL chamada StreamIt, composta por uma linguagem e compilador. Similar à SPar, o objetivo do StreamIt é aumentar o nível de abstração através de transformações *source-to-source* para geração automática de código paralelo. Entretanto, o StreamIt requer aprender uma linguagem nova baseada em Java, enquanto isso a SPar utiliza atributos da sintaxe padrão do C++. O trabalho [del Rio Astorga et al. 2017] introduz o GrPPI (Generic Reusable Parallel Pattern Interface), que como o nome sugere, oferece uma API com padrões paralelos genéricos. Ou seja, o programador implementa

os padrões (Map, Pipeline, etc.) apenas uma única vez e escolhe para qual biblioteca a ferramenta deve gerar código paralelo. Diferente da SPar, no GrPPI o programador é responsável por identificar o melhor padrão e implementá-lo manualmente.

Este trabalho objetiva contribuir diretamente com a SPar, apesar de que as descobertas podem ser implementadas em outras DSLs no futuro. Atualmente, a SPar gera código paralelo automático apenas para o paradigma de paralelismo de *stream*. No entanto, muitas aplicações complexas como inteligência artificial, exploração de recursos naturais, análise de imagens geoespaciais (desmatamentos) e outras, exibem também fluxos intensos de dados, que poderiam ser melhor modelados por padrões do paralelismo de dados. Neste trabalho, investigamos se a partir de uma única interface de programação paralela (SPar) é possível gerar código paralelo automático e eficiente para diferentes domínios do paralelismo (*stream* e de dados) em arquiteturas *multi-core*. Este trabalho está organizado da seguinte forma. A seção 2 apresenta a proposta de extensão da linguagem SPar. A seção 3 discute os resultados e a seção 4 apresenta as conclusões.

2. Proposta de extensão da linguagem

Atualmente, a SPar gera código paralelo automático para os padrões paralelos Pipeline e Farm, ou composições destes. Estes padrões são expressivos a ponto de permitir a modelagem da maioria das aplicações [McCool et al. 2012]. Entretanto, o código gerado nem sempre é eficiente, como será mostrado através de experimentos na seção 3. De fato, algumas vezes o desempenho é tão baixo que o tempo de execução da versão paralela é maior que o tempo sequencial. Assim, este trabalho propõem uma extensão da linguagem SPar, focando em arquiteturas *multi-core*, para suportar também o paralelismo de dados. Os padrões adicionados na SPar serão o Map e MapReduce, devido a sua popularidade. Após a extensão da linguagem, a SPar consegue gerar código paralelo automático para quatro padrões paralelos e composições destes: Pipeline, Farm, Map e MapReduce. A composição de padrões é uma contribuição adicional, visto que a SPar permitirá estudar o comportamento da composição de padrões do paralelismo de *stream* e de dados.

A SPar foi originalmente desenvolvida para suportar apenas o domínio de paralelismo de *stream*, onde possui cinco atributos para expressar informações do fluxo de dados no código: (1) **ToStream** indica onde começa e termina o fluxo de *stream*; (2) **Stage** indica onde começa e termina um estágio/bloco sequencial do fluxo de dados; (3) **Input** e **Output** como o nome sugere, são as entradas e saídas de cada bloco de código; (4) **Replicate** é um atributo especial para replicar um bloco sequencial quando não há dependência de dados. Durante o estudo, concluímos que os atributos não são suficientes para expressar o paralelismo de dados. Assim, propomos a extensão da linguagem SPar para suportar dois novos atributos, que expressam informações do paralelismo de dados: (1) **Pure** significa que um laço de repetição é puro, ou seja, pode ser executado em paralelo já que não apresenta dependência de dados. (2) **Impure** pode ser utilizado para anotar uma região de código impura, a fim de purificá-la. Atualmente este atributo consegue resolver apenas problemas de redução (intrínseco ao padrão MapReduce).

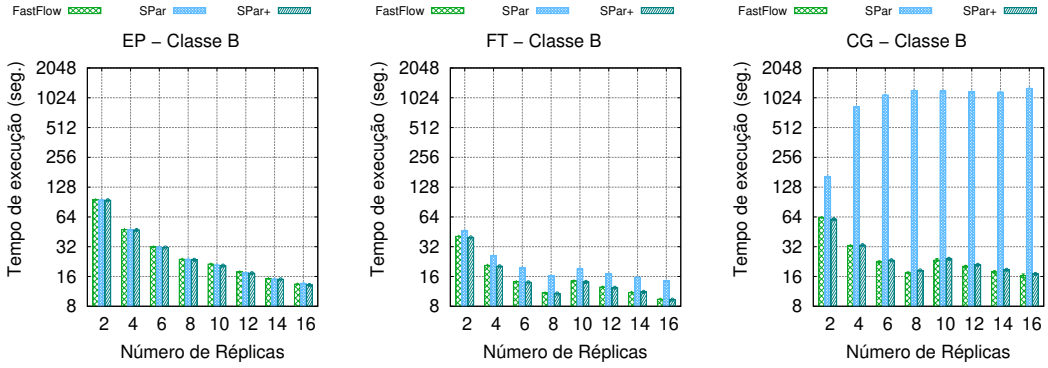
Uma vez que estendemos a linguagem SPar através de dois novos atributos, também foi necessário modificar as definições básicas e regras de transformação. Nós criamos dois conjuntos de regras de transformação: (1) O primeiro conjunto de regras identifica se o código anotado satisfaz todas as restrições dos padrões do paralelismo de dados. Caso satisfaça, então a SPar transforma todo o fluxo de dados (**ToStream**) nos padrões Map ou MapReduce. Alguns exemplos de restrições do paralelismo de dados são: conhecer o conjunto de dados, existir um laço de repetição `for` do C++ e não apresentar sincroni-

zações com regiões externas (`return`, `break`, `socket`, etc.). (2) O segundo conjunto de regras é executado em concomitante com as regras de transformação originais da SPar. Além disso, essas regras também permitem a composição dos novos padrões paralelos adicionados na SPar (Map e MapReduce) com os padrões originais (Pipeline e Farm).

Para implementar a extensão da linguagem SPar e as novas regras de transformação, nós criamos um novo algoritmo para o compilador da SPar (CINCLE). Como a SPar originalmente gera código intermediário para a biblioteca FastFlow [Aldinucci et al. 2017], nós também iremos gerar código paralelo para esta mesma biblioteca. Uma vez que o programador anota uma região de código com **Pure** ou **Impure**, o compilador automaticamente extrai as informações necessárias de acordo com a ISO padrão do C++. Se todas as informações forem localizadas no código e as restrições satisfeitas, então são gerados os padrões Map ou MapReduce através de transformações *source-to-source*. Caso contrário, não é seguro gerar os padrões do paralelismo de dados (código semanticamente incorreto), então a operação é abortada e o fluxo original da SPar assume controle, gerando o mesmo o código paralelo de sempre.

3. Experimentos

Os experimentos foram executados em uma máquina equipada com 64 GB de memória RAM e um processador Intel(R) Xeon(R) Silver 4108 CPU @ 1.80GHz que possui 8 núcleos físicos com suporte *hyper-threading*, totalizando 16 virtuais. O sistema operacional era Ubuntu 18.04 com *kernel* 4.15.0-123-generic. Nós utilizamos GCC 7.5 com a flag de otimização -O3. A versão do FastFlow era v3.0.0. Os testes foram executados com três *kernels* do NAS Parallel Benchmarks (NPB), obtidos de [Griebler et al. 2018]. Utilizamos a classe B do NPB, onde os parâmetros podem ser consultados no site oficial [1]. Os testes foram executados de 1 até o grau máximo de paralelismo. A execução foi repetida 5 vezes e os gráficos representam a média destes valores, com desvio padrão representado através de barras de erros em cima das colunas.



(a) EP **(b)** FT **(c)** CG

Figura 1. NPB com FastFlow [Griebler et al. 2018] vs. SPar e SPar+ (Nova versão).

Os gráficos da Figura 1 resumem os resultados, onde o eixo x mostra o número de réplicas (*threads* responsáveis pela computação intensa) e o eixo y o tempo de execução em segundos (escala logarítmica). Comparamos a SPar original e a nova versão SPar+ estendida neste trabalho, com uma versão manualmente otimizada do FastFlow obtida de [Griebler et al. 2018]. Na Figura 1a as diferenças não são significativas, visto que o

[1] https://www.nas.nasa.gov/publications/npb_problem_sizes.html

EP possui toda a computação intensa em um único laço de repetição. Entretanto, é possível identificar alguns problemas, por exemplo, em 16 réplicas a SPar possui mais *threads* que núcleos no processador (16 réplicas + 1 emissor), resultando em disputa de recursos. Na Figura 1b a sub-utilização de recursos na SPar é mais evidente, visto que o escalonador é centralizado em uma única *thread*. O problema é que as *threads* de computação precisam esperar até receber novas tarefas do escalonador. O tempo de ociosidade aumenta a medida que a fila possui mais *threads* em paralelo. A Figura 1c evidencia ainda mais esse problema, pois o tempo de execução da SPar supera o tempo sequencial em uma ordem de magnitude. O CG apresenta baixa granularidade, computação fragmentada devido às sincronizações e acessos irregulares na memória. Somando esses problemas com o fato da SPar centralizar o escalonamento, o resultado são desempenhos negativos. Em todos os gráficos da Figura 1, a nova SPar+ ficou similar (Figura 1c) e até mesmo melhor (Figuras 1a e 1b) que a versão implementada manualmente. A explicação é que o código paralelo gerado automaticamente pela SPar+ consegue utilizar melhor os recursos computacionais em comparação com o código gerado pela SPar. Nos experimento do NPB, padrões do paralelismo de dados (Map e MapReduce) se mostraram mais eficientes que padrões do paralelismo de *stream* (Pipeline e Farm). Além disso, os resultados da SPar+ são melhores que a versão manual do FastFlow, pois implementamos uma nova estratégia de redução baseada em sobrecarga de operadores (*operator overloading* do C++).

4. Conclusões

Este trabalho apresentou uma extensão da DSL SPar, a fim de melhorar a geração de código paralelo automático. Foram introduzidos dois novos atributos na linguagem (Pure e Impure) para aumentar a expressividade e permitir a geração de padrões do paralelismo de dados. Com os novos atributos, definimos novas regras de transformação, que agora geram automaticamente os padrões paralelos Pipeline, Farm, Map, MapReduce, e composições destes. Além disso, implementamos um novo algoritmo para o compilador da SPar e elaboramos estratégias para extrair as informações do código anotado com a SPar. Experimentos foram executados em três *kernels* do NPB. Os resultados mostraram que a nova versão da SPar possui desempenho similar ou até mesmo melhor do que a versão do FastFlow implementada manualmente. Como trabalhos futuros pretendemos estudar a composição de padrões paralelos do paralelismo de *stream* com o paralelismo de dados.

Referências

Aldinucci, M., Danelutto, M., Kilpatrick, P., and Torquati, M. (2017). *Fastflow: High-Level and Efficient Streaming on Multicore*, chapter 13, pages 261–280. John Wiley Sons, Ltd.

del Rio Astorga, D., Dolz, M. F., Fernández, J., and García, J. D. (2017). A generic parallel pattern interface for stream and data processing. *CCPE*, 29(24):e4175.

Griebler, D., Danelutto, M., Torquati, M., and Fernandes, L. G. (2017). SPar: A DSL for High-Level and Productive Stream Parallelism. *PPL*, 27(01):1740005.

Griebler, D., Loff, J., Mencagli, G., Danelutto, M., and Fernandes, L. G. (2018). Efficient NAS Benchmark Kernels with C++ Parallel Programming. In *26th PDP*, PDP'18, pages 733–740, Cambridge, UK. IEEE.

McCool, M., Robison, A., and Reinders, J. (2012). *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier Science.

Thies, W., Karczmarek, M., and Amarasinghe, S. (2002). Streamit: A language for streaming applications. In *Compiler Construction*, pages 179–196. Springer Berlin Heidel.