Geração Automática de Código TBB na SPar

Renato B. Hoffmann[1], Dalvan Griebler[1,2], Luiz G. Fernandes[1]

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

[2]Laboratório de Pesquisas Avançadas para Computação em Nuvem (LARCC)
Faculdade Três de Maio (SETREM), Três de Maio, Brasil.

{renato.hoffmann,dalvan.griebler}@acad.pucrs.br

Resumo. *Técnicas de programação paralela são necessárias para extrair todo o potencial dos processadores de múltiplos núcleos. Para isso, foi criada a SPar, uma linguagem para abstração do paralelismo de stream. Esse trabalho descreve a implementação da geração de código automática para a biblioteca TBB na SPar, uma vez que gerava-se código para FastFlow. Os testes com aplicações resultaram em tempos de execução até* $12,76$ *vezes mais rápidos.*

1. Introdução

Processadores com múltiplos núcleos já são predominantes. Consequentemente, técnicas de programação paralela são necessárias para explorar todo potencial da arquitetura. Principalmente, essas técnicas objetivam fornecer mecanismos ou algoritmos para lidar com a complexidade do paralelismo. Nesse contexto, pode-se destacar o paralelismo de *stream* ou de fluxo de dados. Esse método é comumente empregado em aplicações de vídeo, compressão ou criptografia, que são caracterizadas por uma sequência de estágios ou filtros de processamento. Para esse padrão, foi desenvolvida a SPar [Griebler et al. 2017], que é uma interface entre o programador e aspectos da programação paralela. Seu objetivo é aliviar o programador de preocupações referentes ao paralelismo, permitindo que dirija seus esforços ao desenvolvimento de soluções específicas de sua área. Nomeadamente, a SPar transforma anotações inseridas no código fonte sequencial C++ em binário paralelo automaticamente. Dessa forma, a estrutura do código original é mantida. Para isso, a SPar gera chamadas para uma biblioteca com suporte para paralelismo de *stream*, que nativamente é o FastFlow [Griebler 2016].

TBB (*Threading Building Blocks*) [Reinders 2007] é uma biblioteca C++ para paralelismo em arquiteturas com múltiplos núcleos. Em especial, o TBB trabalha com o conceito de tarefas e não *threads*. A biblioteca fica responsável por mapear tarefas em *threads* de forma eficiente. Dessa forma, o programador fica livre de conceitos como sincronização, balanceamento de carga e otimização de cache. Além do mais, o TBB manipula questões de escalonamento através do método *work-stealing*, que move tarefas de processadores ocupados para processadores livres. Diferentemente, o FastFlow gerado pela SPar utiliza escalonamento estático. Em determinadas situações, o escalonamento *work-stealing* pode amplificar a escalabilidade do programa [Reinders 2007].

Sendo assim, uma das contribuições deste trabalho é habilitar o suporte da biblioteca TBB na SPar. Conceitualmente, a SPar fornece um nível de abstração acima do TBB, uma vez que não necessita de refatoração de código. Para validar a implementação, foram realizados testes na aplicação de compressão Bzip2 e na de vídeo Detecção

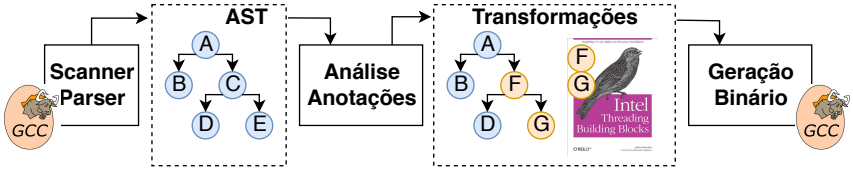


Figura 1. Compilador da SPar

de Pistas [Griebler et al. 2018]. Além do mais, as implementações das duas versões da SPar são idênticas, o que permite o programador utilizar duas soluções diferentes visando obter o melhor desempenho. Na sequência do artigo, a Seção 2 desenvolve a discussão da implementação enquanto que a Seção 3 apresenta e discute os resultados obtidos. Por fim, a Seção 4 descreve brevemente trabalhos similares enquanto que a Seção 5 expõe as considerações finais.

2. Implementação

A SPar utiliza um compilador próprio (CINCLE [Griebler 2016]), desenvolvido para reconhecer sua linguagem e gerar código paralelo. A Figura 1 ilustra o processo de compilação. Ao compilar um programa com a SPar, o GCC é inicialmente chamado para realizar uma análise semântica e de sintaxe do código C++. Depois, o CINCLE faz a análise do código fonte e gera uma AST (Árvore Sintática Abstrata). Essa AST contém as informações de todo o código sequêncial e das anotações da SPar. A partir desse ponto, a SPar realiza uma análise da corretude da sintaxe das anotações. Então, essas anotações são substituídas por chamadas para a biblioteca de paralelismo suportada. Dessa forma, as transformações são realizadas diretamente na AST. O passo final consiste na geração do binário paralelo através da AST, que é posteriormente compilado pelo GCC.

Considerando a implementação SPar com TBB, não são necessárias alterações até a etapa de análise da sintaxe das anotações. Isso porque a sintaxe e semântica das anotações não é alterada. Fundamentalmente, a fase de transformação define as regras adequadas para mapear as anotações em um padrão de paralelismo suportado pela biblioteca base. Por padrão, a SPar utiliza *pipeline*, *farm* ou uma composição desses padrões. A Figura 2 ilustra um exemplo típico de cada uma dessas possibilidades. O *pipeline* é caracterizado por uma sequência de estágios independentes, enquanto que o *farm* consiste em um estágio gerador (G), estágios trabalhadores paralelos (T) um estágio coletor (C) opcional. Já a composição integra um padrão dentro de outro.

Entretanto, o TBB apenas disponibiliza implementação de *pipeline*, o que não é um problema. Isso porquê, por definição, um *farm* é equivalente à um pipeline com pelo menos um estágio replicado. Sendo assim, não foi necessário alterar as regras de transformação. Por outro lado, a sintaxe do TBB exige alterações do comportamento no primeiro estágio da *stream*. Mais especificamente, não é possível utilizar laços de repetição *for* nesse estágio devido à natureza das tarefas do TBB. Isso porque uma tarefa não é fixa à uma *thread* que sempre executa o mesmo código. Uma tarefa do TBB se apega à um elemento da *stream* e o carrega pelo maior número possível de filtros/estágios consecutivos. Sendo assim, a tarefa não executa o campo de expressão do *for*, comumente, o incremento da variável para à iteração seguinte. A solução requer realizar transformações na árvore de modo a substituir o laço *for* por um bloco condicional *if* e reposicionar a inicialização,

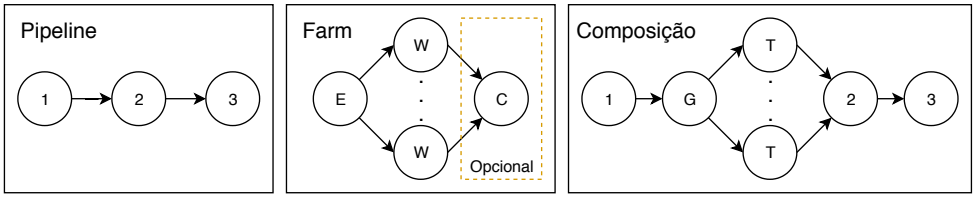
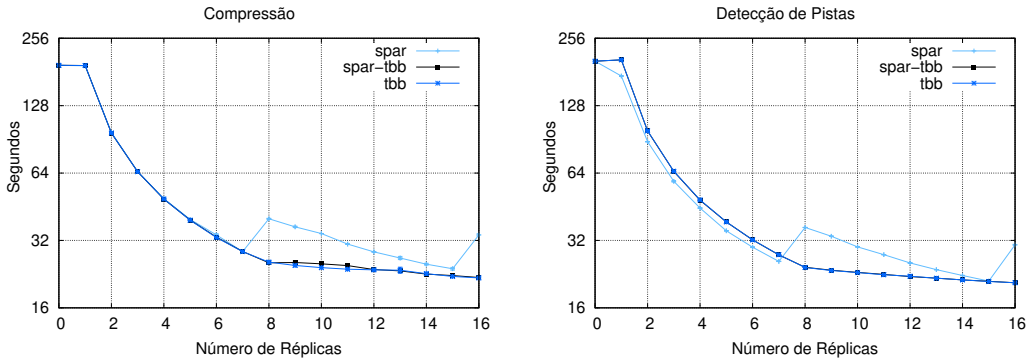


Figura 2. Exemplo Pipeline e Farm

condição e expressão. As alterações restantes do TBB consistem na geração direta das chamadas para o padrão *pipeline* da biblioteca.

3. Experimentos

Essa Seção descreve os resultados obtidos e experimentos realizados. A máquina utilizada possui um processador *Intel(R) Xeon(R) Silver 4108 1.80GHz* e 64GB de memória RAM. O sistema operacional foi *Ubuntu Server 64 bits* (*kernel 4.15.0-74-generic*). As ferramentas utilizadas foram GCC (7.4.0) e biblioteca TBB (4.4 20151115). Além disso, o programa foi compilado utilizando a diretiva de otimização -O3. Os valores foram gerados a partir da média aritmética de 10 execuções. O desvio padrão foi incorporado aos gráficos através de barras de erro. As cargas de trabalho das aplicações Bzip2 e Detecção de Pistas são as mesmas de [Griebler et al. 2018]. As versões paralelas testadas foram SPar original (*spar*), SPar com TBB (*spar-tbb*) e uma implementação manual com TBB (*tbb*).



(a) Compressão Bzip2. (b) Detecção de Pistas.

Figura 3. Resultados dos Tempos de Execução.

Os resultados dos experimentos foram ilustrados na Figura 3. É importante salientar que o número de réplicas 0 é a versão sequencial do programa. Primeiramente, no caso do Bzip2, foi possível observar que a *spar* padrão obteve desempenho similar às outras versões até o número de réplicas 7. Depois desse ponto, a *spar* fica com um resultado um pouco inferior. Nossas investigações indicaram que esse efeito é causado por um desbalanceamento de carga acentuado pelo *hyper-threading*. Isso porquê como o escalonamento é estático, as *threads* posicionadas nos processadores físicos terminam seu processamento mais cedo e ficam ociosas aguardando as *threads* mais lentas dos processadores virtuais. Sendo assim, no caso do Bzip2 paralelo, foi possível observar melhor

desempenho e escalonabilidade geral com o escalonamento *work-stealing* do TBB. Já Detecção de Pistas demonstra uma situação em que o escalonamento estático da *spar* é um pouco superior até o *hyper-threading* e inferior depois. Mesmo assim, o programador pode facilmente escolher a versão mais adequada sem mudanças no código. Além do mais, *spar-tbb* obtém um resultado muito próximo da versão manual *tbb* em todos os casos testados. Esse é um resultado encorajador, já que indica que as abstrações da SPar não tem um peso grande no desempenho dessas aplicações.

4. Trabalhos Relacionados

Em seus estudos, os autores em [del Rio Astorga et al. 2016] desenvolveram uma ferramenta para encapsular padrões de paralelismo fornecidos por bibliotecas base como TBB ou FastFlow. Entretanto, essa estratégia requer a refatoração do código fonte para um *template* padrão pré-determinado. Diferentemente, a estratégia da SPar mantém a estrutura do código original e não requer conhecimento prévio do padrão de paralelismo mais adequado. Já os trabalhos de [Playne and Hawick 2011] geraram automaticamente o padrão *parallel_for* do TBB. Contudo, esse trabalho foi desenvolvido para aplicações matemáticas e permanece não testado em casos de *stream*.

5. Conclusões

Neste trabalho, foram apresentados conceitos básicos sobre as ferramentas de abstração do paralelismo SPar e TBB. Além disso, foi descrito o processo de compilação da SPar e os passos necessários para habilitar o suporte à biblioteca TBB. Por sua vez, essa implementação fui utilizada nos testes realizados nas aplicação Bzip2 e Detecção de Pistas. Mesmo com um nível extra de abstração, foi possível observar que a SPar com TBB atingiu resultado praticamente idêntico à versão original TBB. Como trabalhos futuros, pretende-se estender os testes para outras aplicações do domínio de fluxo de dados.

Referências

del Rio Astorga, D., Dolz, M. F., Sanchez, L. M., Blas, J. G., and García, J. D. (2016). A c++ generic parallel pattern interface for stream processing. In *Algorithms and Architectures for Parallel Processing*, pages 74–87. Springer.

Griebler, D. (2016). *Domain-Specific Language & Support Tool for High-Level Stream Parallelism*. PhD thesis, Faculdade de Informática - PPGCC - PUCRS, Porto Alegre, Brazil.

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

Griebler, D., Hoffmann, R. B., Danelutto, M., and Fernandes, L. G. (2018). Stream Parallelism with Ordered Data Constraints on Multi-Core Systems. *Journal of Supercomputing*, 75(8):4042–4061.

Playne, D. P. and Hawick, K. A. (2011). Auto-generation of parallel finite-differencing code for mpi, tbb and cuda. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops*, pages 1168–1175.

Reinders, J. (2007). *Intel Threading Building Blocks*. O'Reilly, Sebastopol, CA, USA.