Abstraindo o OpenMP no Desenvolvimento de Aplicações de Fluxo de Dados Contínuo

Renato B. Hoffmann[1], **Dalvan Griebler**[1], **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

renato.hoffmann@edu.pucrs.br,{dalvan.griebler,luiz.fernandes}@pucrs.br

Resumo. *OpenMP é complexo quando usado para desenvolver aplicações de fluxo de dados. Com o objetivo de mitigar essa dificuldade, foi utilizada uma metodologia existente, chamada SPar, para aumentar o nível de abstração. Portanto, foram utilizadas anotações mais alto-nível da SPar para gerar código mais baixo-nível de fluxo de dados com OpenMP. Os experimentos revelaram que a SPar teve desempenho 0,86% inferior no caso mais extremo.*

1. Introdução

OpenMP é uma interface de programação paralela consolidada no meio acadêmico e industrial. A principal razão é que a OpenMP possibilita desenvolver aplicações paralelas empregando diretivas de compilação. Dessa forma, as questões complexas do desenvolvimento do paralelismo são resolvidas pela OpenMP. Isso permite ao programador voltar seus esforços ao desenvolvimento de soluções específicas de sua área e ao mesmo tempo aproveitando do paralelismo para melhorar o desempenho de suas aplicações. Entretanto, desenvolver aplicações de fluxo de dados (*stream processing*) com OpenMP é uma tarefa complexa e custosa [Pop and Cohen 2013]. Nesse caso, o programador deve possuir conhecimento de técnicas de paralelismo e buscar mecanismos externos para suprir as necessidades dessas aplicações.

As aplicações de *stream processing* são importantes. Alguns exemplos são filtragem de áudio ou vídeo, monitoramento de sensores, criptografia, análise e estatística, etc. A SPar [Griebler et al. 2017] é uma linguagem específica para expressar o paralelismo pensando nessa classe de aplicações. Seu objetivo é aumentar o nível de abstração utilizando anotações inseridas diretamente no código fonte. Essas anotações são interpretadas e automaticamente transformadas em código de uma interface de programação paralela de mais baixo nível, ou seja, rotinas de sincronização e comunicação. Por padrão, a SPar gera código para a biblioteca de paralelismo FastFlow [Griebler et al. 2017] e TBB [Hoffmann et al. 2020]. Como demonstrado nos experimentos deste artigo, há situações em que o OpenMP pode apresentar menor tempo de execução que o FastFlow. Sendo assim, o objetivo deste trabalho foi expandir o conjunto de definições do compilador da SPar para gerar código de mais baixo nível OpenMP e aumentar o nível de abstração. Dessa forma, facilitando o desenvolvimento de aplicações paralelas de fluxo de dados com OpenMP.

No passado, OmpSS [OmpSs 2020] e OpenStream [Pop and Cohen 2013] propuseram extensões da OpenMP padrão que permitem desenvolver paralelismo de fluxo de dados mais facilmente. Apesar de duas soluções independentes, elas possuem características fundamentais similares. Por exemplo, as duas alternativas utilizam implementações próprias do GCC versão 4 e trabalham com o paradigma de descrição de fluxo de dados. Em relação à solução apresentada neste trabalho, a diferença é que pode-se utilizar qualquer versão mais recente do GCC, facilitando a portabilidade do código. Outra diferença,

é que a solução deste trabalho utiliza o conceito de fluxo de execução, que especifica diretamente os estágios computacionais ao invés de deixar a API decidir quando e como paralelizar um estágio como é o caso dos trabalhos relacionados mencionados. No restante do trabalho, a Seção 2 descreve questões da implementação, a Seção 3 mostra os testes de desempenho realizados e por fim a Seção 4 apresenta as considerações finais.

2. Geração de Código OpenMP para Fluxo de Dados

O desenvolvimento de aplicações de *stream* pode ser realizado utilizando o padrão paralelo *pipeline*, o qual é caracterizado como uma sequência de filtros ou estágios computacionais independentes pelos quais fluem os dados. Cada um desses filtros consome um item do *stream* na entrada e produz um item na saída. Além do mais, filtros sem dependência de dados entre si podem ser replicados para aumentar o grau de paralelismo. Outra questão importante, é que o término do *stream* de entrada é considerado desconhecido e, portanto, não se pode previamente dividi-lo entre as tarefas paralelas como no paradigma de paralelismo de dados. Para atender todos esses critérios, foi desenvolvido o padrão *pipeline* usando o paralelismo de tarefas nativo da OpenMP junto de mecanismos de sincronização padrão do C++ e filas MPMC (Múltiplos Produtores Múltiplos Consumidores).

O *pipeline* estruturado desenvolvido com OpenMP acrescenta código mais baixo nível de paralelismo. Isso adiciona complexidade ao desenvolvimento, derrotando o propósito de simplicidade das diretivas *pragma* usadas pelo OpenMP [de Araujo et al. 2019]. É possível mitigar esse custo e aumentar o nível de abstração do desenvolvimento de aplicações de fluxo de dados usando anotações de código fonte da SPar para gerar *pipeline* mais baixo nível OpenMP. As anotações são padrão do C++ e consistem em colchetes duplos que levam como argumento uma lista de atributos (ex. `[[attr-list]]`). A SPar definiu 5 atributos para representar situações recorrentes em aplicações de fluxo de dados. Na SPar, os atributos podem ser identificadores, que são usados por primeiro na lista de atributos, ou então auxiliares, que podem acompanhar um identificador.

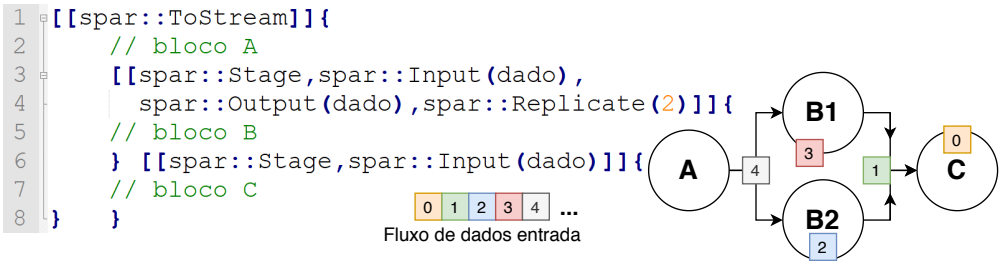


Figura 1. Exemplo do uso de anotações baseado em [Griebler et al. 2017].

A Figura 1 demonstra um exemplo de uso dos atributos da SPar e o *pipeline* que eles modelam. O primeiro atributo identificador anotado na linha 1 é o `ToStream`, que define o escopo da região de *stream* ou fluxo de dados. Além do mais, ele define o estágio sequencial de geração de dados, representado no código pelo bloco A. Já o segundo atributo identificador, anotado nas linhas 3 e 6, é o *Stage*. Ele é usado para definir um estágio computacional dentro da região de *stream* (blocos B e C). O próximo atributo é o auxiliar *Input*, usado nas linhas 3 e 6 para identificar os dados de entrada consumidos pelo estágio ou `ToStream` associado. Por outro lado, os dados produzidos pelo estágio ou `ToStream` associado são especificados pelo atributo auxiliar *Output*. Por fim, *Replicate* é um atributo que paraleliza apenas o *Stage* associado. Esse sistema de anotações abstrai conceitos de criação de tarefas, balanceamento de carga, filas de comunicação e sincronização.

A SPar possui um sistema de compilação próprio que transforma as anotações em código de uma interface de programação paralela mais baixo nível. Nesse trabalho, a interface utilizada é OpenMP estruturado em um *pipeline*. O processo começa com uma análise de sintaxe e semântica que reporta eventuais erros ao programador, tanto padrão do C++, quanto erros no uso dos atributos. Subsequentemente, a SPar realiza transformações do código diretamente na AST (árvore abstrata de sintaxe). Nessa etapa, as anotações são removidas da árvore e no lugar são inseridas as chamadas para OpenMP *pipeline*. Por fim, o binário final é gerado com o GCC do sistema. A vantagem do OpenMP com a SPar ao invés do FastFlow padrão é que permite mais flexibilidade no desenvolvimento.

3. Experimentos

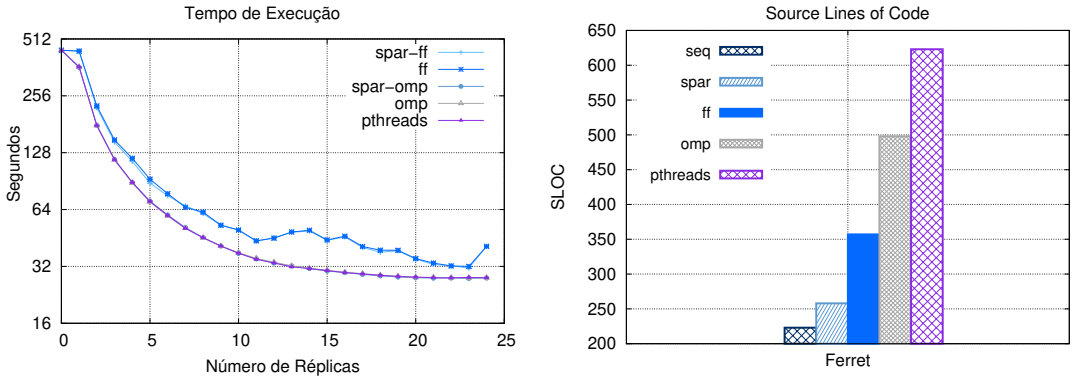
Para avaliar o desempenho da implementação, foi utilizada uma aplicação de fluxo de dados chamada Ferret [Bienia et al. 2008]. Ela é uma aplicação de segmentação e busca de similaridade entre imagens. Como referência de desempenho, foi utilizada a implementação original desenvolvida com Pthreads. Essa implementação possui 6 estágios, dos quais o primeiro e o último são sequenciais de entrada e saída respectivamente. Os 4 estágios do meio são paralelos. Além da versão original Pthreads (`pthreads`), foi desenvolvida uma versão OpenMP (`omp`) e outra com as anotações da SPar gerando OpenMP (`spar-omp`). Também foi demonstrada uma versão codificada manualmente com FastFlow (`ff`) e a SPar padrão (`spar-ff`). Para fins de comparação, todas versões desenvolvem o mesmo algoritmo paralelo do Pthreads. A versão anotada com a SPar pode ser encontrada em [Griebler et al. 2018].

Os experimentos conduzidos mediram o desempenho em tempo de execução e SLOC (linhas de código fonte) das três implementações da aplicação Ferret. O tempo de execução foi medido com a biblioteca padrão C++ `chrono`. Outras três aplicações foram testadas porém com diferenças no desempenho menos expressivas. A máquina utilizada possui dois processadores *Intel(R) Xeon(R) CPU E5-2620 v3 2.40GHz* (12 núcleos físicos e 24 com *hyper-threading*) e 32 GB de RAM. Os dados de entrada foram obtidos do conjunto nativo do Parsec que possui 59695 imagens [Bienia et al. 2008]. Os resultados de desempenho foram obtidos a partir da média aritmética de 10 execuções. O desvio padrão foi representado no gráfico através de barras de erro que são pouco visíveis uma vez que o máximo observado foi $1,01$.

Na Figura 2(a), o tempo de execução foi representado em segundos no eixo y e o número de réplicas de 1 até 24 no eixo x. É importante destacar que o ponto 0 do eixo x representa a versão sequencial do programa. O primeiro aspecto importante é que a escalabilidade da aplicação termina por volta da 12ª réplica. Isso porque a máquina possui apenas 12 núcleos físicos, e a partir disso, se mais Threads forem criadas, o sistema operacional vai usar *hyper-threading*. Em relação ao desempenho, no caso mais extremo, *omp* teve desempenho 1,72% inferior ao *pthreads*. Além do mais, a maior diferença observada entre *omp* e *spar-omp* foi de 0,86%. Por fim, nota-se o ganho de desempenho de até 25,42% ao utilizar a SPar gerando OpenMP no lugar do FastFlow.

O número de linhas do código fonte foi demonstrado no gráfico da Figura 2(b) para as três versões anteriores e também *seq* que representa a versão sequencial. A SPar só possui uma versão já que o código fonte anotado é o mesmo tanto para `spar-omp` quanto para `spar-ff`, o que muda é a geração [Griebler et al. 2018]. A ferramenta *sloccount* foi usada para obter esse número. Com esse resultado, o objetivo foi avaliar o nível de código extra introduzido por cada interface. Apesar da *omp* tradicionalmente ser simples ao desenvolver com diretivas *pragma*, nesse caso obteve o segundo maior SLOC

uma vez que necessita de filas customizadas e mecanismos de sincronização externos. Já o *pthreads* possui um nível de abstração mais baixo com mais opções de customização, o que reflete em um SLOC maior. As anotações da SPar adicionam o menor número de linhas extras em relação à `seq`. Sozinho, SLOC não é suficiente para inferir qual versão é mais produtiva no desenvolvimento. Existem outros fatores como curva de aprendizado, refatoração de código, expressividade da interface, entre outros.



(a) Tempo de Execução.

(b) Linhas de Código Fonte.

Figura 2. Experimentos Aplicação Ferret.

4. Conclusões

Este trabalho apresentou uma solução para abstrair o desenvolvimento de aplicações *stream* com OpenMP. Para isso, foi explicada a ferramenta já existente, chamada SPar, e como que ela transforma anotações do código fonte em código OpenMP no formato de um pipeline. Os experimentos revelaram que, na aplicação Ferret, a SPar gerando OpenMP melhorou o desempenho em até 25.42% na aplicação Ferret enquanto que resulta em um menor número de linhas de código fonte.

Referências

Bienia, C., Kumar, S., Singh, J. P., and Li, K. (2008). The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *17th ICPACT*, pages 72–81, Toronto, Ontario, Canada. ACM.

de Araujo, G. A., Hoffmann, R. B., Griebler, D., and Fernandes, L. G. (2019). Avaliando o Paralelismo de Stream com Pthreads, OpenMP e SPar em Aplicações de Vídeo. In *Escola Regional de Alto Desempenho (ERAD-RS)*, page 4, Três de Maio, BR. Sociedade Brasileira de Computação (SBC).

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). High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2. *International Journal of Parallel Programming*, 47(1):253–271.

Hoffmann, R. B., Griebler, D., Danelutto, M., and Fernandes, L. G. (2020). Stream Parallelism Annotations for Multi-Core Frameworks. In *XXIV Brazilian Symposium on Programming Languages (SBLP)*, SBLP'20, pages 48–55, Natal, Brazil. ACM.

OmpSs (2020). The OmpSs Programming Model. https://pm.bsc.es/ompss.

Pop, A. and Cohen, A. (2013). Openstream: Expressiveness and data-flow compilation of openmp streaming programs. *ACM Trans. Archit. Code Optim.*, 9(4).