Geração de Código OpenMP para o Paralelismo de *Stream*

Renato B. Hoffmann, Dalvan Griebler, Luiz Gustavo Fernandes

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

(renato.hoffmann,dalvan.griebler)@edu.pucrs.br

Abstract. *OpenMP is an industry and academy standard for parallel programming that is complex when using to develop parallel stream processing applications. To tackle this problem, we proposed to use a high-level parallel programming model (named SPar) for generating low-level structured stream processing OpenMP code. We achieved this by employing SPar set of source code annotations to simplify the complexity and verbosity added by OpenMP. The experiments in 4 different stream processing applications demonstrated that the execution time was improved by up to 25.42%. Furthermore, SPar significantly reduced the total source lines of code required to express parallelism.*

Resumo. *OpenMP é uma interface para a programação paralela padrão e amplamente usada na indústria e academia, porém, torna-se complexa quando usada para desenvolver aplicações paralelas de fluxo de dados ou stream. Para resolver esse problema, foi proposto usar uma interface de programação paralela de alto nível (chamada SPar) e seu compilador para a geração de código estruturado de mais baixo nível com OpenMP em aplicações de fluxo de dados. O objetivo é diminuir a complexidade e verbosidade introduzida pelo OpenMP nas aplicações de stream. Nos experimentos em 4 aplicações, notou-se uma redução no tempo de execução de até 25,42%. Além do mais, requer-se um número de linhas de código fonte menor para expressar o paralelismo.*

1. Introdução

A programação paralela não é uma tarefa trivial. Os programadores devem lidar com conceitos como criação e gerenciamento de *theads*, otimização de cache, mecanismos de comunicação, balanceamento de carga, dependência de dados, sincronização, acesso crítico ou mutualmente exclusivo, etc. Mesmo assim, desenvolvedores de aplicações deveriam ser capazes de facilmente paralelizar seus programas. Portanto, as abstrações de paralelismo são essenciais para permitir que o desenvolvedor destine a maior parte dos seus esforços à sua área de trabalho específica e não às complexidades do paralelismo.

Quando abordando abstrações de paralelismo, pode-se optar por uma estratégia estruturada ou não-estruturada. A alternativa estruturada fornece aos desenvolvedores padrões ou estruturas previamente definidas (ex. *map*, *reduce*, *pipeline*). Já as não-estruturadas permitem ao programador criar código de maneira mais flexível, desde que respeitando a sintaxe e semântica da interface. Exemplos estruturados são Intel TBB (*Threading Building Blocks*) [Reinders 2007] e FastFlow [Aldinucci et al. 2014] e não-estruturados OpenMP [OpenMP 2020] e Pthreads [Jacqueline Farrell 1996]. Cada IPP (Interface de Programação Paralela) possui um nível diferente de complexidade, expressividade e limitações. Por exemplo, o Pthreads é mais complexo que o OpenMP mas é

mais flexível nas opções de programação. Além do mais, diferentes IPPs podem desempenhar melhor com diferentes características computacionais já que foram desenvolvidas com diferentes objetivos e técnicas de implementação.

As aplicações de fluxo de dados [Thies et al. 2002] são geralmente computacionalmente intensivas e necessitam de paralelismo para obter resultados em tempo hábil. Essa classe de aplicações é caracterizada por um fluxo de dados executados por uma sequência de estágios ou filtros de computação independentes. Alguns exemplos comuns são manipulação de áudio ou vídeo, monitoramento de sensores, criptografia, análise e estatística, etc. Referente ao paralelismo dessas aplicações, uma estrutura de *pipeline* linear ou não-linear (linear com estágios replicados) é suficiente para representar situações comuns [Aldinucci et al. 2014].

OpenMP não pode ser facilmente desenvolvido em aplicações de fluxo de dados ou *stream*. Isso também foi notado em [Pop and Cohen 2013, Griebler 2016]. Portanto, mecanismos externos devem ser buscados para desenvolver esse domínio de aplicações de forma correta e eficiente. Esse esforço extra é contra o propósito de simplicidade das diretivas de compilação `pragma` usadas pelo OpenMP. Portanto, o objetivo deste trabalho foi providenciar uma metodologia para simplificar o desenvolvimento de aplicações de fluxo de dados com OpenMP. Para isso, foi utilizada a SPar[1] [Griebler et al. 2017], que é uma DSL desenvolvida com foco na abstração do desenvolvimento de aplicações de fluxo de dados. A SPar se beneficia do mecanismo de anotações padrão do C++ inseridas diretamente no código fonte. Ela usa sua própria infraestrutura de compilador para automaticamente transformar anotações em chamadas para uma IPP estruturada mais baixo nível. Tudo isso é feito diretamente na AST (Árvore Abstrata de Sintaxe) e não necessita de refatoração do código fonte por parte do programador nem conhecimento prévio dos padrões paralelos. Em resumo, o objetivo é gerar código OpenMP eficiente para aplicações de fluxo de dados mantendo a simplicidade da sintaxe e semântica original das anotações da SPar. As contribuições desta pesquisa são listadas a seguir:

- Implementação eficiente de *pipeline* linear e não-linear com OpenMP para aplicações de fluxo de dados.
- Implementação de 4 aplicações de fluxo de dados reais com OpenMP.
- Definições e regras de transformação para geração automática de código paralelo OpenMP através das anotações da SPar.
- Avaliação do desempenho e programabilidade das soluções propostas.

O restante do artigo foi organizado como se segue. A Seção 2 caracteriza o trabalho dentro do grupo de pesquisa. Na sequência, a Seção 3 explica o processo de geração de código para aplicações de fluxo de dados com OpenMP. Por fim, a Seção 5 delibera sobre os trabalhos relacionados e a Seção 6 apresenta as considerações finais.

2. Caracterização do Trabalho Dentro do Grupo de Pesquisa

O trabalho proposto neste artigo é parte do projeto da SPar [Griebler 2016] dentro do grupo de pesquisa GMAP. A SPar usa o CINCLE (infraestrutura de compilador para criação de DSLs) para implementar a geração de código de uma interface paralela mais baixo-nível. Essa interface é responsável pelo paralelismo em si.

[1] https://gmap.pucrs.br/spar

Buscando adicionar novas interfaces paralelas na SPar, outros trabalhos no grupo já foram conduzidos com *clusters* (DSParLib [Pieper 2020]) e GPU (GSParLib [Rockenbach 2020]). No contexto multi-núcleo, o padrão da SPar é o FastFlow [Griebler et al. 2017], onde estudos recentes abordaram a adaptatividade automática do número de réplicas paralelas [Vogel et al. 2020], objetivos de nível de serviço [Griebler et al. 2019, Griebler et al. 2018b] e anotações para paralelismo de dados [Löff 2020]. Entretanto, em um trabalho anterior, também foi desenvolvido o suporte para TBB [Hoffmann et al. 2020]. Em relação ao projeto de pesquisa, esse trabalho adiciona o OpenMP como uma interface de exploração de paralelismo da SPar para arquiteturas multi-núcleo.

3. Geração do Código OpenMP com a SPar

Essa Seção explica os mecanismos utilizados na geração de código para aplicações de fluxo de dados com OpenMP. Portanto, primeiro foram explicadas as dificuldades e requerimentos para o desenvolvimento de um *pipeline* estruturado OpenMP na Seção3.1. A partir dai, a Seção 3.2 desenvolve o tema da geração de código do pipeline OpenMP utilizando a SPar.

3.1. Esqueleto de Código OpenMP para Expressar o Fluxo de Dados Paralelo

O OpenMP padrão não consegue lidar bem com processamento de *stream* [Pop and Cohen 2013]. No caso das aplicações de *stream*, o OpenMP pode ser usado, mas requer dois aspectos essenciais que não são diretamente disponibilizados. O primeiro é comunicação MPMC (Múltiplos Produtores Múltiplos Consumidores). OpenMP já possui a diretiva *depend* para expressar relações produtor-consumidor. Entretanto, ela não suporta ciclos iterativos de comunicações e ainda não implementa mecanismos de sincronização. Nesse caso, não há alternativa, um mecanismo externo de comunicação deve ser implementado ou importado. O segundo aspecto essencial é a necessidade de sincronização eficiente entre as *threads* ligadas no canal de comunicação MPMC, que deve ser obtida de meios externos. Outra questão necessária no desenvolvimento de aplicações de fluxo de dados é o paralelismo de tarefas. Nesse caso, OpenMP já fornece abstrações adequadas, nomeadamente as diretivas *section* e *task*. Tendo isso em mente, a estrutura de *pipeline* atende as seguintes especificações:

- **Múltiplos Produtores Múltiplos Consumidores:** Cada canal de comunicação deve ser capaz de gerenciar qualquer número positivo maior ou igual a 0 de produtores ou consumidores.
- **Acesso mutualmente exclusivo dos dados:** Como pode haver múltiplas *threads* produzindo ou consumindo dados, acesso mutualmente exclusivo dos dados é necessário para evitar condições de corrida. Para isso, um único `mutex` padrão do C++ foi usado por canal de comunicação. Os `locks` padrão do OpenMP não podem ser usados já que não permitem a sincronização de produtores e consumidores sem duplicar a operação, o que limitaria a escalabilidade.
- **Final de processamento:** Na eventualidade do final de processamento da *stream*, deve ser possível propagar um sinal de terminação para todos os produtores e consumidor encadeados. Tarefas ociosas devem ser acordadas e terminadas. Além do mais, esse processo deve garantir que nenhum item fique não processado no meio do fluxo de execução.

- **Vazão indeterminada**: Deve-se considerar vazão indeterminada, ou seja, os dados de entrada podem ser caracterizados por momentos de pico ou escassez. Um canal cheio ou um gargalo pode acontecer quando a vazão é maior do que a capacidade de processamento do pipeline. Um canal vazio pode ocorrer quando a vazão é menor do que capacidade de processamento do *pipeline*, o que significa que pode-se escalar o desempenho aumentado o número de réplicas dos estágios.
- **Prioridade de acesso balanceada:** Produtores e consumidores devem ter igual prioridade de acesso ao canal. Dessa forma, operações de inserção ou remoção ocorrem de forma mais balanceada possível.
- **Espera não-ocupada:** Consumidores buscando dados em um canal de comunicação vazio não devem esperar de forma ocupada. Eles devem parar sua execução e acordar apenas quando houver itens disponíveis para consumir. Assim, o núcleo de processamento é liberado para outra potencial tarefa. Da mesma forma, produtores buscando inserir dados em um canal de comunicação cheio não devem esperar de forma bloqueante. Eles devem parar sua execução e acordar apenas quando houver espaço para inserir novos dados. Para isso, foram utilizadas duas condições de espera padrão do C++.
- **Tamanho limitado do canal:** O canal de comunicação deve armazenar um número máximo de elementos para evitar sobrecarregar a memória. Isso deve ser customizável já que um tamanho muito pequeno pode gerar um gargalo e um tamanho muito grande pode desperdiçar recursos de memória.
- **FIFO:** Canal de comunicação *First-in-first-out* (Primeiro a Entrar Primeiro a Sair).
- **Reordenamento:** Devido à natureza não determinística do processamento paralelo, dados podem chegar em determinados estágios de processamento em uma ordem diferente da qual eles foram gerados. Isso é um problema para algumas aplicações em que a integridade dos dados de saída depende da ordem dos itens (ex. ordem dos quadros em um vídeo). Por essa razão, a implementação deve acomodar um algoritmo de reordenamento nos estágios sequenciais quando necessário. O algoritmo de reordenamento utilizado foi [Griebler et al. 2018a].

3.2. Simplificado o Paralelismo com a SPar

Como previamente explicado, o OpenMP apresenta dificuldades quando usado no desenvolvimento de aplicações de fluxo de dados. De fato, o pipeline estruturado desenvolvido de acordo com a Seção 3 requer um grande esforço de programação. Tendo em vista essa dificuldade, a SPar pode ser usada para mitigar esse problema. A SPar é uma DSL (Linguagem Específica de Domínio) que abstrai uma interface de programação paralela estruturada mais baixo-nível. No caso desse trabalho, o *pipeline* estruturado OpenMP. Para isso, a SPar utiliza anotações inseridas pelo programador diretamente no código fonte sem alterar a estrutura sequencial original. Essas anotações contém uma lista de atributos. Focando na simplicidade, a SPar disponibiliza apenas 5 atributos, que, juntos, são capazes de descrever situações corriqueiras das aplicações de *stream*. Para os fins deste trabalho, o sistema de anotações da SPar não foi alterado. Um exemplo de anotações da SPar foi demonstrado na parte direita da Figura 1. Mais detalhes sobre técnicas de implementação das anotações podem ser encontrados em [Griebler 2016].

A continuação do fluxo da metodologia foi ilustrada na parte esquerda da Figura 1. A partir das anotações, a SPar usa o suporte do CINCLE (Infraestrutura de Compilador

para Novas Extensões de Linguagem C/C++) [Griebler 2016] para realizar transformações de código diretamente na AST (Árvore Abstrata de Sintaxe). Efetivamente, a SPar substitui os nodos da AST que contém as anotações pelo código estruturado da interface abstraída. Isso acontece em três etapas. Primeiro, salva-se os blocos de código associados às anotações. Depois, o compilador monta uma sub-árvore com o padrão estruturado usando os blocos de código extraídos como estágios de processamento. No terceiro passo o compilador remove da AST original toda sub-árvore contendo as anotações e substitui pela nova sub-árvore. Depois disso, a AST resultante é recompilada para gerar o binário paralelo final.

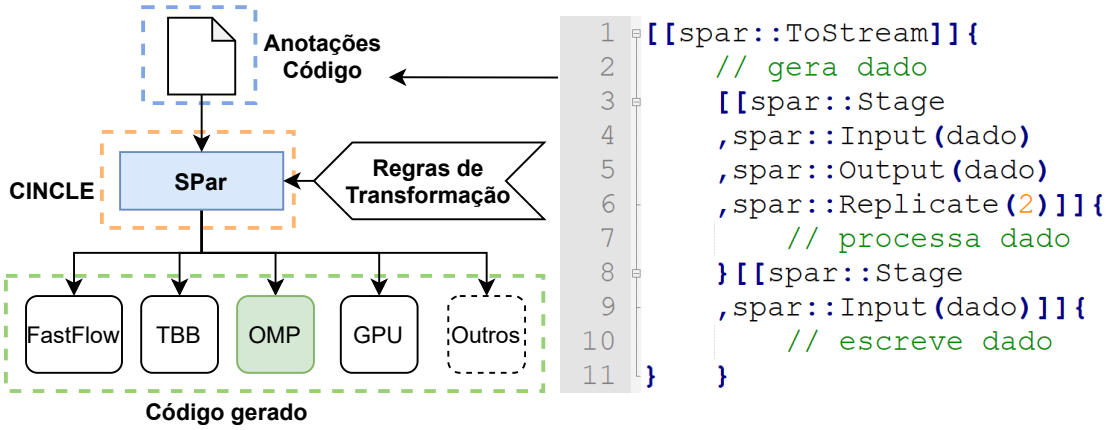


Figura 1. Metodologia de abstração com SPar.

O processo de transformação é realizado de acordo com um conjunto de regras aplicadas durante o período de compilação. Cada interface tem o seu conjunto próprio de regras. Por padrão, a SPar gera código FastFlow e, no passado, foi adicionado na SPar suporte para geração de código TBB [Hoffmann et al. 2020]. Neste trabalho, foi adicionado o suporte à geração de código OpenMP estruturado em *pipeline*. Isso implica na necessidade de um conjunto novo de regras de transformação. Essas regras são uma forma de indicar ao compilador como montar a estrutura paralela correta à partir das anotações. Em resumo, elas indicam quantos estágios possui o *pipeline*, quais deles são paralelos e como montar os canais de comunicação entre os estágios. O detalhamento completo das regras pode ser encontrado em [Hoffmann 2020]. Mesmo assim, independentemente da interface abstraída, o sistema de anotações da SPar é o mesmo. Assim, a SPar permite gerar código para qualquer interface suportada simplesmente trocando uma *flag* de compilação.

Uma pergunta pertinente é sobre a necessidade de adicionar suporte OpenMP considerando que a SPar já gera código para outras interfaces. Nesse caso, além da popularidade da interface OpenMP, o fator motivador é que em alguns casos ele pode desempenhar melhor. Isso será demonstrado a seguir na Seção 4.

4. Experimentos

O objetivo desta Seção foi avaliar o desempenho do *pipeline* estruturado do OpenMP assim como sua geração com a SPar. Para isso, foi utilizado um conjunto de quatro aplicações: Detecção de Pistas [Griebler et al. 2018a], Reconhecimento Facial [Griebler et al. 2018a], Bzip2 [Gilchrist 2004], Ferret [Bienia et al. 2008] previamente anotadas com SPar. Os experimentos conduzidos mediram o desempenho em

tempo de execução e programabilidade com a métrica SLOC (linhas de código fonte). O tempo de execução foi medido com a biblioteca padrão C++ `chrono` e SLOC com a ferramenta Sloccount. Em relação aos gráficos resultantes dos experimentos, o desvio padrão foi representado através de barras de erro. Além do mais, todos os valores foram obtidos através da média aritmética de 5 execuções. A corretude foi garantida comparando o valor *hash* de saída das versões paralelas com o sequencial. A máquina utilizada para os testes 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. Outros detalhes de *software* são: sistema operacional Ubuntu Server 64 bits *kernel 4.15.0-88-generic*, GCC 7.4.0, Opencv 2.4.13.6, TBB 2017 (interface version 9107), e FastFlow 3.0. Além do mais, foi usada a otimização de compilação -O3. Na sequência, a Seção 4.1 demonstra os resultados de desempenho e a Seção 4.2 avalia aspectos de programabilidade.

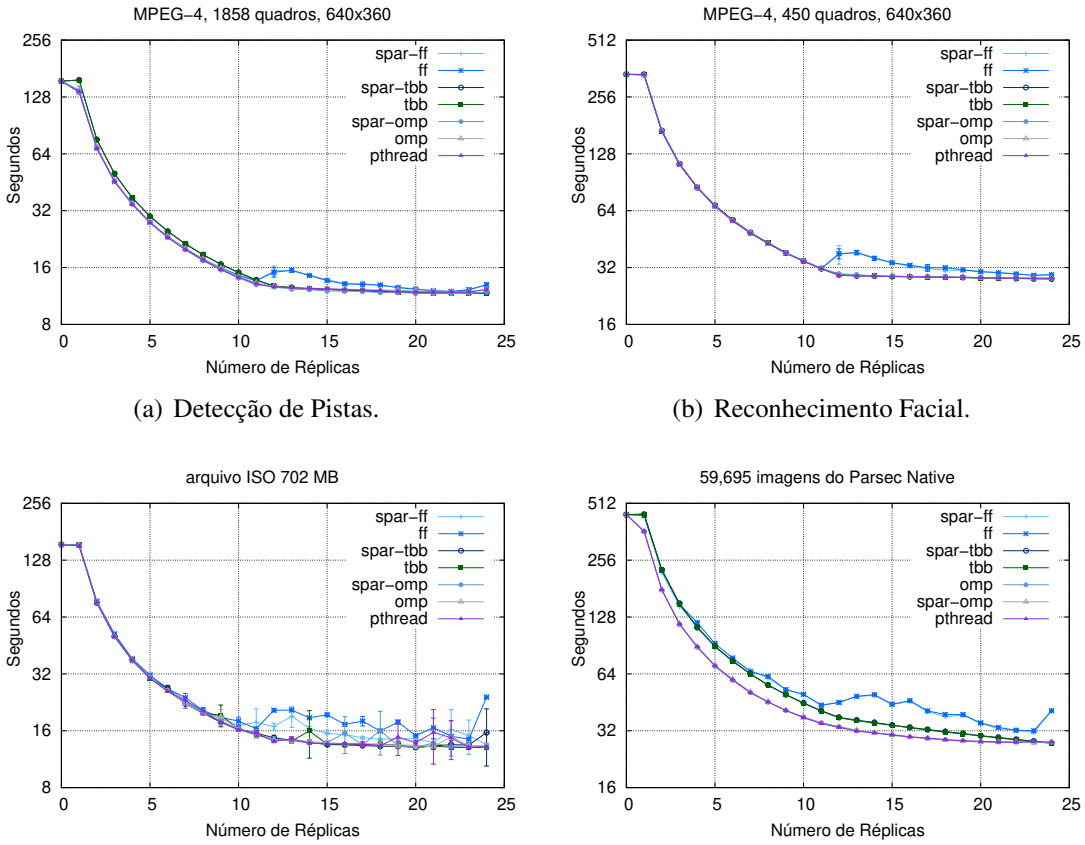
4.1. Desempenho

Em todos os gráficos na Figura 2, o tempo de execução foi representado em segundos no eixo y e o número de réplicas de 1 até 24 no eixo x. Ainda, o ponto 0 do eixo x representa a versão sequencial do programa. Também é importante destacar que o número de réplicas no eixo X não necessariamente indica o número de *threads* executadas no sistema. Em vez disso, cada interface pode escalonar um conjunto de *threads* para cada estágio com tamanho determinado pelo número de réplicas. Além do mais, todos os gráficos foram representados em escala logarítmica com base 2. Por fim, as especificações dos dados de entrada podem ser encontrados na parte superior de cada gráfico.

As versões paralelas demonstradas nos gráficos incluem OpenMP (`omp`), TBB (`tbb`), FastFlow (`ff`), Pthreads (`pthread`) e as versões geradas pela SPar (`spar-ff`, `spar-tbb`, `spar-omp`). Assim, é possível avaliar o impacto das abstrações da SPar comparando a versão manual das implementações em relação à geração de código da SPar. Além disso, o Pthreads representa a implementação base do estado-da-arte.

Os resultados de Detecção de Pista na Figura 2(a) demonstram que a aplicação escala somente até a 12ª réplica. De fato, essa característica ocorre em todas aplicações. Isso porquê a máquina usada possui apenas 12 núcleos físicos. Ainda avaliando Detecção de Pistas, nota-se que as versões `ff` têm uma piora no desempenho após a 11ª réplica. Os estudos conduzidos indicaram que isso acontece por conta de um desbalanceamento de carga no escalonamento estático do FastFlow. Outra questão é que as versões *omp* desempenharam no máximo 0,64% inferiores em relação ao `pthread`, indicando que são comparáveis ao estado-da-arte. Já a aplicação Reconhecimento Facial na Figura 2(b) apresenta resultados similares à Detecção de Pistas. Esse é um dado interessante, uma vez que demonstra a consistência das implementações mesmo sob vazão drasticamente diferente. No caso, Detecção de Pistas tem vazão máxima observada de 157 quadros por segundo e Reconhecimento Facial de 16 quadros por segundo. Outro fator observado foi que, em ambas aplicações, a SPar ficou no máximo 1,77% atrás da versão equivalente codificada manualmente, incluindo *spar-omp* e *omp*.

Os resultados do Bzip2 demonstrados na Figura 2(c) apresentam alto desvio padrão generalizado. No geral, os resultados são similares, porém há instâncias que se diferenciam em até 10%. Esse comportamento também foi observado em [Hoffmann et al. 2020]. Uma possível explicação é a variabilidade de acesso ao disco



(a) Detecção de Pistas. (b) Reconhecimento Facial.

(c) Bzip2 Compressão. (d) Ferret.

Figura 2. Resultados de experimentos.

ou a própria arquitetura onde os testes foram executados. Resultados em uma máquina diferente demonstraram desvio padrão negligenciável com as mesmas configurações de *software* e dados de entrada. Já *spar-omp* e *omp* se diferenciam em até 1,24%.

Por fim, a aplicação Ferret na Figura 2(d) demonstrou uma maior disparidade entre as diferentes versões. As melhores versões foram `pthread` e `omp|spar-omp` com diferença máxima observada de 1,72%. O OpenMP desempenha melhor que o TBB, que por sua vez desempenha melhor que o FastFlow. Trazendo esse resultado para números, na 12ª réplica, TBB é 10,1% inferior do que o OpenMP, FastFlow é 17,04% pior do que o TBB e o FastFlow é 25,42% pior do que OpenMP. Novamente, todas as versões da SPar desempenham muito semelhantemente em relação às versões codificadas manualmente, com a maior diferença observada sendo de 2,49%.

4.2. Programabilidade

A Tabela 1 demonstra os valores de SLOC (linhas de código fonte) para cada versão, incluindo a sequencial `seq`. A `spar` só apresenta uma versão já que `spar-ff`,`spar-tbb` e `spar-omp` usam as mesmas anotações de código fonte. Essa métrica não contabiliza comentários ou linhas em branco. Referente a programabilidade, sozinho, SLOC não é expressivo o suficiente para concluir qual versão é mais produtiva. Em vez disso, o número de SLOC é uma indicação do nível de intrusão de código

resultante de cada interface de programação paralela.

Tabela 1. Número total de linhas de código fonte

Versão	Ferret	Bzip2	Detecção de Pistas	Reconhecimento de Pistas
seq	223	1278	126	126
omp	498	1695	234	292
spar	258	1404	130	132
tbb	376	1483	170	175
ff	357	1607	164	174
pthread	623	1917	415	417

Como esperado, `pthread` apresentou o SLOC mais alto já que é possui alta complexidade no desenvolvimento e menor nível de abstrações de paralelismo. Logo depois vem `omp`, com os segundo maior SLOC. Nesse caso, é possível observar que apesar da simplicidade tradicional das diretivas *pragma*, as aplicações de fluxo de dados com OpenMP demandam um grande número extra de linhas de código fonte. Ao colocar esses resultados juntos dos valores de desempenho apresentados na Seção 4.1, pode-se notar que essas versões com maior SLOC são as que desempenham de forma mais consistente e eficiente. Isso porquê elas providenciam mais opções de customização que podem ser aproveitadas por um desenvolver experiente. Já `tbb` e `ff` ficam no meio do caminho, explicado pelo fato de buscarem um balanço entre customização e abstração. Entre todas as aplicações, `spar` atingiu o menor SLOC extra em relação ao `seq`. Ela é a interface mais com o maior nível de abstração, permitindo reduzir o número de linhas de código fonte extra ao deixar o compilador lidar com os aspectos complexos da implementação. Ainda assim, o desempenho é comparável a implementação manual.

5. Trabalhos Relacionados

As dificuldades trazidas pelo desenvolvimento de aplicações de fluxo de dados com OpenMP já foram notadas por outros pesquisadores [Pop and Cohen 2013, Griebler 2016]. Visando solucionar esse problema, o modelo de programação ompSS [OmpSs 2020] propôs novas diretivas *pragma* capazes de expressar paralelismo de fluxo de dados em ambientes multi-núcleo, aceleradores GPU e FPGA. Utilizando outra abordagem, OpenStream [Pop and Cohen 2013] optou por usar extensões aos *pragmas* já existentes no OpenMP enquanto que ompSS introduz diretivas com novas semânticas. Eles até mesmo reportaram um desempenho superior ao ompSS. Isso porquê o modelo de OpenStream utiliza tarefas persistentes enquanto que ompSS assume tarefas leves combinadas com escalonamento de roubo de tarefas.

Tanto ompSS quanto OpenStream compartilham duas características fundamentais. A primeira é que ambos utilizam versões fixas e antigas do OpenMP combinadas a um protótipo experimental de uma versão fixa e antiga do GCC. Isso significa que desenvolvedores que desejarem utilizar uma dessas soluções são forçados a desenvolver com OpenMP 3 e GCC 4. Por outro lado, a solução apresentada neste trabalho permite ao desenvolvedor utilizar, mesmo que sem desenvolver diretamente, qualquer GCC ou OpenMP disponível em seu sistema. Portanto, pode-se fazer melhor uso das otimizações mais recentes liberadas pelos mantenedores dessas duas ferramentas. A segunda característica é que OpenStream e ompSS trabalham com um conceito diferente de paralelismo

de fluxo de dados. No caso deles, o programador específica um fluxo de dados representado por um grafo acíclico direcionado enquanto que a interface adapta quando e onde paralelizar um determinado código. Por outro lado, este trabalho considera um padrão estruturado que especifica um fluxo de execução dos dados. Efetivamente, isso significa que o programador vai ter a certeza de que um estágio marcado como paralelo será executado de forma concorrente.

No campo da programação estruturada, FastFlow [Aldinucci et al. 2014] e TBB [Reinders 2007] são bibliotecas C++ que providenciam padrões paralelos prontos para uso (ex: *Map*, *Farm* e *Pipeline*). O programador necessita apenas conhecer as implicações de cada padrão e a semântica e sintaxe da biblioteca. GrPPI [del Rio Astorga et al. 2016] é uma IPP que envolopa outras soluções como FastFlow, TBB e até mesmo OpenMP. GrPPI providencia padrões na forma de *templates* C++, o que significa que o programador deve refatorar seu código na interface providenciada. Apesar de não focar em aplicações de fluxo de dados, GrPPI providencia um pipeline. Comparativamente, a alternativa da SPar permite ao programador manter a estrutura original do código sequencial e também abstrai conceitos de caracterização da implementação dos padrões. Mais parecido com a SPar, StreamIT [Thies et al. 2002] é uma linguagem de domínio para paralelismo de fluxo de dados. A diferença fundamental é que ele requer do programador conhecimento de uma nova e restritiva semântica de linguagem para expressar o fluxo de execução.

6. Conclusões

Este trabalho apresentou uma alternativa para facilitar o paralelismo de fluxo de dados/*stream* com OpenMP através de geração automática de código. Para isso, foi usada a SPar, uma Linguagem Específica de Domínio que trabalha com anotações de código fonte para abstrair os detalhes de implementação do OpenMP. Além do mais, a metodologia de geração de código com a SPar foi explicada. Nesse caso, as anotações de código fonte são transformadas em chamadas para um esqueleto *pipeline* estruturado com OpenMP para aplicações de *stream*. Na sequência, foram realizados experimentos com quatro robustas aplicações de fluxo de dados comparando as soluções desenvolvidas. Foi observado que a implementação OpenMP ficou no máximo 1,72% inferior em relação estado-da-arte com Pthreads. Além do mais, o nível extra de abstração da SPar impactou no máximo 2,49% enquanto que adicionou o menor número de linhas de código fonte extra. Com isso, conclui-se que este trabalho de pesquisa contribuiu para a facilitar o desenvolvimento das aplicações de fluxo de dados com OpenMP usando a metodologia de abstração com a SPar. Para garantir a escalabilidade da solução no futuro, um trabalho futuro pertinente consiste na avaliação da utilização de canais de comunicação no *pipeline* OpenMP com operações de exclusão mútua relaxadas.

Agradecimentos

Nós agradecemos pelos recursos computacionais do LAD-PUCRS e do GMAP. Essa pesquisa é parcialmente financiada pela FAPERGS 05/2019-PQG projeto PARAS (Nº 19/2551-0001895-9), Universal MCTIC/CNPq Nº 28/2018 projeto SPARCLOUD (Nº 437693/2018-0), e MCTIC/CNPq call 25/2020 (Nº 130484/2021-0)

Referências

Aldinucci, M., Danelutto, M., Kilpatrick, P., and Torquati, M. (2014). FastFlow: High-Level and Efficient Streaming on Multi-core. In *Programming Multi-core and Many-core Computing Systems*, volume 1 of *PDC*, page 14. Wiley.

Bienia, C., Kumar, S., Singh, J. P., and Li, K. (2008). The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, Toronto. ACM.

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 International Publishing.

Gilchrist, J. (2004). Parallel Compression with BZIP2. In *16th IASTED ICPDCS*, PDCS' 04, pages 559–564, MIT, Cambridge, USA. ACTA Press.

Griebler, D. (2016). *Domain-Specific Language & Support Tool for High-Level Stream Parallelism*. PhD thesis, 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).

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

Griebler, D., Sensi, D. D., Vogel, A., Danelutto, M., and Fernandes, L. G. (2018b). Service Level Objectives via C++11 Attributes. In *Euro-Par 2018: Parallel Processing Workshops*, Lecture Notes in Computer Science, pages 745–756, Turin, Italy. Springer.

Griebler, D., Vogel, A., Sensi, D. D., Danelutto, M., and Fernandes, L. G. (2019). Simplifying and implementing service level objectives for stream parallelism. *Journal of Supercomputing*, 76:4603–4628.

Hoffmann, R. B. (2020). Stream Parallelism Annotations for Automatic OpenMP Code Generation. Technical report, School of Technology - PUCRS, Porto Alegre, Brazil.

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.

Jacqueline Farrell, Dick Buttlar, B. N. (1996). *PThreads Programming*. O'Reilly, Sebastopol, CA, USA.

Löff, J. H. (2020). Aumentando a Expressividade e Melhorando a Geração de Código Paralelo para o Paradigma de Paralelismo de Stream em Arquiteturas Multi-core. Technical report, School of Technology - PPGCC - PUCRS, Porto Alegre, Brazil.

OmpSs (2020). The ompss programming model.

OpenMP (2020). Open multi-processing api specification for parallel programming.

Pieper, R. L. (2020). High-level Programming Abstractions for Distributed Stream Processing. Master's thesis, PPGCC - PUCRS, Porto Alegre, Brazil.

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

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

Rockenbach, D. A. (2020). High-Level Programming Abstractions for Stream Parallelism on GPUs. Master's thesis, PPGCC - PUCRS, Porto Alegre, Brazil.

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

Vogel, A., Griebler, D., and F, L. G. (2020). Providing High-level Self-adaptive Abstractions for Stream Parallelism on Multi-cores. *Software: Practice and Experience*.