Avaliando a Produtividade e o Desempenho da DSL SPar em uma Aplicação de Detecção de Pistas

Renato B. H. Filho, Dalvan Griebler, Cleverson Ledur, Luiz G. Fernandes

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

{renato.hoffmann,dalvan.griebler,cleverson.ledur}@acad.pucrs.br
luiz.fernandes@pucrs.br

Resumo. *A linguagem de domínio específico SPar, embarcada na linguagem C++, fornece através de anotações uma alternativa para explorar o paralelismo de stream em arquiteturas multi-núcleo. Neste artigo, o objetivo é demonstrar indicadores de desempenho e produtividade em uma aplicação de detecção de pistas. Os resultados comprovaram que a SPar apresentou maior produtividade e bom desempenho.*

1. Introdução

Aplicações baseadas em *stream* se tornam cada vez mais comuns no mundo de hoje, como por exemplo, no processamento de imagem, áudio, vídeo, e dados em geral. Como elas são tradicionalmente intensivas, computacionalmente demandam de paralelismo para realizarem suas operações. Uma destas é a aplicação Detecção de Linhas, capaz de reconhecer e escrever padrões lineares sobre o *frame*, que se trata de uma imagem de rodovia. O paralelismo é fundamental para essa aplicação, considerando que o processamento deve ser em tempo real e uma série de filtros são aplicados em cada imagem.

Atualmente, existem diferentes interfaces de programação paralela (bibliotecas/*frameworks*) capazes de explorar o paralelismo de *stream*. Podemos destacar como as mais conhecidas o FastFlow [Aldinucci et al. 2014], e o TBB (*Threading Building Blocks*) [Reinders 2007]. O FastFlow é uma biblioteca C++ que oferece uma interface para implementar diferentes padrões paralelos (ex. *farm*, *pipeline*, *map*, e *reduce*). O TBB é também uma biblioteca em C++ e oferece interface para lidar com padrões como *parallel_for*, *pipeline* e entre outros. Além da interface, ambos tem um paradigma de execução das *threads* bem diferentes. Enquanto que o TBB possui uma única fila que opera no modelo *work-stealing*, o FastFlow opera com filas não bloqueante na comunicação.

Mais recentemente, foi criada uma linguagem específica de domínio (DSL) para simplificar o paralelismo de *stream* em arquiteturas *multi-core*, a qual foi nomeada SPar [Griebler et al. 2015, Griebler 2016]. O principal objetivo dela é fornecer um nível de abstração mais alto para desenvolvedores manejarem a exploração de paralelismo, que não é muito produtivo usando as soluções existentes. Ainda, visa manter o desempenho sem significativa perda de performance em relação ao TBB e FastFlow. Assim, o objetivo deste artigo é avaliar, através da aplicação Detecção de Linhas, a produtividade e o desempenho da SPar e comparar com FastFlow e TBB. O trabalho está organizado da seguinte forma: Seção 2 apresenta a SPar, Seção 3 descreve como a aplicação foi paralelizada com a SPar, Seção 4 discute os resultados e Seção 5 conclui o trabalho.

2. SPar: uma DSL para o Paralelismo de Stream

A SPar é uma DSL embarcada na linguagem C++, capaz de modelar aplicações baseadas em paralelismo de *stream* [Griebler et al. 2015, Griebler 2016]. Ela foi implementada

utilizando-se do mecanismo de atributos do padrão C++ 2014 [ISO/IEC 2014]. A ideia é que o programador apenas introduza anotações no código fonte sem precisar reescrevê-lo.

Na SPar, o programador lida com abstrações que são amigáveis com o vocabulário do domínio de *stream* e as propriedades são especificadas através de atributos nas regiões de código anotadas. Uma anotação é realizada com a especificação de colchetes duplos `[[id-attr, aux-attr, ..]]`, no qual pode haver uma lista de atributos. Pelo menos o primeiro atributo da lista deve ser especificado para que seja considerado uma anotação da SPar. O primeiro atributo é denominado de identificador enquanto que os demais da lista são auxiliares. Uma breve descrição dos atributos disponíveis na SPar é dado a seguir, onde `ToStream` e `Stage` são os identificadores e os demais auxiliares:

- `ToStream` é usado para denotar o início da região de *stream*. Pode ser colocado em frente de qualquer laço ou escopo de código.
- `Stage` é usado para anotar as regiões dentro do escopo `ToStream` que realizam computações sobre os elementos da *stream*.
- `Input(<list-var>)` é usado para demarcar elementos que a região da *stream* vai consumir. Podem haver uma ou mais variáveis como argumento.
- `Output(<list-var>)` é usado para denotar elementos que a região de *stream* vai produzir. Podem haver uma ou mais variáveis como argumento.
- `Replicate(<val-int>)` é usado para replicar um `Stage`. Isso é possível quando a execução é independente entre as réplicas e a sequência de elementos do *stream*. O atributo recebe como parâmetro um inteiro, mas também pode ser deixado vazio e utilizar a variável de ambiente `SPAR_NUM_WORKERS`.

Semanticamente todo `ToStream` deve possuir pelo menos uma anotação `Stage`. É relevante salientar que o código situado entre o `ToStream` e o primeiro `Stage` funciona como o primeiro estágio, consistindo no único trecho de código que pode ser inserido fora dos limites do escopo de uma anotação `Stage`. Ambos atributos identificadores podem utilizar o corpo do laço de repetição para definir seu escopo. Outra restrição é a utilização do auxiliar `Replicate`, que só pode ser inserido junto de um `Stage`.

O compilador da SPar é responsável por reconhecer a linguagem e foi desenvolvido utilizando o CINCLE (*A Compiler Infrastructure for New C/C++ Language Extensions*) [Griebler 2016]. O compilador da SPar realiza um *parser* do código e o representa em uma AST (*Abstract Syntax Tree*). Depois, realiza uma análise semântica da linguagem SPar. Subsequentemente, faz a transformação de código fonte para a AST, na qual são geradas chamadas para a biblioteca FastFlow. Ao final, o compilador chama o GCC para gerar código de máquina. Além disso, algumas otimizações são implementadas no compilador e podem ser acionadas através de *flags* de compilação descritas a seguir:

- `-spar_ondemand`: ativa um escalonar de elementos de *stream* sob demanda, pois por padrão é *round-robin*.
- `-spar_ordered`: permite que os elementos de *stream* sejam processados ordenadamente, pois por padrão eles são computados sem preservar a ordem.
- `-spar_blocking`: ativa um comportamento bloqueante no escalonador gerado pela SPar, por padrão é não-bloqueante.

3. Aplicação de Detecção de Pistas

A aplicação Detecção de Pistas reconhece e escreve padrões lineares em um vídeo. Implementada seguindo um padrão de linha de montagem, a aplicação apresenta três estágios distintos e independentes: leitura do arquivo de entrada, processamento (tratamento da

imagem) e escrita no arquivo de saída. Tem-se no processamento uma série de algoritmos, (*Canny, Hough*, etc.) aplicada individualmente em cada *frame*.

No Código 1 tem-se um representação do código fonte da aplicação de Detecção de Pistas. É possível notar que a região de *stream*, denotada pela anotação `ToStream`, consome apenas o objeto *capture* que é responsável por capturar individualmente cada *frame* do vídeo e armazenar na matriz *image*. Se o *frame* estiver vazio, o laço de repetição infinito é quebrado. Na linha 5, uma anotação `Stage` consumindo um *frame*, demarca o estágio de processamento, capaz de ser replicado e executado de forma independente. O tratamento do *frame* não sofre alteração em relação ao código original.Por fim, outra anotação `Stage`, na linha 8, demarca a região de escrita do *frame* no arquivo de saída (*oVideoWriter*). Nesta aplicação em específico, não é possível replicar este `Stage` pelo fato de que esta região possui um estado interno, que escreve em uma única região de memória. Além disso, para evitar sobreposição de *frames* é necessário compilar com `-spar_ordered` para manter os elementos do *stream* ordenados durante a escrita.

```
1  [[spar::ToStream,spar::Input(capture)]] while(1){
2    cv::Mat image;
3    capture >> image;
4    if(image.empty()) break;
5    [[spar::Stage,spar::Input(image),spar::Output(image),spar::Replicate(4)]]{
6      //tratamento do frame
7    }
8    [[spar::Stage,spar::Input(image)]]{
9      oVideoWriter.write(image);
10   }
11 }
```

Código 1. Código com a SPar:

4. Resultados

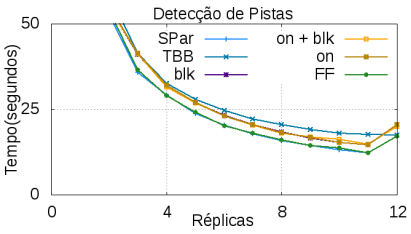
Nesta seção demonstramos os experimentos realizados, usando uma máquina possuindo dois processadores *Intel Xeon Six-Core E5645 2.4GHz Hyper-Threading* e 24GB de memória (no total de 24 *threads*). Foram também implementadas diferentes versões da aplicação de Detecção de Pistas, aonde os resultados foram plotados nos gráficos da Figura 1. Para fins de comparação, as versões TBB e FastFlow foram manualmente desenvolvidas por um especialista na área, considerando somente as que obtiveram o melhor resultado. Na SPar, a implementação foi tal e qual explicada na Seção 3, apenas variando as combinações possíveis das *flags* de compilação `-spar_ondemand` (`ond`), `-spar_blocking` (`blk`). O resultado de um *frame* processado pode ser visto na Figura 1(d), no qual podemos observar a demarcação das linhas em uma pista.

Nos experimentos, realizamos 10 execuções para cada número de réplicas (0 até 12). A média de cada execução foi empregada na formalização dos gráficos. O vídeo utilizado é de formato mp4 com resolução de 640x360 *pixels*, 1858 *frames* e duração de 1:02 minutos. Nos gráficos 1(a) e 1(b), o número 0 do eixo x representa a execução em serial. O máximo desvio padrão observado foi de $4,21$. Os resultados apresentados no gráfico 1(c) foram obtidos através do programa SLOCCount[1], que mediu a versão final do código fonte de cada uma das três implementações.

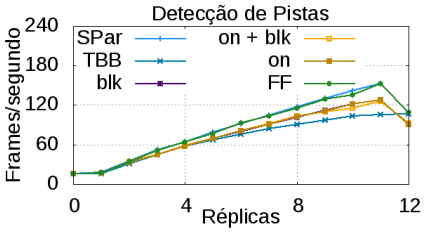
Observando os gráficos 1(a) e 1(b), é possível verificar que a escalabilidade do programa, é de aproximadamente 11 réplicas. A partir desse ponto, os resultados são prejudicados pelo uso do recurso *hyper-threading* nas versões SPar e FastFlow, enquanto que no TBB as perdas na vazão são perceptíveis a partir de 4 réplicas. Nota-se que a

[1]SLOCCount de David A. Wheeler: http://www.dwheeler.com/sloccount/

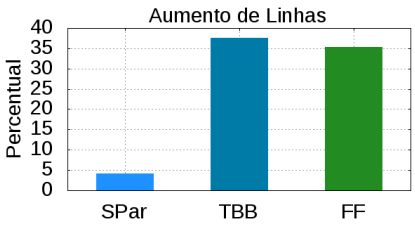
SPar mantém constantemente um desempenho semelhante ao do FastFlow e superior ao do TBB. A utilização das *flags* `ond` e `blk` deterioram o desempenho. Isso porque `ond` necessita comunicar mais vezes e o `blk` acaba não sendo benéfico já que os *frames* são processados intensivamente. Com seu melhor resultado, 11 réplicas, foi possível reduzir o tempo de execução da aplicação em 85%. Podemos concluir que a SPar oferece maior produtividade sem comprometer o desempenho da aplicação de detecção de pistas.



(a) Tempo de Execução.



(b) Vazão dos *frames*.



(c) Produtividade de código.

(d) Imagem processada.

Figura 1. Resultados da aplicação de detecção de pistas.

5. Conclusões

Este artigo apresentou a produtividade e o desempenho da SPar em uma aplicação de Detecção de Pistas. Os resultados foram melhores do que o esperando, sendo que a SPar obteve bom desempenho (similar ao FastFlow e melhor que o TBB) e uma significativa melhora na produtividade de código (aumentando apenas 4% em relação ao código original). Futuramente, pretende-se modelar outras aplicações reais do domínio de *stream* através da SPar. Da mesma forma, comparar, com outras bibliotecas, o desempenho e o esforço de programação ao aprender a SPar.

Referências

[Aldinucci et al. 2014] 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 *Parallel and Distributed Computing*, page 14. Wiley.

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

[Griebler et al. 2015] Griebler, D., Danelutto, M., Torquati, M., and Fernandes, L. G. (2015). An Embedded C++ Domain-Specific Language for Stream Parallelism. In *Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing*, ParCo'15, pages 317–326, Edinburgh, UK. IOS Press.

[ISO/IEC 2014] ISO/IEC, . (2014). Information Technology - Programming Languages - C++. Technical report, International Standard, Geneva, Switzerland.

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