Avaliando o Paralelismo de *Stream* com Pthreads, OpenMP e SPar em Aplicações de Vídeo

Gabriell A. de Araujo[1], Renato B. Hoffmann[1], Junior Loff[1], Dalvan Griebler[1,2], Luiz Gustavo Fernandes[1]

[1] Escola Politécnica, Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), Porto Alegre – RS – Brasil

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

{gabriell.araujo,renato.hoffmann,junior.loff,dalvan.griebler}@acad.pucrs.br, luiz.fernandes@pucrs.br

Resumo. *Visando estender os estudos de avaliação da* `SPar`*, efetuamos uma análise comparativa entre* `SPar`*,* `Pthreads` *e* `OpenMP` *em aplicações de stream. Os resultados revelam que o desempenho do código paralelo gerado pela SPar se equipara com as implementações robustas nas consolidadas bibliotecas* `Pthreads` *e* `OpenMP`*. Não obstante, também encontramos pontos de possíveis melhorias na* `SPar`*.*

1. Introdução

Aplicações de *streaming* são caracterizadas por um fluxo contínuo de dados, comumente encontrado em aplicações de *streaming* de vídeo, imagem e áudio. Devido à necessidade da obtenção de resultados com qualidade, o poder computacional exigido tem aumentado. Isto faz com que a intrusão do paralelismo seja necessária para aumentar o desempenho das aplicações. Além disso, o paralelismo nativo pode ser uma atividade complexa. Para tanto, existem ferramentas que auxiliam no desenvolvimento de programas paralelos. Dentre essas ferramentas, podemos destacar a seguir. `Pthreads`, uma biblioteca na linguagem C que oferece suporte para programação com *threads*. `OpenMP`, uma biblioteca para memória compartilhada com foco no paralelismo de dados. `SPar` [Griebler et al. 2017a], uma DSL (Linguagem Específica de Domínio) que oferece abstrações de alto nível para expressar o paralelismo de *stream* através de anotações na linguagem `C++`.

Este trabalho visa expandir a pesquisa [Griebler et al. 2017b], cujo comparou a `SPar` com o `Thread Building Blocks` e `FastFlow`. Pretende-se agora verificar se a `SPar` apresenta desempenho similar a implementações manuais e robustas com `Pthreads` e `OpenMP`, as quais são bibliotecas consolidadas e amplamente utilizadas pela indústria e academia. Para tal, são paralelizadas duas aplicações reais do domínio de *streaming* e os resultados são comparados. A primeira aplicação, *Detecção de Pistas* (DP) é utilizada para processar os limites de uma rodovia, através das imagens de uma câmera instalada na parte frontal do veículo, e assim controlar carros autônomos. A segunda aplicação, *Reconhecimento Facial* (RF) pode ser utilizada na detecção de indivíduos de interesse (criminosos, desaparecidos e crianças) em multidões. Os trabalhos relacionados que envolvem estas aplicações são poucos. [Mahmood et al. 2015] desenvolveu uma aplicação para detecção de veículos e reconhecimento da face do motorista bem como a paralelizou através da vetorização de instruções e reaproveitamento de cache em arquiteturas *many-core*. Em [Xu et al. 2017], os autores propõem um algoritmo de

detecção de pistas com execução paralela também em arquiteturas *many-core*. O presente trabalho inova através do foco no paralelismo de *stream* e arquitetura alvo, e também contribui com a análise das aplicações *DP* e *RF* com `Pthreads` e `OpenMP`, além da avaliação comparativa destas bibliotecas com a `SPar`. Este artigo está organizado da seguinte forma. Seção 2 discorre sobre as implementações. Seção 3 discute os resultados. Seção 4 apresenta as conclusões.

2. Implementação do Paralelismo de Stream

As aplicações *DP* e *RF* possuem padrão de fluxo de dados similar conforme destacado em [Griebler et al. 2017b] e podem ser desmembradas em três estágios. No primeiro, é feita a leitura de um quadro do vídeo de entrada. No segundo, uma sequência de filtros é aplicada sobre o quadro de forma a extrair as informações desejadas. No terceiro, o quadro é escrito em um vídeo de saída. Uma vez que as aplicações apresentam o mesmo comportamento, foi adotada a mesma estratégia de paralelismo em ambas, onde a única diferença reside na função que aplica filtros nos quadros do vídeo. A implementação modela um *pipeline* com três estágios, cuja comunicação entre os estágios é realizada através de filas encadeadas bloqueantes (apenas uma *thread* pode acessar cada fila por vez). A Figura 1 ilustra a estratégia de paralelismo adotada. O estágio A é executado por

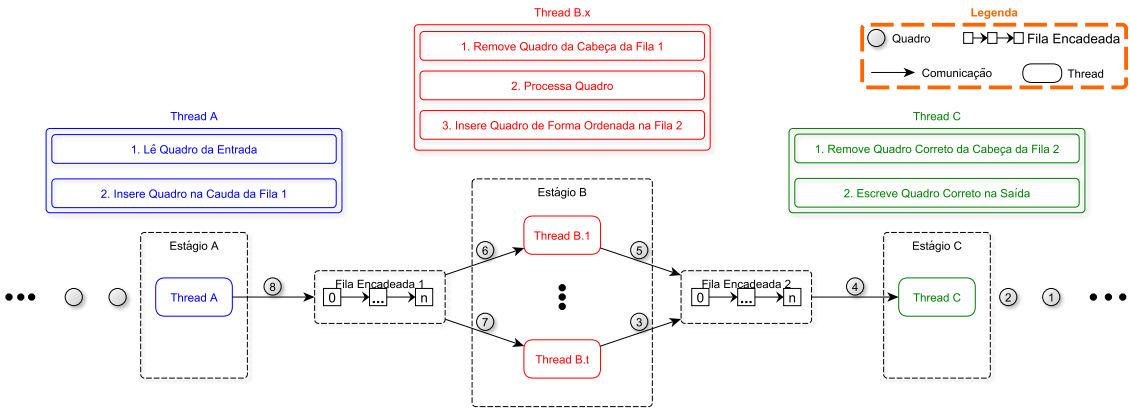


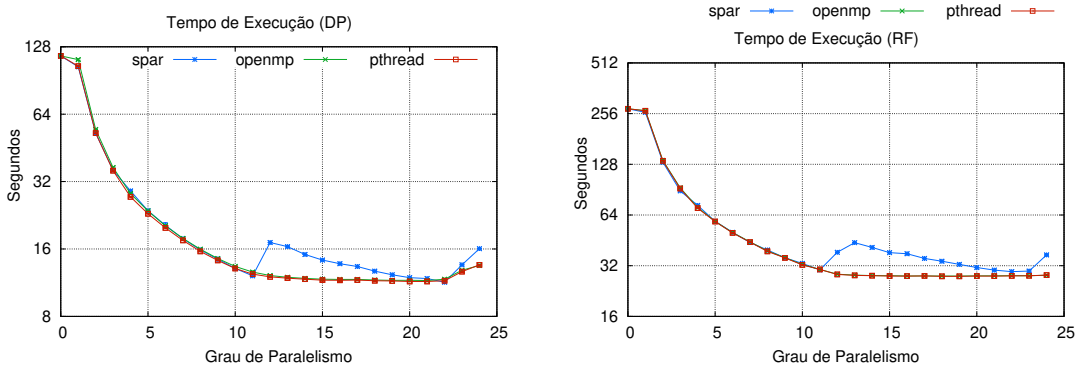
Figura 1. Estratégia de paralelização.

uma única *thread*. A *thread* lê um quadro da entrada. Coloca o quadro no final da fila 1, acessando diretamente o ponteiro da cauda. Envia um sinal para as *threads* do estágio B, indicando que existe um novo quadro a ser processado. Estes passos são repetidos até o término da *stream*. O estágio B é executado por *t threads*. Cada *thread* dorme até receber um sinal do estágio A. Ao receber um sinal, indicando um novo quadro na fila, a *thread* remove o quadro da cabeça da fila e o processa. Após, insere o quadro de forma ordenada na fila encadeada 2 e envia um sinal para a *thread* do estágio C, indicando que o quadro foi processado. Se ainda houver quadros na fila 1, repete o processo, caso contrário, dorme. O estágio C também é executado por uma única *thread*. A *thread* dorme até receber um sinal de uma das *threads* do estágio B, indicando que um novo quadro foi processado. Ao acordar, a *thread* verifica se o identificador do quadro que está inserido na cabeça da fila 2 é o correto. Se sim, remove o quadro da fila, escreve-o na saída e volta a dormir. Se não, apenas volta a dormir. A atribuição dos identificadores é feita de forma incremental no estágio A, portanto, incrementalmente a *thread* C sabe qual é o próximo quadro que deve ser escrito na saída.

Adentrando em detalhes específicos de programação. Em `Pthreads` foram usadas as funções $pthread\_mutex\_lock$ e $pthread\_mutex\_unlock$ para exclusão mútua e as funções $cond\_wait$ e $cond\_signal$ para a espera condicional dos estágios B e C. Em `OpenMP` foi mantida estrutura de código similar a `Pthreads`. No entanto, buscou-se usar ao máximo todas as rotinas da biblioteca `OpenMP`. Utilizou-se a diretiva $\#pragma\ omp\ parallel$ para a criação das *threads*. Através da função $omp\_get\_thread\_num$ as *threads* são identificadas e após distribuídas entre os estágios. A exclusão mútua foi implementada com omp_set_lock e omp_set_unlock, e para programar a espera condicional, utilizando essas duas funções básicas, foram implementadas funções equivalentes a $cond\_wait$ e $cond\_signal$ da biblioteca `Pthreads`. Além disso, foi atribuído o valor $PASSIVE$ na variável de ambiente OMP_WAIT_POLICY, cujo faz as *threads* dormirem ao serem bloqueadas. Isso é relevante pelo motivo de que dependendo do ambiente ou compilador o comportamento padrão é esperada ocupada, o que pode degradar o desempenho. Em relação a detalhes da implementação com `SPar`, estes podem ser conferidos em [Griebler et al. 2017b].

3. Resultados

Esta seção apresenta e descreve os resultados e experimentos realizados. A máquina utilizada possui dois processadores *Intel(R) Xeon(R) CPU E5-2620 v3 2.40GHz* e 24GB de memória RAM. Os vídeos de entrada são no formato MPEG4 com 5,5 e 14 MB de tamanho respectivamente para DP e RF. Os valores dos gráficos foram gerados a partir da média aritmética de 10 execuções e os desvios padrões também plotados através de *error-bars*, os quais foram insignificantes. Na Figura 2 são apresentados os tempos de execução das implementações. Pode-se notar que em ambas aplicações todas as interfaces de programação paralela atingiram até em torno de onze de aceleração (*speed-up*) em relação ao código sequencial (na Figura 2, grau de paralelismo zero). Isso é um resultado satisfatório levando em conta que a arquitetura dispõe de 12 núcleos físicos de processamento. No entanto, nota-se que as implementações `Pthreads` e `OpenMP` se comportam de maneira mais estável do que `SPar`. Ao utilizar *hyper-threading*, enquanto as implementações `Pthreads` e `OpenMP` mantém a mesma margem de desempenho, `SPar` apresenta uma queda significativa. Isso é um fator decorrente dos mecanismos, estruturas e escalonamento do código gerado pela `SPar` e pode ser alvo de futura melhoria. Em termos de



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

Figura 2. Tempos de execução.

programação, foi necessário um período de duas semanas para estudo e pesquisa sobre

programação e estratégias adequadas ao domínio de *stream* (modelos de programação produtor/consumidor), uma semana programando e estudando com `Pthreads`, uma semana estudando e programando com `OpenMP`, além dos testes, ajustes e correções. Em contraste, as implementações `SPar` levaram não mais que um dia. Além disso, através da ferramenta `sloccount` verificamos que a quantidade de linhas de código da `SPar` ficou apenas 4% maior do que as implementações sequencias, enquanto `Pthreads` 211% e `OpenMP` 215%. Embora essas bibliotecas de baixo nível de programação possuam equilibrio no quesito desempenho, o mesmo não aconteceu em termos de programabilidade. Embora existam abstrações do OpenMP para o paralelismo de dados, essas não refletem a mesma simplicidade no contexto do paralelismo de *stream*. O programador precisa conhecer técnicas de Sistemas Operacionais como espera condicional e implementá-las manualmente utilizando funções básicas do `OpenMP`. Além disso, o `OpenMP` não oferece descritores para acessos das `threads` e nem funções como envio de sinal *broadcast* a um grupo de `threads`. Em `OpenMP` calcula-se a soma do total de `threads` em cada estágio e então cria-se a quantidade específica de `threads`. Manualmente, o programador define um intervalo de *threads* para cada estágio (as `threads` são identificadas por um número inteiro), e também é necessário implementar `broadcast` manualmente utilizando laços e o intervalo específico de `threads` para cada estágio. Na medida que o número de estágios cresce, esses controles são cada vez mais complexos.

4. Conclusões

Esse trabalho demonstrou que a `SPar` consegue atingir desempenho similar a implementações robustas com bibliotecas consolidadas e de baixo nível de programação. Ainda, destacou-se um ponto de possível melhoria na `SPar` e foi acentuada a diferença do esforço necessário ao paralelizar aplicações de *stream* utilizando `SPar`, `Pthreads` e `OpenMP`. `SPar` é um destaque na programabilidade mesmo quando comparada a `OpenMP`, que embora ofereça recursos que facilitam a programação. Porém, demonstrou-se que esta premissa não é a mesma para algoritmos que adentram o domínio de *stream*. A implementação de estratégias eficientes acaba exigindo ainda mais esforço que `Pthreads`. Futuramente, pretende-se efetuar estudos comparativos com outras interfaces de programação paralela e otimizar a geração de código paralelo no compilador da `SPar`, buscando melhorias na estabilidade do desempenho com o uso de *hyper-threading*.

Referências

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

[Griebler et al. 2017b] Griebler, D., Hoffmann, R. B., Danelutto, M., and Fernandes, L. G. (2017b). Higher-Level Parallelism Abstractions for Video Applications with SPar. In *Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing*, ParCo'17, pages 698–707, Bologna, Italy. IOS Press.

[Mahmood et al. 2015] Mahmood, Z., Khan, M. U. S., Jawad, M., Khan, S. U., and Yang, L. T. (2015). A parallel framework for object detection and recognition for secure vehicle parking. In *2015 IEEE 17th International Conference on High Performance Computing and Communications*, pages 892–895.

[Xu et al. 2017] Xu, Y., Fang, B., Wu, X., and Yang, W. (2017). Research and implementation of parallel lane detection algorithm based on gpu. In *2017 International Conference on Security, Pattern Analysis, and Cybernetics (SPAC)*, pages 351–355.