Explorando a Flexibilidade e o Desempenho da Biblioteca FastFlow com o Padrão Paralelo Farm

Júnior Löff, 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

{junior.loff,dalvan.griebler,cleverson.ledur}@acad.pucrs.br
luiz.fernandes@pucrs.br

Resumo. *O paralelismo é uma tarefa para especialistas, onde o desafio é utilizar abstrações que ofereçam a flexibilidade e expressividade necessária para atingir o melhor desempenho. Este artigo visa explorar variações na implementação do padrão Farm utilizando a biblioteca FastFlow nos algoritmos K-means (domínio da mineração de dados) e Mandelbrot Set (domínio da matemática). Concluímos que o padrão Farm oferece boa flexibilidade e bom desempenho.*

1. Introdução

Devido ao limite físico entre transistores e problemas de resfriamento que vinham sendo encontrados nos processadores, o surgimento de *multi-cores* tornou-se necessário [Blake et al. 2009]. O mesmo consiste na adição de camadas empilhadas em um único processador que se comportam como outros processadores. Não obstante, ocorreu um movimento para acompanhar esta tecnologia onde aplicações defasadas precisaram ser modificadas para suportar a tecnologia *multi-core*.

A implementação de algoritmos que denominam-se paralelos é considerada complexa e enfrenta uma série de obstáculos que não eram encontrados no cenário de aplicações sequenciais. Muitos foram solucionados enquanto outros persistem no ramo da pesquisa, mas a maioria baseiam-se nos padrões de paralelismo, que podem ser gerais ou aplicarem-se para um grupo em específico [Mattson et al. 2005, Griebler and Fernandes 2013]. Ao mesmo tempo surgiram *frameworks* que abstraem a programação. Tem-se então APIs (interface de programação de aplicações) que implementam o conceito de *threads* (ex. Solaris threads e POSIX threads). O outro grupo oferece abstrações de mais alto nível utilizando estas APIs, por exemplo, TBB, Cilk, OpenMP e FastFlow [Aldinucci et al. 2014].

Portanto, o objetivo deste artigo é contribuir com a difusão e usabilidade do FastFlow, explorando a flexibilidade em expressar o padrão de programação paralela Farm e o desempenho resultante. Não obstante, foram implementados os algoritmos Mandelbrot Set e K-Means por se tratarem de aplicações maleáveis à paralelização e com fluxos de *stream* de dados com granularidades alta e baixa, respectivamente. Com isso, é possível estender os resultados obtidos com a expressividade do FastFlow no padrão Farm e suas variações em aplicações gerais. O trabalho está organizado da seguinte maneira. A Seção 2 descreve o FastFlow e sua flexibilidade no padrão paralelo Farm. A seção 3 apresenta duas aplicações reais implementadas com o FastFlow. Na seção 4 estão descritos os resultados obtidos explorando a flexibilidade nas aplicações Mandelbrot Set e K-Means.

2. FastFlow: Explorando a Flexibilidade do Padrão Farm

O FastFlow [Aldinucci et al. 2014] é uma interface de programação baseada em *skeletons* para C++ . Esta suporta paralelismo de tarefas e dados. Além do mais, implementa três ní-

veis de abstração cujos programadores podem explorar: padrões de alto nível (encontram-se os laços de repetição e paralelismo de *streams*), padrões de fluxo (Pipeline e Farm, mas não limitando-se à estes) e padrões de baixo nível (consiste na estrutura onde o FastFlow abstrai código em blocos e filas).

No Código 1 está representado em alto nível um Farm implementado no FastFlow. Este possui o seguinte fluxo: um *emitter* (emissor) processa e divide a entrada de dados em tarefas menores que são enviadas para *workers* (trabalhadores), que executam em paralelo. Logo após terminarem a execução, o resultado é enviado ao *collector* (coletor), que recebe todas as partes processadas e concatena em uma única estrutura para gerar a saída esperada. Em certas aplicações, é necessário manter a ordem das informações *FIFO(first in, first out)*. Para isto, o FastFlow possui implementado uma variação de Farm denotado como Ordered Farm. Como pode ser visto no Código 1, para declará-la é necessário somente alterar a estrutura de `ff_Farm` (linha 21) para `ff_OFarm` (linha 22) e respeitar algumas divergências para definir o *emitter* e o *collector*.

```
1  struct Worker: ff_node{
2      void * svc(void * task){
3          //processa uma tarefa
4      }
5  };
6  struct Emitter: ff_node{
7      void * svc(void * task){
8          //cria uma tarefa e envia
9      }
10 }Emitter;
11 struct Collector: ff_node{
12     void * svc(void * task){
13         //recolhe as tarefas
14     }
15 }Collector;
```

```
16 int main(int argc, char **argv){
17     std::vector<std::unique_ptr<ff_node>>
       workers;
18     for(int i = 0; i < atoi(argv[1]); i++)
19   workers.push_back(make_unique<Worker>())
20
21     ff_Farm farm(move(workers), Emitter,
       Collector);
22     ff_OFarm ofarm(move(workers));
23     ofarm.setEmitterF(Emitter);
24     ofarm.setCollectorF(Collector);
25     farm.run_and_wait_end();
26     ofarm.run_and_wait_end();
27 }
```

Código 1. Uso do *farm*.

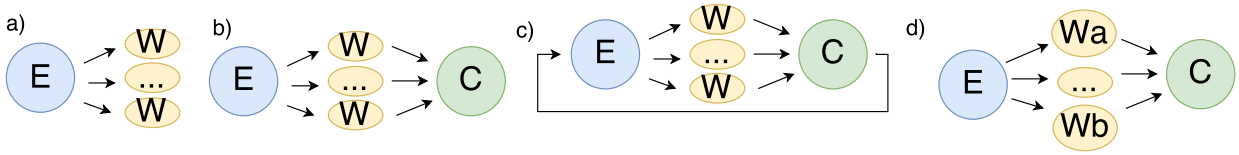


Figura 1. Estruturas de possíveis versões do padrão *farm*.

Apesar do fluxo padrão do Farm implementado, ver Figura 1.b, o FastFlow permite alterar esta rotina. Na Figura 1 há exemplos de combinações, como: 1.a remoção do *collector*, 1.c adição do *feed-back* entre as entidades, assim como 1.d distribuição de tarefa entre dois tipos de *workers* distintos entre si. Nesta imagem, E representa o *Emitter*, W são os *Workers* e C é o *Collector*, com cada seta representando uma fila no respectivo sentido. O dinamismo das combinações tem sido uma das principais características do FastFlow, onde é possível modificá-las para obter o melhor aproveitamento no fluxo de uma aplicação. Isto é possível pois cada entidade herda os métodos da classe *abstract ff_node*, linhas: 1, 6 e 11 do Código 1. Uma vez que cada trecho de código está encapsulado, é possível definir vários tipos de filas de comunicação entre si [Aldinucci et al. 2014].

Outra aplicabilidade que o FastFlow implementou recentemente é a função de *blocking* (bloqueante) e *non-blocking* (não bloqueante). Por padrão o programa é *non-blocking*, ou seja, os blocos (*emitter, workers*, etc.) continuam utilizando o processador, mesmo que estejam ociosos, até que a *flag* EOS(*End of Stream*) seja recebida. Como exemplo prático pode-se utilizar o *emitter*, como já foi dito, ele distribui as tarefas de modo que preencha as filas de cada *worker*. No entanto, mesmo após encher as filas,

o *emitter* utiliza uma parte do processador onde executa um laço para verificar o *status* das filas. Através do modo *blocking*, que é definido por uma *flag* na compilação (`DBLOCKING_MODE`), é possível pausar o bloco desse *emitter* e liberar *clocks* do processador para outros blocos até que o mesmo seja requisitado. Na seguinte seção serão discutidas algumas destas possíveis implementações do padrão Farm em aplicações reais.

3. Paralelização das Aplicações com o Padrão Farm

MandelBrot Set é uma aplicação que pertence ao conjunto de visualizações matemáticas, onde uma imagem de largura x e altura y tem cada um de seus píxeis processados através de um fractal no plano complexo que determina uma cor RGB baseado em sua localização. Como resultado, tem-se uma imagem que segue padrões recursivos. Para paralelizá-la com o FastFlow, foi identificado primeiramente a parte mais custosa do algoritmo, que se trata do cálculo realizado em cada píxel para definir sua cor. Com isso, foi estudado a melhor forma de modelar o Farm nesta região do código. O fluxo resultante está representado na Figura 1.a. A implementação é similar ao exemplo do Código 1, porém não possui um *collector* implementado, visando melhorar o desempenho. Ao final do programa, tem-se uma estrutura na forma de matriz com todos os dados necessários para gerar a imagem no padrão Mandelbrot Set. Neste caso, é necessário manter a ordem das linhas de píxeis para não ocorrerem distorções na imagem, portanto, foi utilizada uma `ff_OFarm`, também representada no Código 1 (linhas 22 à 24) com fluxo da Figura 1.b.

Classificado como um algoritmo de mineração de dados, o **K-Means** é um método de agrupamento que consiste em juntar n pontos para k centroides. Esta interação acontece em um plano cartesiano que tem duas ou mais dimensões, sendo que cada ponto representa uma coordenada e cada centroide é um ponto médio para determinado grupo de coordenadas. A parte central de processamento deste algoritmo são dois laços aninhados. No primeiro laço, para cada ponto é calculado sua distância em relação aos centroides. Depois, este ponto é associado ao centroide mais próximo. No término desta execução, inicia-se outro laço cujo são somados os valores de cada ponto para calcular a média e definir o novo centroide. Este processo é repetido n vezes enquanto houver alternância de pontos para outros centroides. Para paralelizar esta aplicação foi aplicada um `ff_Farm` para cada um dos laços, cujo fluxo está representado na Figura 1.a. Neste caso, também foi removido o *collector* em prol de otimizar esta iteração nesta aplicação.

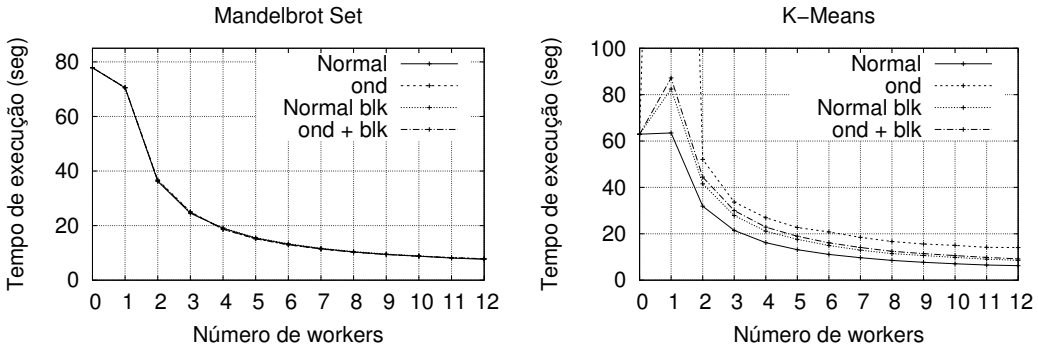
4. Resultados

Os experimentos foram realizados em uma máquina que possui dois processadores *Intel Xeon Six-Core E5645 2.4GHz Hyper-Threading* e $24GB$ de memória (no total são 24 *threads* e 12 núcleos físicos). As duas aplicações foram executadas com duas possíveis otimizações: escalonador sob demanda (ond), chamando o método `set_scheduling_ondemand()` e modo bloqueante (blk), especificando `DBLOCKING_MODE`. Os programas foram compilados usando `-O3` e executados para cada amostra 10 vezes para efetuar a média aritmética. As amostras obtiveram um desvio padrão abaixo de 1% que não foi considerado relevante, os resultados podem ser observados na Figura 2.

Na aplicação Mandelbrot Set (Figura 2(a)), pode-se observar que as opções de otimização não causaram impacto positivo e negativo. No modo `blk` nesta aplicação, o tempo de execução dos *workers* e o tempo de incrementar elementos na fila através do *emitter* são sincronizados. Ou seja, quando um *worker* solicitar uma tarefa, o *emitter* a terá pronta e vice-versa. Desta forma, as instâncias praticamente não são pausadas. Além do mais, quando operando em `ond` também há resultados neutros. Pois não há desperdício de *clocks* tentando incluir tarefas em filas cheias ou problemas de ociosidade dos *workers*

esperando receber novas tarefas. Este comportamento é associado à alta granularidade, já que foi optado em passar uma linha de píxeis ao invés de píxeis isolados.

Para o algoritmo K-means (Figura 2(b)), tem-se um contra exemplo no uso da flexibilidade do FastFlow. Como pode ser visto na Figura 2.b, com 1 *worker* em especial e depois com menos impacto, no impasse de *workers* e *emitter* monitorarem a fila, vários *clocks* são desperdiçados escalonando as tarefas `ond`, aumentando o tempo de execução. Não obstante, no modo `blk`, o tempo é estabilizado comparando com o `ond`, mas também piora quanto à execução normal, pois encher uma fila não representa ociosidade dos blocos. Esta aplicação representa um fluxo de dados extenso com granularidade baixa.



(a) Tempos para Mandelbrot Set. (b) Tempos para K-Means.

Figura 2. Resultados do desempenho.

5. Conclusão

Neste artigo foi apresentado e discutido os principais conceitos de flexibilidade e desempenho proporcionadas pelo FastFlow com o padrão Farm. Como resultado deste estudo tem-se dados empíricos obtidos na paralelização de duas aplicações do domínio de mineração de dados e da matemática, contribuindo para difundir a biblioteca do FastFlow. Além do mais, as aplicações podem ser simplificadas à granularidade alta e baixa, representadas pelo Mandelbrot Set e K-means, respectivamente. Desta forma, se os experimentos forem refeitos com aplicações que seguem estas características, os resultados seguirão o comportamento demonstrado neste trabalho. As principais dificuldades encontradas foram referentes à documentação, pois demasiados detalhes de implementação aqui discutidos não estão evidentes na documentação oficial, apenas constam no código fonte da biblioteca. Além disso, ressaltamos que ainda existem funcionalidades que não foram discutidas neste trabalho, mas que vão ser consideradas em um trabalho de investigação futura (*e.g.*, a customização do escalonador e a implementação de canais de *feed-back*).

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*, PDC. Wiley.

[Blake et al. 2009] Blake, G., Dreslinski, R. G., and Mudge, T. (2009). A Survey of Multicore Processors. *IEEE Signal Processing Magazine*, 26(6):26–37.

[Griebler and Fernandes 2013] Griebler, D. and Fernandes, L. G. (2013). Towards a Domain-Specific Language for Patterns-Oriented Parallel Programming. In *17th Brazilian Symposium Programming Languages (SBLP)*, LNCS, Brasilia, Brazil. Springer.

[Mattson et al. 2005] Mattson, T. G., Sanders, B. A., and Massingill, B. L. (2005). *Patterns for Parallel Programming*. Addison-Wesley, Boston, USA.