Proposta de Modelo de Custo para Paralelização de Estágios em *Pipelines* Lineares

Guilherme Vásquez[1], **Dalvan Griebler**[1]

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

guilherme.malgarizi@edu.pucrs.br, dalvan.griebler@pucrs.br

Resumo. *A programação paralela continua sendo uma tarefa difícil, que exige programadores especializados. Portanto, busca-se soluções que auxiliem o desenvolvimento de aplicações paralelas. Uma dessas soluções é a paralelização automática no nível de compilador. Para aplicações de paralelismo de stream, pode-se utilizar a DSL SPar. No entanto, o programador ainda precisa escolher explicitamente o grau de paralelismo para cada estágio do pipeline. Nesta pesquisa, propõe-se um modelo de custo para determinar quais estágios podem se beneficiar da replicação, com o objetivo de dispensar ao programador a especificação do grau de paralelismo.*

1. Introdução

A busca por desempenho computacional sempre foi um tema fundamental na área da ciência da computação. Por décadas, melhorias de *hardware* seguindo a Lei de Moore sustentaram o crescimento do desempenho para programas sequenciais. À medida que essas tendências desaceleram e nos aproximamos dos limites da Lei de Moore [Thompson and Spanuth 2021], novos ganhos de desempenho passam a depender cada vez mais do paralelismo. O surgimento da programação paralela trouxe consigo um problema que evidenciou a complexidade de escrever e otimizar código paralelo. A programação paralela é, em geral, mais difícil do que a programação sequencial [Polychronopoulos 1993]. Ela introduz uma série de novos desafios, exigindo que os programadores considerem cuidadosamente como dividir adequadamente as regiões do programa, se possível, para explorar as capacidades paralelas da arquitetura *multicore*.

Na tentativa de aliviar algumas das dificuldades da programação paralela, surgiu a área de pesquisa voltada à paralelização automática [Bacon et al. 1994]. O objetivo de um compilador de paralelização automática é realizar análise de programas, transformar regiões de código sequenciais em versões paralelas e otimizar esse código. Para alcançar esses resultados, é necessário garantir que a paralelização seja correta para todos os programas e determinar se ela é vantajosa [Barakhshan and Eigenmann 2022]. Atualmente, implementar paralelismo no código ainda é uma tarefa difícil para programadores. Existem diversas ferramentas que podem ser usadas para desenvolver um código paralelo. Por exemplo, é possível utilizar anotações de código que permitem ao compilador gerar, ao final, uma versão paralela com OpenMP. Há também linguagens de domínio específico (DSLs) que podem utilizar anotações para indicar ao compilador como paralelizar o código, uma dessas sendo a SPar.

2. SPar

A SPar [Griebler et al. 2017] é uma DSL baseada em C++ voltada ao paralelismo de *stream*. Ela permite o uso de anotações de alto nível para gerar código paralelo. A SPar possui seu próprio compilador, que utiliza uma infraestrutura de transpilação *source-to-source* para substituir as anotações por chamadas à *runtime* FastFlow. Inicialmente, a DSL foi projetada para arquiteturas de memória compartilhada, mas desde então foi estendida para oferecer suporte a uma gama mais ampla de arquiteturas, principalmente à memória distribuída [Griebler and Fernandes 2017] e à GPU [Araujo et al. 2025]. A SPar utiliza duas anotações identificadoras, os atributos *ToStream* e *Stage*, e três atributos auxiliares, *Input*, *Output* e *Replicate*. As anotações identificadoras marcam as regiões de código responsáveis pelo processamento de *stream*, enquanto os atributos auxiliares indicam o comportamento de *I/O* dessas regiões e o grau de paralelismo nos estágios da SPar.

Para utilizar a SPar, primeiro anota-se, com *ToStream*, as seções de código que serão regiões de processamento de *stream*. Dentro do escopo do *ToStream*, pode-se utilizar as anotações *Stage* para montar, explicitamente, estágios do *pipeline*. É possível adicionar, a cada anotação, um atributo do tipo *Input* para indicar que a entrada é produzida fora da região anotada. Semelhantemente, o atributo *Output* indica que algo produzido na região anotada será utilizado fora dela. O atributo *Replicate* só pode ser utilizado com a anotação *Stage* e serve para controlar o grau de paralelismo do estágio, configurando mais réplicas de um determinado estágio na região de *stream*. Como a SPar não possui suporte à paralelização totalmente automática, ela depende do desenvolvedor para fornecer as anotações no código e os atributos auxiliares apropriados. Dado esse contexto, surge a oportunidade de estender a ferramenta com o objetivo de simplificar seu uso, reduzindo a dependência das decisões do programador para obter ganhos de desempenho.

A Figura 1 apresenta um trecho de código que utiliza anotações SPar com o objetivo de realizar a soma de vetores. Cada estágio engloba uma parte do processamento aplicado aos elementos do fluxo de dados. O primeiro estágio realiza a soma dos elementos dos vetores e pode ser replicado de acordo com o valor da variável *n_workers*. O segundo estágio acumula as somas parciais calculadas a um total global.

```
1  [[spar::ToStream, spar::Input(n_vetor, tam_vetor, vets1, vets2, results)]]
2  for (int i = 0; i < n_vetor; i++) {
3    // Pegar os i-ésimos vetores de entrada e saída e inicializar a soma parcial
4    [[spar::Stage,
5      spar::Input(tam_vetor, vet1, vet2, result, soma_parcial),
6      spar::Output(soma_parcial, result),
7      spar::Replicate(n_workers)]]
8    for (int j = 0; j < tam_vetor; j++) {
9      // Somar os j-ésimos elementos e acumular a soma parcial
10   }
11   [[spar::Stage, spar::Input(soma_parcial), spar::Output(total)]]
12   {
13     // Somar a soma parcial ao total
14   }
15 }
```

Figura 1. Um trecho de código utilizando anotações SPar.

3. Proposta

Propõe-se a definição de um modelo de custo que pode ser implementado no caso de uso da SPar. O modelo é pensado para que possa ser acoplado ao *frontend* do compila-

dor, funcionando por meio da travessia na árvore sintática abstrata (AST) do código C++ gerada pelo Clang. Utiliza-se a plataforma nativa *LibTooling* para construção de ferramentas em nível de código-fonte fornecida pelo projeto Clang. Esse ferramental fornece controle total para navegar pela AST com *ASTFrontendActions*, e utilizando o mecanismo *RecursiveASTVisitor* consegue-se visitar nós específicos da AST e executar ações personalizadas sobre eles. Dessa forma, o modelo de custo é avaliado recursivamente sobre a AST. Deve-se representar o custo como um valor inteiro não negativo, sem unidade, destinado a comparações relativas, e não para previsões precisas do tempo de execução. Seja $C(n)$ o custo de um nó n da AST. Então, os custos totais são computados agregando os custos dos nós da região que está sendo calculada. Para um nó n com filhos $child(n)$, a regra geral de agregação pode ser definida como:

$$C(n) = f_n(\{C(c) \mid c \in child(n)\}) \tag{1}$$

onde f_n é uma função de agregação de custo específica do nó. Para a maioria dos nós, que representam operações simples, sua modelagem seria $C(n) = w(n)$, onde $w(n) \in \mathbb{N}$ e $w(n)$ é o peso do custo associado à operação do nó. Considerando que cada região *ToStream* T contém um ou mais estágios s, podemos representar a construção como uma sequência de estágios definida por $T = \{s_1, s_2, ..., s_k\}$. O custo total é, portanto, agregado por:

$$C(T) = \sum_{i=1}^{k} C(s_i) \tag{2}$$

Da mesma forma, um *Stage* S é representado como o conjunto de nós da AST que ele contém. Assim, seja $S = \{n_1, n_2, ..., n_k\}$ a representação de um estágio que possui a seguinte função de agregação de custo:

$$C(S) = \sum_{i=1}^{k} C(n_i) \tag{3}$$

Uma grande variedade de programas, desde as aplicações mais simples até sistemas complexos, depende de dois aspectos fundamentais da programação: controle de laços e desvio condicional. Dada sua importância, define-se as funções de agregação para os nós de *if* e laços *for*. Para um laço *for* F, seja $B_{for} = \{n_1, n_2, ..., n_k\}$ o conjunto de nós dentro do corpo do laço e L o número de iterações do laço, que, inicialmente, assume-se estar definido no arquivo-fonte ou nos limites do laço. Assim, o custo de um nó *ForStmt* do Clang pode ser modelado por:

$$C(F) = L \cdot \sum_{i=1}^{k} C(n_i) \tag{4}$$

Para modelar a condicional *if*, deve-se levar em consideração a possibilidade de também existir um corpo *else*. Considerando que, em um contexto de execução paralela, o desempenho é limitado pela porção mais lenta, modela-se a função de agregação do *if* assumindo um cenário de pior caso. Sejam $B_{if} = \{n_1, n_2, ..., n_k\}$ e $B_{else} = \{e_1, e_2, ..., e_m\}$ os conjuntos de nós presentes nos corpos do *if* e do *else* respectivamente. Assim, pode-se modelar os custos de cada ramo como:

$$C_{if} = \sum_{i=1}^{k} C(n_i) \tag{5}$$

$$C_{else} = \begin{cases} \sum_{i=1}^{m} C(e_i), & \text{se } B_{else} \neq \emptyset \\ 0, & \text{se } B_{else} = \emptyset \end{cases} \tag{6}$$

E é possível combiná-los para definir a função de agregação para a condicional *if* I como:

$$C(I) = \max(C_{if},\ C_{else}) \tag{7}$$

Uma vez que o custo de cada estágio pode ser calculado a partir das funções de agregação, utiliza-se desses valores para estimar o grau de paralelismo que cada estágio necessita. A intuição é que os mais custosos tendem a ser os gargalos na execução. Sabendo disso, mais réplicas são atribuídas aos estágios mais custosos para distribuir o trabalho total entre *workers*. Em teoria, por meio da replicação, se buscaria uma proporção que deixasse a distribuição de trabalho nivelada entre cada estágio de uma região de *stream*. Para avaliar a performance do modelo proposto, pretende-se utilizar um conjunto de *benchmarks* implementados com a SPar. O *benchmark* possui 11 aplicações com anotações SPar. O objetivo será comparar o desempenho das versões nas quais o modelo de custo guiou o uso do atributo *Replicate* para decidir o número de *workers* dos estágios e as versões presentes no *benchmark*.

Os resultados desta pesquisa visam à modelagem e à implementação de um modelo de custo em nível de compilador, que poderá ser acoplado à SPar. Essa modelagem fornecerá à área de paralelismo de *stream* uma heurística para avaliar o custo de estágios em *pipeline* linear. Adicionalmente, a implementação fortalecerá a SPar, tornando-a mais automática e menos dependente do programador, facilitando a programação paralela e aumentando a produtividade dos programadores.

Referências

Araujo, G., Rockenbach, D. A., Löff, J., Griebler, D., and Fernandes, L. G. (2025). A C++ annotation-based domain-specific language for expressing stream and data parallelism supporting CPU and GPU. *Journal of Computer Languages*, 85:101369.

Bacon, D. F., Graham, S. L., and Sharp, O. J. (1994). Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)*, 26(4):345–420.

Barakhshan, P. and Eigenmann, R. (2022). icetus: A semi-automatic parallel programming assistant. In Li, X. and Chandrasekaran, S., editors, *Languages and Compilers for Parallel Computing*, pages 18–32, Cham. Springer International Publishing.

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. and Fernandes, L. G. (2017). Towards Distributed Parallel Programming Support for the SPar DSL. In *Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing*, ParCo'17, pages 563–572, Bologna, Italy. IOS Press.

Polychronopoulos, C. (1993). Parallel programming issues. *International Journal of High Speed Computing*, 05(03):413–473.

Thompson, N. C. and Spanuth, S. (2021). The decline of computers as a general purpose technology. *Communications of the ACM*, 64(3):64–72.