SPar
Stream . Parallelism

A Domain-Specific Language for Stream Parallelism (SPar)

Manual de Usuário:

Implementando Paralelismo de Stream

Dalvan Griebler

Renato Hoffmann Filho

Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)

Grupo de Modelagem de Aplicações Paralelas (GMAP)

Porto Alegre, 2016

PUCRS
Pontifícia Universidade Católica do Rio Grande do Sul

GMAP

SPar
Stream . Parallelism

1. Introdução

A SPar é uma DSL (*Domain-Specific Language*) embutida na linguagem C++, capaz de modelar aplicações baseadas em paralelismo de *stream*. Ela foi implementada utilizando-se do mecanismo de atributos do padrão C++ 2014. A ideia base é que o programador apenas introduza anotações no código fonte para permitir o paralelismo sem precisar reescrevê-lo, deixando que o compilador da linguagem gere o código paralelo.

Esta documentação visa oferecer uma visão geral sobre o uso anotações de paralelismo de *stream* na **SPar**, apresentando conceitos e terminologias fundamentais para a compreensão e implementação em um código fonte.

2. Compilando e Executando Programas

A SPar possui um compilador próprio que reconhece C/C++ e a sua linguagem, que possui um conjunto de atributos que são anotados no código fonte. O compilador da SPar reconhece estes atributos e realiza a análise semântica, retornando um erro caso existe alguma inconsistência. Em geral, o compilador aceita todos os parâmetros do GCC e também oferece suas próprias flags. A mais importante é a flag **-spar_file**, que é usada para especificar o código C++ fonte, no qual o compilador deve realizar o parser e transformar as anotações em código paralelo. Para uma compilação simples, o comando é o seguinte:

$ spar_dir/bin/spar -std=c++1y -spar_file <arquivo.cpp> -o <arquivo>

Para executar um programa da SPar é simples e da mesma forma que qualquer outro binário, tal como segue:

$ ./seu_arquivo [argumentos]

Note que o compilador da SPar é um executável, de nome **spar**, que está dentro da pasta “bin/” na pasta raiz do compilador. Por isso, é necessário passar o caminho completo, uma vez que ele não é instalado diretamente no sistema. É válido ressaltar também que o programa anotado com a SPar pode ser compilado utilizando o g++. No entanto, ele não irá gerar código paralelo, apenas ignorando as anotações da SPar. Além disso, a SPar permite que o programador especifique *flags* para otimização da geração de código, as opções são descritas a seguir:

-spar_ordered: permite que os elementos de *stream* sejam processados ordenadamente, pois por padrão eles são computados sem preservar a ordem.

-spar_ondemand: ativa um escalonador de elementos de *stream* sob demanda, pois por padrão é *round-robin*.

-spar_blocking: ativa um comportamento bloqueante no escalonador gerado pela SPar, por padrão é não-bloqueante.

3. Interface de Suporte ao Paralelismo de Stream

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. Isso refere-se ao cenário de uma linha de produção, onde existe uma sequência de estágios que realizam operações sobre os elementos de um *stream*. Estes elementos também podem ser vistos como tarefas independentes que cada estágio deve realizar ao receber o resultado do estágio anterior. Assim, cada estágio consome uma tarefa e produz outra.

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 (**id-attr**) e os outros são auxiliares (**aux-attr**):

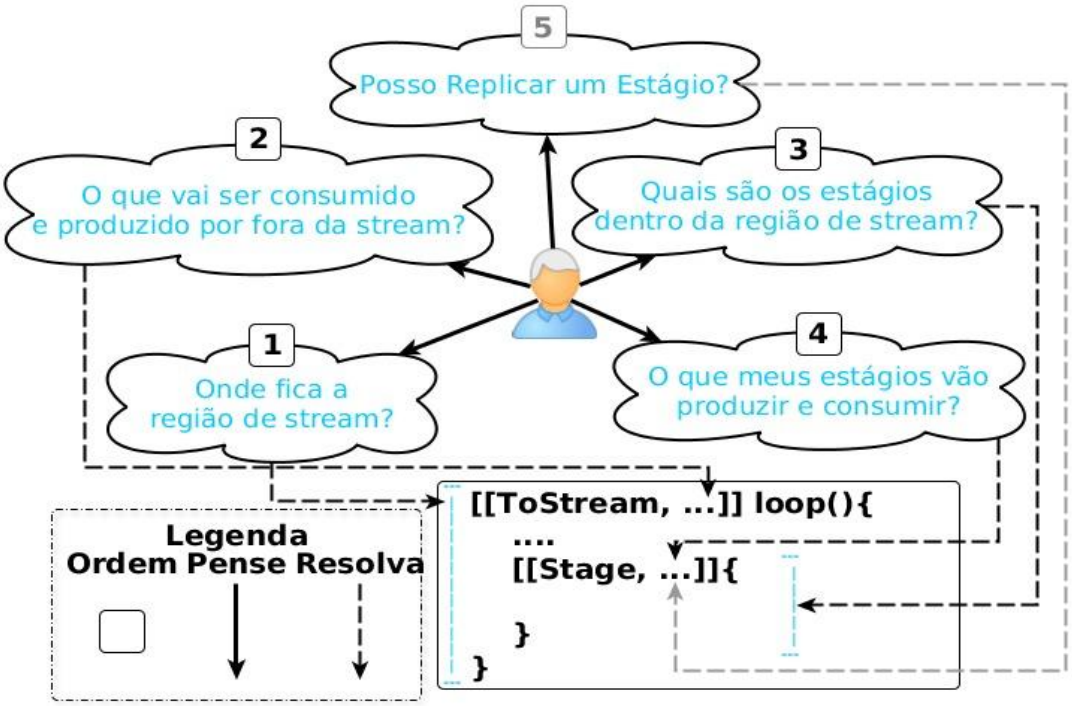
- **ToStream** é usado para denotar o início da região de *stream*. Pode ser colocado em frente de qualquer laço ou escopo de código.
- **Stage** é usado para anotar as regiões dentro do escopo **ToStream** que realizam computações sobre os elementos da stream.
- **Input(<list-var>)** é usado para demarcar 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** (ex. *export SPAR_NUM_WORKERS=2*).

Ao anotar um código fonte com a SPar, é necessário que sejam colocados com o *namespace* **spar**, que identifica o atributo como sendo parte da linguagem SPar. Segue uma demonstração simples:

```
[[spar::ToStream]] loop {
        ….
        [[spar::Stage, spar::Input(a,b)]]{
                ….
        }
}
```

Semanticamente, como pode ser visto na demonstração acima, para cada **ToStream** pelo menos um **Stage** precisa ser anotado dentro de sua região. Outra restrição é que nenhum trecho de código é aceito após o último **Stage** e entre as

declarações de **Stage**. Observe que a parte inicial da região **ToStream** será o primeiro estágio de processamento do *stream*. É nele que normalmente os elementos de *stream* (ou tarefas) são gerados e o controle de parada é realizado (note que é a região que antecede uma anotação de **Stage**). Para ajudar os programadores a anotar códigos de aplicações de *stream*, temos uma figura a seguir que guia o programador para uma anotação correta através de uma sequência de perguntas.



Seguindo a ordem, a primeira coisa a se perguntar é onde fica a região de *stream* (computacionalmente intensiva). Assim, para localizá-lo no código, a dica é que ela geralmente está localizada sobre um laço de repetição que produzirá um novo elemento de *stream* a cada iteração. Já identificada a região, basta anotar ela com o atributo **ToStream** como segue na Figura acima. Em seguida, questiona-se o que será consumido e produzido por essa região, anotando consequentemente com os atributos **Input (**consume**)** e **Output (**produz**)** na lista da anotação **ToStream**. Note que a declaração é opcional, e somente se necessário tais atributos devem ser especificados com as variáveis que serão consumidas (para dentro da região **ToStream**) e produzidas (para fora da região **ToStream**).

Após respondidas as perguntas anteriores, o terceiro passo é demarcar os estágios com anotações **Stage**. Lembre-se que a área entre o **ToStream** e o primeiro **Stage** funciona como um estágio. Depois, o quarto passo é anotar o consumo e produção desses estágios quando necessário com os atributos **Input(**consume**)** e **Output(**produz**)**. Na última etapa (pergunta 5) é identificada a possibilidade de replicar uma determinada região anotada com **Stage**. Isso deve ser analisado individualmente para cada região **Stage** anotada. Ao concluir que a região Stage pode processar em paralelo diferentes elementos do *stream*, é possível incluir na lista de atributos da anotação **Stage** o atributo **Replicate()**.

No trecho de código a seguir (abaixo), demonstramos a usabilidade da SPar. Observando-se o código, é possível perceber um padrão de linha de montagem, onde cada elemento é manipulado por diferentes estágios independentes. Na linha 1, uma anotação **ToStream** foi utilizada para demarcar o início da região de *stream*. Ela consome dois objetos, inseridos em **Input()**, e, já que nada nessa região será consumido em outras áreas do programa após executar a região anotada, não é necessário um **Output()**. Dentro de um laço de repetição infinito, a leitura dos dados de entrada é feita. Se chegou ao fim, o laço é quebrado na linha 5.

```
1: [[spar::ToStream,spar::Input(entrada_stream, saida_stream)]]
2: while(1){
3:     obj elemento_stream;
4:     elemento_stream = entrada_stream.leitura();
5:     if(elemento_stream.vazio()) break;
6:     [[spar::Stage,spar::Input(elemento_stream),
       spar::Output(elemento_stream),spar::Replicate(4)]]{
7:             //computação sobre o elemento_stream
8:     }
9:     [[spar::Stage,spar::Input(elemento_stream,saida_stream)]]{
10:            saida_stream.escrita(elemento_stream);
11:    }
12: }
```

O trecho de código situado entre **ToStream** e a primeira anotação de **Stage** funciona como o primeiro estágio. Todo **ToStream** possui pelo menos uma anotação

Stage. Logo após, na linha 6, uma anotação **Stage** é utilizada, demarcando o estágio de processamento, que consumirá (**Input**) do estágio anterior e produzirá (**Output**) para o estágio seguinte o objeto *elemento_stream*. Note também que a anotação **Replicate()** foi inserida, tornando a execução de réplicas desse código em paralelo. Ainda na última anotação, um inteiro é passado como parâmetro, delimitando a quantidade de vezes que o estágio será replicado. Caso esse número não seja passado, a variável de ambiente **SPAR_NUM_WORKERS** será utilizada. Por fim, o último estágio, responsável pela escrita é anotado na linha 9, consumindo o elemento processado pelo estágio anterior e o objeto que será gravado o resultado.

4. Exemplo de Uso no Paralelismo de Stream

Considere o seguinte código, capaz de ler um arquivo de entrada e acrescentar um colchetes nas extremidades de cada palavra, para em seguida, escrevê-las em um arquivo de saída.

Com o objetivo de anotar este programa em C++ com a SPar e seguindo o passo a passo da figura que foi explicada anteriormente, o primeiro passo é identificar a região de stream. Observando o código em sem considerar as anotações, é possível definir três diferentes estágios na execução do programa. O primeiro é responsável por ler o arquivo de entrada, o segundo por adicionar as palavras entre dois colchetes e o terceiro por escrever o arquivo de saída. Essa é então a região de *stream*, onde um fluxo de dados passa por uma série de estágios independentes, que realizarão uma operação sobre os elementos do *stream* (tarefas). A anotação **ToStream** na linha 20, marca a região. Note que ela utiliza o corpo do laço de repetição para definir sua extensão. O segundo passo é anotar o que será consumido (**Input**) e produzido (**Output**) por essa região. No entanto, para este caso não há necessidade, pois nada é consumido para dentro da região que vem de um pré-processamento ou produzido internamente para ser usado no pós-processamento.

Em seguida, os estágios devem ser anotados. O primeiro estágio, responsável pela leitura, ocorre entre as linhas 21 e 23. O segundo que é de processamento está na

linha 25, o qual é demarcado por uma anotação **Stage** na linha anterior. Lembre-se que a àrea entre o **ToStream** e a primeira anotação **Stage** corresponde à um estágio. O último estágio realiza a operação de escrita em arquivo na linha 27 do programa, o qual é demarcado por outra anotação **Stage** na linha anterior. O quarto passo é anotar as relações de consumo e produção, se existentes nos estágios. O segundo estágio consome o *elemento_stream* lido anteriormente e após processá-lo, ele o entrega para o estágio de escrita. Portanto, na linha 24 um atributo **Input( )** e outro **Output( )** são necessários, enquanto que na linha 26 apenas **Input( )**.

```
1  #include <iostream>
2  #include <fstream>
3
4  std::ofstream stream_out;
5  std::ifstream stream_in;
6
7  int main(int argc, char const *argv[]){
8      stream_in.open("input.txt",std::ios::in);
9      if (stream_in.fail()){
10         std::cerr << "Error in: " << "input.txt"  << std::endl;
11         stream_in.close();
12         return -1;
13     }
14     stream_out.open("ouput.txt",std::ios::out);
15     if (stream_out.fail()){
16         std::cerr << "Error in: " << "ouput.txt"  << std::endl;
17         stream_out.close();
18         return -1;
19     }
20     [[spar::ToStream]] while(1){
21         std::string stream_element;
22         if(stream_in.eof()) break;
23         stream_in >> stream_element;
24         [[spar::Stage,spar::Input(stream_element),spar::Output(stream_element)]]
25         { stream_element = "[" + stream_element + "]\n"; }
26         [[spar::Stage,spar::Input(stream_element)]]
27         { stream_out << stream_element; }
28     }
29     stream_in.close();
30     stream_out.close();
31     return 0;
32 }
```

Por fim, o quinto e último passo seria verificar a possibilidade de replicar um estágio. Nesse exemplo, as palavras pode ser processada de forma independente. Então, o atributo **Replicate()** poderia ser adicionado na anotação **Stage** da linha 24. Desta forma, cada réplica deste estágio estaria adicionando colchetes em diferentes palavras. Caso a ordenação de escrita necessita ser preservada, a *flag* **-spar_ordered** deve ser utilizada na compilação.