30 de Abril de 2020

1 Lazy Evaluation ou avaliação preguiçosa

A avaliação preguiçosa é uma técnica de programação que atrasa a execução de um código, até que o seu valor seja necessário e evita chamadas repetidas. Como normalmente não partilhamos o resultado das nossas operações entre threads/funcões, estamos a sobrecarregar o sistema ao executar as mesmas operações varias vezes, resultando numa perda de performance, no pior dos casos tendo um tempo exponencial.

Como por exemplo, no case de um algoritmo que recorra ao calculo do produto entre 2 matrizes A e B com alguma ou muita frequência, a avaliação preguiçosa propõe calcular uma única vez o produto das matrizes e reutilizar o resultado sempre que o produto seja utilizado. Deste modo evita-se o custo computacional associado a repetição da mesma operação, o que contribui para o aumento da performance.

No entanto, pode dar-se o caso do resultado não ser necessário, o que consome desnecessariamente os recursos computacionais. Assim sendo o cálculo é atrasado até que seja objectivamente necessário, para tal, criamos uma função para calcular o produto entre duas matrizes:

```
auto P = [A, B] {
    return A * B;
};
```

No contexto de programação paralela, onde o resultado de operações pode ser reutilizado por múltiplas threads, pretende-se partilhar esse valor para que outros fios de execução tenham acesso a esse resultado, evitando cálculos desnecessários.

2 Laziness in C++

Em C++ temos de ser nós a programar dessa forma garantindo uma avaliação preguiçosa. Para isso criamos um **template** com uma função que é a operação de se pretende executar, uma flag que indica se a operação ja foi calculada e por fim o resultado da função. O processo associado a utilização deste template

denomina-se por *memorização*, visto que os resultados são memorizados.

- Função.
- Uma flag que indica se já calculamos o resultado da função.
- O resultado calculado.

```
template <typename F>
class lazy_val {
private:
    F m_computation;
    mutable bool m_cache_initialized;
    mutable decltype(m_computation()) m_cache;
    mutable std::mutex m_cache_mutex;

public:
    ...
};
```

É necessária a utilização de um mutex para evitar a execução paralela da função. As variáveis são declaradas como *mutable* e não *conts* pois eventualmente vamos ter de alterar esse valores. Como a dedução automática para tipos de tamplates só é suportada a partir do C++17, temos de ter uma abordagem diferente. Na maioria das vezes, não podemos especificar explicitamente o tipo da função pois pode ser definindo por "lambdas", e não é posível determinar o seu tipo. É necessário criar uma função auxiliar para que a dedução dos argumentos da função em tamplates seja automática, assim podemos usar o modelo principal com compiladores que não suportam C++17.

```
template <typename F>
class lazy_val {
private:
    ...

public:
    lazy_val(F computation)
        : m_computation(computation)
        , m_cache_initialized(false)
{
}
};

template <typename F>
inline lazy_val<F> make_lazy_val(F&& computation)
```

```
{
    return lazy_val<F>(std::forward<F>(computation));
}
```

O construtor precisa de armazenar a função e definir como *false* a flag que indica se já calculamos o resultado.

NOTA Como não sabemos o tipo de retorno da função, temos de garantir que o membro m_cache tenha um construtor por defeito.

Para finalizar falta criar a função responsável pelo calculo e/ou retorno do valor da operação. A representação em C++ traduz-se na aplicação do mutex no inicio da função, para evitar acesso simultâneo a cache, seguido da verificação da condição da inicialização e da execução da operação caso a cache não tenha sido inicializada. A função retorna o valor calculado.

```
template <typename F>
class lazy_val {
private:
    ...

public:
    ...

    operator const decltype(m_computation())& () const
    {
        std::unique_lock<std::mutex>
            lock{m_cache_mutex};

        if (!m_cache_initialized) {
            m_cache = m_computation();
            m_cache_initialized = true;
        }

        return m_cache;
    }
};
```

Como se pode observar este algoritmo não é óptimo, pois apesar de o mutex ser necessário apenas na primeira execução este é utilizado em todas as execuções.

Uma solução alternativa ao mutex é a utilização de um operador de casting que permite limitar a execução de uma porção do código uma única vez por instancia. A função correspondente a operação casting na biblioteca padrão é std :: call_once:

```
template <typename F>
class lazy_val {
private:
    F m_computation;
    mutable decltype(m_computation ()) m_cache;
    \textbf{mutable std::once_flag m_value_flag;}

public:
    ...
    operator const decltype(m_computation())& () const
    {
        std::call_once(m_value_flag, [this] {
            m_cache = m_computation();
        });
    return m_cache;
    }
};
```

Assim substituímos o mutex pela função std::call_once que recebe como argumentos a flag e a operação que deve executar uma única vez. A flag será actualizada automaticamente pela operação de casting.

3 Laziness as an optimization technique

...