Programação com Threads

Raquel Lopes de Oliveira[1]

Resumo
Este é um relatório desenvolvido para a disciplina de Programação Corrente, período 2017.2, professor Everton Ranielly de Sousa Cavalcante. O relatório consiste na análise do registro dos tempos de execução realizando o mesmo o objetivo, multiplicação de matrizes, mas comparando a abordagem sequecial e concorrente.

Palavras-chave
Threads — Programação — Concorrência

[1] 2013023946

1. Introdução

O propósito desse trabalho é realizar um teste para validar ou não a intuição em relação ao uso da concorrência. Para isso devemos fazer testes fazendo uso da concorrência e uso da solução sequencial nas mesmas condições e com diferentes parâmetros(por exemplo, número de threads).
No presente trabalho o algoritmo implementado foi o da multiplicação de matrizes.

2. Implementação

As implementações dos códigos foram feitas na linguagem *C++*. Para computarmos o tempo, foi usada a biblioteca chrono[1]. Uma classe *Matrix* foi criada para representar uma matrix, uma versão dessa classe já tinha sido previamente parcialmente implementada para outra disciplina e pode ser verificada no repositório numerical-analysis no github, essa classe faz uso de template e já possui diversos métodos, que fazem jus ao nome da classe, implementados.
A multiplicação do algoritmo sequencial foi feita usando a sobrecarga do operador * e para o algoritmo que faz uso de concorrência foi criado dois métodos, um responsável por fazer a atualização da matriz para determinados elementos da matriz resultante e outro(multiply) responsável por chamar o método anterior(multiplyAtomic) dado o número de threads que for ser usado.
O número de threads que é aceito pelo programa é no mínimo 1 e no máximo igual ao número de linhas e colunas das matrizes, dado o fato do procedimento que é realizado para a multiplicação de matrizes. Caso o número de threads seja menor que o número de linhas[2], então o número de operações atômicas(multiplicação de uma linha por uma coluna que resulta em uma célula da matriz produto) é determinado pelo seguinte cálculo:

$$nb\_op = \frac{num\_linhas \times num\_colunas}{num\_threads}$$

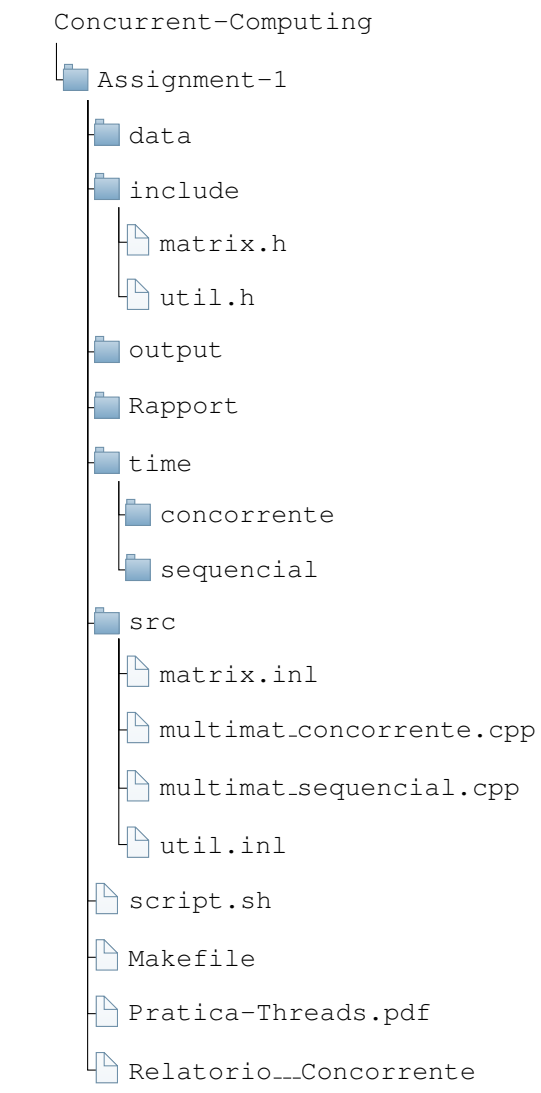
Para a última thread o cálculo tem uma alteração para que ela possua o valor do restante das operações(atômicas) que não foram realizadas:

$$nb\_op = \frac{num\_linhas \times num\_colunas}{num\_threads} + (num\_linhas \times num\_colunas)\%num\_threads$$

Ao usar threads, usamos a chamada *join()* de forma que possamos garantir que tudo já tenha sido computado antes que o programa chegue ao fim, ou seja, é a forma que temos para confirmar que todo o fluxo execução destinado aquela thread foi computado. Caso tivéssemos usado a chamada *detach()* não poderíamos garantir isso, normalmente esse mecanismo é indicado para coisas que queremos que rode em *background*.

[1] http://www.cplusplus.com/reference/chrono/
[2] ou colunas já que se trata de uma matriz quadrada

2.1 Organização



Os arquivos de entrada devem estar necessariamente dentro da pasta **data**, os output das matrizes produtos serão criados na pasta **output** e dentro da pasta **time** os outputs caso execute o script.sh

2.2 Como executar

Uma vez dentro do diretório *Assignment-1* basta excutar o comando:

```
$ make
```

que será criado dois executáveis: **multimat_sequencial** e **multimat_concorrente**. Ele funciona da mesma forma como descrita na especificação do projeto: "O programa principal deverá ser executado via linha de comando da seguinte forma:

```
$./multimat_sequencial 2
```

em que o número inteiro seguido do nome do programa representa a dimensão das matrizes quadradas que serão tratadas pelo programa. Todas as matrizes utilizadas como casos de teste para este trabalho possuem dimensões como potências de base 2, logo qualquer valor fornecido como argumento de linha de comando ao programa deve atender a essa restrição. No caso da solução concorrente, o programa principal deverá ser executado via linha de comando da seguinte forma:

```
$ ./multimat_concorrente 2 2
```

em que os números inteiros seguidos do nome do programa representam, respectivamente, a dimensão das matrizes quadradas que serão tratadas pelo programa e o número de threads a serem utilizadas."No caso da solução concorrente, caso não seja definido o número de threads ele vai setado com o valor máximo (definido pela constante **NUMBER_THREADS**(10) e caso a dimensão da matriz seja menor que a constante, o número de threads é igual ao valor máximo(número de colunas/linhas) dado que a a sub-tarefa mínima para uma thread é o cálculo de um elemento da matriz produto, ou seja, a multiplicação de uma linha por uma coluna.
Os documentos de entrada devem estar dentro da pasta *data* e as matriz resultante da multiplicação se encontra na pasta *output*, ambas as pastas já estão previamente criadas e é possível verificar a hierarquia na seção 2.1.

3. Metodologia

A máquina usada para a realização de testes tem as seguintes características:

Processador : 2,7 GHz Intel Core i5
Memória RAM : 16Go 1867 MHz DDR3
Sistema Operacional : macOS Sierra - version 10.12.16

As implementações foram feitas em C++11.
Compilador: g++
Foi dado prioridade máxima para o programa na realização dos testes, além do uso da flag -O3 para (suposta) otimização. O tempo de fato diminuiu usando a flag, porém depois li algumas críticas que me fizeram repensar no seu uso, tais como: *Compiling with -O3 is not a guaranteed way to improve performance, and in fact in many cases can slow down a system due to larger binaries and increased memory usage. -O3 is also known to break several packages. Therefore, using -O3 is not recommended.*

4. Resultados

Os resultados foram lançados numa planilha para realizar a devida análise, a planilha pode ser encontrada neste link(clique aqui).

Tabela 1. Tabela - Resultados

Dimensão	Número de Treads	(em milisegundos)			
		Min	Max	Média	Desvio padrão
4x4	0	0,41	0,58	0,45	0,04
	1	0,48	0,94	0,55	0,12
	2	0,53	1,14	0,59	0,14
	4	0,55	1,75	0,62	0,25
8x8	0				
	1				
	2				
	4				
	8				
16x16	0				
	1				
	2				
	4				
	8				
	16				
32x32	0				
	1				
	2				
	4				
	8				
	16				
	32				
64x64	0				
	1				
	2				
	4				
	8				
	16				
	32				
	64				

Tabela 2. Tabela - Resultados

Dimensão	Número de Treads	(em milisegundos) Min	Max	Média	Desvio padrão
128x128	0				
	1				
	2				
	4				
	8				
	16				
	32				
	64				
256x256	0				
	1				
	2				
	4				
	8				
	16				
	32				
	64				
512x512	0				
	1				
	2				
	4				
	8				
	16				
	32				
	64				
1024x1024	0	7.714,75	17.958	16.488,75	3.516,56
	1	10.934,89	15.776	14.823,35	964,57
	2				
	4				
	8				
	16				
	32				
	64	9.726,19	10.330,80	9.847,31	122,69
2048x2048	0				
	1				
	2				
	4				
	8				
	16				
	32				
	64				

5. Discussão

Claramente fiz algo errado, pois os dados dos testes não me parecem corretos.
Mas o número de threads foi confirmado através do *Monitor de atividade* do MacOS. Talvez o desempenho não melhore tanto depois de quatro threads por alguma limitação/configuração para o número de threads que podem ser executadas em paralelo.
Para as matrizes com dimensão menor que 256x256, eu não vi muita vantagem no uso de concorrencia. A partir da dimensão 256x256 a configuração que apresentou o melhor desempenho foi usando o parâmetro ideal para as threads igual a 4. Ainda sim não achei a melhora tão alta quanto eu tinha idealizado antes dos testes.