# ESTUDO SOBRE PARALELIZAÇÃO DE MULTIPLICAÇÃO DE MATRIZES UTILIZANDO OPENMP

**Fabiano Cassol de Vargas**
Acadêmico do Curso de Ciência da Computação da Universidade Federal do Pampa
cassol.fabiano@gmail.com
**Márcia Cristina Cera**
Professora do Campus Alegrete da Universidade Federal do Pampa
marciacera@unipampa.edu.br

***Resumo.*** *Atualmente, processadores multi-core são comuns e estão cada vez mais sendo otimizados. Este trabalho apresenta um estudo realizado sobre a multiplicação de matrizes paralelizada com OpenMP em um computador com arquitetura multi-core, visando obter um melhor desempenho da aplicação. São descritos aqui o funcionamento das diretivas, cláusulas e construtores OpenMP utilizados e como foram aplicados ao problema abordado. Com isso, analisamos o impacto da paralelização sobre a aplicação, sendo que a mesma apresentou um desempenho final cerca de 3,6 vezes mais rápido que a versão sequencial.*

***Palavras-chave:*** *OpenMP. Multiplicação de matrizes. Desempenho.*

## 1. INTRODUÇÃO

Nos anos 70, surgiram as primeiras arquiteturas paralelas de computador devido à necessidade de se obter melhores desempenhos em tarefas computacionais, as quais demandam muito tempo de processamento. Com o passar dos anos – e décadas – os avanços tecnológicos e estudos nessa área promoveram grandes avanços. A programação paralela visa obter o melhor desempenho na execução de problemas computacionais em arquiteturas paralelas. Tendo em vista isso, deve-se observar as diferentes arquiteturas paralelas e o modo de programa-las. Conforme Chapman et al. (2008), a interface de programação OpenMP foi desenvolvida para permitir programação paralela em memória compartilhada. Isso se aplica aos processadores multi-core – são arquiteturas de memória compartilhada que possuem, em um único chip de processador, várias unidades de processamento que compartilham a mesma memória - presentes na maioria dos computadores hoje em dia, e o OpenMP nos permite explorá-los a fim de obter melhores desempenhos das aplicações, visto que possuem um grande potencial de processamento.

OpenMP apresenta um modelo de programação *Fork-Join*. Nele, a execução inicia com um fluxo de execução principal e em certos momentos criam-se outros fluxos de execução que passam a executar concorrentemente, ou seja, em paralelo. Para isso, acontece um *Fork*: o OpenMP cria *threads* que executam o mesmo trecho de código - que se trata da região paralela do programa e que demanda maior trabalho/tempo computacional. Ao fim da região paralela compartilhada pelas *threads* acontece um *Join*, onde os resultados computados paralelamente são reunidos para compor o resultado final.

O objetivo deste trabalho é analisar o desempenho da paralelização da multiplicação de matrizes com OpenMP. Mais especificamente, verificaremos o impacto causado no tempo de execução da aplicação quando varia-se o modo como a carga de trabalho é distribuída entre as *threads* por meio dos diferentes tipos de políticas de distribuição de carga.

O restante do artigo encontra-se estruturado da seguinte forma. Primeiramente, é feita uma contextualização acerca do problema abordado e da interface de programação paralela escolhida para melhorar o seu desempenho. Em seguida, são apresentadas as decisões tomadas na paralelização do código. Continuando, há uma apresentação e análise dos resultados obtidos nos testes realizados, finalizando com as conclusões e planos para trabalhos futuros.

## 2. CONTEXTUALIZAÇÃO

### 2.1. Multiplicação de matrizes

A multiplicação de matrizes foi escolhida devido a sua grande carga de trabalho, principalmente para matrizes de grande ordem. A principal característica da aplicação que multiplica matrizes quadradas é que os cálculos da matriz resposta são feitos em laços aninhados. O código da aplicação foi escrito na linguagem de programação C.

### 2.2. OpenMP

O OpenMP possui um construtor paralelo (*#pragma parallel*), o qual define que o processo será executado em paralelo pelas *threads* (Chapman et al. 2008). Sua sintaxe pode ser vista abaixo:

```
#pragma omp parallel [clause[[,]clause]...]
        structured block
```

Outro construtor OpenMP é o construtor de laço paralelo. Ele é utilizado para que as iterações de um *loop* sejam executadas em paralelo quando distribuídas entre as *threads*. Ressalta-se que, em linguagem C, esse construtor deve ser usado em laços onde o número de iterações pode ser contado, e esse número deve aumentar ou diminuir em uma quantidade fixa a cada iteração até atingir o limite estabelecido. A sintaxe do *loop construct*, combinada com o construtor paralelo, encontra-se abaixo:

```
#pragma omp parallel for[clause[[,]clause]...]
        for-loop
```

O OpenMP possui algumas cláusulas para controle de construtores paralelos, e dentre elas está a cláusula *private*. Esta cláusula define uma lista de variáveis que serão privadas para cada *thread.* Sua sintaxe é *private(list).* O que acontece é que esta cláusula cria uma cópia das variáveis de sua lista para cada *thread*, que então podem manipular seus valores apenas em seu escopo, sem alterar os valores das variáveis das demais *threads*.

Outra cláusula do OpenMP é a *schedule*, que é utilizada apenas no construtor de laço paralelo. Sua função é controlar o modo como as iterações do laço são distribuídas entre as *threads*. A sua sintaxe é:

```
schedule(tipo[, tamanho_chunk])
```

Analisando a sua sintaxe, vemos que deve ser definido o tipo de *schedule* e um tamanho de *chunk*. O *chunk* é um subconjunto contíguo de iterações do laço a ser paralelizado. O tamanho do *chunk* define a quantidade de iterações do laço que serão atribuídas para uma *thead* por vez – definir o tamanho do *chunk* é opcional. A seguir, são explicados os três tipos de *schedule* utilizados:

a) *static*: se o tamanho do *chunk* for definido, os *chunks* serão atribuídos às *threads* estaticamente seguindo uma ordem cíclica baseada nos números das *threads* (0, 1, 2, …, n-1). Quando o tamanho do *chunk* não é definido, cada *thread* receberá apenas um *chunk* de iterações, que terá aproximadamente o mesmo tamanho para cada *thread,* dependendo da quantidade de iterações a serem divididas entre *elas*. A atribuição dos *chunks* é sempre feita na ordem dos números das *threads* em tempo de compilação.

b) *dynamic*: os *chunks* são atribuídos às *threads* dinamicamente em tempo de

execução à medida que estas estão aptas a receberem a carga de trabalho. Cada *thread* executa as suas respectivas iterações e, assim que as termina, requisita outro *chunk*, até que não haja mais trabalho a ser distribuído. Quando o tamanho do *chunk* não é especificado, o padrão é 1.

c) *guided*: a distribuição dos *chunks* ocorre do mesmo modo do tipo *dynamic*. Porém o tamanho dos *chunks* não é constante. Para um tamanho de *chunk* definido por um valor *k* (*k* >= 1), o tamanho de cada *chunk* será o equivalente ao número de iterações ainda não atribuídas para alguma *thread* dividido pelo número de *threads*. O valor *k* é o tamanho mínimo que um *chunk* pode assumir, e o *chunk* decresce até o limite *k*. Quando o tamanho do *chunk* não é especificado, o padrão é 1.

## 3. PARALELIZAÇÃO DA MULTIPLICAÇÃO DE MATRIZES

Sendo que no código sequencial da aplicação o produto das matrizes é realizado por uma combinação de laços de repetição aninhados, percebeu-se que ali se concentrava quase todo o tempo de processamento da aplicação. Assim foi definida a região paralela, a qual abrange o trecho de código do laço de repetição mais externo, utilizando o construtor paralelo. Para paralelizar o laço e permitir a distribuição das iterações entre as *threads*, foi utilizado o construtor de laço.

Visto como a multiplicação das matrizes ocorre, e sendo que cada laço possui uma variável para contagem das iterações, foi necessário adicionar a cláusula *private* ao final da linha que estabelece a região do laço paralelo para que cada *thread* mantenha em seu escopo uma cópia das variáveis contadores de cada laço. Se não houvesse esse controle das variáveis privadas, todas as *threads* teriam acesso aos mesmos dados e, assim, iriam alterar indevidamente seus valores, prejudicando a execução e correção do produto de matrizes.

O próximo passo do processo de paralelização do código foi aplicar a cláusula *schedule* para analisar o desempenho das diferentes políticas de distribuição de carga de trabalho.

## 4. RESULTADOS EXPERIMENTAIS

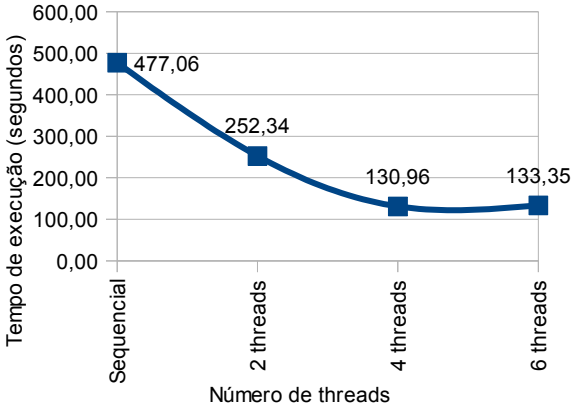
Todos os testes foram realizados em um computador Intel® Core™ i5-2310 CPU @ 2.90GHz × 4 *Cores,* L1 32kB, L2 256kB, L3 6144kB*,* 8GB RAM, Sistema Operacional GNU/Linux Ubuntu 12.04 LTS. A fim de se obter dados fieis, cada versão da aplicação foi executada 30 vezes. Vale ressaltar que os tempos de execução apresentados correspondem apenas à região paralelizada do código. Os tempos apresentados são as médias aritméticas simples das amostras, com um desvio padrão médio em torno de 0,97 segundos.

Os primeiros testes executados sobre o código sequencial serviram para definir a ordem das matrizes. As duas matrizes foram definidas com dimensões de 3000 x 3000 – pois a aplicação apresentou um tempo de execução considerável para ser paralelizada - e seriam inicializadas com números gerados por um cálculo qualquer de somas e multiplicações.

A seguir, comparamos o tempo de execução da aplicação sequencial (sem nenhuma diretiva OpenMP de paralelização) com os tempos da aplicação paralelizada com diferentes números de *threads* – 2, 4 e 6 *threads*. Até então não foi aplicada a cláusula *schedule*, portanto a política de distribuição das iterações do laço entre as *threads* foi a padrão do compilador. Com isso foi constatado um melhor desempenho na versão de 4 *threads*, como visto na Fig. 1. Tal resultado se deve ao fato de que a aplicação foi testada em um processador de 4 *cores* (núcleos de processamento), sendo que cada *core* ficou responsável pela execução de uma *thread*. À medida que se aumenta o número de *threads* aproximando-o do número de *cores* o desempenho

melhora. Quando o número de *threads* ultrapassa o de *cores* a tendência é deixar de ganhar desempenho, pois haverão mais *threads* concorrendo a uma unidade de processamento, fato que prejudica o desempenho da aplicação.

Figura 1. Tempos de execução da aplicação com diferentes números de *threads*



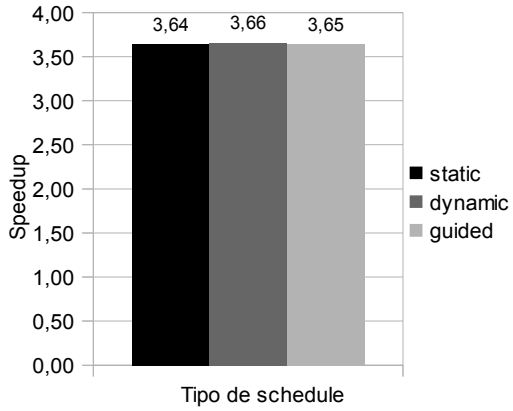

Também deve-se observar o quão mais rápidos foram os tempos das versões paralelas em relação à versão sequencial. Para isso, a Tabela 1 apresenta os *speedups* obtidos.

Tabela 1. Speedups obtidos com diferentes números de *threads*

| Versão | Tempo (seg.) | *Speedup* |
|---|---|---|
| Sequencial | 477,06 | |
| 2 *threads* | 252,34 | **1,89** |
| 4 *threads* | 130,96 | **3,64** |
| 6 *threads* | 133,35 | **3,58** |

O próximo passo foi utilizar a cláusula *schedule* para verificar qual das políticas de distribuição de carga se mostraria mais eficaz. Para isso, escolheu-se a versão com 4 *threads* por esta ter se mostrado a mais eficiente. Os *speedups* resultantes desta etapa são mostrados na Fig. 2, onde se vê que o desempenho foi muito próximo para todas as cláusulas *schedule*. Isto justifica-se pelo fato de que o tempo de computação das iterações do laço paralelizado são muito similares entre si. Logo, qualquer um dos tipos de *schedule* pode ser utilizado na multiplicação de matrizes, pois a diferença entre definir ou não a política de distribuição de carga é insignificante neste caso.

Figura 2. *Speedups* da aplicação com 4 *threads* variando o tipo de *schedule*




## 5. CONCLUSÃO

Inicialmente, foi estudada uma forma adequada de paralelizar a aplicação para obter um melhor desempenho, passando por várias versões da mesma.

Após a análise dos resultados, nota-se que todas as políticas de distribuição de carga de trabalho apresentaram desempenho muito semelhante. Assim, em trabalhos futuros, pretende-se realizar mais testes definindo tamanhos variados para o *chunk* de iterações e abordar mais de uma aplicação.

## 6. REFERÊNCIAS

CHAPMAN, B.; JOST, G.; VAN DER PAS, R. **Using OpenMP: portable shared memory parallel programming**. Cambridge, Massachusetts, USA: The MIT Press. 2008

The OpenMP® API Specification for Parallel Programming. **OpenMP**. Disponível em: <http://openmp.org/wp/>. Acesso em 20 jul. 2013.

CERA, M.; MAILLARD, N. Programação Paralela Avançada. In: 10ª Escola Regional de Alto Desempenho, 2010, Passo Fundo.