UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Gustavo Teixeira Perche Mahlow

Paralelização da função de avaliação aplicada ao problema de escalonamento distribuído de tarefas

Uberlândia, Brasil

2018

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Gustavo Teixeira Perche Mahlow

Paralelização da função de avaliação aplicada ao problema de escalonamento distribuído de tarefas

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Sistemas de Informação.

Orientador: Paulo Henrique Ribeiro Gabriel

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Sistemas de Informação

Uberlândia, Brasil

2018

Dedico este trabalho a minha família, aos meus amigos e a todos que de alguma forma contribuíram para meu crescimento pessoal e profissional.

Resumo

Este trabalho de conclusão de curso tem como objetivo o desenvolvimento de uma API para execução assíncrona de operações com matrizes, na linguagem de programação Java. Essa API é necessária para a resolução eficiente de operações matemáticas relacionadas a um modelo de otimização de escalonamento de tarefas em sistemas distribuídos (GABRIEL, 2013). Nessa monografia, é descrito o processo de paralelização e são apresentados resultados comparando a abordagem proposta com uma implementação sequencial. Resultados indicam que a implementação proposta apresenta melhor desempenho quando comparada a outras versões e, principalmente, com a implementação sequencial.

Palavras-chave: Java, Multithreading, Matriz

Lista de abreviaturas e siglas

JDK	Kit de desenvolvimento Java, necessário para a criação de aplicações com a linguagem
API	Application Programming Interface. É um conjunto de implementações e funcionalidades fornecidas por um terceiro, para serem incorporadas em um outro software
SPC3 PM	Serial, Parallel, Concurrent Core to Core Programming Model
OpenMP	Open Multi-Processing
MPI	Message Passing Interface

Sumário

1 Introdução

1.1 Contextualização e Motivação

A limitação física de recursos de hardware e a demanda por computadores cada vez mais poderosos, juntamente com o barateamento dos processadores e recursos de rede, motivou a criação de sistemas computacionais distribuídos. Com essa tecnologia, em vez da utilização de um único computador com alta capacidade de processamento, há a utilização de diferentes computadores com poder computacional inferior (DONG; AKL, 2006). Segundo van Steen e Tanenbaum (2017) a definição de sistema distribuído é dada como uma coleção de computadores independentes que aparece ao seus usuários como um único sistema coerente.

O desenvolvimento de sistemas distribuídos, porém, possui um grau de complexidade maior, visto que os computadores estarão distribuídos pela rede, não necessariamente no mesmo local geográfico. Além disso, há muitos cenários possíveis de erros, os quais, por sua vez, podem acarretar em atraso de processamento (DONG; AKL, 2006).

Com o intuito de evitar os atrasos de processamento, modelos e algoritmos de escalonamento de tarefas têm sido propostos (KAYA; UÇAR, 2009; ELLOUMI, 2004). Nesse contexto, as tarefas (ou seja, conjuntos de aplicações paralelas) são distribuídas de acordo com a capacidade de processamento de cada computador, levando em consideração, também, características como largura de banda da rede, atraso de comunicação e carga de trabalho gerada pela respectiva tarefa (KAYA; UÇAR, 2009).

A fim de permitir uma distribuição adequada de carga entre os computadores, diversos estudos focam no desenvolvimento de modelos de otimização matemática com o objetivo de representar diferentes cenários do mundo real. Assim, esses modelos buscam capturar características de tarefas e recursos computacionais, as quais são representadas por uma função objetivo e um conjunto de restrições.

Uma vez estabelecido o modelo, é necessário executar um algoritmo de otimização. No entanto, os modelos gerados tendem a ser computacionalmente complexos, demandando cálculos computacionais dispendiosos (GABRIEL, 2013). Isso se deve ao fato de que esses cálculos baseiam-se em operações com matrizes de grande dimensão. Em experimentos preliminares (GABRIEL, 2013; SOARES, 2018) observou-se que esses cálculos consomem grande parte do tempo de processamento. Por esse motivo, a paralelização dessas operações torna-se importante, de modo a reduzir o alto custo computacional de sua avaliação.

A ideia de paralelização consiste, portanto, em executar mais de uma tarefa ao

mesmo tempo, em núcleos diferentes do processador, reduzindo assim, o tempo de execução (PACHECO, 2011).

1.2 Objetivos

O objetivo deste trabalho é desenvolver uma interface de programação de aplicações (API, do inglês *application programming interface*), escrita na linguagem de programação Java, utilizando somente ferramentas já embutidas na plataforma, para a realização de operações de multiplicação e divisão de matrizes, de forma assíncrona. A ideia é que a solução proposta por esse projeto possa ser utilizada, futuramente, para a resolução das operações matemáticas baseadas no modelo proposto por Gabriel (2013), que aborda o escalonamento de tarefas em sistemas distribuídos e a descoberta do escalonamento ótimo.

A API desenvolvida é, então, comparada, em termos de tempo de execução, com outras implementações e modelos propostos por outros autores. No desenvolvimento desta monografia, utilizou-se referências à linguagem Java, a bibliotecas do sistema UNIX, bem como artigos com proposição de modelos obtidos a partir do site *Research Gate*[1] e implementações em repositórios abertos do site GitHub[2].

1.3 Organização da Monografia

Esta monografia encontra-se organizada da seguinte maneira: no Capítulo 2 são descritos conceitos adotados e trabalhos correlatos; no Capitulo 3 são mostrados detalhes da implementação da API; no Capítulo 4 mostram-se os experimentos e resultados deste trabalho; finalmente, no Capítulo 5 apresentam-se conclusões e trabalhos futuros relacionados a esta monografia.

[1] <https://www.researchgate.net/>
[2] <https://github.com/>

2 Referencial Teórico

2.1 Conceitos Adotados

2.1.1 Escalonamento de Tarefas

A necessidade por computação de alto desempenho, assim como o crescente tamanho de centros de armazenamento de dados (do inglês, *datacenters*), tem motivado estudos sobre algoritmos eficientes de escalonamento em sistemas distribuídos (XHAFA; ABRAHAM, 2009).

Conceitualmente, escalonamento de tarefas consiste na alocação eficiente de aplicações paralelas sobre os recursos disponíveis em um sistema distribuído (XHAFA; BAROLLI; DURRESI, 2007). A motivação principal para o estudo do escalonamento é o desejo de aumentar a velocidade na execução de uma carga de trabalho. Partes da carga de trabalho, chamadas tarefas, podem ser espalhadas por vários processadores (ou computadores) e, portanto, serem executadas mais rapidamente do que em um único processador. O problema do escalonamento de processos em sistemas multiprocessados pode ser geralmente formulado pela questão (XHAFA; ABRAHAM, 2009): "Como podemos executar um conjunto de tarefas T em um conjunto de processadores P sujeito a algum conjunto de critérios de otimização C?"

O objetivo mais comum do escalonamento é minimizar o tempo de execução esperado de um conjunto de tarefas. Exemplos de outros critérios de agendamento incluem a minimização do custo, diminuindo o atraso na comunicação, dando prioridade aos processos de determinados usuários, ou necessidade de dispositivos de hardware especializados. A política de escalonamento para um sistema multiprocessado geralmente incorpora uma combinação de vários desses critérios (XHAFA; BAROLLI; DURRESI, 2007).

Devido à alta complexidade do problema de escalonamento não é possível determinar uma solução ideal porque o tempo gasto para explorar todo o espaço de pesquisa não é razoável. Dessa forma, é necessário aproximar métodos para encontrar uma boa solução com alta eficiência computacional em vez de algoritmos que realizam buscas exaustivas e exigem um grande tempo computacional.

Nesse contexto, diversos trabalhos têm focado na modelagem do problema, como forma de buscar algoritmos de otimização eficientes. Kaya e Uçar (2009) descrevem o escalonamento de tarefas como sendo um caso especial do problema de designação quadrática (ELLOUMI, 2004). Formalmente, esse problema é modelado da seguinte maneira: seja um conjunto de n tarefas, um conjunto de m processadores, um custo de execução

e_{ij} da tarefa i sobre a máquina j e um custo de comunicação $c_{ijk\ell}$ entre as tarefas i e j se elas forem atribuídas, respectivamente, aos processadores k e ℓ. O objetivo é encontrar uma atribuição de tarefas sobre processadores tal que o custo total de processamento e execução seja minimizado. Tem-se, assim, o seguinte modelo (Equação (2.1)):

$$\min F(x) = \sum_{i=1}^{n}\sum_{j=1}^{m} e_{ij}x_{ij} + \sum_{i=1}^{n-1}\sum_{j=1}^{m}\sum_{k=i+1}^{n}\sum_{\ell=1}^{m} c_{ijk\ell}x_{ij}x_{k\ell} \tag{2.1}$$

sujeito a:

$$\sum_{j=1}^{m} x_{ij} = 1, \quad i = 1, \ldots, n \tag{2.2}$$

$$x_{ij} \in \{0, 1\}, \quad i = 1, \ldots, n, \quad j = 1, \ldots, m \tag{2.3}$$

Nesse modelo, a variável x_{ij} assume valor 1 se, e somente se, a tarefa i for atribuída ao processador j. As restrições (2.2) e (2.3) garantem uma atribuição válida.

Posteriormente, Gabriel (2013) estendeu esse modelo, adicionando novos componentes, de acordo com a Equação (2.4):

$$\min F(x) = \sum_{i=1}^{n}\sum_{j=1}^{m} \frac{x_{ij} \cdot w_i}{c_j} + \sum_{i=1}^{n}\sum_{h=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{m} \frac{x_{ij} \cdot d_{ih} \cdot x_{hk}}{b_{jk}} \tag{2.4}$$

Nessa nova versão, w_i representa a carga de trabalho das tarefas, c_j a capacidade de CPU dos processadores, d_{ih} o total de dados transmitidos entre as tarefas i e h e b_{ik} é a capacidade de comunicação (largura de banda) entre os processadores i e k. As restrições são as mesmas mostradas anteriormente.

No contexto desse trabalho, Soares (2018) implementou um algoritmo genético (GOLDBERG, 1989) com o objetivo de encontrar o melhor valor para esse modelo. No entanto, o algoritmo implementado demandou longo tempo de processamento para avaliar a Equação (2.4), uma vez que, na prática, são necessários diversas operações de multiplicação de matrizes (de fato, essas matrizes armazenam informações referentes a comunicação). O alto tempo de processamento requerido pelo algoritmo genético proposto motivou este trabalho de conclusão de curso a implementar um versão paralela da Equação (2.4), como forma de permitir o desenvolvimento de algoritmos mais eficientes.

2.1.2 Programação Paralela

O paradigma da programação paralela surgiu com o intuito de possibilitar a execução de diferentes tarefas simultaneamente em um mesmo computador.

Em 2005 foi lançado o primeiro processador multicore, o *Intel Pentium D*, que possuia dois núcleos de processamento. Até então, a execução de tarefas concorrentes não era possível, devido a processadores com somente um núcleo, além de limitações por parte dos sistemas operacionais.

Segundo Göetz *et al.* (2006), um dos motivos para a evolução dessa área é a necessidade de que, durante operações bloqueantes, como a leitura de entradas, ou a escrita de saídas, o computador possa executar outros programas.

Existem diversas formas de se implementar esse paradigma, sendo que as duas mais comuns utilizam os conceitos de subprocesso e *thread* (BRYANT; O'HALLARON, 2016).

Um subprocesso é um processo criado a partir de um outro processo. O processo que dá origem ao subprocesso é chamado de **processo pai**, e o subprocesso de **processo filho**. No contexto de um sistema operacional, se o processo pai for morto, seus filhos serão mortos também. Na utilização da técnica de subprocesso, todas as áreas do processo pai são duplicadas para os filhos, sendo assim, cada subprocesso possui um contexto independente. Para a comunicação entre subprocessos, devem-se utilizar mecanismos como pipes e memória compartilhada, segundo Bryant e O'Hallaron (2016).

Já a definição de *thread* é dada como um fluxo lógico de execução que é executado dentro do contexto de um único processo, e sua execução é agendada pelo *kernel* do sistema operacional. Diferentemente do subprocesso, as *threads* compartilham a mesma região de memória do processo ao qual foi originada, sendo assim são mais leves (BRYANT; O'HALLARON, 2016) .

A maioria das linguagens de programação modernas possuem bibliotecas que possibilitam a implementação do paradigma paralelo. Algumas possuem funcionalidades embutidas na linguagem, inclusive (THE OPEN GROUP, 1997a; ORACLE, 2018b).

Na linguagem C, por exemplo, existem as bibliotecas *unistd* (THE OPEN GROUP, 1997b) e *pthread* (THE OPEN GROUP, 1997a), implementadas para a utilização de subprocessos e *threads* respectivamente. A *unistd* provê a função *fork* que inicia um subprocesso com o contexto das linhas de código subsequente a chamada. Por outro lado, a *pthread* conta com as funções *pthread_create* e *pthread_join* para, respectivamente, criação e espera do fim de execução de uma *thread* (KERNIGHAN; PIKE, 1984).

Na linguagem Java, existe o pacote *java.util.concurrent* (ORACLE, 2018b), embutido no Java Development Kit (JDK), que conta com utilitários para a implementação de operações de concorrência, além do pacote *java.lang* (ORACLE, 2018a) que conta com classes e interfaces para a criações de *threads*, como a própria classe denominada *Thread* e as interfaces como *Runnable* e *Callable*.

Existem também plataformas que visam a conversão de programas síncronos em assíncronos, como a *OpenMP* (ARB, 2013). Essa biblioteca é uma ferramenta de pré-processamento que analisa o código à procura de trechos específicos, como por exemplo *#pragma omp parallel*, que faz com que o bloco subsequente seja executado paralelamente de forma assíncrona. A *OpenMP* converte essas instruções para chamadas de sua

API que realiza as operações de acordo com a referida instrução. Segundo o manual da *OpenMP* (ARB, 2013), essa API provê diretivas, rotinas e variáveis de ambiente, que, quando explicitadas pelo desenvolvedor, permitem a execução de tarefas em paralelo.

A linguagem Java, adotada para este trabalho, é orientada a objetos, multiplataforma e com uma vasta quantidade de bibliotecas disponíveis. Todas as funcionalidades embutidas na linguagem são documentadas e é possível encontrar facilmente referências de como utilizá-las. Sua API para concorrência, se enquadra em todos os requisitos necessários ao projeto.

Assim, no desenvolvimento deste trabalho de conclusão de curso, são utilizadas as ferramentes de concorrência disponibilizadas no próprio JDK, como citado anteriormente, e o paradigma adotado implementa o uso de *threads*.

2.2 Trabalhos Correlatos

Existem diversos trabalhos que propõem a multiplicação de matrizes de forma paralela. Pode-se citar, por exemplo, os trabalhos de Ismail, Mirza e Altaf (2011), Qasem e Qatawneh (2017) e Myint e Kyi (2014), uma vez que, além de servirem como referência para a implementação deste trabalho, apresentam comparações entre o tempo de execução da implementação assíncrona dos algoritmos com uma versão síncrona dos mesmos. Isso ajuda a ter um embasamento para o resultado que é esperado no projeto.

Ismail, Mirza e Altaf (2011) propõem uma implementação de multiplicação assíncrona de matrizes baseada em uma API denominada SPC^3 *PM*, que se assemelha bastante ao propósito da *OpenMP*. Os autores descrevem funcionalidades para paralelização de tarefas com foco na comparação dos tempos de execução de algoritmos desenvolvidos nas duas ferramentas. A API SPC^3 *PM* teve um resultado superior ao *OpenMP*. Na comparação utilizando 24 *threads*, a ferramenta obteve uma redução de $63,24\%$ no tempo de execução na operação com matrizes de escala 10000×10000.

A implementação demonstrada no trabalho, porém, não se adéqua à proposta deste projeto. O algoritmo foi desenvolvido utilizando um pré-processador de código, enquanto, neste trabalho, optou-se por utilizar ferramentas embutidas na linguagem. Contudo, os resultados demonstrados tem grande valia para este trabalho. A partir deles, foi possível ter uma ideia do desempenho da implementação baseada no *OpenMP*, alvo de comparação de resultados deste projeto.

Qasem e Qatawneh (2017) apresentam duas alternativas para multiplicação de matrizes. A ideia é a criação de uma implementação eficiente para o mundo corporativo. Para isso, é utilizada, como exemplo, uma rotina para cálculo do lucro de uma empresa denominada Market Shop. A primeira proposta é a utilização de uma biblioteca cha-

mada MPI (*Message Passing Interface*) (OPENMPI, 2016), que suporta as linguagens C e Fortran77, para troca de mensagens entre múltiplos computadores executando um programa em paralelo. Assim, particionam-se as tarefas de multiplicação entre diferentes computadores do sistema.

A segunda proposta implementada por Qasem e Qatawneh (2017) utiliza um modelo de programação chamado *MapReduce* (GASPAROTTO, 2014), da plataforma *Hadoop* (APACHE, 2018), que segue os conceitos do paradigma de programação funcional (SÁ; SILVA, 2006) para o particionamento das tarefas entre nós. O conceito de *Map*, consiste na aplicação de uma função a cada elemento de uma lista. Isso gera como resultado uma nova lista com elementos modificados. O *Reduce*, por outro lado, visa transformar os elementos de uma lista em um único elemento. Uma função *Reduce* recebe o elemento anterior e o próximo da lista, e retorna uma única saída. Assim, no final é gerado apenas um elemento. No trabalho apresentado, os dados são particionados em pares, sendo o primeiro elemento um valor da linha da matriz A, e o segundo um valor da coluna da matriz B. As funções *map* e *reduce* são utilizadas para a multiplicação dos pares, e consequente soma, respectivamente. Por fim os autores comparam o desempenho entre as duas propostas com uma implementação *multithread*.

Foram realizados experimentos com matrizes densas, nas quais todo e qualquer valor é armazenado na matriz, e matrizes esparsas, quando somente são armazenados na matriz valores diferentes de zero. Foram utilizados sete servidores.

Como resultado, o modelo desenvolvido utilizando MPI, obteve $80,94\%$ de redução no tempo de execução quando comparado com o modelo sequencial. Para o teste da operação de multiplicação foram utilizadas duas matrizes de escala 4000×4000. O modelo *MapReduce* por sua vez, obteve $65,22\%$.

O cálculo utilizando MPI demonstrou-se mais eficiente do que a implementação do modelo *MapReduce*.

A proposta foi de grande valia para este trabalho, visto que serviu como base para a ideia de um trabalho futuro, que visa o cálculo da multiplicação de matrizes, de forma distribuída, entre vários elementos de processamento.

Finalmente, Myint e Kyi (2014) detalham a multiplicação paralela de matrizes a partir da divisão das operações entre diferentes servidores. Um servidor mestre particiona a matriz A e B entre os servidores escravos. No final, o resultado é agrupado no mestre novamente. Os autores apresentam a comparação entre o tempo de execução do algoritmo com a forma serial (ou sequencial) do mesmo. O conceito do trabalho é bastante próximo ao que é proposto pelo modelo *MapReduce* da plataforma *Hadoop*. A finalidade é a mesma, particionar o cálculo em diferentes servidores, e por fim unir o resultado. Como resultado, foi obtido um ganho $65,69\%$ de redução no tempo de execução quando comparado com

o modelo sequencial.

Todos os trabalhos citados nesta seção, comprovaram, em seus experimentos, que o modelo paralelo é mais eficiente do que o modelo sequencial para cálculos envolvendo matrizes de alta dimensionalidade.

3 Desenvolvimento

3.1 Implementação

A arquitetura da API assíncrona foi baseada no conceito de *callback*. O usuário da API realiza a chamada de uma operação, multiplicação ou divisão matricial, e passa, como argumento: 1) as matrizes para o cálculo; 2) um objeto que implementa a interface de *callback*; e, opcionalmente, 3) a quantidade de *threads*, sendo que o valor padrão desse último argumento é a quantidade de núcleos no processador do cliente. Quando a operação assíncrona for finalizada, o usuário será notificado por meio do objeto que ele passou na chamada, recebendo o resultado da operação.

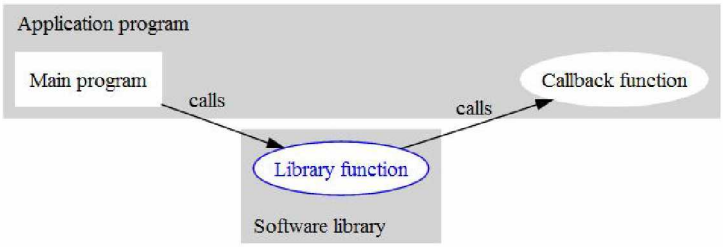

Figura 1 – Esquematização de Callback

3.1.1 Multiplicação de Matrizes

A implementação síncrona da multiplicação foi baseada no Algoritmo 1, mostrado a seguir.

Algoritmo 1 Cálculo da multiplicação de duas matrizes.

Entrada: matrizes A e B
Saída: matriz C resultante da multiplicação $(A \cdot B)$

Condição: A quantidade de colunas da matriz A deve ser a mesma da quantidade de linhas da matriz B.

$l \leftarrow$ *quantidade de linhas da matriz* A
$n \leftarrow$ *quantidade de colunas da matriz* B
$m \leftarrow$ *quantidade de colunas da matriz* A
para $i \leftarrow 0$ **até** l **faça**
  para $j \leftarrow 0$ **até** n **faça**
    para $k \leftarrow 0$ **até** m **faça**
      $C_{ij} \leftarrow C_{ij} + (A_{ik} \cdot B_{kj})$
    fim para
  fim para
fim para

Para a execução assíncrona da multiplicação, o algoritmo subdivide as operações em pequenas tarefas. Essas tarefas por sua vez, são despachadas para uma estrutura de dados denominada *Executor Service*, do pacote *java.util.concurrent* (ORACLE, 2018b). Essa estrutura possui uma fila interna na qual as tarefas são colocadas em espera e um *pool*, ou seja, uma coleção de recursos já inicializados, disponíveis para uso que são alocados e destruídos sob demanda, por meio de *threads*. Assim que uma *thread* for liberada, uma nova tarefa é despachada.

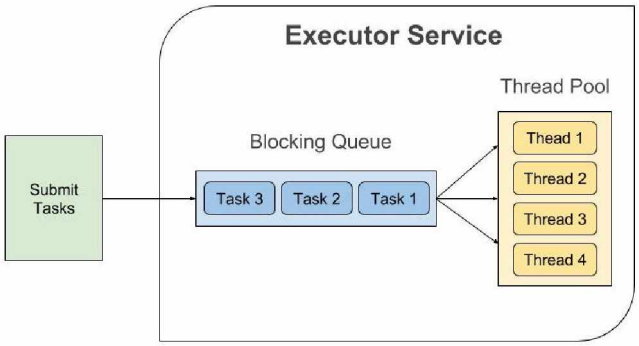

Figura 2 – Esquematização do ExecutorService

O Algoritmo 2 realiza a criação das tarefas de multiplicação e itera a quantidade de linhas da matriz A e colunas da matriz B. Por meio desse algoritmo, é criada uma tarefa de multiplicação para cada índice das colunas. A ideia é a mesma dos dois primeiros laços do algoritmo síncrono (Algoritmo 1).

Algoritmo 2 Subdivisão para multiplicação.

```
l ← quantidade de linhas da matriz A
n ← quantidade de colunas da matriz B
para i ← 0 até l faça
  para j ← 0 até n faça
    Cria tarefa de multiplicação (i, j)
  fim para
fim para
```

A tarefa de multiplicação, por sua vez, realiza o que é representado pelo terceiro laço do Algoritmo 1 (síncrono), ou seja, a iteração pela quantidade das colunas da matriz A. Essa tarefa é representada, em pseudocódigo, no Algoritmo 3.

Algoritmo 3 Tarefa de multiplicação.

```
m ← quantidade de colunas da matriz A
para k ← 0 até m faça
  C_ij ← C_ij + (A_ik · B_kj)
fim para
```

Por fim, assim que todas as tarefas foram executadas, o cliente é notificado a partir do *callback* com a matriz resultante.

3.1.2 Divisão entre Matrizes

A divisão de duas matrizes de mesma dimensão é realizada termo a termo. A implementação da versão síncrona dessa operação foi baseada no Algoritmo 4.

Algoritmo 4 Cálculo da divisão de duas matrizes.

Entrada: matrizes A e B.
Saída: matriz C resultante da divisão $(A \cdot B)$.

Condição: A quantidade de colunas da matriz A deve ser a mesma da quantidade de linhas da matriz B.

```
l ← quantidade de linhas da matriz A
n ← quantidade de colunas da matriz B
para i ← 0 até l faça
  para j ← 0 até n faça
    C_ij ← A_ij / B_ij
  fim para
fim para
```

A operação de divisão assíncrona foi elaborada seguindo os mesmos princípios da multiplicação. A principal diferença é que, agora, não é criada uma tarefa para cada coluna e para cada linha, mas apenas uma tarefa por linha, conforme ilustrado nos Algoritmo 5 e 6.

Algoritmo 5 Subdivisão para divisão.

$l \leftarrow$ *quantidade de linhas da matriz A*
para $i \leftarrow 0$ **até** l **faça**
  Cria tarefa de divisão (i)
fim para

Algoritmo 6 Tarefa de divisão

$m \leftarrow$ *quantidade de colunas da matriz A*
para $j \leftarrow 0$ **até** m **faça**
  $C_{ij} \leftarrow \frac{A_{ij}}{B_{ij}}$
fim para

4 Experimentos e Resultados

Para assegurar que os resultados sejam iguais em ambas as implementações, foram utilizadas as mesmas matrizes para os testes unitários, de dimensões 1000 por 1000, geradas aleatoriamente com números de ponto flutuante com valores entre 1 e 5.

Os experimentos para a avaliação de desempenho foram executados 100 vezes para cada implementação. O computador utilizado nessa avaliação possui um processador Intel Core i5-4690K, 8,0Gb de memória principal e sistema operacional Ubuntu 18.04. A versão do JDK utilizada, para a execução do código Java da API, foi a *8u181*, última versão do Java 8 disponibilizada pela Oracle até então.

4.1 Experimentos Iniciais

4.1.1 Multiplicação de Matrizes

As médias de tempo de execução da multiplicação podem ser observadas na Tabela 1 e no gráfico da Figura 3.

Tabela 1 – Tabela comparativa de tempo médio em segundos para a realização da multiplicação de duas matrizes 1000×1000. O símbolo "-" representa avaliações não realizado uma vez que a implementação síncrona é sequencial e, portanto, utiliza apenas uma *thread*.

	1 thread	5 threads	10 threads	20 threads
Assíncrona	27s	10s	9s	9s
Síncrona	60s	-	-	-

4.1.2 Divisão de Matrizes

Com relação à divisão entre duas matrizes, não se observou um ganho de desempenho significativo entre as diferentes implementações, ou seja, síncrona e assíncrona. Também não se observou melhoras na implementação assíncrona com diferentes números de *threads*.

De fato, a divisão é uma operação realizada termo a termo, ou seja, dadas duas matrizes A e B, de mesma dimensão, divide-se o termo A_{ij} por B_{ij}. Tal operação não é computacionalmente dispendiosa, não sendo um gargalo de execução como, por exemplo, a multiplicação descrita na seção anterior.

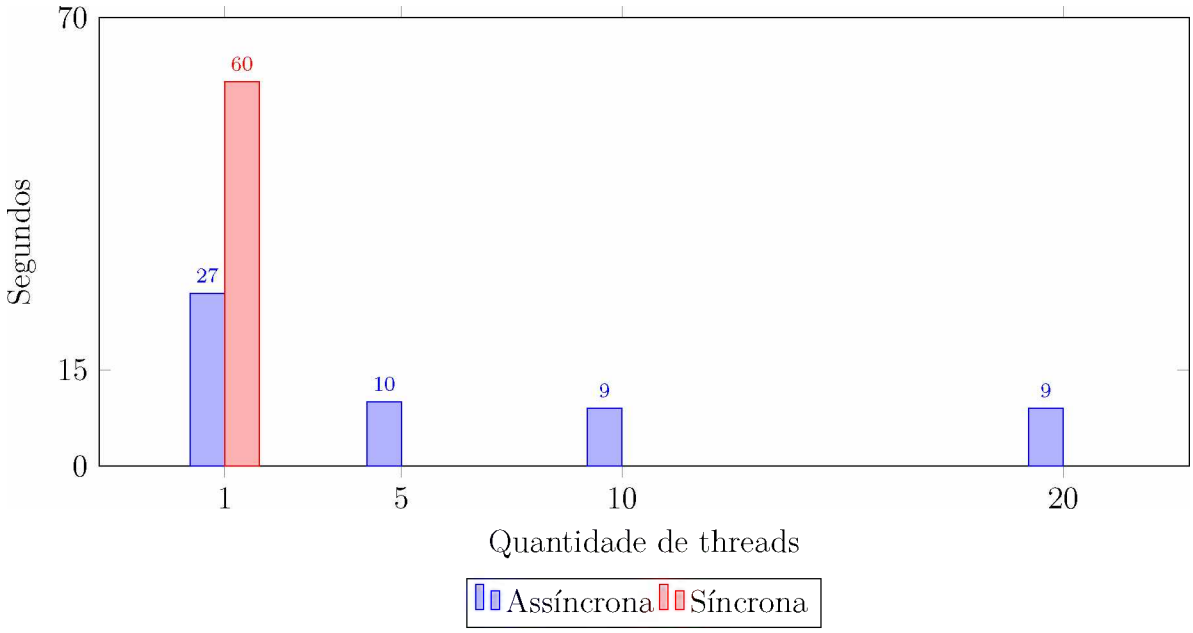

Figura 3 – Gráfico de comparativos de tempo médio em segundos para a realização da multiplicação de duas matrizes 1000×1000. Não foram realizados experimentos para a implementação síncrona com mais de uma *thread*, pois a mesma utiliza apenas uma *thread*.

4.2 Experimentos Adicionais

Após o final da implementação, para fins de comparação, foi implementada uma versão do algoritmo síncrono de multiplicação com o uso de diretivas presentes na biblioteca *omp4j* (BĚLOHLÁVEK, 2015), pré-processador de código para Java baseado no *OpenMP*, que transforma implementações síncronas em assíncronas.

A ideia desses experimentos é comparar a implementação proposta com uma outra solução disponível na literatura. Além disso, não foram encontradas implementações relevantes em relação ao que é apresentado neste trabalho para Java; por isso, optou-se por *omp4j* para comparação.

Foi implementada somente a multiplicação de matrizes devido ao que foi observado nos experimentos mostrados na seção anterior. Os resultados podem ser observados na tabela Tabela 2 e no gráfico Figura 4.

A partir desses resultados, observa-se que a estratégia proposta neste trabalho atinge resultados comparativamente relevantes. De fato, ao se utilizar uma biblioteca pré-processada (*omp4j*), o desempenho torna-se significativamente inferior. Em ambos os casos, no entanto, tem-se um ganho de desempenho à medida em que o número de *threads* aumenta[1].

[1] O código fonte dos programas utilizado nos experimentos deste capítulo estão disponíveis em: <https://bitbucket.org/parallelmatrixoperations/parallel-matrix-operations-api>.

Tabela 2 – Tabela comparativa de tempo médio em segundos para a realização da multiplicação de duas matrizes 1000 × 1000 no algoritmo assíncrono, síncrono e na implementação utilizando o *framework omp4j*. O símbolo "-" representa um experimento não realizado. Não foram realizados experimentos para a implementação síncrona com mais de uma *thread*, afinal, a implementação síncrona é sequencial e portanto, utiliza apenas uma *thread*.

	1 thread	5 threads	10 threads	20 threads
Assíncrona	27s	10s	9s	9s
Síncrona	60s	-	-	-
omp4j	63s	20s	20s	19s

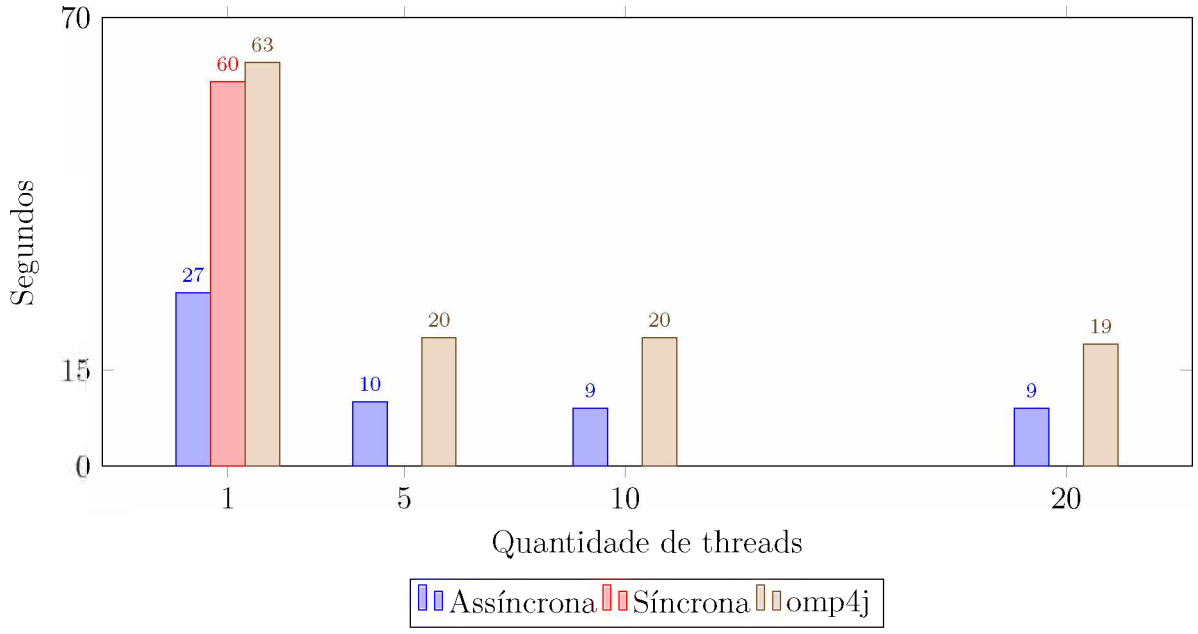

Figura 4 – Gráfico comparativo de tempo médio em segundos para a realização da multiplicação de duas matrizes 1000 × 1000 no algoritmo assíncrono, síncrono e na implementação utilizando o framework *omp4j*. Não foram realizados experimentos para a implementação síncrona com mais de uma *thread*, afinal, a implementação síncrona é sequencial e portanto, utiliza uma única *thread*. Por este motivo, não são mostrados valores referentes aos tempos de execução da implementação síncrona com mais de uma *thread*.

5 Conclusão

Neste trabalho de conclusão de curso, foi proposta a criação de uma API, na linguagem de programação Java, para a multiplicação e divisão de matrizes de forma assíncrona. O objetivo é que o produto deste trabalho seja utilizado futuramente para a resolução de modelos propostos por Gabriel (2013).

O paradigma paralelo propicia um melhor uso dos recursos de hardware e, na maioria casos, isso acarreta em um tempo de execução menor para uma tarefa. Uma vez que as operações com grandes matrizes são dispendiosas e, portanto, demoram muito para serem efetuadas, optou-se pelo paradigma paralelo para a implementação, com o objetivo de diminuir o tempo de execução.

Para a elaboração deste trabalho de conclusão de curso, inicialmente foram estudadas algumas publicações que propunham implementações eficientes para a multiplicação assíncrona de matrizes. Poucos referenciais foram encontrados de modelos que utilizassem a linguagem de programação Java (a maioria das propostas utilizam linguagem C, ou são voltadas para a criação de sistemas distribuídos, que particionam as operações em diferentes nós).

A implementação adotada foi baseada no conceito de *callback*, um modelo bastante avaliado e consolidado, que é adotado por diversos *frameworks* e linguagens de programação. O cliente, no caso o desenvolvedor que está utilizando a API, realiza a chamada de uma operação, multiplicação ou divisão, passando um objeto de *callback*. Quando a operação for concluída, ele é notificado através deste.

Após a fase de implementação, com o intuito de assegurar o objetivo proposto pelo projeto e verificar se o objetivo esperado foi atingido, realizou-se a avaliação de desempenho, comparando diferentes implementações para a multiplicação e divisão de matrizes. Obteve-se uma melhora significativa no tempo de execução da operação de multiplicação de matrizes, em comparação com o algoritmo síncrono, bom como a implementação com a utilização do pré-processador *omp4j*, que se baseia na ferramenta *OpenMP*. Na operação de divisão porém, não se obteve melhora significativa.

5.1 Trabalhos futuros

Neste trabalho, as tarefas são executadas em um único nó, ou seja, um único computador. Porém, utiliza-se os diversos núcleos disponíveis no processador.

Com o objetivo de diminuir ainda mais o tempo de execução das tarefas, as operações poderiam ser distribuídas em diferentes nós, ou seja, diferentes computadores. A

ideia seria criar um sistema distribuído que controlasse a execução e particionamento das tarefas de matrizes.

Uma possível solução seria utilizar a plataforma *Hadoop* (APACHE, 2018) para este propósito. A plataforma é utilizada para processamento distribuído de operações com um grande volume de informações. Assim, é possível utilizar um modelo implementado no *Hadoop* denominado *MapReduce*.

Nesse caso, a operação *Map* é utilizada para subdividir as operações de multiplicação de uma matriz e o *Reduce* para o unir os resultados das operações, ou seja, gerar a matriz resultante e devolver ao nó que solicitou a operação.

Em grandes matrizes, esse novo processo possivelmente teria um tempo de execução muito inferior ao que é proposto nesse trabalho. Uma possível implementação pode ser encontrada no site *lendapp*[1].

[1] https://lendap.wordpress.com/2015/02/16/matrix-multiplication-with-mapreduce/

Referências

APACHE. *Hadoop*. 2018. Versão 2.9.1. Disponível em: <http://hadoop.apache.org/docs/current/>. Acesso em: 15 nov. 2018. Citado 2 vezes nas páginas 12 e 22.

BĚLOHLÁVEK, P. *OpenMP for Java 6/7/8*. 2015. Disponível em: <http://www.omp4j.org>. Acesso em: 03 dez. 2018. Citado na página 19.

BRYANT, D. E.; O'HALLARON, R. *Computer Systems*: A programmer's perspective. 2. ed. Índia: Pearson Education India, 2016. 1120 p. Citado na página 10.

DONG, F.; AKL, S. G. *Scheduling Algorithms for Grid Computing*: State of the art and open problems. 55 p. Monografia (Relatório Técnico) — School of Computing, Queen's University, Kingston, Ontario, 2006. Citado na página 6.

ELLOUMI, S. *The Task Assignment Problem*. 2004. Disponível em: <http://cedric.cnam.fr/oc/TAP/TAP.html>. Acesso em: 12 nov. 2018. Citado 2 vezes nas páginas 6 e 8.

GABRIEL, P. H. R. *Uma abordagem orientada a sistemas para otimização de escalonamento de processos em grades computacionais*. 96 p. Tese (Doutorado) — Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo, São Carlos, 2013. Citado 5 vezes nas páginas 3, 6, 7, 9 e 21.

GASPAROTTO, H. M. *Hadoop MapReduce*: Introdução a big data. DevMedia, 2014. Disponível em: <https://www.devmedia.com.br/hadoop-mapreduce-introducao-a-big-data/30034>. Acesso em: 24 nov. 2018. Citado na página 12.

GÖETZ, B.; PEIERLS, T.; BLOCH, J.; BOWBEER, J.; HOLMES, D.; LEA, D. *Java Concurrency In Pratice*. EUA: Addison-Wesley Professional, 2006. 218 p. Citado na página 10.

GOLDBERG, D. E. *Genetic algorithms in search, optimization, and machine learning*. EUA: Addison-Wesley, 1989. 412 p. Citado na página 9.

ISMAIL, M. A.; MIRZA, S. H.; ALTAF, T. Concurrent matrix multiplication on multi-core processors. *International Journal of Computer Science and Security*, v. 5, n. 2, p. 208–220, 2011. Citado na página 11.

KAYA, K.; UÇAR, B. Exact algorithms for a task assignment problem. *Parallel Processing Letters*, v. 19, p. 451–465, 2009. Citado 2 vezes nas páginas 6 e 8.

KERNIGHAN, B. W.; PIKE, R. *The Unix Programming Environment*. EUA: Prentice Hall, 1984. 357 p. Citado na página 10.

MYINT, E. E.; KYI, L. L. W. Design and analysis of parallel matrix multiplication. *International Journal of Computer & Communication Engineering Research*, v. 2, n. 2, p. 62–66, 2014. Citado 2 vezes nas páginas 11 e 12.

OPENMP ARCHITECTURE REVIEW BOARD. *OpenMP Application Program Interface*. 2013. Versão 4.5. Disponível em: <https://www.openmp.org/resources/refguides/>. Acesso em: 12 nov. 2018. Citado 2 vezes nas páginas 10 e 11.

ORACLE. *Package java.lang*. 2018. JDK 7. Disponível em: <https://docs.oracle.com/javase/7/docs/api/java/lang/package-summary.html>. Acesso em: 12 nov. 2018. Citado na página 10.

_____. *Package java.util.concurrent*. 2018. JDK 7. Disponível em: <https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html>. Acesso em: 12 nov. 2018. Citado 2 vezes nas páginas 10 e 15.

PACHECO, P. S. *An Introduction to Parallel Programming*. EUA: Morgan Kaufmann Publishers, 2011. 370 p. Citado na página 7.

QASEM, M. H.; QATAWNEH, M. Parallel matrix multiplication for business applications. In: SILHAVY, R.; SILHAVY, P.; PROKOPOVA, Z. (Ed.). *Applied Computational Intelligence and Mathematical Methods*. Cham: Springer, 2017. (Advances in Intelligent Systems and Computing, v. 662), p. 24–36. Citado 2 vezes nas páginas 11 e 12.

SÁ, C. C. de; SILVA, M. F. da. *Haskell*: Uma abordagem prática. São Paulo: Editora Novatec, 2006. 296 p. Citado na página 12.

SOARES, T. F. *Algoritmos evolutivos e modelo TIG para escalonamento de processos em ambientes distribuídos*. 35 p. Monografia (Trabalho de Conclusão de Curso) — Faculdade de Computação, Universidade Federal de Uberlândia, Uberlândia, 2018. Citado 2 vezes nas páginas 6 e 9.

THE OPEN GROUP. *pthread.h*: Posix threads. 1997. Disponível em: <http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html>. Acesso em: 12 nov. 2018. Citado na página 10.

_____. *unistd.h*: standard symbolic constants and types. 1997. Disponível em: <http://pubs.opengroup.org/onlinepubs/7908799/xsh/unistd.h.html>. Acesso em: 12 nov. 2018. Citado na página 10.

THE OPEN MPI PROJECT. *Open MPI*: Open source high performance computing. 2016. Disponível em: <https://www.open-mpi.org/>. Acesso em: 24 nov. 2018. Citado na página 12.

VAN STEEN, M.; TANENBAUM, A. S. *Distributed Systems*. 3. ed. EUA: CreateSpace Independent Publishing Platform, 2017. 582 p. Citado na página 6.

XHAFA, F.; ABRAHAM, A. A compendium of heuristic methods for scheduling in computational grids. In: CORCHADO, E.; YIN, H. (Ed.). *Intelligent Data Engineering and Automated Learning*. Berlin: Springer, 2009, (Lecture Notes in Computer Science, v. 5788). p. 751–758. Citado na página 8.

XHAFA, F.; BAROLLI, L.; DURRESI, A. Batch mode scheduling in grid systems. *International Journal of Web and Grid Services*, v. 3, n. 1, p. 19–37, 2007. Citado na página 8.