UNIVERSIDADE FEDERAL DE VIÇOSA
CAMPUS DE RIO PARANAÍBA
SISTEMAS DE INFORMAÇÃO

MATHEUS VIEIRA DA SILVA - 6002
RICARDO BORTOLUCCI - 5948
VICTÓRIA SANTOS DE SOUZA - 5969

RELATÓRIO DE PROJETO DE PROGRAMAÇÃO CONCORRENTE E DISTRIBUÍDA

RIO PARANAÍBA
2019

MATHEUS VIEIRA DA SILVA - 6002
RICARDO BORTOLUCCI - 5948
VICTÓRIA SANTOS DE SOUZA - 5969

RELATÓRIO DE PROJETO DE PROGRAMAÇÃO CONCORRENTE E DISTRIBUÍDA

Relatório apresentado à Universidade Federal de Viçosa como um dos projetos exigidos para aprovação na disciplina Programação Concorrente e Distribuída

Orientador: João Batista Ribeiro

RIO PARANAÍBA

2019

Sumário

1 Introdução

A programação concorrente surgiu com os Sistemas Operacionais (CHOFFNES, 2005). Porém, atualmente seu contexto não é tão restrito e esse paradigma é utilizado em diversas áreas da computação. Para entender a programação concorrente e as análises do trabalho serão apresentados alguns conceitos:

1.1 Programas, processos e *threads*

A tecnologia, bem como qualquer grande campo de estudo, possui terminologias básicas inerentes a qualquer atividade a ser desenvolvida na área, por mínima que seja. Dentre elas, um dos objetos de maior foco dentro da subárea de programação estão os resultantes de sua aplicação através de uma ordem específica de instruções de código a serem executadas, ou seja, os programas. Um processo é uma unidade de programa, possível de ser executada juntamente com outros de forma concorrente, que suporta uma única thread de controle. *Threads* são as linhas de execução de determinado processo (ou tarefa), delineando o acontecimento em tempo real daquela porção do programa.

1.2 Sequencial VS Concorrente

Quando se dá o desenvolvimento de um programa, os conceitos acima são amplamente abordados e têm grande importância, o que se estende para qual tipo de programação será usada para melhor resolver os problemas daquela aplicação, dois desses sendo a abordagem sequencial e a concorrente. A concorrência, caracterizada rudemente por dois ou mais acontecimentos ocorrendo simultaneamente, pode se manifestar em um programa através de instruções, declarações, unidades e até mesmo os programas em si; sendo assim, há várias threads e processos ocorrendo (SEBESTA, 2015). Inversamente, a execução sequencial se dá exatamente na ordem em que as instruções foram escritas pelo programador, tendo seu sentido, portanto, pré-estabelecido com apenas uma thread.

1.3 CPU *bound*, I/O *bound* e Memory *bound*

Outros conceitos sempre associados com a área da computação são CPU (em português, unidade central de processamento) e I/O (dispositivos de entrada e saída). Certas tarefas dentro dos computadores têm seu desempenho pautado puramente na capacidade de processamento da máquina, caracterizando problemas CPU bound; ao contrário destas, existem as tarefas I/O bounds, baseadas no tempo de espera entre as operações de

entrada e saída, envolvendo a requisição de dados e o processamento dos mesmos; e, por fim, quando o tempo de execução de um tarefa depende majoritariamente da quantidade de memória necessária para armazenar os dados nela envolvidos tem-se um problema Memory bound.

1.4 Programação concorrente em Java

Java é uma das linguagens mais utilizadas no mundo e principalmente na esfera comercial (DOEDERLEIN, 2011).Sendo assim, a presença da programação concorrente na linguagem é substancial, por isso Java atende aos requisitos do paradigma, aplicando conceitos como *multithreading*, por meio ,principalmente, das interfaces: *Thread* e *Runnable*, utilizadas para criar várias linhas de execução diferentes e paralelas (CAELUM, 2013).

2 Avaliação de Desempenho

2.1 Planejamento de experimentos e boas práticas

Para o desenvolvimento de qualquer software é de suma importância medir o desempenho da aplicação, caracterizado como a relação entre a facilidade de realizar as funções necessárias conforme o poder computacional aumenta, englobando desde tempo de execução a uso de memória.

Com o crescimento de aplicações para teste e simulação, além da capacidade tecnológica ascendente característica das últimas décadas, utiliza-se de projetos de experimentos visando obter a maior quantidade possível de informações, fazendo-se essencial o planejamento para conseguir atingir os objetivos com o menor número de testes quanto possível e obter análises com mais qualidade (DUARTE, 2015). Acrescenta-se que "sua aplicação auxilia na construção de uma estrutura de análise adequada que permite identificar os principais fatores, bem como a interação entre eles"(NAZáRIO, 1998).

Visando os mesmos fins, porém ainda ajudando a poupar tempo na criação, leitura e manutenção de programas, destacam-se as boas práticas de programação. Dentro da linguagem escolhida para o desenvolvimento do projeto (Java), há a preocupação com vários aspectos técnicos, sendo alguns deles: indentação adequada, nomenclatura de classes, métodos e atributos que, além de seguir os padrões sintáticos específicos, precisam ser auto-descritivos, pois, como dito pelo Devmedia — plataforma informativa para programadores — se um nome [...] requer um comentário, ele não está revelando sua real intenção"; e comentários significativos onde necessário, facilitando saber o que cada parte do código faz de fato. (DEVMEDIA, 2014)

2.2 Fatorial completo versus fatorial 2^k

Existem diferentes tipos de projetos de experimentos, sendo eles os simples, fatoriais completos e fracionados, e fatoriais 2^k. É relevante para este trabalho apenas o completo e o fatorial 2^k.

Os projetos com fatorial completo abrangem todas as combinações possíveis (ACTION, 2019) entre os níveis (n) dos fatores (k) do experimento, testando-se cada uma delas para no fim ter a informação completa sobre a interação como um todo. No entanto, o trabalho de realizar cada uma dessas verificações, especialmente com um n grande, torna essa abordagem inviável para muitos casos.

Buscando contornar o trabalho excessivo dos fatoriais completos são utilizados

os fatoriais fracionados, com medição somente de uma combinação específica de níveis de fatores. No caso do fatorial 2^k tem-se k fatores com dois níveis cada, normalmente medindo o máximo e o mínimo. Caso seja conhecido antes dos testes que certos fatores não interagem entre si esse tipo de projeto pode ser vantajoso, uma vez que resulta em menos trabalho para testar as interações, embora seja importante ressaltar que a chance de imprecisões nos resultados é maior. Ainda assim, é uma ferramenta boa para ser aplicada antes de estudos mais detalhados e gerar novas ideias sobre as interações em questão.(DUARTE, 2015)

3 Desenvolvimento

3.1 Configuração dos Experimentos

3.1.1 Hardware e Software utilizados

A implementação dos "problemas" I/O bound e CPU bound foram feitas em ambiente linux, ubuntu e fedora, respectivamente. Para o desenvolvimento da parte I/O bound foi utilizado o Visual Studio Code em uma máquina com 8G de memória RAM, 1TB de HD e processador intel CORE i7 (oitava geração). Já a implementação de CPU bound foi feita por meio da IDE pache NetBeans em um computador com 4G de memória RAM, 1TB de HD e processador intel CORE i3 (quinta geração) com windows 10.

3.1.2 Manual de utilização dos programas

a) **Utilização dos arquivos .java I/O *bound***

O problema I/O Bound selecionado foi a transferência de arquivos com verificação de integridade.A implementação dos programas (sequencial e concorrente) foi feita em linguagem java e pode ser encontrada no github[1].

O programa sequencial foi implementado utilizando apenas duas classes, na qual todas operações de I/O Bound foram efetuadas em uma classe, enquanto a outra serve como classe principal (interação com o usuário). Enquanto isso, o programa concorrente consiste nas classes: PrincipalConcorrente.java (interação com usuário e organização das threads), TransferenciaConcorrente.java (realiza transferencia de diretorio), HashConcorrente.java (recolhe tamanho do arquivo a ser transferido), HashConcorrente2.java (compara tamanho do arquivo original com o tamanho da cópia e decide se a transferência será concluída).

Para o programa sequencial as classes necessárias para execução do programa são Principal.java e Arquivo.java. Caso seja utilizada alguma IDE como NetBeans basta criar um projeto java e adicionar as classes. Senão inserir ambas em um diretório comum é o suficiente para executá-las por meio do terminal. Lembrando que é necessário atender aos requisitos para interpretação da linguagem java.

Já para os programas concorrentes estão nomeados como "[...]Concorrente.java" (PrincipalConcorrente.java, HashConcorrente.java, etc). Para execução desses

[1] https://github.com/patriani/Projeto-Concorrente

as instruções são semelhantes às apresentadas para o programa sequencial.

b) **Utilização dos arquivos .java CPU *bound***

O código fonte dos programas podem ser encontrados nos seguintes links[2,3] O problema CPU Bound escolhido foi o Merge Sort (PYTHONDS, 2019) com suas classes implementadas na linguagem de programação java, em sua versão 8.

O programa sequencial pode ser executado por terminal, ou qualquer IDE disponível, basta importar o arquivo "Merge_sort_S.java". Tenha em mente que o código pode não funcionar em versões mais recentes do java. Enquanto o programa concorrente segue a lógica do anterior, os passos para a execução são os mesmos.

3.2 Resultados e Análises

3.2.1 Resultados (das, pelo menos 10, replicações dos experimentos – com e sem concorrência))

a) **I/O *bound***

Teste	1	2	3	4	5	6	7	8	9	10
Concorrente	6s	3s	2s	2s	3s	2s	2s	7s	1s	2s
Sequencial	0,85s	0,86s	0,78s	0,85s	0,91s	0,82s	0,95s	0,86s	0,85s	1s

Figura 1 – Tempo de Execução I/O *bound*

Por meio da tabela é possível notar, medindo o tempo de execução das tarefas utilizando System.currentTimeMillis() (função própria do java), que a tarefa executada pelo algoritmo é mais eficiente em sua forma sequencial. Isso ocorre porque a tarefa não é muito complexa para um número reduzido de requisições, logo o tempo de espera para resposta entre classes ou evitando *deadlock* acaba sendo uma desvantagem.

b) **CPU *bound***

Ao analisar os tempos de execução dos programas apresentados na Figura 2, é possível notar que o tamanho do vetor de números inteiros, está relacionado com a diferença no

[2] https://github.com/Matheuspp/merge-sort-with-threads
[3] https://github.com/Matheuspp/merge-sort-sequencial

Tamanho	1	5	10	100	500	1000	3000	4000	5000	10000
Concorrente	0.001s	0.006s	0.02s	0.06s	0.607s	0.685s	2.36s	2.511s	3.458s	14.187s
Sequencial	0.001s	0.005s	0.001s	0.008s	0.027s	0.04s	0.129s	0.166s	0.294s	0.532s

Figura 2 – Tempo de Execução CPU *bound*

tempo de execução dos programas. Nota-se também, que um vetor de tamanho 10000, leva 26 vezes mais tempo em um programa concorrente. Sendo assim, para vetores maiores esse valor deve aumentar ainda mais.

3.2.2 Tempo de execução (coloque uma tabela com dos dados e calcule a média, variância e desvio padrão)

a) **I/O *bound***

*	Sequencial	Concorrente
Variância	0,004s	3,778s
Desvio Padrão	0,068s	1,943s
Média	0,873s	3s

Figura 3 – Medidas I/O *bound*

Se analisarmos a razão entre a media do tempo do programa concorrente e a media do tempo do programa sequencial percebemos que o programa sequencial é aproximadamente 3,436 vezes mais rápido que o concorrente.

b) **CPU *bound***

*	Sequencial	Concorrente
Variância	0,027136s	16,884806s
Desvio Padrão	0,164729s	4,109113s
Média	0,120300s	2,389500s

Figura 4 – Medidas CPU *bound*

As métricas estatísticas deixam ainda mais evidente como a abordagem concorrente foi ineficiente para resolver este tipo de problema. A variância foi uma delas, onde teve valores muito altos para casos de teste próximos numericamente.

4 Conclusão

Aproximando os dados de análise (baseados em tempo de execução), é possível notar o impacto da dinâmica do algoritmo (se utiliza threads ou não) segundo o contexto inserido (natureza do problema - tipo e tamanho de entrada):

4.1 I/O Bound

Conforme os dados analisados nas sessões 3.2.1 e 3.2.2 (resultados e tempo de execução) percebemos que há uma discrepância entre as diferentes aplicações do mesmo problema. Isso ocorre porque o computador pode efetuar transferência de arquivos e verificação de integridade rapidamente, sem precisar que a estrutura sequencial seja rearranjada em operações atômicas.

O quadro para o exercício efetivo das threads seria uma requisição muito grande de arquivos a serem transferidos ou copiados como em um backup de um computador. Dessa forma, as operações seriam mais eficientes se divididas como o programa concorrente desenvolvido para esse projeto. A organização em tarefas proporcionaria melhor aproveitamento dos recursos computacionais em função do tempo.

4.2 CPU Bound

A aplicação sequencial do algoritmo de ordenação *Merge Sort* se mostrou mais eficiente nos casos teste quando comparada com a abordagem concorrente do problema. O que não era esperado, já que teoricamente um número maior de *threads* nesse algoritmo resultaria em um cálculo mais rápido devido a natureza binária das divisões do vetor de números inteiros.

4.3 CPU *Bound* Vs I/O *Bound*

Como podemos observar a abordagem concorrente no problema I/O Bound foi mais efetiva em relação ao programa *CPU Bound* concorrente. Tendo como base as estimativas estatísticas extraídas nos casos de teste nas duas abordagens, observa-se que a dinâmica concorrente na abordagem de *CPU Bound* obtiveram altas taxas de variância, desvio padrão e média, em relação ao *I/O Bound*.

Referências

ACTION, P. **Experimento Fatorial Completo**. [S.l.], 2019. Disponível em: <http://www.portalaction.com.br/planejamento-de-experimento/experimentos-fatoriais-completos>.

CAELUM. **Programação Concorrente e Threads**. [S.l.], 2013. Disponível em: <https://www.caelum.com.br/apostila-java-orientacao-objetos/apendice-programacao-concorrente-e-threads/>.

CHOFFNES, D. Sistemas operacionais. **Editora Person**, 2005.

DEVMEDIA. **Boas práticas de programação**. [S.l.], 2014. Disponível em: <https://www.devmedia.com.br/boas-praticas-de-programacao/31163>.

DOEDERLEIN, O. **Por Quê Java ?** [S.l.], 2011. Disponível em: <https://www.devmedia.com.br/por-que-java/20384>.

DUARTE, A. **Projeto de Experimentos**. [S.l.], 2015. Disponível em: <https://www.slideshare.net/alexandrend/projeto-de-experimentos>.

NAZáRIO, P. **PROJETO DE EXPERIMENTOS FATORIAL: APLICAÇÃO NA SIMULAÇÃO DE UM PORTO**. [S.l.], 1998. Disponível em: <https://www.ilos.com.br/web/modelo-para-artigo-9/>.

PYTHONDS. **Algoritmo Merge Sort**. [S.l.], 2019. Disponível em: <https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheMergeSort.html>.

SEBESTA, R. **Concepts of Programming Languages, Eleventh Edition, Global Edition**. [S.l.], 2015.