Estudo Experimental sobre Máquinas Virtuais Java em Sistemas Operacionais Linux

Fabiane C. Araújo, Paulo Eduardo Nogueira, Carlos Euardo Pagani

Instituto Federal de Educação, Ciência e Tecnologia de São Paulo – Campus Hortolândia – Caixa Postal 13183-250 - São Paulo – SP – Brasil

araujo112@hotmail.com, nogueira.paulo@ifsp.edu.br, pagani@ifsp.edu.br

Abstract. *The Virtual Machine is a computer environment software that runs programs virtually, therefore, it creates abstractions of computational resources and subdivides a program into several simulated images. This article compares three Java Virtual Machines available for the Linux operating system. The objective is to characterize Java Virtual Machines available in the market, observing the total expense used to execute a program and the total use of HEAP memory as parameters. Thus, the study can be used as a guide for choosing the most viable software for developing an application.*

Resumo. *A Máquina Virtual é um software de ambiente computacional que executa programas virtualmente, consequentemente, cria abstrações de recursos computacionais e subdividi um programa em várias imagens simuladas. O presente artigo, compara três Máquinas Virtuais Java disponíveis para o sistema operacional Linux. O objetivo é caracterizar Máquinas Virtuais Java disponíveis no mercado observando o gasto total utilizado para a execução de um programa e o uso total de memoria HEAP como parâmetros. Desse modo, o estudo pode ser usado como um guia para a escolha do software mais viável no desenvolvimento de uma aplicação.*

1. Introdução

Como apontado por PRODEST (2020), utilizar uma aplicação para pedir uma refeição, ouvir música, ou mesmo assistir a vídeo aulas de um curso, tornou-se algo frequente para a população. O que determina o bom funcionamento destas aplicações são as ferramentas envolvidas em sua elaboração. A *Java Virtual Machine* (JVM) é uma tecnologia padronizada pelo processo *Java Community Process* (JCP) [JCP 2021] que ganhou fama por oferecer grandes vantagens no desenvolvimento de software auxiliando na celeridade do processo de criação das aplicações e garantindo a portabilidade para diversos sistemas operacionais viabilizando a execução dos programas em múltiplas plataformas.

A possibilidade de se escrever um programa que execute em qualquer plataforma de hardware e software atraiu muitos desenvolvedores e impulsionou a popularidade da linguagem de programação Java. Conforme apresentado na plataforma TIOBE (2021), índice de linguagens de programação mais utilizadas na atualidade, o Java vem se mantendo nas primeiras colocações desde o início dos anos 2000 e ocupa a terceira posição no ranking perdendo para as linguagens de programação PYTHON e C.

Por se tratar de uma linguagem OpenSource, é permitido a qualquer um criar e oferecer uma JVM [DevMedia 2013]. Assim sendo, é possível encontrar no mercado diversas JVMs oferecidas por comunidades ou mesmo empresas de software. Dentre as máquinas virtuais pode-se citar: a OpenJDK, implementação livre e gratuita da plataforma Java desenvolvida pela Sun Microsystems com contribuições da comunidade Java [OPENJDK 2021], o AdoptOpenJDK projetada pela IBM, que surgiu após anos de discussões sobre a

falta de um sistema de construção e teste em que o código-fonte fosse aberto e com execução disponível em várias plataformas [AdoptOpenJDK 2021] e o OpenJ9, também desenvolvido pela IBM, com a proposta de solucionar problemas de execução do programa utilizando técnicas para melhorar o desempenho das aplicações [JAXENTER 2018]. No entanto, diante das opções disponíveis o programador que está iniciando sua carreira pode ter dúvidas sobre qual JVM escolher para executar os seus programas.

Assim sendo, o presente trabalho pretende apresentar um estudo sobre as características e o desempenho de JVMs que possam ser instaladas e operadas em sistemas operacionais Linux. Programas como HTOP [REDEHOST 2015] capturam a execução do processo em tempo real, no entanto, para se obter uma sequência de valores e informações mais precisas é necessário um trabalho de configuração que possibilite armazenar os resultados obtidos, para um usuário sem familiaridade com a ferramenta, uma configuração correta pode consumir algumas horas, ou mesmo dias, de aprendizado. Desse modo, com o intuito de obter informações rápidas e de forma simples, neste trabalho, foram realizados experimentos utilizando um programa TESTE contendo algoritmos de ordenação, com o intuito de observar o comportamento de cada JVM estudada.

Este artigo está estruturado do seguinte modo, a Seção 2 apresenta uma breve descrição de cada JVM, a Seção 3 descreve o planejamento do experimento realizado, a Seção 4 discute os resultados encontrados e a Seção 5 as conclusões e considerações finais.

2. Java Virtual Machine

Com a proposta de integrar dispositivos eletrônicos com conteúdo dinâmico, a equipe de programadores chefiada por James Gosling inicia em 1991, dentro da Sun Microsystems, o desenvolvimento de uma nova linguagem de programação. A popularidade da Web, a partir de 1993, foi a oportuniade identificada pelos desenvolvedores para integrar o Java à Internet, porém, apenas em 1995 as alterações necessárias foram concluídas [Deitel 2009].

Em seu trabalho, SOUZA (2005), aponta que a portabilidade da linguagem Java foi um dos impulsionadores do seu sucesso. Uma vez que há no mercado uma variedade de sistemas operacionais, os programas escritos para um determinado sistema podem apresentar vários erros ou mesmo incompatibilidade total em outras plataformas, exigindo que eles fossem reescritos. O conceito de máquina virtual surgiu na tentativa de oferecer uma solução para este problema, com a JVM sendo a responsável por converter os dados para a arquitetura real.

O funcionamento do programa Java depende da instalação prévia do JRE (Java Runtime Enviroment) no ambiente computacional do usuário, pois, é este software o responsável por preparar o dispositivo para que ocorra a execução correta do programa [DevMedia 2013]. Dentro da JRE existe uma série de componente, neste estudo iremos citar apenas os principais deles: o *Java Class Libraries*, *o Class Loader*, o JVM, o *Java Interpreter* e o *Garbage Collecction*.

Segundo SOUZA (2005), o *Java Class Library* é um conjunto de bibliotecas carregáveis dinamicamente que a JVM pode chamar em tempo de execução do programa. O *Java Interpreter* transforma o dado para o tamanho de byte correspondente de acordo com uma tabela pré-existente na JVM e que é compatível com a máquina do usuário. A JVM reproduz as funções de um determinado ambiente utilizando um conjunto de instruções próprias, atua em áreas de gerenciamento de memória, trabalha com especificações compatíveis com diversas arquiteturas e opera diretamente no arquivo de formato *class* onde estão localizados os *bytecodes*. Cabe a JVM localizar e importar os dados binários para as

classes por meio do componente *class loader*, esta classe também ativa o *heap* que é a estrutura responsável por alocar a memória e que irá executar as instâncias da classe e as variáveis do programa dentro da JVM. A cada processo apenas um *heap* é criado na JVM, assim, caso não seja possível a criação de um novo processo, os processos que já estão em execução não serão afetados. O *Class Loader* também é responsável pela resolução de símbolos. Por fim, caso exista processos que não sejam utilizados pela JVM, o *Garbage Collection* é o componente responsável por identificar e remover da JRE estes processos [SOUZA, 2005]. A Figura 1 ilustra a iteração entre esses componentes.

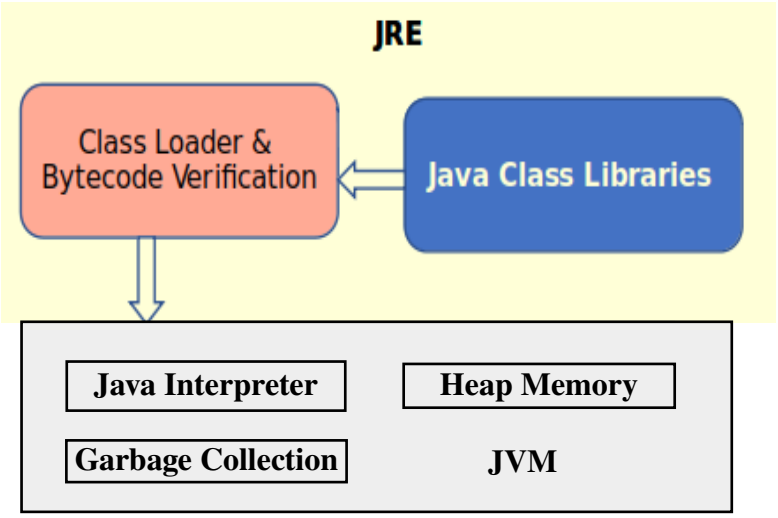

Figura 1. Principais componentes da JRE

No entanto, para o desenvolvimento de programas JAVA é necessário utilizar um outro pacote, o *Java Development Kit* (JDK) [Cipoli 2020]. Esse pacote pode ser disponibilizado tanto por empresas como Oracle e IBM, quanto pela própria comunidade. Neste trabalho, optou-se por trabalhar com 3 opções de JDK disponíveis no mercado, são elas: OpenJDK [OpenDK 2021], AdoptOpenJDK [AdoptOpenJDK 2021] e OpenJ9 [Open9 2021]. Esses pacotes foram escolhidos por serem de licença aberta e por serem indicados para o sistema operacional utilizado na pesquisa: o Linux.

2.1 OpenJDK

O OpenJDK foi um projeto da Sun, iniciado em 2006, porém, nos dias atuais é mantido pela Oracle e o seu desenvolvimento é realizado pela comunidade Java. Esse projeto surgiu como uma proposta OpenSource e gratuita para o JDK7. Sua construção possui uma implementação com base na arquitetura Java SE e no sistema operacional Linux [Kasko et al 2015]. A abertura do código foi uma opção da Sun ao percebe que com a contribuição da comunidade o software evoluiria de forma mais rápida e eficiente [OpenJDK 2021].

Em seu artigo a DevMedia (2012) aponta algumas características do OpenJDK, como por exemplo, o desenvolvimento de software em pares, que se mostra ideal para grandes projetos uma vez que facilita o manuseio de grandes pedaços de código por ser mais flexível. O OpenJDK permite que segmentos de uma aplicação sejam usados e re-usados em operações maiores, proporcionando assim uma molecularização do JDK. Ele permite também a remoção de classes desnecessárias, ou seja, ele aplica refatoração, utilizando o mínimo necessário com responsabilidades bem definidas, aumentando assim as chances de aprovações em testes. Uma

outra característica apontada é o controle de versão no qual é atribuído um nome único ou uma numeração única para indicar o estado de um programa de computador facilitando a recuperação de versões anteriores caso necessário. Além disso, permite a integração de módulos, assim, o acoplamento de outras tarefas podem ser adicionados a aplicação.

Como apresentado pelo Wikipedia (2021a), o OpenJDK foi desenvolvido para executar em ambientes Linux, FreeBSD, Mac OS X, Microsoft Windows, OpenIndiana e existe outros portes em andamento.

2.2 AdoptOpenJDK

O AdoptOpenJDK (Adopt) é um dos vários projetos comunitários co-criados pela London Java Community. Esse projeto surgiu em 2017 após anos de discussões sobre a falta geral de um sistema direcionado a construção, teste e reproduzível para o código-fonte disponivél a várias plataformas [AdoptOpenJDK 2021]. Atualmente, o Adopt é mantido pela IBM e está em processo de migração para o Eclipse Foundation.

O artigo do Costlow (2019) conta que o Adopt inclui o certificado *AdoptOpenJDK Quality Assurance* (AQA) que é o compromisso de suprir as expectativas dos clientes corporativos até mesmo para aqueles que não desejam arcar com custos. O autor cita que o Adopt foi projetado para garantir um melhor desempenho visto que executa uma tarefa ou ação em um ambiente padronizado e controlado fornecendo uma compilação completa do código fonte do OpenJDK utilizando também uma série de testes de desempenho que estressam os sistemas, visando reforçado a garantia de um bom desempenho. Os status destas execuções bem sucedidas, registram casos em que a execução levam 15 minutos ou menos. Costlow comenta que o JDK pode executar o próprio teste de maneira mais rápida e alerta que o Adopt inadvertidamente causar uma lentidão em outros projetos.

Uma boa característica do Adopt é que ele oferece uma fonte confiável de binários do OpenJDK, sendo ideal para o desenvolvedor que deseja uma aplicação com funcionamento à longo prazo em qualquer plataforma.

2.3 OpenJ9

O OpenJ9 teve seu início com o projeto ENVY / Smalltalk, desenvolvido pela Object Technology International (OTI). Em 1996 a IBM adquiriu a OTI com o intuito de agregar conhecimento ao seu próprio Smaltalk, no entanto, ela decidiu adaptar a máquina virtual do Smaltalk para processar os *bytecodes* Java, devido ao crescimento dessa linguagem [Servant 2017].

Com o desenvolvimento continuo o OpenJ9 atua sendo o principal componente de muitos produtos de software IBM Enterprise, tem garantido uma colaboração mais ampla entre a comunidade e gerado a oportunidade de influenciar a próxima geração de aplicativos Java. Esta JVM trabalha com sistemas dedicados e serviços de hospedagem que permitem executar operações automatizadas de construção e teste, o que ajuda a garantir a qualidade e a confiabilidade do produto [Craik 2018].

O OpenJ9 JVM é totalmente compatível com a versões posteriores, o que significa que muitos recursos e melhorias podem ser explorados por aplicativos executados em diferentes versões do Java. O desenvolvedor terá sua aplicação executada por usuário de versões anteriores do Java sem problemas de versão. Outra característica é que OpenJ9 incorpora o Eclipse OMR, que fornece componentes de tempo de execução principais que podem ser usados para construir ambientes de tempo de execução para diferentes linguagens de

programação. O OpenJ9 é ideal para ambientes Linux, AIX, Windows, MacOs Z/os e IBM i [Wikipedia 2021b].

3. Planejamento Experimental

3.1 Método

Neste estudo, tanto o planejamento e execução do experimento, quanto a análise dos resultados, foram realizados conforme o método DOE (*Design of Experiment)* [Montgomery 2000]. O DOE instrui a modificar de forma controlada os fatores sendo estudados, permitindo assim, observar os efeitos dessas alterações sobre a variável resposta. Para este estudo, o consumo de memória e CPU, pela JVM, serão utilizados como variáveis resposta, obtidos em cada cenário de teste (tratamento). Em relação aos tratamentos, é importante informar que é obtido pela combinação de fatores e níveis [Montgomery 2000].

Cada fator estudado em combinação com o nível de cada um configura um tratamento. Uma vez que cada tratamento possui mais de dois níveis, foi adotado a seguinte definição: 0 para o nível mais baixo, 1 para o nível médio e 2 para o nível mais alto.

3.2. Programa TESTE

Uma vez que neste estudo, o algoritmo de ordenação também foi definido como um fator a ser observado, um conjunto de programas foram utilizados com certo procedimento para comparações, foi escrito um programa especifico para cada algoritmo de ordenação utilizado. O algoritmo Bubble Sort foi escolhido com o intuito de estressar o sistema já que é esperado um maior tempo de execução, os demais algoritmos foram escolhidos por serem adequados na execução de programas com número elevado de entradas.

Os programas foram compilados e executados diretamente no terminal utilizando o compilador disponibilizado pelo ambiente JAVA. Durante a execução do programa a conexão com a INTERNET foi interrompida. Cada programa TESTE foi executado 31 vezes, sendo que a primeira execução foi descartada com o intuito de evitar a influência do *buffer*, tanto do disco rígido quanto da memória.

Com o intuito de obter os dados referentes ao tempo de execução de um processo e o consumo da memória HEAP pela máquina virtual, optou-se por utilizar a classe *ManagementFactory*. Essa classe é composta por métodos estáticos que retornam *MXBeans* da plataforma que representa a interface de gerenciamento de um componente da máquina virtual Java [Java SE Documentation 2020]. A sequência a, b e c da Figura 2, apresenta os trechos de código responsáveis por capturar os valores obtidos pelo tempo de execução do programa.

3.2.1 Bubble Sort

O algoritmo de ordenação *Bubble Sort* efetua a ordenação entre os dados armazenados. Cada elemento de posição i será comparado com o elemento de posição i-1, e quando a ordenação procurada (crescente ou decrescente) é encontrada, uma troca de posições entre os elementos é feita. Assim, um laço com a quantidade de elementos do vetor será executada (for (j-1; j<=n; j++)), e dentro deste, um outro laço que percorre da primeira à penúltima posição do vetor (for (i-0; i<n-1;i++)) [Ascencio e Araújo 2010]. A Figura 3 ilustra a execução resumida do algoritmo Bubble Sort usado no Programa Teste do experimento.

a.

```
MemoryMXBean memoryMXBean = ManagementFactory.getMemoryMXBean();
ThreadMXBean threadMXBean = ManagementFactory.getThreadMXBean();
```

b.

```
for(Long threadID : threadMXBean.getAllThreadIds()) {
    ThreadInfo info = threadMXBean.getThreadInfo(threadID);
    System.out.println("Thread name: " + info.getThreadName());
    System.out.println("Thread State: " + info.getThreadState());
    System.out.println(String.format("Final CPU time: %s ns",
      threadMXBean.getThreadCpuTime(threadID)));
  }
```

c.

```
System.out.println(String.format("Used heap memory.: %.2f GB",
(double)memoryMXBean.getHeapMemoryUsage().getUsed()/1073741824));
```

Figura 2. a) declaração do objeto da classe ManagementFactory; b) trecho de código responsável por capturar o tempo de execução na CPU, e; c) trecho de código responsável por capturar o consumo de memória HEAP.

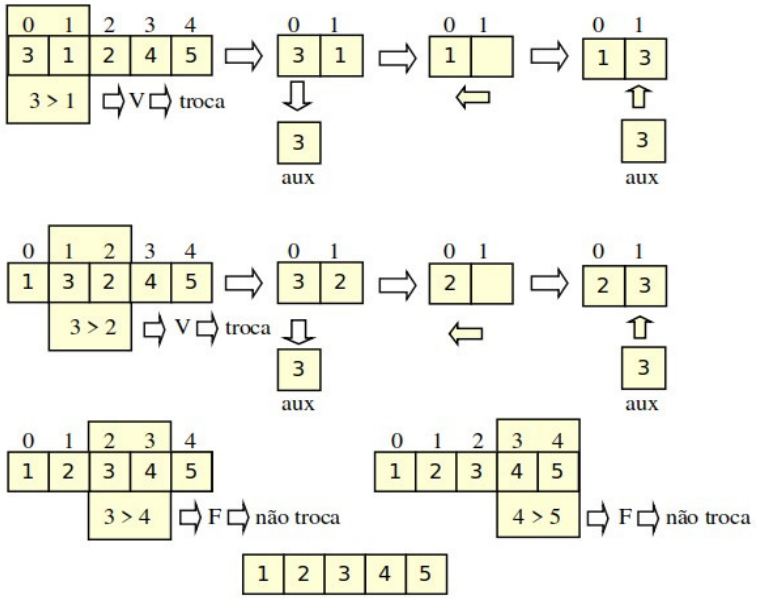

Figura 3. Execução simplificada do Burble Sort

3.2.2 Merge Sort

Neste algoritmo de ordenação o vetor é dividido em vetores com a metade do tamanho do original por meio de um procedimento recursivo. Essa divisão ocorre até que o vetor fique com apenas um elemento e estes sejam ordenados e intercalados. Assim, no algoritmo de ordenação por intercalação, Merge Sort, tem-se a técnica da divisão e conquista da seguinte forma: dividir a sequência de n elementos a serem ordenados em duas subsequências de n/elementos cada, ordenar as duas subsequências recursivamente utilizando a ordenação por intercalação e intercalar as duas subsequências ordenadas para produzir a solução [Ascencio e

Araújo 2010]. A Figura 4 ilustra o algoritmo de ordenação *Merge Sort* em sua versão simplificada.

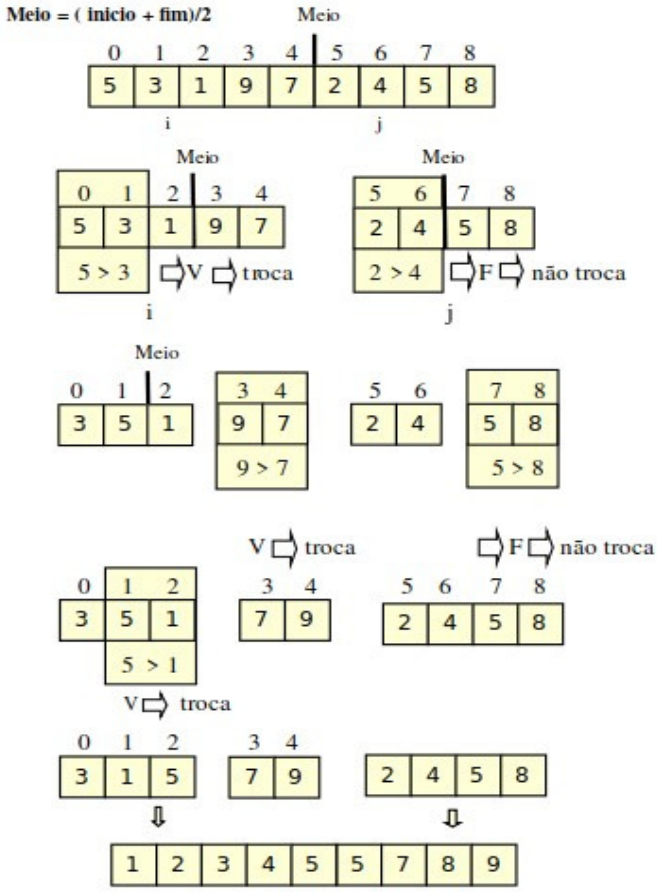

Figura 4. Execução do Merge Sort

3.2.3 QuickSort

O QuickSort é o algoritmo mais eficiente na ordenação por comparação. Nele se escolhe um elemento chamado de pivô, a partir disto é organizada a lista para que todos os números anteriores a ele sejam menores que ele, e todos os números posteriores a ele sejam maiores que ele. Ao final desse processo o número pivô já está em sua posição final. Os dois grupos desordenados recursivamente sofreram o mesmo processo até que a lista esteja ordenada. Esse algoritmo também é baseado na técnica da divisão e conquista mencionada na seção do algoritmo *Merge Sort*. O vetor é particionado (reagindo) em dois subvetores não vazios [Ascencio e Araújo 2010]. A seguir, a Figura 5 ilustra a execução simplificada do Quick Sort.

3. 4 Experimento

Esse experimento tem por objetivo observar o comportamento de cada JVM em relação ao gasto total do tempo para a execução e a utilização de memória HEAP. Por isso, cada programa TESTE foi executado em uma determinada JVM (ver Tabela 1). O que determinou a escolha das máquinas virtuais foi a compatibilidade da máquina com o computador utilizado no experimento e, além disso, algumas máquinas virtuais apresentavam algum custo em sua aquisição.

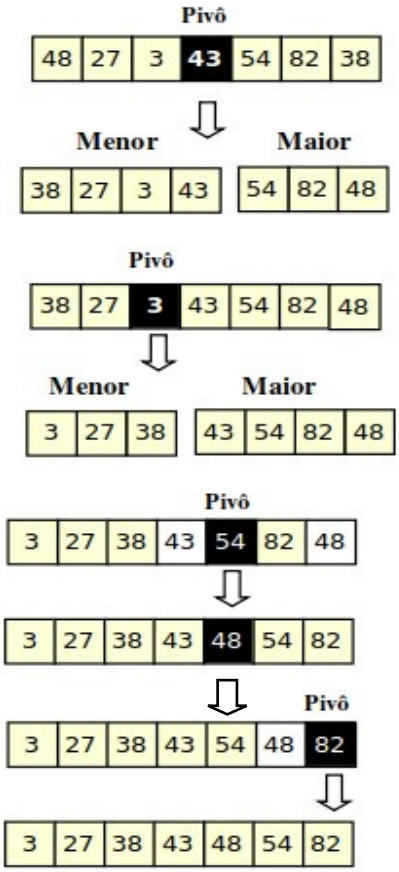

Figura 5. Execução simplificada do Quick Sort

O ambiente computacional representou um ambiente de trabalho presente no cotidiano dos desenvolvedores de software e, portanto, o nível de execução do sistema operacional foi mantido em *runlevel* 5 e não houve nenhuma alteração nos processos do sistema que executam em *background*. Neste trabalho foram utilizadas duas cargas de trabalho, que foram processados em cada programa TESTE, a primeira definida como um vetor de 100.000 números inteiros e a segunda como um vetor de 500.000 números inteiros, ambas as cargas foram carregadas com números inteiros aleatórios de 0 a 1.000.000. Números iguais foram mantidos.

Tabela 1. Fatores e níveis analisados no experimento.

		0	1	2
Fatores	Java Virtual Machine (JVM)	OpenJDK	AdoptOpenJDK	OpenJ9
	Algoritmo Ordenação (AO)	BubbleSort	MergeSort	QuickSort

O experimento foi executado em computador com processador Intel I5 de 7a geração e 8GB de memória RAM. O sistema operacional utilizado foi o Linux UBUNTU 20.04.1

4. Resultados

Após a coleta dos tempos de execução, os dados foram tabulados e uma análise descritiva foi realizada a partir dos resultados da média, mediana, desvio padrão e a variância das amostras.A Figura 6 ilustra um exemplo de um resultado obtido derivado da execução do programa TESTE:

```
Bubblesort
Initial memory.: 0,12 GB
Thread name: main
Thread State: RUNNABLE
Initial CPU time: 333067232 ns
Thread name: Reference Handler
Thread State: RUNNABLE
Initial CPU time: 376102 ns
Thread name: Finalizer
Thread State: WAITING
Initial CPU time: 337676 ns
Thread name: Signal Dispatcher
Thread State: RUNNABLE
Initial CPU time: 143920 ns
Thread name: Common-Cleaner
Thread State: TIMED_WAITING
Initial CPU time: 286356 ns
Fim da execução do bubblesort!
Thread name: main
Thread State: RUNNABLE
Final CPU time: 380904679 ns
Thread name: Reference Handler
Thread State: RUNNABLE
Final CPU time: 376102 ns
Thread name: Finalizer
Thread State: WAITING
Final CPU time: 337676 ns
Thread name: Signal Dispatcher
Thread State: RUNNABLE
Final CPU time: 143920 ns
Thread name: Common-Cleaner
Thread State: TIMED_WAITING
Final CPU time: 286356 ns
Used heap memory.: 0,02 GB
Elapsed Time: 73ms
```

Figura 6. Execução do Programa Teste

A Tabela 2 apresenta os dados obtidos para a execução dos programas TESTE com uma carga de trabalho de 100.000 números e a Tabela 3 apresenta os dados obtidos com uma carga de trabalho de 500.000 números.

Como é possível observar, de um modo geral, o algoritmo BubbleSort foi o que levou mais tempo para concluir a execução, no entanto, o consumo de memória HEAP, por este algoritmo, não foi excessivamente maior que os demais. Em alguns casos, como no uso da máquina virtual OpenJDK, o consumo de memória HEAP se mostrou menor se comparado aos demais algoritmos quando a carga de 100.000 números foi utilizada. Em relação a carga com 500.000 números, sua média só não foi maior do que o algoritmo QuickSort, porém, seu desvio padrão e variância não apresentaram valores significantes, isso significa que seus valores encontrados não apresentam dispersão significativa em relação aos valores médios.

Tabela 2. Estatística descritiva obtida com carga de trabalho de 100.000 números inteiros

	BubbleSort					
	Tempo Total			**Heap Memory**		
	OpenJDK	**AdoptJDK**	**Open J9**	**OpenJDK**	**AdoptJDK**	**Open J9**
Média	21296,167	23625,700	411681,800	0,063	0,010	0,012
Mediana	21465,000	24048,000	401887,000	0,090	0,010	0,010
DesvioPadrão	638,640	1433,450	22297,926	0,033	0,000	0,004
Variância	407861,072	2054779,677	497197500, 960	0,001	0,000	0,000
	MergeSort					
	Tempo total			**Heap Memory**		
	OpenJDK	**AdoptJDK**	**Open J9**	**OpenJDK**	**AdoptJDK**	**Open J9**
Média	49,400	68,467	58,200	0,086	0,012	0,010
Mediana	47,000	68,000	48,000	0,100	0,010	0,010
DesvioPadrão	9,351	5,239	22,158	0,028	0,004	0,000
Variância	87,440	27,449	22,158	0,001	0,000	0,000
	QuickSort					
	Tempo total			**Heap Memory**		
	OpenJDK	**AdoptJDK**	**Open J9**	**OpenJDK**	**AdoptJDK**	**Open J9**
Média	40,733	63,967	60,567	0,077	0,010	0,010
Mediana	38,000	65,000	52,000	0,090	0,010	0,010
DesvioPadrão	7,698	7,735	24,115	0,025	0,000	0,000
Variância	59,262	59,832	581,512	0,001	0,000	0,000

Obs: i. os tempos de execução estão em microssegundos; ii. Os valores da HEAP estão em Gigabyte

Além disso, quando se observa os tempos de execução do programa TESTE com uma carga de 500.000 números, é possível verificar que a máquina virtual OpenJ9 apresenta uma variação nos tempos de execução muito maior que as demais, quando se utiliza os algoritmos de ordenação Merge e QuickSort. Essa variação talvez possa ser explicada pela quantidade de threads que essa máquina cria para gerenciar a execução dos processos. Nesse caso, um estudo mais aprofundado se mostra necessário para comprovar essa hipótese.

Em relação aos algoritmos implementados, é possível observar que os tempos de execução, a variância e o consumo de memória HEAP são maiores quando os algoritmos MergeSort e QuickSort são utilizados, isso pode ser explicado pela quantidade de manipulação de dados que é necessária para colocar os valores em ordem.

Tabela 3. Estatística descritiva obtida com carga de trabalho de 500.000 números inteiros

	BubbleSort					
	Tempo total			**Heap Memory**		
	OpenJDK	**AdoptJDK**	**Open J9**	**OpenJDK**	**AdoptJDK**	**Open J9**
Média	568846,233	628426,367	390597,267	0,420	0,070	0,013
Mediana	527281,000	600332,271	389946,000	0,420	0,080	0,010
DesvioPadrão	59112,168	87023,271	2770,979	0,000	0,027	0,005
Variância	3494248437,446	7573049632,232	7678323,062	0,000	0,001	0,000
	MergeSort					
	Tempo total			**Heap Memory**		
	OpenJDK	**AdoptJDK**	**Open J9**	**OpenJDK**	**AdoptJDK**	**Open J9**
Média	139,500	133,700	412,500	0,453	0,046	0,012
Mediana	138,500	131,000	189,500	0,470	0,050	0,010
DesvioPadrão	10,056	13,685	304,088	0,082	0,005	0,004
Variância	101,117	187,277	92469,450	0,007	0,000	0,000
	QuickSort					
	Tempo total			**Heap Memory**		
	OpenJDK	**AdoptJDK**	**Open J9**	**OpenJDK**	**AdoptJDK**	**Open J9**
Média	83,500	103,067	568,400	0,414	0,105	0,011
Mediana	82,500	104,000	743,500	0,420	0,110	0,010
DesvioPadrão	11,141	7,141	375,266	0,019	0,015	0,002
Variância	124,117	50,996	140824,907	0,000	0,000	0,000

Obs: i. os tempos de execução estão em microssegundos; ii. Os valores da HEAP estão em Gigabyte

5. Conclusões

Devido o crescimento das aplicações auxiliando a humanidade em seu cotidiano surge também uma infinidade de arquiteturas de software dispostos a suprir a demanda dos consumidores. A necessidade de resolver problemas em relação a portabilidade destes diferentes sistemas fez surgir a JVM, responsavél por reserva e prepara os recursos que o computador necessita para executar um programa em diferentes ambientes computacionais.

Neste trabalho, buscando encontrar diferenças entre JVMs foi comparado três máquinas disponíveis que utilizam licenças de software livre. Para isso, foi criado um conjunto de programas TESTE com ordenação de números inteiros e este foi executado seguindo os cenários propostos.

Como variáveis de observação optou-se por analisar o tempo de execução total do programa e o consumo de memória HEAP em cada máquina virtual. O algoritmo Bubble Sort, como esperado, foi o que apresentou maior tempo de execução, de 10 a 15 minutos . O estudo permitiu identificar uma diferença considerável nos tempos de execução e no consumo de memória HEAP, apresentados por cada máquina virtual. Além disso, foi possível observar que há uma diferença na quantidade de threads geradas por cada máquina virtual para atender a

execução do programa. No entanto, observar apenas as variáveis escolhidas não permitiu uma análise mais profunda do comportamento das máquinas virtuais escolhidas.Assim sendo, como trabalho futuro propõe-se um estudo aprofundado em relação as threads criadas por cada máquina virtual. Desse modo, entende-se que é possível alcançar uma melhor compreensão sobre o comportamento de cada máquina virtual.

Referências

AdoptOpenJDK. (2021) "Sobre", http://adoptopenjdk.net, Outubro.

Ascencio. A. F. G. Araújo. G. S. (2010) "Estrutura de Dados: algoritmos, análise da complexidade e implementações em Java e C/C++", pág. 21-6, pág 53-59 e pág 61-72, Editora Pearson.

Cipoli, P. (2020) "O que é Java, JRE, JVM e JDK", https://canaltech.com.br/software/O-que-e-Java-JRE-JVM-e-JDK, Outubro.

Craik, A. (2018), "IBM Open Souce graduated to Eclipse Open J9", https://developer.ibm.com/languages/java/projects/eclipse-openj9, Outubro.

Costlow, E. (2019), "AdoptOpenJDK introduz programa de garantia da qualidade", https://www.infoq.com/br/news/2019/11/adoptopenjdk-quality/, Outubro.

Deitel (2009), "Java Como Programar", pág. 06–07. Pearson, 8th edição.

DevMedia. (2012) "Java 8 – O que esperar?", https://www.devmedia.com.br/java-8-o-que-esperar/24811, Novembro.

DevMedia. (2013) "Introdução ao Java Virtual Machine (JVM)", https://www.devmedia.com.br/introducao-ao-java-virtual-machine-jvm/27624, Outubro.

REDEHOST (2015) "Monitorando processos no Linux com o Htop", https://www.vivaolinux.com.br/artigo/Monitorando-processos-no-Linux-com-o-Htop, Outubro.

JavaSEDocumentation. (2020) "ClassManagementFactory.", https://docs.oracle.com/javase/7/docs/api/java/lang/management/ManagementFactory.html, Outubro.

JCP. (2021), "Java Community Process", https://www.jcp.org/en/home/index Novembro.

JAXENTER. (2018), "Eclipse OpenJ9: Not just any Java Virtual Machine", https://jaxenter.com/eclipse-openj9-145182, Outubro.

Kasko, A. K. S. Alexey, Mironcheko. (2009) "OpenDK CookBook", pág. 08. Packt Publish.

Montgomery, D. C. (2000), "Design and Analysis of Experiments", John Wiley, 3 rd edition.

OpenJ9. (2021), "Sobre", https://www.eclipse.org/openj9/, Outubro.

OPENJDK. (2021), "OpenJDK", https://openjdk.java.net/, Outubro.

PRODEST, Instituto de Tecnologia da Informação e Comunicação do Espírito Santo. (2021), "O uso de aplicativos na sociedade", https://prodest.es.gov.br/o-uso-de-aplicativos-na-sociedade, Outubro.

Servanti, R. (2017), "How did the J9 in OpenJ9 get its name?", https://medium.com/@rservant/how-did-the-j9-in-openj9-get-its-name-95a6416b4cb9, Outubro.

Souza, J. (2002), “Arquitetura da Máquina Virtual Java”, http://wwww.ic.unicamp.br/~rodolfo/Cursos/mo401/2s2005/Trabalho/991899-java.pdf, Outubro.

TIOBE, The Software Quality Company. (2021), “TIOBE Index for October 2021”, https://www.tiobe.com/tiobe-index/, Outubro.

Wikipedia. (2021 a), “OpenJ9”, https://en.wikipedia.org/wiki/OpenJ9, Novembro.

Wikipedia. (2021 b), “OpenJDK”, https://pt.wikipedia.org/wiki/OpenJDK, Novembro.

Documento Digitalizado Público

Artigo do Trabalho de Conclusão de Curso

Assunto: Artigo do Trabalho de Conclusão de Curso
Assinado por: Paulo Nogueira
Tipo do Documento: Outro
Situação: Finalizado
Nível de Acesso: Público
Tipo do Conferência: Documento Digital

Documento assinado eletronicamente por:

- **Paulo Eduardo Nogueira**, **PROFESSOR ENS BASICO TECN TECNOLOGICO**, em 16/11/2021 11:07:01.

Este documento foi armazenado no SUAP em 16/11/2021. Para comprovar sua integridade, faça a leitura do QRCode ao lado ou acesse https://suap.ifsp.edu.br/verificar-documento-externo/ e forneça os dados abaixo:

Código Verificador: 815142
Código de Autenticação: 9b3ebb1f5d