Análise Comparativa do Código Gerado por Compiladores Java e C++

Ricardo Terra, Jussara Almeida, Roberto S. Bigonha, Marco Túlio Valente

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG)
31.270-010 – Belo Horizonte – MG

{terra,jussara,bigonha,mtov}@dcc.ufmg.br

Abstract. *Platform independence provided by interpreters is usually achieved at the expense of efficiency. Although such inefficiency can be tackled by several optimization techniques, the efficiency of interpreted languages such as Java is still questioned nowadays. This article initially describes what types of operations are relatively slow in Java and presents a comparative analysis of several classes of algorithms implemented both in Java and C/C++. It also compares the performance gains that have been achieved by versions of the GNU GCC and JDK compilers delivered in the last five years.*

Resumo. *Independência de plataforma normalmente é obtida às custas da eficiência de execução de sistemas, que nesses casos são interpretados a partir de códigos intermediários. Embora esses tipos de ineficiência têm sido resolvidos via novas técnicas de compilação, a eficiência da linguagem Java é ainda muito questionada nos dias de hoje. Diante disso, este artigo procura inicialmente enumerar quais tipos de operações ainda são relativamente lentas em Java e apresenta uma análise comparativa entre implementações Java e C/C++ de diversas classes de algoritmos. Compara-se também o impacto das melhorias dos compiladores GNU GCC e JDK no tempo de execução do código gerado nesses últimos cinco anos.*

1. Introdução

A linguagem Java vem obtendo destaque no desenvolvimento de sistemas. Contudo, sua eficiência é muito discutida nos dias de hoje, sendo usualmente considerada lenta e com alta demanda de memória [8]. Mesmo assim, é amplamente adotada no desenvolvimento de sistemas, devido a diversos fatores como: (i) facilidade de aprendizagem e utilização; (ii) diversos arcabouços e bibliotecas públicas disponíveis; (iii) ampla gama de fóruns, grupos de discussão devido ao grande número de comunidades de desenvolvedores; (iv) portabilidade, o que atualmente é considerado fundamental pela quantidade de diferentes plataformas; (v) sem custo de aquisição ou utilização; entre outros [2].

Analisando unicamente o tempo de execução, é de se esperar que sistemas desenvolvidos em C/C++ tenham melhor desempenho do que sistemas desenvolvidos em Java [8, 9, 5, 4, 1]. Claramente, esse raciocínio se deve ao fato de se conhecer o processo de compilação dessas linguagens. Enquanto uma compilação C/C++ gera diretamente código de máquina, uma compilação Java gera um código intermediário (*bytecode*) que posteriormente é interpretado pela Máquina Virtual Java (*Java Virtual Machine* ou JVM).

Por outro lado, compiladores estão em constante aprimoramento, incorporando novas técnicas e recursos de otimização. Apenas para citar um exemplo, a JVM incorpora um recurso conhecido como compilador JIT (*Just-In-Time*) que consiste na tradução do *byte-code* para código de máquina nativo em tempo de execução [7].

Diante disto, este artigo tem o intuito de avaliar o desempenho da linguagem Java em comparação à linguagem C/C++ em duas esferas. Primeiramente, vários algoritmos foram implementados e executados tanto em Java quanto em C/C++, com o objetivo de mostrar se realmente Java sempre tem pior desempenho e quais tipos de operações são notoriamente mais lentas. Por outro lado, é realizada uma análise dos compiladores de Java e C/C++ de cinco anos atrás e os de atualmente a fim de avaliar o impacto das melhorias dos compiladores no tempo de execução do código gerado.

O restante deste artigo está organizado conforme descrito a seguir. A Seção 2 descreve a metodologia aplicada, incluindo a descrição completa do ambiente experimental e da carga de trabalho utilizada. Em seguida, a Seção 3 descreve um comparativo entre as linguagens por meio de observações pareadas de diversos algoritmos, e a Seção 4 analisa quantitativamente o impacto das melhorias dos compiladores no tempo de execução do código gerado nesses últimos cinco anos. Por fim, a Seção 5 descreve trabalhos relacionados e a Seção 6 conclui.

2. Metodologia

Para atingir os objetivos propostos neste artigo, as seguintes atividades foram executadas:

1. O experimento foi todo realizado em uma máquina dedicada do Laboratório de Linguagem de Programação configurada exclusivamente para o desenvolvimento dos experimentos. Isso foi realizado a fim de prover um ambiente com validade experimental. O ambiente é descrito em detalhes na Subseção 2.1.

2. Realizou-se uma análise comparativa entre Java e C/C++.

   (a) Desenvolveu-se um *benchmark* chamado `TerraBM` que contém uma série de algoritmos implementados em C/C++ e em Java. Como critério de seleção, foram selecionados algoritmos de alta complexidade assintótica, preferencialmente $O(n^2)$. Além disso, tais algoritmos foram agrupados em classes específicas, tais como algoritmos de estresse à pilha de execução, à memória primária, à Unidade Lógica e Aritmética (ULA) etc. A carga de trabalho é descrita em detalhes na Subseção 2.2.

   (b) Em seguida, foi realizado um processo experimental com validade estatística visando avaliar o desempenho de cada um desses algoritmos e inferir quais tipos de operações são notoriamente mais lentas.

   Essa atividade é descrita em detalhes na Subseção 3.

3. Realizou-se uma análise do impacto das melhorias dos compiladores C/C++ e Java no tempo de execução do código gerado.

(a) Desenvolveu-se um sistema Java chamado `Simula` que envolve pelo menos um algoritmo de cada classe definida na Seção 3.

(b) Em seguida, foi realizado um Projeto Fatorial $2^k r$ visando avaliar a influência dos parâmetros de customização da compilação e a versão do compilador no tempo de execução de sistemas nessas duas linguagens.

Essa atividade é descrita em detalhes na Seção 4.

2.1. Ambiente

Todos os tempos neste artigo foram medidos em uma máquina AMD Sempron 2.8GHz com 1GB de RAM, sob sistema operacional Linux Debian 5.04 *Basic Shell*. É importante mencionar que somente os serviços indispensáveis ao experimento estavam ativados e somente os aplicativos estritamente necessários foram instalados[1].

Para o experimento comparativo das linguagens foram utilizadas a JDK versão 1.6.0_20 e GNU GCC 4.5.0 – que correspondem às últimas versões dos compiladores. E, para a análise do impacto das melhorias dos compiladores C/C++ e Java no tempo de execução do código gerado foram também utilizadas a JDK versão 1.5.0_22[2] e GNU GCC 4.0.4[3].

Além disso, para que sistemas C/C++ e Java obtenham acesso equivalente aos recursos e, consequentemente, haja uma comparação justa, foram estipuladas as seguintes configurações: (i) o espaço da pilha de execução (*stack*) e do espaço de alocação dinâmica (*heap*) foram fixados em 32MB; (ii) os compiladores foram configurados para otimizar o código o máximo possível; (iii) as informações de depuração em tempo de execução foram desativadas; (iv) desativou-se a compilação em *background* na JVM. Caso ativa, essa configuração permitiria que o código fosse inicialmente interpretado até que a compilação estivesse completa, o que invalidaria os resultados.

Portanto, em termos técnicos, para atender a configuração (i), no quesito espaço da pilha de execução, foi inserido o parâmetro `stack` para C/C++ e `Xss` para Java. Já no quesito espaço da *heap*, foi inserido o parâmetro `heap` para C/C++ e os parâmetros `Xms` e `Xmx` para Java que, respectivamente, correspondem à memória inicial e máxima. Para atender a configuração (ii), foi inserido o parâmetro `O3` para C/C++ e `server` para Java. Para atender a configuração (iii), foi inserido o parâmetro `g0` para C/C++ e os parâmetros `g:none`, `Xlint:none` e `dsa` para Java. Por fim, para atender a configuração (iv), foi passado o parâmetro `Xbatch` para a JVM.

2.2. Carga de Trabalho

Não foram utilizados *benchmarks* existentes, simplesmente porque não foi encontrado um *benchmark* que contenha implementações de algoritmos tanto em C/C++ quanto em Java. Os *benchmarks* encontrados ou apresentam uma gama de algoritmos em C/C++ ou então

[1]Na página `http://www.dcc.ufmg.br/~terra/sblp2010` encontra-se publicamente disponível o roteiro de configuração do ambiente e detalhamento das linhas de compilação e execução.

[2]A JDK 1.5 foi lançada em setembro de 2004.

[3]O GNU GCC 4.0 foi lançado em abril de 2005.

uma gama de algoritmos em Java, entretanto nenhum continha implementações nas duas linguagens.

Diante disso, com o intuito de possibilitar uma observação pareada, a qual requer uma carga de trabalho equivalente para as duas linguagens na execução dos experimentos, desenvolveu-se um *benchmark* de algoritmos chamado `TerraBM`[4] que consiste na implementação – tanto em C/C++ quanto em Java – de uma série de algoritmos, conforme descritos na Tabela 1. Esse *benchmark* contemplou as operações estressadas nos *benchmarks* existentes. Por observação pareada, designa-se um tipo de observação em que são conduzidos n experimentos em cada um dos sistemas de tal forma que exista uma correspondência um-para-um entre o i-ésimo teste do sistema A e o i-ésimo teste do sistema B [3]. Para fins de esclarecimento, neste artigo, os $sistemas$ são as linguagens de programação e os $testes$ são os algoritmos. Convém mencionar que esses algoritmos foram implementados da melhor forma possível que a linguagem permite, isto é, fez-se a utilização preferencial de recursos que possibilitem uma maior eficiência. Por exemplo, a iteração de um arranjo na linguagem C/C++, o qual pode ser feito com indexação clássica, optou-se por aritmética de ponteiros que é sabidamente mais eficiente.

Tabela 1. Algoritmos do TerraBM

Objetivo	**Algoritmo**	**Complexidade**
Estresse à pilha de execução	*Fibonacci*	$O(\Phi^n)$
Estresse à memória primária		
- Busca de valores	*Busca Sequencial*	$O(n)$
- Troca de valores	*Ordenação – Bolha*	$O(n^2)$
	Ordenação – Seleção	$O(n^2)$
	Ordenação – Inserção	$O(n^2)$
Estresse à memória secundária	*Cópia de Arquivo*	$O(n)$
Estresse à ULA	*Mult. de Matrizes Inteiras e Decimais*	$O(n^3)$
Simulação de um sistema	*Simula*	$O(n^3)$

Esclarecendo: $\Phi = \frac{1+\sqrt{5}}{2} \approx 1,61803...$

Como pode ser observado na Tabela 1, optou-se por algoritmos de alta complexidade assintótica que estressam certa região de execução. A lógica por trás de cada algoritmo é fazer o uso repetitivo de determinados tipos de operação.

O objetivo do algoritmo recursivo *Fibonacci* é analisar o comportamento de sistemas C/C++ e Java em relação ao estresse da pilha de execução. Em relação ao estresse à memória primária, o objetivo é analisar o comportamento de sistemas em relação à leitura e escrita de valores em memória primária. O algoritmo *Busca Sequencial* visa analisar o comportamento de sistemas na recuperação de valores sequenciais em memória. Em virtude do diferente comportamento adotado para a ordenação, foram selecionados três algoritmos para troca de valores. Por exemplo, o *Bolha* tende a realizar mais trocas que o *Seleção*, e o *Inserção* tende a realizar trocas mais próximas.

[4]Na página `http://www.dcc.ufmg.br/~terra/sblp2010` encontra-se publicamente disponível o *benchmark* desenvolvido.

O algoritmo *Cópia de Arquivo* consiste basicamente na leitura de um arquivo binário e posterior gravação em um outro arquivo. Logo, esse algoritmo estressa a memória secundária – disco rígido, no caso – por meio de diversas operações de entrada e saída (E/S). E, o algoritmo *Multiplicação de Matrizes* – que possui a maior complexidade – tem o intuito de analisar o comportamento de sistemas em relação a execução repetitiva de operações matemáticas inteiras e decimais (ponto flutuante).

Por fim, foi idealizado e desenvolvido um aplicativo completo chamado `Simula` que tem o intuito de realizar atividades fundamentais realizadas em aplicativos, tais como recursões, acesso à disco, ordenações, operações matemáticas etc. Sua funcionalidade consiste em, inicialmente, declarar e inicializar matrizes inteiras quadradas de 200 linhas e colunas (`A` e `B`) seguindo uma distribuição uniforme. Em seguida, o valor de cada posição da matriz `A` é substituído pelo fatorial de seu valor e o valor de cada posição da matriz `B` é substituído pelo seu valor na série de *fibonacci*. Em seguida, as matrizes são multiplicadas e a matriz resultante é convertida para um arranjo unidimensional que é ordenado pelo algoritmo de seleção e então é realizada uma busca sequencial procurando encontrar um determinado valor. De forma sintética, o `Simula` tem seu *pseudo*-código descrito na Listagem 1.

```
1 x := {1, 2, 3, ...} /* Incrementa quando utilizado */
2 para todo elemento em vA[200][200] faça fat(x mod 30)
3 para todo elemento em vB[200][200] faça fib(x mod 30)
4 vAB := mult(vA,vB)
5 v[40000] := linear(vAB)
6 selecao(v)
7 sequencial(v, x mod 30)
```

Listagem 1. *Pseudo*-código do Aplicativo `Simula`

3. Comparativo das Linguagens

Nesta seção, descreve-se uma análise comparativa da eficiência das linguagens. O ambiente do experimento – máquina, versões, parâmetros de customização da compilação etc – já foram identificados e descritos na Subseção 2.1, e a carga de trabalho já foi apresentada na Subseção 2.2. A análise comparativa consiste na avaliação do desempenho dos algoritmos do `TerraBM`, exceto o *Simula* que foi implementado para a definição do tamanho da amostra e para a avaliação do impacto das melhorias dos compiladores C/C++ e Java no tempo de execução do código gerado (realizada na Seção 4). É importante mencionar que a métrica utilizada para a experimentação é o tempo de CPU[5] utilizado por cada algoritmo, em contrapartida a diversos artigos que consideraram o tempo decorrido, o que pode levar a conclusões incorretas [1].

3.1. Determinando o Tamanho da Amostra

Para a realização de observações pareadas, o primeiro passo é determinar o tamanho da amostra para que os resultados possuam validade estatística. A fórmula estatística que

[5]Na linguagem C, utilizou a estrutura `rusage` da biblioteca `sys/resource.h` e, na linguagem Java, utilizou-se a classe `java.lang.management.ThreadMXBean`. O código completo para se medir o tempo de execução pode ser observado nos arquivos fontes publicamente disponíveis.

permite determinar o tamanho da amostra considerando a quantidade de observações preliminares (n), a média aritmética ($\overline{x}$), o desvio padrão (s) dos resultados, a confiança desejada (p), a taxa de erro considerada (r) e o valor da distribuição *student-t* referente (pois a amostra preliminar é inferior a 30) é transcrita a seguir [3]:

$$tamanho\ da\ amostra = \sqrt{\frac{100.t_{[p;\ n-1]}.s}{r.\overline{x}}} \quad (1)$$

Inicialmente, para essa determinação, foi definida uma confiança de 99,9% e taxa de erro de 0,05%[6]. Desse modo, as versões do aplicativo `Simula` para C/C++ e para Java foram executadas 10 vezes. Em seguida, determinou-se o tamanho da amostra por meio da resolução da fórmula supracitada cujos valores e resultados obtidos são apresentados na Tabela 2.

Tabela 2. Definição do Número de Amostras

	C/C++	**Java**
Observação Preliminares (n)	10	10
Média Aritmética das Observações ($\overline{x}$)	29,4674	31,9720
Desvio Padrão (s)	0,0021	0,0162
$t_{[p;r]} = t_{[0,9995;9]}$	4.781	4.781
Taxa de Erro (r)	0,05	0,05
Tamanho da Amostra	**1**	**24**

Como pode ser observado na Tabela 2, o desvio padrão (s) das observações de C/C++ foi muito baixo, indicando que apenas um única execução em C/C++ garantiria validade estatística, o que retrata a estabilidade da linguagem e do ambiente de execução. Por outro lado, como o desvio padrão de Java é superior ao de C/C++, o número de amostras para validade estatística para sistemas Java é de 24 observações. Logo, seguindo uma abordagem estatística conservadora, adotou-se 24 observações como o tamanho da amostra para a observação pareada.

3.2. Observações Pareadas

Calculado o tamanho da amostra, os seguintes algoritmos do `TerraBM` foram executados com seus devidos parâmetros:

- *Fibonacci*: foi avaliado o desempenho para $fib(40)$ e $fib(45)$;
- *Busca Sequencial*: foi avaliado o desempenho para uma busca de um número inexistente em um arranjo de $10E6$ posições. O fato do tamanho do arranjo ser consideravelmente maior quando comparado com os outros experimentos se deve ao fato da complexidade desse algoritmo ser $O(n)$;
- *Bolha*, *Seleção* e *Inserção*: foi avaliado o desempenho para ordenação de um arranjo em ordem descendente de 30.000 e de 100.000 posições sem repetição de valores;

[6] É uma confiança altamente adequada em comparação de algoritmos [1, 3].

- *Cópia de Arquivo (CP)*: foi avaliado o desempenho para cópia de um arquivo de 100MB. Para avaliar a melhoria proporcionada pela nova biblioteca de Java para manipulação de *streams*, comparou-se o desempenho do algoritmo C/C++ com o de Java utilizando `java.io` (IO) e utilizando `java.nio` (NIO);

- *Multiplicação de Matrizes Inteiras (MMI) e Decimais (MMD)*: foi avaliado o desempenho da multiplicação de uma matriz quadrada de tamanho 800 com uma outra matriz de mesmo tamanho.

A Figura 1 apresenta a média aritmética ($\overline{x}$) com o intervalo de confiança (IC) do tempo de execução. Todos os algoritmos tiveram o mesmo ambiente de execução e foram executados 24 vezes. Diante disso, os resultados apresentados são estatisticamente válidos com uma confiança de 99,9% e com um taxa de erro de 0,1%.

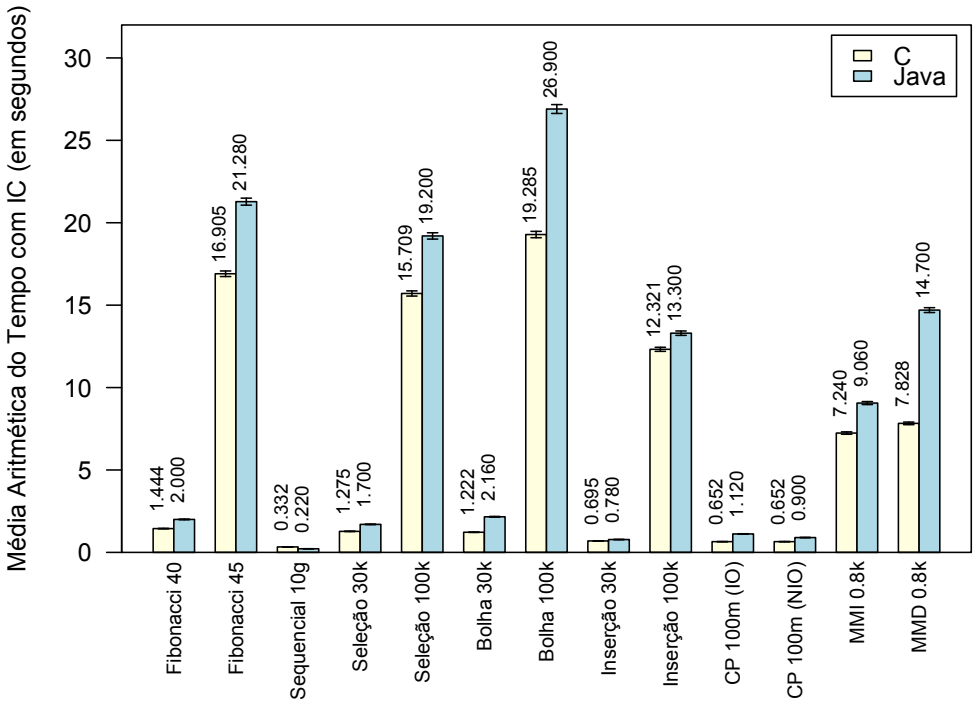

Figura 1. Tempo de Execução dos Algoritmos em C/C++ e Java do TerraBM

3.3. Análise dos Resultados

É de se esperar que todos os algoritmos implementados em C/C++ sejam mais eficientes que seu correspondente em Java, justamente pelo processo de compilação. Entretanto, o algoritmo *Busca Sequencial* apresentou melhor eficiência em Java. Isso é justificado por pelo menos duas razões: (i) a partir do Java 5, a JVM possui um recurso chamado compilador *Just In Time* (JIT) que consiste na tradução, em tempo de execução, do *bytecode* para código de máquina nativo de trechos de código com grande custo de processamento [7]; (ii) Somente o compilador JIT não explicaria o desempenho ser melhor que C. Por isso, suspeita-se que a compilação dinâmica (em tempo de execução) deu ao compilador mais oportunidades para otimização [9].

Com o intuito de validar os resultados das observações pareadas, foi aplicada sobre a diferença das médias aritméticas um teste chamado *t-test*, com o objetivo de comprovar que os desempenhos dos algoritmos das duas linguagens são estatisticamente diferentes. Assim, após a condução desse teste para cada par de observações, foi possível garantir que todos os algoritmos em C/C++ possuem desempenho significativamente diferente do seu correspondente em Java.

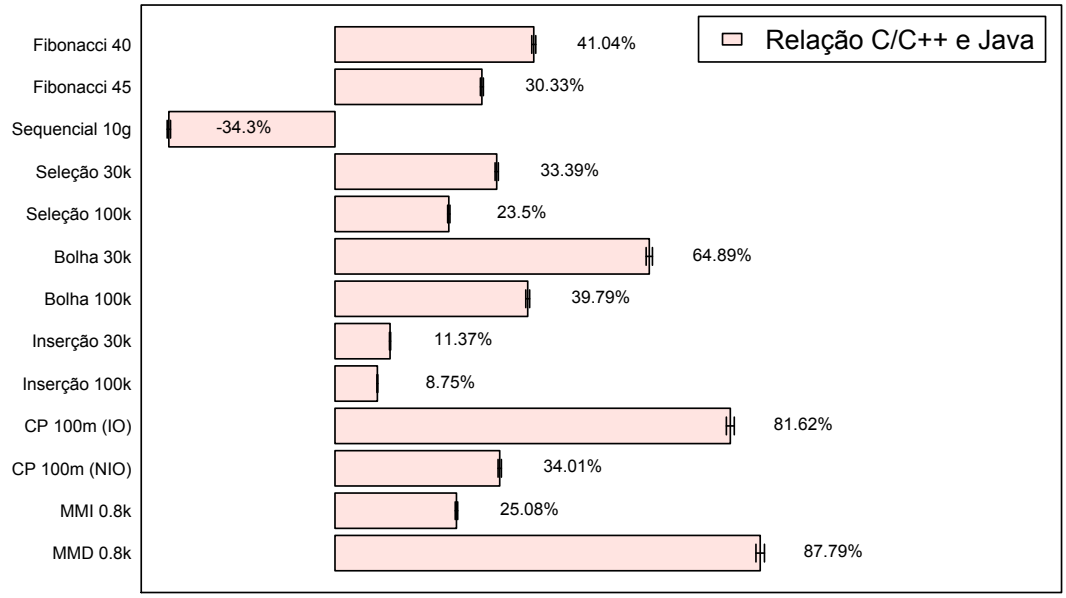

Figura 2. Percentual da Melhoria dos Algoritmos em C/C++ e Java

Ainda, para facilitar o raciocínio sobre os resultados e apontar o quanto melhor o aplicativo em C/C++ é do seu correspondente em Java, foram sumarizados os resultados sob a razão das médias aritméticas para cada algoritmo, conforme ilustrado na Figura 2. Convém mencionar que a diferença também foi calculada com base em uma confiança de 99,9%.

Assim, com base nos resultados apresentados nas Figuras 1 e 2, foram realizadas análises por classe de algoritmo:

Estresse à pilha de execução: Mesmo Java possuindo uma arquitetura de pilha, o algoritmo recursivo *Fibonacci* em Java possui desempenho aproximadamente um terço pior que em C/C++. Contudo, a diferença caiu de 41,04% com $n = 40$ para 30,33% com $n = 45$. Experimentalmente, observa-se que a diferença cai com o aumento de n.

Estresse à memória primária - Busca de Valores: Em relação ao algoritmo *Busca Sequencial*, Java teve desempenho quase 35% melhor que em C/C++. Pode-se dizer que o resultado foi um tanto quanto inesperado principalmente pela linguagem Java não possuir ponteiros para tipos primitivos, isto é, o uso de indexação em arranjos é inevitável, o que foi facilmente evitado, na implementação em C/C++, por meio de ponteiros.

Por outro lado, o resultado possibilitou as seguintes inferências: (i) afirmar que iterar um arranjo em Java é mais lento que em C devido a não existir aritmética de ponteiros não é correto. É observável que o compilador de Java otimiza a iteração internamente;

(ii) o compilador JIT foi utilizado na conversão do laço de procura. Afirma-se isso devido ao fato que na execução com arranjos pequenos, o desempenho de C/C++ foi superior ao de Java, pois o compilador JIT é utilizado somente quando o trecho de código é de alto custo de processamento [7]; (iii) suspeita-se que informações em tempo de execução proporcionadas pela compilação dinâmica de Java foram utilizadas, pois os outros recursos somente garantiriam um desempenho equiparável, mas não melhor.

Estresse à memória primária - Troca de Valores: As diferenças entre os algoritmos de ordenação variaram de 8,75% até 64,89%. A menor diferença ocorreu no algoritmo *Inserção* seguido do *Seleção* e *Bolha*. O fato de a diferença entre as linguagens ficar mais evidente nos algoritmos *Seleção* e *Bolha* – algoritmos com maior número de trocas de valores – indica que existe um gargalo significativo em Java no quesito de alteração de valores em memória primária.

Além disso, não se pode atribuir gargalo à não existência de ponteiros, uma vez que o algoritmo *Busca Sequencial* demonstrou que internamente Java otimiza a iteração de arranjos.

Estresse à memória secundária: Há pouco tempo atrás, a API `java.io` foi muito criticada devido a seu desempenho ser bem inferior ao desempenho de outras linguagens [5, 6]. Por exemplo, no algoritmo *Cópia de Arquivo* utilizando `java.io`, o desempenho de Java foi mais de 80% pior que o correspondente em C/C++. Contudo, ao se utilizar a API `java.nio` – biblioteca desenvolvida pela Sun em respostas às críticas de desempenho de `java.io` – essa diferença caiu substancialmente, sendo agora pouco mais de 30%.

Convém mencionar que o desempenho de cópia de arquivos é diretamente influenciado pelo tamanho do *buffer* utilizado. Assim, definiu-se o tamanho do *buffer* em 32KB tanto para os algoritmos em C/C++ quanto para os de Java.

Estresse à ULA: Em relação aos algoritmos de estresse à ULA, o desempenho de Java é bem inferior ao de C/C++. Por exemplo, no algoritmo *Multiplicação de Matrizes* com valores inteiros, o desempenho de Java foi mais de 25% pior que o desempenho em C. No entanto, essa diferença vai para mais de 85% quando utiliza-se valores decimais. Assim, conclui-se que Java ainda é lenta em operações aritméticas, principalmente quando os operandos são pontos flutuantes.

3.4. Riscos à Validade do Estudo

A comparação de desempenho entre Java e C++ apresentada nesta seção foi baseada em um conjunto de aplicações que fazem parte do *benchmark* `TerraBM`. Procurou-se com esses programas exercitar diversas operações importantes nas duas linguagens, incluindo acesso à memória principal, acesso à memória secundária, operações aritméticas etc. No entanto, não se pode argumentar que esses aplicativos – e as operações principais utilizadas na implementação dos mesmos – são representativos de qualquer programa que possa ser implementado em Java ou C++. Particularmente, em sua atual versão, o *benchmark* `TerraBM` não inclui programas que fazem uso intensivo de operações para gerenciamento de objetos (criação, remoção via coletor de lixo etc).

Adicionalmente, mediu-se os tempos de execução de cada programa e, com base

nos resultados obtidos, foram formuladas hipóteses para explicar as diferenças de desempenho observadas entre Java e C++. No entanto, comprovações e explicações definitivas para as diferenças medidas nos experimentos devem demandar uma análise detalhada dos códigos gerados e das possíveis otimizações realizadas pelos compiladores.

4. Avaliação do Impacto das Melhorias dos Compiladores

Nesta seção, realiza-se uma análise do impacto das melhorias dos compiladores no tempo de execução do código gerado em relação a dois principais fatores: parâmetros de customização da compilação e versões do compilador. Em outras palavras, deseja-se avaliar o quanto os parâmetros de customização da compilação e a versão do compilador influenciam no tempo de execução de sistemas. Para isso, foi realizado um projeto estatístico conhecido como Projeto Fatorial $2^k r$ cujo principal objetivo é classificar a importância desses fatores no tempo de execução de sistemas.

4.1. Projeto Fatorial $2^k r$

Um Projeto Fatorial é um tipo especial de projeto experimental estatístico que determina os efeitos de k fatores, cada qual com dois níveis [3]. Esse tipo de projeto merece uma discussão especial, pois seus resultados são de fácil análise e ajudam consideravelmente na classificação dos fatores pelo seu impacto na resposta. O Projeto Fatorial $2^k r$ consiste em, além de observar todas as possíveis combinações, replicar todas as observações r vezes, isto é, tem-se $2^k r$ observações. Isso permite estipular intervalo de confiança para os efeitos de cada fator, das interações entre eles e ainda obter o impacto relativo ao erro experimental.

Sumarizando, o Projeto Fatorial $2^k r$ é utilizado para determinar o efeito de k fatores – em que cada fator possui dois níveis – em um experimento com uma confiança p e com r replicações. Um exemplo clássico desse tipo de projeto consiste na avaliação do impacto da memória e do processador no desempenho de um sistema. Os fatores seriam o tamanho da memória e a velocidade do processador. Como é um projeto fatorial $2^k r$, cada fator possui dois níveis, por exemplo, 1GB e 4GB para memória e 1GHz e 2GHz para o processador. A solução desse projeto permite avaliar qual o impacto da memória e do processador no desempenho do sistema. Logo, permitindo definir qual investimento seria mais apropriado para otimizar o desempenho do sistema: memória e/ou processador [3].

Neste estudo, os fatores considerados são os parâmetros de customização da compilação (A) e versões do compilador (B), pois se deseja obter o impacto desses fatores no tempo de execução de sistemas em cada linguagem. Similarmente ao processo experimental anterior, adotou-se a confiança de 99,9% e, ainda seguindo uma abordagem estatística conservadora, adotou-se 24 replicações para ambas as linguagens.

Para os parâmetros de customização da compilação, os níveis se baseiam na existência (+1), mais especificamente na utilização dos parâmetros identificados e descritos na Subseção 2.1, e na não existência (-1) que consiste no uso da compilação padrão da linguagem. Isso permite que se avalie o quanto a utilização de parâmetros de customização da compilação impactam no tempo de execução de sistemas. Para as versões dos compilador, foram selecionadas as versões de cinco anos atrás (-1) e as versões mais recentes (+1). Mais especificamente, foram selecionadas as JDKs 1.5.0_22 e 1.6.0_20 para o modelo de regressão de Java e os compiladores GNU GCC 4.0.4 e 4.5.0 para o modelo de regressão

de C/C++. Isso permite que se avalie o quanto cinco anos de desenvolvimento de compiladores impactam no tempo de execução de sistemas. Com a definição do problema, utilizou-se o seguinte modelo de regressão não linear [3]:

$$y = q_0 + q_A x_A + q_B x_B + q_{AB} x_A x_B + e \quad (2)$$

Nesse modelo, q_0 indica a média aritmética do tempo de execução, q_A indica o efeito dos parâmetros de customização do compilador, q_B indica o efeito da versão do compilador, q_{AB} indica o efeito da interação entre os parâmetros de customização e a versão do compilador e, por fim, e indica o efeito relativo aos erros experimentais. Ainda, x_A é o nível do fator A que pode ser +1 ou -1 dependendo da utilização ou não de parâmetros de customização da compilação, x_B é o nível do fator B que pode ser +1 ou -1 dependendo da versão do compilador utilizada. Por fim, y é a variável resposta.

Para um melhor entendimento dos experimentos, a Tabela 3 apresenta as médias aritméticas do tempo de execução de cada uma das combinações descritas anteriormente para C/C++ e Java.

Tabela 3. Médias aritméticas dos tempos de execução do Projeto Fatorial $2^k r$ para C/C++ e Java

	A **GCC**		*A* **JDK**	
	(-1) **4.0.4**	(+1) **4.5.0**	(-1) **1.5.0_22**	(+1) **1.6.0_20**
B (-1) **Sem otimização**	61,388	59,088	43,408	34,354
(+1) **Com otimização**	27,636	29,467	37,388	31,975

A partir dos resultados, determinou-se o seguinte modelo de regressão para a linguagem C/C++:

$$y_{C/C++} = 44,395 - 15,844 x_A - 0,117 x_B + 1,033 x_A x_B + e \quad (3)$$

Segue-se a interpretação do modelo de regressão: o tempo médio é de 44,395s; o efeito dos parâmetros de otimização do compilador é de -15,844s; o efeito da versão do compilador é de -0,117s; e o efeito da interação entre os parâmetros de customização e a versão do compilador é de 1,033s.

A partir do modelo de regressão, pode-se calcular e entender a fração da variação total causada por cada efeito. A fração explicada pelos parâmetros de customização da compilação foi de 99,57% e pela versão do compilador foi de 0,01%. A fração explicada pela interação entre os fatores foi de apenas 0,42% e praticamente nada foi atribuído como erro experimental. É importante mencionar que a fração explicada pelos erros experimentais é praticamente nula devido ao desvio padrão dos algoritmos C/C++ serem muito baixos.

A partir dos resultados, determinou-se o seguinte modelo de regressão para a linguagem Java:

$$y_{JAVA} = 36,781 - 2,1 x_A - 3,617 x_B + 0,91 x_A x_B + e \quad (4)$$

Segue-se a interpretação do modelo de regressão: o tempo médio é de 36,781s; o efeito dos parâmetros de otimização do compilador é de -2,1s; o efeito da versão do compilador é de -3,617s; e o efeito da interação entre os parâmetros de customização e a versão do compilador é de 0,910s.

A partir do modelo de regressão, pode-se calcular e entender a fração da variação total causada por cada efeito. A fração explicada pelos parâmetros de customização da compilação foi de 24,06% e pela versão do compilador foi de 71,41%. A fração explicada pela interação entre os fatores foi de 4,52% e praticamente nada foi atribuído como erro experimental.

4.2. Análise dos Resultados

A partir dos resultados obtidos para a linguagem C/C++, suspeita-se da estabilidade dos compiladores C/C++, uma vez que não vem apresentando grande melhoria de uma versão para outra. Isso se justifica pelo fato dos compiladores GNU GCC já terem 23 anos de desenvolvimento. Diante disso, conclui-se que, em busca de um melhor desempenho em C, mais esforços devem ser dados aos parâmetros de customização da compilação, os quais representaram o efeito dominante.

Em contrapartida, a partir dos resultados obtidos para a linguagem Java, foi constatado que os compiladores ainda estão em constante evolução, uma vez que representaram mais de 70% da melhoria do tempo de execução. Isso era esperado já que a linguagem Java é relativamente nova.

Além disso, pode-se perceber que os parâmetros de customização da compilação também são importantes, representando cerca de um quarto do tempo de execução. Contudo, essa fração é bem inferior àquela resultante no modelo da linguagem C. Isto é, na última versão do compilador C/C++, o tempo sem otimização foi o dobro do tempo com otimização, uma diferença de quase 30s. Por outro lado, na linguagem Java, essa diferença foi de menos de 8%, uma diferença de menos de 3s. Isso leva a conclusão de que os compiladores Java já agregam diversos parâmetros de customização em sua compilação padrão, deixando-a bem próxima à máxima otimização.

Convém mencionar que o desempenho otimizado da última versão do compilador de C/C++ foi inferior ao desempenho otimizado da versão do compilador de cinco anos atrás. O motivo disso podem ser vários, contudo dois podem ser mencionados: (i) a última versão do compilador C/C++ foi lançada há pouco tempo e ainda está na liberação 0 – 4.5.$\underline{0}$ – enquanto que a versão de cinco anos atrás parou na liberação 4 (4.0.$\underline{4}$); (ii) nos últimos cinco anos, várias modificações foram realizadas no intuito de sistemas serem otimizados para máquinas 64 bits e multi-núcleos, o que pode ter impactado o desempenho em máquinas 32 bits de apenas um núcleo, a qual caracteriza a máquina utilizada pelo experimento.

4.3. Riscos à Validade do Estudo

No estudo descrito, considerou-se apenas duas versões de compiladores C++ e duas versões de compiladores Java, com uma diferença de cinco anos entre cada um deles. Evidentemente, esse estudo poderia ser ampliado, considerando um número maior de versões de compiladores.

5. Trabalhos Relacionados

Desempenho de linguagens é um assunto de grande interesse à academia. Logo, vários trabalhos – cada qual com seu enfoque – foram propostos. Inicialmente, Georges *et al.* afirmam que as metodologias dominantes de avaliação de desempenho podem estar se enganando e que diversos artigos podem estar apresentando conclusões incorretas. Logo, realizam uma avaliação estaticamente rigorosa do desempenho da linguagem Java para diversos *benchmarks* validando seus resultados [1]. No entanto, não comparou o desempenho com outras linguagens.

Em relação a pesquisas comparativas, Martins e Botelho estudaram a evolução do desempenho da linguagem Java entre as suas versões e compararam o desempenho de Java com C++ em relação ao tempo de processamento e ao tempo de leitura sequencial em disco concluindo que a diferença em relação ao tempo de processamento não é mais tão significativa [5]. Contudo, nenhuma técnica estatística foi utilizada e o espectro de aplicações é relativamente pequeno.

Em relação a otimização da linguagem Java, Reinholtz analisou a compilação dinâmica de Java e concluiu que é possível que Java tenha um desempenho melhor do que C++, pois a compilação dinâmica fornece ao compilador Java acesso a informações em tempo de execução que não estão disponíveis a um compilador C++ [9].

Em uma outra linha, Prechelt compara o desempenho de 40 diferentes implementações de um mesmo problema escritas por 38 diferentes desenvolvedores [8]. Diferenciando-se da análise realizada pela maioria dos *benchmarks*, que comparam uma única implementação de um problema em C/C++ com uma única implementação em Java. Com esse estudo, Prechelt concluiu que a variação que ocorre entre as implementações de cada desenvolvedor supera a diferença de desempenho da linguagem em si.

6. Considerações Finais

A portabilidade da linguagem Java é uma das suas maiores vantagens e, também, o fator causador de uma de suas maiores críticas: desempenho. Desenvolvedores normalmente optam pela linguagem Java visando distribuir seus aplicativos para vários sistemas operacionais. Contudo, como é de se esperar, querem a portabilidade sem abrir mão da eficiência de uma linguagem compilada como C/C++.

Para avaliar o desempenho da linguagem Java, inicialmente, elaborou-se uma análise comparativa entre as linguagens C/C++ e Java. Essa análise utilizou toda fundamentação estatística para a validação dos experimentos e consistiu basicamente em observações pareadas de diversas classes de algoritmos que visam estressar certa região de execução, por exemplo, memória primária, ULA, pilha etc.

Em um segundo momento, foi elaborada uma análise do impacto das melhorias dos compiladores C/C++ e Java no tempo de execução do código gerado nesses últimos cinco anos. Essa análise foi realizada por meio de um Projeto Fatorial $2^k r$ em relação a dois fatores: versão do compilador e parâmetros de customização da compilação.

A partir desses estudos, formou-se argumentos para se responder às perguntas enunciadas no início deste estudo. A linguagem Java não tem sempre pior desempenho que a linguagem C. Em situações em que o compilador JIT é utilizado e informações obtidas pela compilação dinâmica permitem a otimização do código, o desempenho de

Java pode ser superior. Isso foi constatado pelo algoritmo *Sequencial* que obteve um desempenho quase 35% melhor. Além disso, observou-se que operações de troca de valores em memória primária e operações aritméticas com pontos flutuantes tem um desempenho pouco satisfatório em Java. Logo, mais pesquisas devem ser conduzidas no intuito de melhorar o compilador Java para essas operações.

Em relação ao impacto das melhorias dos compiladores no tempo de execução do código gerado, foi constatado que os compiladores Java estão em constante aperfeiçoamento, sendo sua nova versão responsável por 70% da melhoria de desempenho. Isso contrasta com o resultado do compilador C, em que praticamente toda a melhoria foi proporcionada pelos parâmetros de customização da compilação. Isso indica que, em busca de um melhor desempenho em Java, a atualização para uma nova versão do compilador é uma boa opção, enquanto que em C, mais esforços devem ser dados aos parâmetros de otimização da compilação.

Possíveis linhas de trabalho futuro incluem: (i) realizar uma análise comparativa entre todas as versões dos compiladores dos últimos cinco anos, a fim de obter um estudo da evolução entre as versões; (ii) realizar uma análise do comportamento dos compiladores em processadores 64 bits; (iii) realizar uma análise do comportamento dos compiladores em máquinas multi-núcleos.

Agradecimentos: Este trabalho foi apoiado pela FAPEMIG e CNPq.

Referências

[1] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, 2007.

[2] C. S. Horstmann and G. Cornell. *Core Java, Volume I – Fundamentals*. Prentice Hall, 8 edition, 2007.

[3] R. Jain. *The art of computer system performance analysis: techniques for experimental design, measurement, simulation, and modeling*. John Wiley & Sons, 1991.

[4] J. P. Lewis and U. Neumann. Performance of Java versus C++, 2004. Disponível em: http://www.idiom.com/ zilla/Computer/javaCbenchmark.html.

[5] H. B. Martins and F. C. Botelho. Java não é mais tão lento. *Revista Eletrônica de Iniciação Científica*, 2008.

[6] S. Microsystems. JSR 51: New I/O APIs for the Java platform, 2010. Disponível em: http://www.jcp.org/en/jsr/detail?id=51.

[7] S. Microsystems. Performance features and tools, 2010. Disponível em: http://java.sun.com/developer/onlineTraining/Programming/JDCBook/perf2.html.

[8] L. Prechelt. Technical opinion: comparing Java vs. C/C++ efficiency differences to interpersonal differences. *Commun. ACM*, 42(10):109–112, 1999.

[9] K. Reinholtz. Java will be faster than C++. *SIGPLAN Not.*, 35(2):25–28, 2000.