Estudo de Técnicas para Melhora do desempenho de Aplicações Java

Elton N. Mathias[*], Guilherme Koslovski[*], Márcia C. Cera[†], Rodrigo da R. Righi[‡], Marcelo Pasin

Universidade Federal de Santa Maria - Laboratório de Sistemas de Computação
Faixa de Camobi, Km 9 - CEP 97105-900 - Santa Maria - RS - Fone/Fax: (55) 220 8523
{emathias, koslovski, cera, rodrigor, pasin}@inf.ufsm.br

Introdução

A linguagem Java vem sendo cada vez mais utilizada por ser uma linguagem de programação simples e flexível. Ela segue o modelo de programação orientado a objetos apresentando características como herança, polimorfismo, facilidade no reuso de código, portabilidade, coleta de lixo automática, entre outras. Além de ser utilizada na implementação de programas seqüenciais, Java também tem sido utilizada na programação paralela e distribuída por oferecer suporte nativo à programação com múltiplos fluxos de execução (*multithreading*) e com memória distribuída.

A fim de proporcionar flexibilidade e portabilidade, os códigos fonte são geralmente traduzidos para código binário de uma arquitetura neutra chamada de *bytecode*. Um arquivo de *bytecode* pode ser interpretado em qualquer plataforma através de uma implementação da Máquina Virtual Java (JVM). As facilidades agregadas pelo paradigma orientado a objetos, o processo de interpretação de *bytecodes*, a verificação de exceções e coleta automática de lixo, entre outros, acabam por prejudicar o desempenho final de aplicações Java.

O objetivo desse artigo é apresentar algumas iniciativas que buscam proporcionar ganho de desempenho em aplicações Java. Para testar a eficiência de algumas dessas iniciativas, foram implementadas duas aplicações com características distintas, a partir das quais será possível identificar situações onde tais iniciativas podem ser eficazes.

Java e Alto Desempenho

O mecanismo padrão para a execução de programas Java é a interpretação; entretanto, existem outros métodos que exploram a compilação e a tradução de códigos fonte Java para linguagens de programação intermediárias. Esses métodos podem ser vistos na figura 1 e serão melhor explicados no decorrer dessa seção.

A **interpretação** emula a operação de um processador no momento da execução de um programa. Esse tipo de abordagem oferece algumas vantagens, como a simplicidade de implementação e portabilidade [KAZ 2000]. Contudo, a sua principal penalidade sobrecai sobre o seu desempenho, que normalmente é baixo.

[*]Fomento: CNPq.
[†]Fomento: CAPES e FIPE.
[‡]Fomento: CNPq e FIPE.

Existem também, como nas linguagens tradicionais, compiladores que transformam arquivos fonte em binários para serem executados no processador alvo. Um exemplo são os **compiladores JIT** (*Just In Time*), que efetuam a compilação de partes do código fonte em tempo de execução. A maioria das JVMs modernas incorpora compilador JIT por padrão.

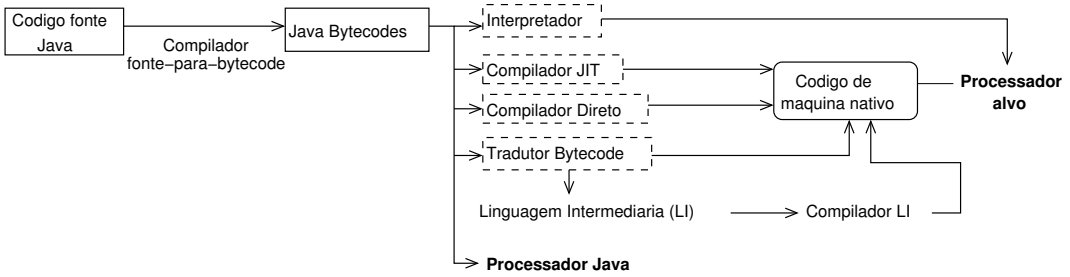

Figura 1: Mecanismos para execução de programas Java.

Outra alternativa entre as técnicas de compilação é a utilização de **compiladores diretos**, que traduzem em tempo de compilação um código fonte Java, ou mesmo um *bytecode*, para instruções de código de máquina. Compiladores diretos resultam na perda da portabilidade, uma vez que o código gerado pode ser executado somente em um processador específico.

Ainda referente a compiladores, existem aqueles que traduzem *bytecode* para arquivos fonte intermediários, chamados de **compiladores bytecode-para-fonte**. Eles geram código fonte para uma linguagem de programação de alto nível, como C. Após esse processo, são usados compiladores tradicionais para produzir código nativo. O compilador GCJ [TRO 2004] é um exemplo desse tipo de compilador.

Há também, integrado à distribuição Java, desde 1997 [LIA 99], a **JNI** (*Java Native Interface*). Ela atua como uma interface entre aplicativos escritos em Java e códigos compilados escritos em linguagem C ou C++. Essa alternativa favorece o programador pois permite a ele a utilização de bibliotecas já escritas nessas linguagens.

Aplicações Desenvolvidas

Para uma avaliação mais consistente das alternativas previamente abordadas foram implementados duas aplicações com características distintas. A primeira aplicação trata-se do cálculo do determinante de uma matriz segundo o teorema de Laplace. A segunda realiza o método de força bruta sobre o algoritmo de criptografia DES.

Cálculo de Determinantes Segundo o Teorema de Laplace

O cálculo de determinante de matrizes está presente em diversas soluções de sistemas relacionados à álgebra linear e cálculo numérico O teorema de Laplace permite o cálculo do determinante de uma matriz de qualquer ordem através do cálculo de menor complementar. Esse cálculo é gerado a partir da redução recursiva da ordem dessa matriz até obter uma matriz de ordem dois. Nessas matrizes, o cálculo do determinante é trivialmente realizado através da subtração do produto da diagonal principal pelo produto da diagonal secundária. A característica principal dessa aplicação é a grande necessidade de acessos a memória.

Força Bruta sobre o Algoritmo DES

O algoritmo de criptografia simétrica DES (*Data encription Standard*) [MEN 97], desenvolvido na década de 70, tornou-se o algorítmo padrão, mundialmente utilizado de proteção de documentos e sistemas operacionais. O algoritmo é baseado em uma grande quantidade operações básicas. Tais operações incluem translações, transposição, transformações lineares em vetores e operações aritméticas [MEN 97]. Dado à dificuldade de quebra analítica do algoritmo, o método mais utilizado, em aplicações forenses, por exemplo, é o ataque força bruta sobre o algoritmo [MEN 97]. Esta aplicação diferentemente da anterior, necessita poucos acessos à memória; entrentanto, o processamento realizado contém operações mais complexas.

Análise dos resultados

O conjunto de aplicações referidas na seção anterior foi implementado em quatro diferentes cenários. O primeiro cenário se refere à implementação em linguagem C. O segundo e o terceiro utilizam Java, através dos métodos de interpretação pura e do uso de compilação JIT, respectivamente. O quarto cenário é uma implementação híbrida que utiliza a interface JNI para a chamada de métodos nativos implementados em C. Tais alternativas foram escolhidas em detrimento das demais por serem disponibilizadas junto ao SDK (*Software Development Kit*) da Sun.

O ambiente de execução utilizado foi uma máquina Pentium 4, com memória de 512 MB, cache de 512 kb, e sistema operacional Gnu/Linux Gentoo, com núcleo 2.4.26 e a versão JVM utilizada é a *Blackdown-1.4.1-01*. Os resultados foram obtidos a partir da média aritmética de 50 execuções de cada implementação, em cada cenário.

Na Tabela 1 pode-se verificar que com o aumento da ordem da matriz, o Java Interpretado tem desempenho muito aquém da implementação em C. A versão Java com JIT, no entanto, mostra uma boa aproximação em relação ao resultado apresentado pela implementação em C. A diferença de tempo deve-se, principalmente, ao fato de que o acesso á memória pelo código nativo é mais rápido. Isso ocorre por que o armazenamento das estruturas ocorre de forma linear, permitindo um acesso mais veloz aos dados. A implementação híbrida, nesse caso, teve desempenho inferior à implementação feita em Java. Isso acontece porque o processamento exigido é bastante simples, tornando-se significativo o tempo perdido em trocas de contexto entre a linguagem Java e C.

Ordem da Matriz	C	Java Interpretado	Java +JIT	Java + JNI
7	0.0020 s	0.0113 s	0.0135 s	0.0124 s
8	0.0284 s	0.1398 s	0.2262 s	0.1355 s
9	0.4690 s	2.3283 s	0.7664 s	0.8396 s
10	9.1622 s	46.1665 s	10.5051 s	13.1987 s
11	205.9448 s	1026.0725 s	226.9916 s	288.5520 s

Tabela 1: Tempos de Execução em segundos para o Teorema de Laplace

Na Tabela 2 pode-se observar uma análise diferenciada se comparada à tabela anterior. Para essa aplicação o uso de Java e C integrados pela interface JNI mostrou-se bastante satisfatória. Isso mostra que nesse caso o tempo necessário ao processamento

compensou o tempo gasto com trocas de contexto. Pelo fato do processamento ser mais complexo e significativo no tempo final, e de que compiladores normais apresentarem uma melhor otimização que compiladores JIT, a implementação em Java com JIT apresentou desempenho bastante inferior à implementação em C e híbrida. Mais uma vez a execução em Java completamente interpretado mostrou-se bastante lenta.

Número de Iterações	C	Java Interpretado	Java +JIT	Java + JNI
100	0.0096 s	0.4138 s	0.6655 s	0.0104 s
1000	0.0869 s	3.7409 s	1.4438 s	0.0936 s
10000	0.8802 s	37.0341 s	5.2935 s	0.9226 s
1000000	8.8118 s	369.3642 s	43.0287 s	9.0240 s
10000000	88.1529 s	3693.4832 s	420.2414 s	90.0336 s

Tabela 2: Tempos de Execução em segundos para o Algoritmo DES

Considerações Finais e Trabalhos Futuros

Os resultados obtidos mostraram que a utilização de iniciativas que buscam melhorar o desempenho de Java, como as analisadas, é importante, e pode trazer ganhos expressivos. A utilização de compilador JIT alcançou melhores resultados em aplicações que exigiam um processamento mais simples, mesmo com necessidade de mais operações de entrada e saída. JNI por sua vez mostrou-se mais efetivo quando existe uma quantia de cálculos que podem ser executados em código nativo.

Embora os ensaios realizados sejam bastante simples, eles encorajam a continuidade do estudo. Pretende-se, futuramente prosseguir com a avaliação de outras iniciativas que buscam a obtenção de melhoras no desempenho, buscando comparações para o sistema de rede, *threads*, além de estender a análise a bibliotecas distribuídas para processamento de alto desempenho em Java, tais como o Proactive[CAR 98] e Java Party[PHI 97].

Referências

[CAR 98] CAROMEL, D.; KLAUSER, W.; VAYSSIERE, J. Towards seamless computing and metacomputing in java. In: CONCURRENCY PRACTICE AND EXPERIENCE, 1998. **Anais...** Wiley & Sons: Ltd., 1998.

[KAZ 2000] KAZI, I. H. et al. Techniques for obtaining high performance in java programs. **ACM Comput. Surv.**, v.32, n.3, p.213–240, 2000.

[LIA 99] LIANG, S. **Java native interface**: programmer's guide and reference. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 1999.

[MEN 97] MENEZES, A.; OORSCHOT, P. V.; VANSTONE, S. **Handbook of applied cryptography**. [S.l.: s.n.], 1997.

[PHI 97] PHILIPPSEN, M.; ZENGER, M. JavaParty – transparent remote objects in Java. **Concurrency: Practice and Experience**, v.9, n.11, p.1225–1242, Nov. 1997.

[TRO 2004] TROMEY, T. Gcj: the new ABI and its implications. In: GCC Developers Summit, 2004, Ottawa, Ontario, Canada, 2004. **Proceedings...** [S.l.: s.n.], 2004. p.169–174.