6

Programação Paralela e Distribuída em Java

Marinho P. Barcellos[1] *(PIPCA/UNISINOS, marinho@exatas.unisinos.br)*

Resumo:

A linguagem Java oferece extenso suporte à programação paralela e distribuída. Este trabalho aborda as facilidades de Java que auxiliam no desenvolvimento dessa classe de software. Dentre elas, destacam-se o suporte a *threads*, os mecanismos de sincronização (`synchonized`, `wait` e `notify`) e a comunicação em rede (via *Sockets* ou *Remote Method Invocation*). Discute-se então o uso de Java em Processamento de Alto Desempenho, apresentando-se alguns dos principais projetos de pesquisa que seguem essa abordagem.

[1]Doutor em Ciência da Computação pela University of Newcastle upon Tyne, Inglaterra, e Mestre em Ciência da Computação junto ao Curso de Pós-graduação em Ciência da Computação da UFRGS. Professor e Pesquisador junto ao PIPCA - Programa Interdisciplinar de Pós-graduação em Computação Aplicada, UNISINOS. Áreas de Interesse: Redes de Computadores, Processamento de Alto Desempenho e Sistemas Distribuídos.

6.1. Introdução

Desde o seu surgimento (em 1995), Java vem conquistando crescente popularidade entre programadores, engenheiros de software e usuários de aplicativos de rede. Uma das principais razões para o sucesso de Java são suas facilidades para programação paralela e distribuída, fazendo dessa linguagem uma excelente alternativa para o Processamento de Alto Desempenho (PAD). Este trabalho revisa essas facilidades de Java, fornecendo uma introdução ao assunto e apontando referências.

Java não é apenas uma *linguagem* de programação, pois inclui a especificação da linguagem ([GOS96]), de uma API (Application Program Interface) e de uma *máquina virtual*. Juntos, eles definem um ambiente completo de programação e execução. A API é formada por um conjunto de pacotes de classes que implementam funcionalidade não nativa da linguagem. A máquina virtual Java (JVM) executa um código Java através da interpretação de um arquivo em formato especial (`class`) contendo *bytecodes*. A JVM está presente, por exemplo, no comando `java` ou em navegadores Web.

Várias razões contribuiram para que Java atingisse rápido sucesso. É uma linguagem moderna, totalmente orientada a objetos, independente de plataforma e com suporte à programação paralela e distribuída. Threads, mecanismos de sincronização e de comunicação via rede como Sockets e RMI são exemplos desse suporte.

O restante deste texto está organizado da seguinte maneira. As duas primeiras seções tratam de aspectos particularmente pertinentes à programação concorrente: a Seção 6.2. aborda o uso de threads em Java, enquanto a Seção 6.3. discute recursos de sincronização entre as mesmas. A Seção 6.4. trata dos mecanismos de comunicação em rede, fundametal à programação distribuída. A Seção 6.5. discute a recente aplicação de Java ao PAD, enquanto a Seção 6.6. encerra o trabalho com comentários finais.

6.2. Threads

Existem diversas razões para se usar threads em Java ([OAK99]), tais como operações de E/S não bloqueantes, gerência de alarmes e temporizadores, execução de tarefas que são independentes no programa e execução de algoritmos paralelizáveis em multiprocessadores. Java permite que threads sejam criadas de duas maneiras distintas: *herança* ou *interface*. No primeiro caso, uma classe do usuário herda da classe `Thread`, e com ela um método `start()`, que deve ser invocado para que a thread inicie após a mesma ser instanciada. A thread inicia execução em um método especial, `run()`.

O segundo esquema emprega a interface *Runnable*, que possui um único método, `run()`. Um objeto executável é criado e uma referência ao mesmo obtida; essa referência é passada como parâmetro na criação de uma nova thread, que é então iniciada invocando-se o método `start()`. Quando isso ocorre, o método default `run()` da classe `Thread` se limita a invocar `run()` presente na classe do usuário. Considerando que Java não permite herança múltipla, o esquema de criação de threads via interface tem a vantagem de permitir que a classe de usuário herde de outra classe que não `Thread` ([OAK99]).

Outras primitivas importantes da API de threads são `sleep()`, `isAlive()`, `join()` e `interrupt()`. O método `sleep()` bloqueia a thread por um intervalo de tempo passado como argumento. Apesar do tempo de bloqueio poder ser especificado até em nanosegundos, o tempo exibido dependerá da resolução do *timer* do sistema operacional

(usualmente da ordem de millisegundos), do modelo de threads sendo usado (*green threads* ou *nativas*) e do escalonamento aplicado pelo sistema operacional. Já `isAlive()` indica se uma thread está 'viva' ou não. O método `join()` permite a uma thread esperar pela morte de outra, com a possibilidade de especificar um tempo máximo de espera (similarmente a `sleep()`). Por fim, `interrupt()` permite liberar uma (outra) thread que esteja bloqueada em um `sleep()`, `join()` ou `wait()` (vide a seguir). Se `th` é uma thread não bloqueada, quando outra thread executa `th.interrupt()`, um flag do objeto `th` é ativado para indicar que este foi alvo de um `interrupt()`. O flag, que pode ser consultado através de `th.isInterrupted()`, é desligado apenas depois que `th` executa o método estático `interrupted()` ou `sleep()`, ou então `join()` ou `wait()` lançam uma exceção de interrupção ([OAK99], [LEA00]).

Finalmente, um dos atributos da classe `Thread` é um valor inteiro entre 1 e 10 que representa a prioridade da thread, sendo 10 a maior prioridade. Threads são escalonadas de maneira *preemptiva* pela JVM. Por exemplo, se uma thread de prioridade 5 está em execução quando uma outra thread, de prioridade 6, passa ao estado pronta para executar, a JVM logo preemptará a thread de prioridade 5 em favor daquela de prioridade 6. O escalonamento de threads de igual prioridade é deixado em aberto (!): dependendo da implementação do escalonador da JVM, poderá exibir comportamento cooperativo ou preemptivo. Como era de se esperar, esse indeterminismo dificulta bastante a depuração de programas concorrentes.

6.3. Sincronização de Threads

A semântica do modelo de memória empregado por Java corresponde a uma arquitetura simplificada de SMP com memória compartilhada e uma thread por processador com cache e registradores próprios ([LEA00]). Cada thread possui uma *memória de trabalho* (cache e registradores). Uma thread pode manipular valores na memória aliúres a outras threads. O modelo oferece como garantia mínima o acesso atômico a todos os tipos de dados que não `long` ou `double`, embora o modificador `volatile` possa ser usado com esses tipos para garantir atomicidade.

Múltiplas threads concorrendo pelo acesso ao mesmo conjunto de recursos pode, devido à ordem em que as threads são escalonadas, levar a resultados diferentes a cada execução. Esta situação é denominada *condições de corrida*, e usualmente representa um sério problema para a aplicação. Para exemplificar, em uma linguagem puramente seqüencial, o método `verifica()` da Figura 6.1 sempre retornaria `true` ([LEA00]). Em Java, existe a possibilidade, embora pequena, que o método retorne `false` ocasionalmente, devido a um intercalamento das execuções de `liga()` e `verifica()` por duas threads diferentes.

O problema acima se deve à ausência de sincronização na execução dos métodos. Java oferece dois mecanismos de sincronização entre processos, *monitores* e *variáveis condicionais* ([AND00]), a partir dos quais outros mecanismos podem ser elaborados. Em Java, o problema no exemplo acima é facilmente resolvido incluindo-se o modificador `synchronized` nos métodos `liga()` e `verifica()`, o que previne que os mesmos sejam executados simultaneamente por mais de uma thread sobre um mesmo objeto.

A implementação de `synchronized` está baseada em um *lock* pertencente à classe `Object`, da qual todos objetos em Java herdam: existe um lock por objeto. Tipicamen-

```
final class LigaEverifica {
   private int  a = 0;
   private long b = 0;
   void liga() {
      a = 1;
      b = -1;
   }
   boolean verifica() {
     return ((b == 0) ||
             (b == -1 && a == 1));
   }
}
```

Figura 6.1: exemplo de condições de corrida.

te, quando modificado por um `synchronized`, um método que executa sobre um objeto obtém o lock do objeto, processa e então libera o lock. Considere uma classe `Classe`, contendo métodos `m1` e `m2`, ambos `synchronized`, e `o1` e `o2` como objetos instanciados da classe `Classe`. Em primeiro lugar, não há interferência entre métodos de `o1` e `o2`, pois são objetos diferentes e por isso podem ser acessados simultaneamente. Em segundo lugar, considerando um objeto e método arbitrários, digamos `o1` e `m1`, respectivamente, a semântica de Java garante que uma vez iniciada a execução de `m1`, o mesmo não será *preemptado*. Em outras palavras, `m1` completará sua execução ou então intencionalmente travará e abdicará do lock. Este último caso acontece quando um método (como `m1`) em `synchronized` invoca o método `wait()` sobre um objeto. A execução de `m1` será retomada apenas após certas condições serem satisfeitas, incluindo a retomada do lock. O mecanismo de `wait()`/`notify()` é explicado a seguir.

O método `wait()` é utilizado em conjunto com `notify()` em diversas situações com threads, notadamente quando uma thread precisa esperar que uma variável de condição se torne verdadeira. A opção sem `wait()` e `notify()` exige que a thread faça um *polling* periódico da variável. É importante, e frequentemente difícil, determinar um perído adequado para o polling, pois se o período for curto, a thread desperdiçará ciclos de processador acessando a variável desnecessariamente, e sobrecarregará o mesmo; no entanto, se for longo demais, haverá um período de espera prolongado entre a ativação da variável e a liberação da thread. Com `wait()`, uma thread pode esperar por uma condição sem desperdiçar tempo de processador, pois é bloqueada. Mais precisamente, se `o1` é um objeto, com `o1.wait()` a thread (libera o lock sobre `o1` e) é bloqueada até que a thread seja interrompida (via `interrupt()`, gerando uma excessão), ou a thread seja acordada via `o1.notify()` ou `o1.notifyAll()`.

O modificador `synchronized` pode ser usado também diretamente sobre blocos de código, ao invés de métodos. Neste caso, junto ao `synchronized` é especificado um objeto, cujo lock deverá ser obtido. Tipicamente, ou é especificado o objeto a ser acessado dentro do bloco (como mostrado na Figura 6.2), ou `this` é usado. O `synchronized` de bloco permite diminuir o escopo de um lock, aspecto fundamental no desempenho de um programa concorrente em Java: um escopo grande demais limita a concorrência, enquanto um pequeno demais aumenta o overhead em função da frequente aquisição e liberação de locks.

Por fim, a sincronização é afetada diretamente pelo escalonamento das threads.

```
public void tranca() { ...
synchronized (ob) {
   try {
     ob.wait();  // espera por notify em ob
   } catch (Exception e) {...
   }
}
...
public void libera() { ...
synchronized (ob) {
   sb.notify(); // libera thread qualquer em wait de ob
}
```

Figura 6.2: exemplo do uso de synchronized.

Existem diversas razões diferentes pelas quais uma thread pode perder o processador, como por exemplo: operação de E/S; acesso a um objeto com `synchronized`; execução dos métodos `yield()`, `sleep()` ou `join()`; execução de `wait()` em um objeto; término; e por fim, preempção por thread de mais alta prioridade. Em algumas dessas situações os locks serão liberados, em outras não, dando margem a inesperados *deadlocks* e condições de corrida.

6.4. Comunicação em Rede

Das linguagens largamente utilizadas, Java foi a primeira a ser projetada com interligação via redes de computadores em mente. Com isso, Java incorpora uma série de benefícios para desenvolvimento de aplicações distribuídas, como por exemplo independência de plataforma, conjuntos de caracteres internacionais e mecanismos para carga e execução segura de programas remotos. Portanto, as peças de um dado aplicativo em Java podem interoperar através da Internet apesar de estarem fisicamente espalhadas pelo mundo e executando em arquiteturas de hardware e sistemas operacionais absolutamente diversos[2].

Comunicação em rede é uma forma de E/S, e portanto compreender estes mecanismos é uma premissa básica para o entendimento das facilidades de comunicação de Java. Em sistemas Unix, toda E/S parece ao desenvolvedor como operações sobre arquivos. A API de Sockets adere a essa filosofia: é uma *extensão* da interface de arquivos, permitindo a usuários trabalhar com (abrir e escrever em) conexões de rede de maneira similar a arquivos. Em Java, diferentemente, a filosofia de E/S é construída sobre *streams*: escrever e ler em um arquivo, ou pela rede, implica criar uma stream e invocar métodos que são comuns a boa parte das streams.

A stream básica de saída é `OutputStream`; dois dos seus principais métodos são `write()`, para escrita de um ou mais bytes, e `flush()`, para forçar escrita dos bytes que estão no buffer. A stream básica de entrada é `InputStream`, e seus principais métodos são `read()`, `skip()` e `available()`, que permitem ler um ou mais bytes, descartar bytes na stream, ou verificar se existem bytes prontos para serem lidos, respectivamente. A rotina de leitura retorna o número de bytes lidos com sucesso. Duas das streams mais

[2]desde que exista uma implementação da JVM para aquela arquitetura e sistema operacional.

importantes são `DataInputStream` e `DataOutputStream`, que fornecem métodos para ler e escrever dados primitivos e strings de Java em um formato binário, adequado à transmissão via rede.

Streams de dados podem ser 'concatenadas' em seqüência, havendo dois tipos: filtros de streams e leitores/escritores. Filtros trabalham sobre dados brutos, e servem por exemplo para armazenamento em buffers, compactação, cifragem ou conversão de códigos de caracteres. Leitores e escritores podem ser empilhados sobre filtros ou outros leitores e escritores. Enquanto streams operam sobre bytes, leitores e escritores utilizam caracteres Unicode. Caracteres Unicode comportam codificações de diferentes alfabetos, como por exemplo Chinês e Russo. Aplicações distribuídas na Internet não podem assumir como condição a utilização de códigos ASCII. As duas classes mais importantes são `InputStreamReader` e `OutputStreamWriter`, que atuam entre o programa e uma stream subjacente para realizar a conversão entre caracteres Unicode e bytes.

6.4.1. Sockets

Sockets são usados tipicamente para troca de dados via TCP ou UDP, o que é refletido no conjunto de classes que os representam. A API de sockets permite as seguintes operações ([HAR00]): amarrar (*bind*) um socket a uma porta de comunicação TCP/IP; aceitar conexões de máquinas remotas na porta amarrada; esperar a chegada de dados; conectar a uma máquina remota; enviar e receber dados, e fechar uma conexão. Entretanto, em Java, a interface de Sockets é de mais alto nível, e esconde/agrega algumas dessas operações.

No caso da interface de sockets baseada no protocolo TCP, existe uma divisão clara entre os papéis de *cliente* e *servidor*: cliente é o 'lado' que solicita a abertura de uma conexão, enquanto o servidor espera por uma solicitação de conexão. Esta divisão entre cliente e servidor está restrita ao estabelecimento de conexão; não necessariamente a aplicação distribuída seguirá o modelo cliente/servidor, podendo apresentar uma organização descentralizada *peer-to-peer* ou *serveless.* Duas classes representam objetos do tipo sockets sobre TCP: `Socket` e `ServerSocket`, a serem usados por cliente e servidor, respectivamente. Existem diversos construtores para `Socket`, dependendo dos argumentos tomados (tipicamente, o endereço IP e porta do servidor a ser contactado). Similarmente, existem diferentes construtores para `ServerSocket`, mas na maioria das vezes é especificado apenas a porta através da qual devem ser recebidos os pedidos de conexão TCP. Um servidor pode manter múltiplas conexões com clientes ao mesmo tempo. Para aceitar uma conexão, um servidor utiliza o método `accept()`, que bloqueia até que seja recebida uma solicitação de conexão à porta amarrada ao socket, e a conexão estabelecida com sucesso. O `accept()` retorna uma referência a um novo objeto (da classe `Socket`), que deve ser usado na comunicação com o novo cliente.

A comunicação com Sockets sobre UDP se baseia na troca de *datagramas* entre dois ou mais elementos. Com UDP, não há conexão ponto-a-ponto nem stream confiável de bytes. Cada datagrama é enviado independentemente dos demais, e pode ser perdido, reordenado ou duplicado; a cada transmissão, o endereço destino (endereço IP e porta) deve ser fornecido. UDP é utilizado, por exemplo, para transmissões de conteúdo multimídia em tempo real, para um ou mais destinatários (neste último caso, com auxílio de IP multicast). Em Java, a comunicação com sockets sobre UDP está baseada em duas classes principais: `DatagramPacket` permite criar um datagrama e 'empacotar' dados no mesmo, assim como 'desempacotar' os dados de um datagrama; `DatagramSocket` permite

enviar e receber datagramas através da rede ([HAR00]). Como não há distinção entre cliente e servidor, há apenas uma tipo de socket (classe `DatagramSocket`); o construtor dessa classe cria um socket e faz a amarração do mesmo a uma dada porta, que pode ser especificada no construtor ou escolhida aleatoriamente pelo sistema.

A mesma classe `DatagramPacket` é usada para enviar e receber datagramas, variando apenas o construtor. Por exemplo, para preparar um datagrama a ser enviado, os argumentos são um array de bytes, o tamanho desse array, o endereço IP e o número da porta destinos; para receber, basta os dois primeiros argumentos, sendo que o array estará vazio. Entre outros métodos importantes, `getData()` e `setData()` podem ser usados para obter ou configurar o *payload* do datagrama, respectivamente. Tipicamente, dados são *serializados* antes de serem enviados no array de bytes do datagrama, de maneira a transformar tipos primitivos e objetos complexos em uma seqüência de bytes que possa ser recomposta no destinatário; para tal, é possível utilizar as classes `ByteArrayInputStream` e `ByteArrayOutputStream`.

Para enviar um datagrama a um grupo multicast, utiliza-se um endereço IP da faixa reservada a multicast. Java possui uma classe específica para multicast, `MulticastSocket`, que estende `DatagramSocket`, e métodos que permitem assinar (`join()`) ou deixar um grupo (`leave()`).

6.4.2. RMI - *Remote Method Invocation*

RMI é um mecanismo de comunicação entre processos que permite a invocação de um método sobre um *objeto remoto*. O conceito de RMI é similar ao de RPC (chamada remota de procedimento), embora aplicado à filosofia de programação orientada a objetos.

Em Java, RMI é implementado através de uma API e classe básicas. Com RMI, partes de um programa podem executar em diferentes computadores na rede (sobre diferentes JVMs). Objetos remotos são objetos que podem ter seus métodos invocados a partir de uma JVM diferente da qual o objeto reside. Cada objeto remoto implementa uma ou mais interfaces remotas, que declaram quais métodos podem ser invocados remotamente. A implementação de RMI trata da passagem de argumentos entre métodos, transmitindo os mesmos através da rede. Diferentes mecanismos são usados, dependendo do tipo de argumento: (a) tipos primitivos são passados por valor, tal como em chamadas locais; (b) referências a objetos que implementam a interface `Remote` são passados como *referências remotas*, permitindo que o método remoto invoque um segundo método remoto (que poderia, por exemplo, ser no computador que originou a RMI); (c) referências a objetos que *não* implementam `Remote`, fazendo com que o objeto seja inteiramente serializado e enviado através da rede[3]. O único parâmetro de saída é o argumento de retorno.

Assim como RPC, RMI baseia-se na utilização de *stubs* e interações segundo o modelo cliente/servidor. Um cliente invoca um método local, implementado por um *stub*, que por sua vez toma as providências e redireciona a chamada ao servidor remoto; da mesma forma, recebe uma resposta e retorna a chamada, tal como se ela tivesse ocorrido localmente. No lado do servidor, há um *esqueleto*, que recebe a comunicação remota e a transforma em uma invocação local ao método do objeto no servidor; os argumentos de retorno são preparados e enviados de volta ao cliente. Stubs e esqueletos são gerados automaticamente pelo compilador `rmic`, parte da JDK.

Por fim, o mecanismo de RMI necessita de um esquema de registro de objetos, para

[3] neste caso, é uma condição que o objeto possa ser serializado.

que um cliente possa referenciar um objeto e método que foram previamente 'exportados' por um servidor. Em Java, essa função é feita pelo *registry*, um daemon que contém a lista de todos os objetos e métodos que a JVM local está preparada para exportar e os nomes (strings) através dos quais eles podem ser contactados.

6.5. Java para Processamento de Alto Desempenho

Java surgiu como a linguagem para aplicativos na Internet, possibilitando o nascimento de novas arquiteturas de software orientadas a rede. Uma série de decisões de projeto quanto à linguagem e à máquina virtual foram tomadas favorecendo a computação em rede, em detrimento do desempenho de Java ([VEN99]), como explicado a seguir. Por causa da independência de plataforma, o código fonte é compilado em um *bytecode*, que é mais tarde interpretado por uma implementação qualquer de JVM. A simples e pura interpretação de bytecode é muito menos eficiente do que a execução de uma versão em formato binário, nativo da arquitetura. Igualmente, Java promove o use de caracteres Unicode, que são implementados através de 2 bytes ao invés de 1. Finalmente, a gerência de memória, que estando baseada em *garbage collection*, sacrifica desempenho em prol da robustez ao eliminar erros clássicos de programação ligados à gerência dinâmica de memória.

Por outro lado, a ineficiência comumente associada a Java é uma limitação da implementação da linguagem, e não da linguagem em si ([PAN01]). Recentemente houve uma série de avanços significativos quanto ao desempenho de Java, particularmente em termos de compiladores. Primeiro, compiladores passaram a gerar código bytecode mais eficiente. Segundo, a técnica de compilação *just-in-time* é aplicada para que um método seja compilado de bytecode para o formato nativo na primeira vez que o mesmo é executado, e então armazenando a versão em formato nativo; com isso, todas as invocações subsequentes utilizem o código já em formato nativo da arquitetura. Terceiro, quando ligação dinâmica não é necessária, os compiladores *ahead-of-time* podem gerar código de máquina nativo para uma arquitetura ([VEN99]).

O *JavaGrande Forum* (JavaGrande `http://www.javagrande.org`) representa os interesses de PAD para desenvolvedores de Java. Seus objetivos principais são: (a) avaliar e melhorar a usabilidade do ambiente Java em aplicações do tipo *Grande* (em grosso modo, problemas exigindo computação em larga escala); (b) unir a comunidade Java Grande para promover um consenso sobre requisitos; (c) criar protótipos de implementações, *benchmarks*, especificações de API, e recomendações de melhorias para tornar Java mais útil para aplicações Grande ([GET01]). Questiona-se nesse fórum se as aplicações de PAD devem ser adaptadas às restrições da linguagem (e como ajustar as implementações de Java de forma a explorar completamente o potencial de desempenho da linguagem) ou, ao contrário, a linguagem deve ser estendida (e que extensões são essas, apenas de classes ou modificações na especificação da linguagem).

Java exibe uma série de limitações para uso em computação científica, envolvendo processamento numérico ([MOR01]):

- Falta de arrays multi-dimensionais e com formato regular: em Java, a implementação de arrays multi-dimensionais está baseada em *array de arrays*, onde cada array pode em tese possuir um tamanho diferente.

- Verificação de exceções: cada acesso em Java deve ser verificado em relação a ponteiros nulos ou elementos com índices fora do limite do array. O modelo adotado em Java oferece robustez em detrimento do desempenho (pois verificações aumentam *overhead*). Além disso, a semântica adotada no tratamento de exceções impossibilita a otimização através do reordenamento de código em laços.

- Suporte pobre para números complexos e outros sistemas aritméticos: Java opera apenas com números reais. Dados não primitivos são representados como objetos, como da classe `Complex`, ao invés de estruturas de dados 'de baixo custo'.

NINJA ([MOR01]) demonstra que é possível obter em Java desempenho comparável ao de Fortran, oferencendo potenciais soluções para as limitações acima através da criação de novas classes e no uso de *expansão semântica*, técnica em que o compilador procura por invocações de métodos e as troca por código eficiente. Quanto ao problema da checagem nos acessos, novas classes de arrays e números complexos são criadas para eliminar a necessidade de checar cada acesso. A idéia é criar regiões livres do risco de exceção, sobre as quais o compilador pode então executar as otmizações tradicionais, reorganizando o código. Para criar essa regiões livres, o compilador cria duas versões para cada laço otimizado, uma segura e outra insegura, e executa uma delas de acordo com um conjunto de testes em tempo de execução. Foram comparadas implementações de aplicações numéricas em Java original, Java otimizado e Fortran 90. Embora Fortran 90 tenha obtido o melhor desempenho em virtualmente todos os experimentos, Java otimizado obteve desempenho bastante próximo ao de Fortran em muitos casos (vide resultados em [MOR01]).

Embora compilação *just-in-time* acelere a execução seqüencial, em computação de alto desempenho isso não é o bastante, uma vez que a comunicação entre processos (ou threads) deve ser bastante eficiente, pois um programa executa em múltiplos processadores. Invocação de método é a principal forma de comunicação em Java: em uma máquina com um ou mais processadores e memória compartilhada, threads interagem através de invocações sincronizadas de métodos; em sistemas de memória distribuída, Java oferece RMI (vide Seção 6.4.). A combinação de compiladores poderosos e sistemas de execução eficientes permite explorar a capacidade computacional de computadores com memória compartilhada distribuída, escalando para tamanhos de sistema inatingíveis em abordagens com memória compartilhada pura ([KIE01]).

Hyperion ([ANT01]) está baseado na construção de uma máquina virtual distribuída que permite o compartilhamento de objetos entre threads residentes em diferentes computadores. A máquina virtual distribuída esconde do programador a existência dos múltiplos processadores, e transparentemente distribui threads entre os nós de um agregado. A comunicação entre threads se dá através do compartilhamento de um espaço comum de endereçamento, disponibilizado através de um substrato DSM (esquema de memória compartilhada distribuída) que mantém a semântica do modelo de memória de Java (vide Seção 6.3.). Para aumentar a eficiência da execução seqüencial, Hyperion primeiro compila executáveis no formato class (bytecodes) para C, e então de C para o formato nativo da arquitetura alvo (com um compilador otimizado para a arquitetura de hardware em questão). O desempenho é aumentado sem comprometer a independência de plataforma.

Manta ([MAS99]) se baseia na utilização de uma reimplementação mais eficiente de RMI no acesso a objetos remotos. O modelo de programação Java não é alterado, a não

ser por uma extensão que permite que o programador aumente a localidade especificando que objetos devem ser replicados. Manta emprega um compilador Java que previamente (*ahead of time*) compila código Java em formato binário nativo da arquitetura. A compilação estática do programa permite otimizações substanciais. A implementação de RMI no Mantra está baseada nos seguintes componentes: (a) um protocolo de RMI mais leve, implementado em C e sem camadas; (b) serialização otimizada de objetos, sem as verificações de exceções em tempo de execução típico de Java; (c) comunicação eficiente, com RPC e troca de mensagens (incluindo *broadcast*) baseada na biblioteca Panda.

Finalmente, Java é uma alternativa para computação de Grid (*Grid Computing*). Um grid é uma infraestrutura computacional interligando computadores em uma 'organização virtual' formada dinamicamente por indivíduos e instituições com interesses comuns, visando o compartilhamento de recursos. Segundo [GET01], existem diversas razões para usar Java em aplicações Grid, incluindo sua portabilidade, suporte à comunicação, mecanismos de concorrência e recursos de engenharia de software (programação baseada em componentes, ferramenta de documentação integrada, etc.) Existem diversos outros projetos relacionados a Java e PAD, tais como o *JavaParty* ([PHI00]). Por restrições de espaço, não é possível apresentá-los neste documento.

6.6. Comentários Finais

A bibliografia sobre programação paralela e distribuída em Java é extensa e qualificada. Apenas para citar alguns exemplos, [LEA00] discute programação concorrente em Java, com sólida base conceitual e ênfase em *frameworks* e *design patterns*. Com similar importância para princípios e padrões, [AND00] aborda programação concorrente, tendo Java como um dos estudos de caso. [OAK99] é uma boa referência sobre o desenvolvimento de programas concorrentes em Java, com ótica pragmática centrada em threads. [HAR00] aborda a programação em rede usando Java, com explicações claras e exemplos elucidativos. [VEN99], embora um pouco desatualizado, apresenta a JVM por dentro, incluindo fatores que afetam o desempenho de programas Java. Em todos esses livros, conhecimentos sobre Java são recomendáveis, pelo menos básicos. Quanto ao Java em PAD, existe uma boa quantidade de artigos publicados; recomenda-se como recurso inicial para consulta o portal JavaGrande `http://www.javagrande.org`, pois este aponta para páginas relacionadas. Computação de alto desempenho em Java está bastante em voga; por exemplo, parte da edição de outubro de 2001 da *Communications of the ACM* é dedicada ao assunto (os artigos da mesma são comentados na seção anterior).

Java tem se apresentado como a escolha natural para desenvolvimento de sistemas distribuídos na Internet, em função das facilidades para criação e sincronização de threads, suporte à comunicação em rede, independência de plataforma, segurança e robustez. Ela oferece facilidades nativas que não são encontradas em outras abordagens. No entanto, essas facilidades impactam negativamente no desempenho do Java. As primeiras implementações de Java, baseadas somente na interpretação de bytecode, apresentavam desempenho bastante fraco quando comparadas a abordagens tradicionais como C e Fortran.

No entanto, a desvantagem acima tem sido encurtada cada vez mais através da evolução dos compiladores e JVMs, que compilam bytecode para formato binário em tempo de execução e sob demanda, e otimizam código a medida que o mesmo é execu-

tado. A diferença em desempenho encurtou a ponto da comunidade científica considerar, pelo menos de maneira incipiente, a utilização de Java em PAD. Dependendo da aplicação e da implementação de Java utilizada, é possível obter desempenho comparável ao de outras linguagens tradicionalmente empregadas em PAD. Este foi o caso em uma recente avaliação de desempenho realizada pelos alunos na disciplina de Computação Gráfica (Informática da Unisinos), que desenvolveram e avaliaram o desempenho de uma aplicação *Ray-Tracing* distribuído em diversas linguagens/plataformas. Existem diversos projetos de pesquisa que buscam alternativas para aumentar a eficiência de Java em aplicações de alto desempenho, tendo os mesmos obtidos resultados promissores.

6.7. Bibliografia

[AND00] ANDREWS, G. **Foundations of Multithreaded, Parallel, and Distributed Programming**, Boston: Addison-Wesley, 2000. 664p.

[ANT01] ANTONIU, G. et alli. The Hyperion System: compiling multi-threaded Java bytecode for distributed execution. **Parallel Computing**, v.27, n.10, Sept. 2001, pp1279-1297.

[GET01] GETOV, V. et alli, Multiparadigm Communications in Java for Grid Computing, , **Communications of the ACM**, v.44, n.10, pp.118-125. Oct. 2001.

[GOS96] GOSLING, J.; JOY, B.; STEELE, G.; **The Java Language Specification**, Boston: Addison-Wesley, 1996.

[HAR00] HAROLD, E. **Java Network Programming**, 2nd. Ed., Sebastopol: O'Reilly, 2000. 731p.

[KIE01] KIELMAN, T et alli, Enabling Java for High-Performance Computing, **Communications of the ACM**, v.44, n.10, pp.102-109, Oct. 2001.

[LEA00] LEA, D. **Concurrent Programming in Java**: design principles and patterns, 2nd. Ed., Boston: Addison-Wesley, 2000. 411p.

[MAS99] MASSEN, J. et alli. An Efficient Implementation of Java's Remote Method Invocation. In: **Proc. of 7th ACM SIGPLAN**, Atlanta May 1999. New York: ACM Press, 1999, pp.173-182.

[MOR01] MOREIRA, J.E. et alli, The Ninja Project, **Communications of the ACM**, v.44, n.10, pp.102-109, Oct. 2001.

[OAK99] OAKS, S.; WONG, H. **Java Threads**, 2nd. Ed., Sebastopol: O'Reilly, 1999. 319p.

[PAN01] PANCAKE, C.M.; LENGAUER, C. High-Performance Java, **Communications of the ACM**, v.44, n.10, pp.98-101. Oct. 2001.

[PHI00] PHILIPPSEN, M.; HAUMACHER, B; NESTER, C. More Efficient Serialization and RMI for Java. **Concurrency: Practice & Experience**, v.12, n.7, pp495-518. May 2000.

[VEN99] VENNERS, B. **Inside the JAVA 2 Virtual Machine**, 2nd. Ed., New York: McGraw-Hill, 1999. 703p.