UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE FÍSICA DE SÃO CARLOS
DEPARTAMENTO DE FÍSICA E INFORMÁTICA

**PROPOSTA DE UMA LINGUAGEM JAVA PARA UM AMBIENTE PARALELO -
JAPAR**

ANTONIO FERNANDO TRAINA

Tese apresentada ao Instituto de Física de São Carlos, Universidade de São Paulo, para
obtenção do título de Doutor em Ciências “Física Aplicada-opção Física Computacional”

Orientador: Prof. Dr. Carlos Antonio Ruggiero

OK

USP/IFSC/SBI

8-2-001313

São Carlos – São Paulo
2000

IFSC-USP SERVIÇO DE BIBLIOTECA
INFORMAÇÃO

Traina, Antonio Fernando

“Proposta de uma linguagem Java para um ambiente paralelo-JAPAR”/
Antonio Fernando Traina – São Carlos, 2000

Tese (Doutorado) – Instituto de Física de São Carlos da Universidade de
São Paulo, 2000 – páginas: 148

Área: Física Aplicada-opção: Física Computacional

Prof. Dr. Carlos Antonio Ruggiero

1. Compiladores. 2. Arquitetura Paralela. 3. Linguagem Java.

I. Título

IFSC**UNIVERSIDADE
DE SÃO PAULO****Instituto de Física de São Carlos**

Av. Dr. Carlos Botelho, 1465
CEP 13560-250 - São Carlos - SP
Brasil

Fone (016) 273-9333
Fax (016) 272-2218

MEMBROS DA COMISSÃO JULGADORA DA TESE DE DOUTORADO DE ANTONIO
FERNANDO TRAINA APRESENTADA AO INSTITUTO DE FÍSICA DE SÃO CARLOS, DA
UNIVERSIDADE DE SÃO PAULO, EM 10 DE MARÇO DE 2000.

COMISSÃO JULGADORA:

Prof. Dr. Carlos Antonio Ruggiero/IFSC-USP

Prof. Dr. Álvaro Garcia Neto/IFSC/USP

Profa. Dra. Patrícia Magna/IFSC/USP

Profa. Dra. Liria Matsumoto Sato/EP/USP

Prof. Dr. Marcos Antônio Cavenaghi/UNESP

**Aos meus pais Caetano e Aurora
e à minha esposa Maria do Carmo,
que tanto colaboraram e, pela
dedicação, compreensão e paciência,
eu dedico este trabalho**

Agradecimentos

Gostaria de agradecer a todos que, de uma forma ou de outra, contribuíram para a realização deste trabalho. Em especial:

- Ao professor Doutor Carlos Antonio Ruggiero, pela paciência e dedicação, e pela constante sabedoria, e observações tão valiosas,
- À Maria do Carmo, por saber dar apoio na hora certa e acima de tudo dedicação nos momentos mais difíceis de minha vida,
- Ao meu irmão Caetano e a Agma, lá de Pittsburg, por seu apoio e compreensão,
- Meus pais, que jamais deixaram de acreditar em mim,
- Todos amigos do departamento de Física, a Cláudia, o prof. Jan, o prof. Gonzalo, prof Álvaro, Patrícia, Lírio, Marcelo Freitas,
- À Wladerez Aparecida Gounella Caiado, a Wlá, por sua paciência e compreensão,
- Ao amigo Luiz Antonio Guimarães Ribas, que me mostrou uma oportunidade de ver o mundo de uma nova forma,
- Ao amigo Ernesto Luis Pereira Lopes, o Xyko, que me estimulou a ir sempre em frente,
- Ao amigo Renato Vairo Belhot, por suas sugestões e conselhos precisos,
- Ao meu tio Carlos, *in memoriam*, onde quer que esteja,
- Aos tantos amigos de Ribeirão Preto,
- E muitos outros...

Sumário

LISTA DE FIGURA

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

RESUMO

ABSTRACT

1. INTRODUÇÃO..... 01

2. COMPUTAÇÃO PARALELA..... 05

 2.1 Introdução..... 05

 2.2 Arquiteturas Paralelas..... 07

 2.2.1 Taxinomia das arquiteturas..... 09

 2.2.2 Considerações sobre projeto e performance..... 13

 2.2.2.1 *Speed-up* e eficiência..... 14

 2.2.2.2 Granularidade..... 15

 2.2.2.3 Balanceamento de carga e distribuição de processos..... 17

 2.2.2.4 Problemas que podem ocorrer na paralelização..... 18

 2.2.2.5 Modelos de acesso à memória..... 20

 2.3 Programação Paralela..... 21

 2.3.1 Paralelização de programas..... 23

 2.3.2 Formas de paralelizar um programa..... 27

 2.3.3 Linguagens de programação..... 28

 2.4 Considerações Finais..... 30

3. A LINGUAGEM JAVA..... 32

 3.1 Introdução..... 32

 3.1.1 Conceitos Básicos..... 32

3.1.2 Histórico.....	35
3.2 O Paradigma Java.....	36
3.2.1 O programa Java.....	37
3.2.2 O arquivo de classes Java.....	37
3.2.3 O API Java.....	38
3.2.4 A máquina virtual Java	39
3.2.5 A rede.....	41
3.3 A Plataforma Java.....	41
3.3.1 A plataforma Java básica	43
3.3.2 Visão interna da plataforma Java.....	43
3.3.3 Máquina virtual Java.....	45
3.3.4 Plataformas para Java.....	46
3.3.4.1 Aplicação em hospedeiros (<i>hosted</i>).....	47
3.3.4.2 Sistema operacional hospedeiro.....	47
3.3.4.3 JavaOS™	48
3.3.4.4 JavaPC.....	49
3.3.4.5 <i>PersonalJava</i> e <i>EmbeddedJava</i>	49
3.3.4.6 Cartão Java (Java Card).....	49
3.3.4.7 Processadores Java.....	50
3.4 O Compilador Java e Ambientes de Execução.....	51
3.5 O Compilador JIT.....	52
3.6 Considerações finais.....	54
4. PROPOSTAS JAVA EM ARQUITETURAS PARALELAS.....	56
4.1 Introdução.....	56
4.2 Possibilidades de Paralelismo em Java... ..	57

4.2.1 Paralelização automática de <i>loops</i> do <i>bytecode</i> (Solução para o paralelismo intrínseco).....	64
4.2.3 Paralelismo em bibliotecas especiais	71
4.2.3.1 JVMP.....	72
4.2.3.2 Outros exemplos de bibliotecas especiais.....	79
5. PROPOSTA E IMPLEMENTAÇÃO DO AMBIENTE PARALELO.....	80
5.1 Introdução.....	80
5.2 Proposta.....	81
5.2.1 Características do ambiente paralelo proposto.....	81
5.2.2 O conversor para reestruturar Java.....	85
5.2.2.1 Um filtro especial: <i>ConJava</i>	86
5.2.2.2 A BNF e o YACC.....	87
5.2.3 O compilador paralelo.....	90
5.2.3.1 O compilador Japar.....	91
5.2.3.2 A BNF utilizada.....	91
5.2.4 A máquina virtual paralela simplificada para Japar: MVJP.....	96
5.2.4.1 Conceitos da máquina virtual paralela.....	97
5.2.4.2 Tratamento do código paralelo.....	98
5.3 Implementações Realizadas.....	98
5.3.1 ConJava.....	98
5.3.2 Japar.....	101
5.3.3 Rotinas de biblioteca para os recursos de paralelismo: japar.par	105
5.3.4 O carregador de classes VM Java: Japar.par.ClassLoaderPar()	106
5.3.5 A máquina virtual Java.....	108
5.4 Resultados Obtidos.....	111
5.4.1 ConJava.....	111

5.4.2 Japar e a máquina virtual paralela (MVJP).....	113
5.5 Observações Finais.....	127
6. CONCLUSÕES E PERSPECTIVAS FUTURAS.....	129
6.1 Conclusões.....	129
6.2 Perspectivas Futuras.....	132
BIBLIOGRAFIA.....	134
Apêndice A – Estrutura gramatical do conversor ConJava.....	A.1
Apêndice A.1 – Estrutura BNF da linguagem Java.....	A.1
Apêndice A.2 - Arquivo de entrada para o YACC (ConJava).....	A.9
Apêndice B – Estrutura JAPAR.....	B.1
Apêndice B.1 – A BNF complementar da linguagem Japar.....	B.1
Apêndice B.2 - <i>Opcodes</i> da linguagem Java.....	B.6
Apêndice B.3 – Mnemônicos dos <i>Opcodes</i> de Java.....	B.11
Apêndice B.4 – Programas usados em testes.....	B.18
Apêndice B.5 – Programas de Otimização.....	B.34
Apêndice C – Exemplos de linguagens e aplicações paralelas.....	C.1
Apêndice D- Conceitos e Ferramentas para o Ambiente Java.....	D.1
Apêndice E- Características da linguagem Java.....	E.1

LISTA DE FIGURAS

FIGURA 2.1 - Taxinomia de arquiteturas segundo Flynn.....	9
FIGURA 2.2 - Granulosidade fina.....	16
FIGURA 2.3 - Granulosidade grossa.....	16
FIGURA 3.1 - Fluxo de execução de um programa Java.....	33
FIGURA 3.2 - A independência de plataforma de um programa Java.....	34
FIGURA 3.3 - Esquema da plataforma básica Java em diversos sistemas operacionais-.....	44
FIGURA 3.4 - Compilação de código fonte gerando e executando <i>bytecodes</i>	52
FIGURA 4.1 - Funcionamento básico de um compilador.....	57
FIGURA 4.2 - O ambiente Java simplificado – o javac	58
FIGURA 4.3 - O ambiente Java simplificado – a máquina virtual Java.....	58
FIGURA 4.4 - Possibilidades de explorar paralelismo em Java.....	61
FIGURA 4.5 - Detalhamento da fase de otimização de um compilador.....	62
FIGURA 4.6 - Mecanismo de paralelização automática de <i>bytecodes</i> pela ferramenta javab	64
FIGURA 4.7 - Fragmento de Java com seu <i>bytecode</i> correspondente produzido por javab	67
FIGURA 4.7 – Outro fragmento de Java com seu <i>bytecode</i> correspondente produzido por javab	69
FIGURA 4.9 - Implementação da comunicação de JPVM.....	76
FIGURA 5.1 - Resultados da execução do programa <i>for</i> simples.....	126
FIGURA 5.2 – Resultados da execução do programa <i>quicksort</i>	127

LISTA DE TABELAS

TABELA 2.1 - Softwares disponíveis na Internet para paralelização..... 25

TABELA 3.1 - Bibliotecas que compõe a API de Java..... 39

TABELA 5.1 – CPU’s implementadas..... 109

TABELA 5.2 - Relação dos programas de teste..... 114

TABELA 5.3 - Relação dos computadores utilizados..... 115

TABELA 5.4 - Comparação entre os programas cpi.c e cpimpi.c..... 116

TABELA 5.5a - Comparação com o programa cpimpi.c usando vários computadores..... 117

TABELA 5.5b - Comparação com o programa cpimpi.c usando vários computadores..... 118

TABELA 5.6 - Comparação com o programa em linguagem Java cpijava.java..... 119

TABELA 5.7a - Testes de execução do programa cpijava.java no computador Atm5. 121

TABELA 5.7b - Testes de execução do programa cpijava.java no computador Atm5. 122

TABELA 5.8 - Teste com o programa cpijava.java (paralelo) em ambiente sem e 122
com MPI.....

TABELA 5.9 - Testes de execução do programa paralelo cpijava.java no computador 123
Atm5.....

LISTA DE SIGLAS E ABREVIATURAS

ANSI	American National Standards Institute
API	Application Program Interface
CGI	Common Grafic Interface
CLI	Call Level Interface
FTP	File Transfer Protocol
GPF	General Protection Fault
GUI	Grafic User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
ISDN	Integrated Services Digital Network
JDBC	Java Database Connectivity
JPVM	Java Parallel Virtual Machine
JDK	Java Developer’s Kit
JVM	Java Virtual Machine
JIT	Just in Time
MPI	Message Passing Interface
MSIE	Microsoft Internet Explorer
OO	Orientected Object
PDA	Digital Assistence e Portable Data Assistentes
PVM	Parallel Virtual Machine
SQL	Structure Query Language
TCP	Transmission Control Protocol
URL	Uniform Resource Locator
WORA	Write Once Run Anywhere
WWW	World Wide Web

Marcas Registradas:

Sun, o logotipo da Sun, Sun Microsystems, Solaris, HotJava e Java são marcas registradas de Sun Microsystems, Inc. nos Estados Unidos e em alguns outros países.

UNIX é marca registrada nos Estados Unidos e outros países, exclusivamente licenciada por X/Open Company, Ltd.

Netscape Navigator é marca registrada de: “Netscape Communications Corporation”.

Intel A Marca Intel é marca registrada pela Intel.

Resumo

TRAINA, A. F. (2000) Proposta de uma linguagem Java para um ambiente paralelo – Japar. São Carlos, 2000, 148p. Tese (Doutorado) – Instituto de Física de São Carlos, Universidade de São Carlos

Com o crescente número de usuários de computadores, novas ferramentas têm sido apresentadas com a finalidade de aumentar a eficiência dos computadores, meio para que seus usuários disponham de recursos automatizados. Mais recentemente, máquinas ligadas em rede de computadores e o fenômeno Internet tornaram necessárias ferramentas específicas para este tipo particular de uso. Entre as principais respostas à essas necessidades surgiu a linguagem Java, que tem ganho adeptos tanto na comunidade científica como no mercado. Surge daí a necessidade de buscar formas alternativas para o uso de computadores em rede. Entre as soluções propostas encontra-se a de arquiteturas e linguagens paralelas. Estas ferramentas, ainda em fase experimental, apresentam soluções que podem a médio e longo prazo serem viáveis, desde que estudos e pesquisas mostrem sua viabilidade. Neste trabalho investiga-se a aplicação do conceito de paralelismo em linguagens para rede, mais especificamente a linguagem Java. A proposta é estudar as possíveis formas para se explorar o paradigma de linguagens paralelas em ambientes Java. Para isto, apresenta-se uma pesquisa relativa às principais linguagens paralelas disponíveis na literatura, de forma a conhecer as melhores soluções apresentadas por essas linguagens. O trabalho apresenta também um estudo realizado em um conjunto de ferramentas Java disponíveis no mercado. Finalmente, propõe-se um novo ambiente que possa disponibilizar ao usuário os melhores recursos da linguagem Java, explorando as melhores soluções encontradas na literatura.

Palavras Chave: linguagem Java; linguagens paralelas; paralelismo; compiladores

Abstract

TRAINA, A. F. (2000) Proposal of a language Java for a environment parallel – Japar. São Carlos, 2000, 148p. Tese (Doutorado) – Instituto de Física de São Carlos, Universidade de São Carlos

With the increasing number of computer users, new tools have been presented to improve the computers efficiency and to make automated resources available for those users. Nowadays, the use of computers in a network and the Internet phenomenon requires specific tools. Among them, the Java language appears as an important tool which has been attracting users in both the scientific and commercial communities. At the same time the computer networks are becoming more popular and some problems have emergent concerned to the networks assessment and connections. It is necessary to look for alternative ways of handling the network computer systems. The parallel architectures and languages appear among the proposed solutions. These tools are still in an experimental phase, studies and researches being necessary additional to confirm their feasibility. In this work we investigate the application of parallelism concepts in languages for networks, and particular we deal with the Java language. The aim is to study the possible approaches for exploring the parallel languages paradigm in Java environments. Research about the main parallel languages available in the literature is presented, in order to check the best solutions proposed by those languages. The work also presents an investigation about the Java tools available in the commercial market. Finally a new environment, is proposed that makes some of the best resources of language Java available to the users by exploring the best solutions found in the literature.

Keywords: language Java; parallel languages; parallelism; compilers

Capítulo 1

Introdução

<p>Resumo Neste capítulo são detalhadas as estruturas deste trabalho, construindo-se um mapa geral de cada capítulo.</p>

Nos últimos anos tem-se notado a diminuição do custo de *hardware*, e o oferecimento de máquinas cada vez mais complexas e poderosas. A necessidade de processamento de informações de forma cada vez mais rápida, em ambientes crescentemente mais complexos, tem exigido a construção de arquiteturas e organização de computadores com desempenho melhor e mais completo. O aumento da velocidade de processamento utilizando as tecnologias tradicionais do tipo formalizado por John von Neumann, ou seja seqüencial com monoprocessador e com armazenamento interno de instruções, tem caminhado para um limite físico de velocidade. A alternativa é buscar as soluções de arquiteturas multiprocessadas, paralelas e em rede.

Por outro lado, novas ferramentas têm sido apresentadas com a finalidade de aumentar a eficiência dos computadores. Como nem sempre os usuários finais são especialistas em computação, essas ferramentas devem oferecer meios para que seus usuários disponham de recursos automatizados. E com o advento da Internet, tornaram-se necessárias ferramentas específicas para este tipo particular de uso. Entre as principais respostas a essas necessidades surgiu a linguagem Java.

Mas a conseqüente popularização dos computadores tem criado gargalos no acesso à rede que, caso nada seja feito, caminha para uma situação de caos, onde o limite das conexões atinge os meios físicos de transmissão. Surge daí a necessidade de buscar formas alternativas para o uso de computadores em rede. Entre as soluções propostas encontra-se a de arquiteturas e linguagens paralelas.

Neste trabalho investiga-se a aplicação do conceito de paralelismo na linguagem Java. A proposta é estudar as possíveis formas para se explorar o paradigma de linguagens paralelas em ambientes Java. Foi necessário então um estudo que envolvesse tanto as linguagens e arquiteturas paralelas e o ambiente Java, quanto todas as propostas que já haviam sido feitas unindo essas duas disciplinas.

No **Capítulo 2**, intitulado “**Computação Paralela**”, apresenta-se uma pesquisa relativa às principais linguagens e arquiteturas paralelas disponíveis na literatura, de forma a conhecer as melhores soluções apresentadas por essas propostas. São revistos conceitos de processamento paralelo e distribuído, as possibilidades de taxinomia das arquiteturas, e são feitas diversas considerações sobre projetos de arquiteturas e desempenho das mesmas. Finalmente são descritos os modelos de acesso à memória. Entre as considerações feitas destacam-se itens como o *speed-up* e eficiência da arquitetura, a granularidade que é possível de ser explorada, etc. São descritos também os aspectos de balanceamento de carga e distribuição de processos. Neste capítulo é apresentada ainda uma discussão a respeito dos problemas que podem ocorrer na paralelização. Um segundo bloco neste capítulo apresenta o tema “Programação Paralela” e discute a paralelização de programa e as melhores formas de se paralelizar um programa. Um resumo aborda as principais linguagens de programação, sendo ainda apresentados exemplos de linguagens e aplicações paralelas. Ainda como complemento a esse capítulo encontram-se no Apêndice C um resumo de diversas linguagens paralelas encontradas na literatura.

No **Capítulo 3**, intitulado “**A linguagem Java**”, são considerados os aspectos mais importantes da ferramenta de programação Java. Neste capítulo serão revistos o paradigma e as especificações da linguagem Java. Dessa forma são conceituados como se define um programa Java, o que é o arquivo de classes Java, e o que é um API Java. É então discutida a Máquina Virtual Java, e seu funcionamento em rede. Para a compreensão de sua evolução um breve histórico é apresentado. São descritos a plataforma Java básica, uma

visão interna desta, e quais são as propostas de plataformas para Java e suas aplicações em hospedeiros (entre os quais o JavaOSTM, o JavaPC, *PersonalJava* e *EmbeddedJava*, Cartão Java (Java Card) Processadores Java). São apresentadas definições a respeito do compilador Java e de seus ambientes de execução, e ainda sobre o compilador JIT (*Just-in-time*). Dentro da estrutura de Java são destacados os conceitos de *Applets* e suas aplicações, o recurso de Java Script, e o conceito fundamental dos *bytecodes*. Fechando essa sessão conceituam-se os **Threads**. Após isso, é apresentada uma sinopse das ferramentas para Java encontradas na literatura. Como complemento desse capítulo, muitas ferramentas encontradas e estudadas podem ser vistas nos Apêndices D e E. Nesses apêndices são tratados os componentes Java, os geradores de compiladores para Java, dos ambientes de desenvolvimento, dos montadores para Java, de descompiladores Java, dos tradutores de Java para C. Na sequência são analisados alguns aspectos da Linguagem Java, como as diferenças entre Java e C/C++, a construção léxica de Java e também sua construção sintática.

No **Capítulo 4**, intitulado “**Propostas JAVA em Arquiteturas paralelas**” é feita uma ponte entre os dois conceitos apresentados nos dois capítulos anteriores e são apresentadas as propostas de exploração dos aspectos paralelos para a ferramenta Java já existentes. Uma abordagem geral levando em consideração os aspectos que podem ser paralelizados, tanto no ambiente, como na linguagem é apresentada. Assim, é vista a paralelização automática de **loops** do *Bytecode* (solução para o paralelismo intrínseco), e a solução de paralelismo em bibliotecas especiais. Com o intuito de apresentar o caminho a ser seguido no capítulo subsequente, estas propostas são analisadas criando uma base teórica para dar suporte a isso.

No **Capítulo 5**, intitulado “**Proposta e Implementação do Ambiente Paralelo**”, é apresentada uma proposta de criação de um ambiente paralelo para Java chamada Japar e a máquina paralela MVJP, apresentando as características do ambiente paralelo proposto. A seguir serão apresentadas as implementações que foram realizadas. Entre essas implementações encontram-se o conversor para reestruturar Java, o filtro especial *ConJava*, o compilador paralelo *Japar*, e as rotinas de biblioteca para os recursos de paralelismo (japar.par) Em seguida é apresentado o carregador de classes VM Java

(Japar.par.ClassLoaderPar) e a máquina Virtual Simplificada para Japar (MVJP). São discutidos os aspectos de geração do compilador com base na ferramenta YACC, que permite uma geração de código para compiladores de forma rápida e segura. É feito então, com o uso do ambiente **Japar** e da máquina MVJP construídos, uma série de verificações testes e avaliações desta proposta.

Finalmente no **Capítulo 6**, intitulado “**Conclusões e Perspectivas Futuras**”, são discutidos os resultados e conclusões a que este trabalho chegou. Como sua implementação pode ainda seguir muitos caminhos, e ser aprofundada em diversos aspectos, este capítulo finaliza oferecendo um conjunto de idéias e sugestões para o prosseguimento desse projeto.

Capítulo 2

Computação Paralela

Resumo Este capítulo descreve os aspectos do processamento paralelo e distribuído computacional, envolvendo desde as arquiteturas até as linguagens paralelas. Neste capítulo é feita uma revisão dos assuntos mais relevantes para a proposta deste trabalho, onde são enfocados os aspectos que constituem a base conceitual do presente trabalho.

2.1 Introdução

A paralelização de aplicações tem como grande atrativo a redução do tempo computacional necessário para a sua execução. O desenvolvimento de aplicações paralelas eficientes, entretanto, é freqüentemente uma tarefa árdua, principalmente em razão da complexidade das aplicações, arquiteturas e interações entre elas. A experiência tem mostrado que a programação paralela é significativamente mais difícil que a programação seqüencial, principalmente porque envolve a necessidade de sincronização entre tarefas, e a análise de dependência de dados.

A implementação de uma aplicação paralela se inicia com uma primeira paralelização, normalmente derivada de uma versão seqüencial da aplicação. O programador então analisa o desempenho dessa primeira implementação, de forma a determinar se a aplicação demanda modificações para melhorar o seu desempenho. Se este for o caso, a aplicação é aperfeiçoada pelo programador através de uma série de sessões onde se realizam medições e modificações.

Inicialmente a aplicação paralela é executada e o seu desempenho é medido. Neste ponto o programador avalia as medições e decide se são necessárias

melhorias. Nesta fase, ferramentas de desempenho auxiliam o programador a analisar a aplicação e a sua execução, no intuito de entender o comportamento da aplicação em termos de desempenho do programa paralelizado. Finalmente, o programador efetua mudanças na aplicação e uma nova sessão de medições e modificações se inicia. Uma vez que o desempenho da aplicação paralela corresponda às expectativas do programador, este frequentemente precisa estimar o desempenho da aplicação para parâmetros de execução ou configurações de sistema diferentes. Se o desempenho estimado, ou o desempenho de uma execução nas novas condições, não for satisfatório, o programador inicia uma nova série de sessões de medições e modificações com o objetivo de melhorar o desempenho da aplicação para as novas condições de execução.

Com base nessa descrição da construção de uma aplicação paralela, há três atividades distintas que são de responsabilidade do programador:

(1) **análise do desempenho da aplicação paralela** de forma a determinar quão bem os recursos disponíveis estão sendo utilizados;

(2) **aperfeiçoamento do desempenho da aplicação paralela**, executando o processo de inferência que relaciona desempenho observado com modificações de código; e

(3) **estimativa do desempenho da aplicação para diferentes condições de execução**, pois todas essas atividades requerem que o programador entenda o comportamento da aplicação e as causas do seu desempenho.

Desta forma, entendimento de desempenho é a tarefa de identificar decisões de implementação como por exemplo, estratégia de paralelização, escalonamento de iterações, distribuição de dados, ou características da arquitetura que expliquem o desempenho de uma aplicação paralela. Abordagens tradicionais de análise de desempenho não são normalmente suficientes para o entendimento do comportamento dinâmico de aplicações, o que obriga o programador a derivar as aplicações baseado em sua experiência e conhecimento a respeito da aplicação e da arquitetura.

2.2 Arquiteturas Paralelas

Atualmente há uma tendência crescente no uso de sistemas de computação paralela e distribuída em uma vasta gama de aplicações. Esses sistemas são compostos por vários processadores que operam concorrentemente, cooperando na execução de uma determinada tarefa. Nas chamadas arquiteturas paralelas o objetivo principal é o aumento da capacidade de processamento, utilizando o potencial oferecido por um grande número de processadores. A comunicação dos processadores é realizada através de redes especiais de conexão ou por meio de uma memória compartilhada, implicando em estruturas fisicamente reunidas. Por outro lado, nas arquiteturas distribuídas o atrativo principal é a flexibilidade, obtida pela integração de computadores de diversos tipos em um mesmo sistema, sem restrições quanto à distribuição física dos componentes. O desafio é dominar essa flexibilidade e usá-la na construção de sistemas que atendam às necessidades atuais.

Com o recente avanço tecnológico nas diversas áreas do conhecimento, buscam-se cada vez mais novos métodos de pesquisa, no sentido de reduzir custos e otimizar a obtenção de resultados. Assim atentou-se para o fato de que a computação, com suas máquinas cada vez mais poderosas e plataformas de alto desempenho, torna cada vez mais viável a substituição, com vantagens, dos experimentos de bancada por simulações em modelos computacionais, seja de experimentos que envolvem o próprio homem ou até mesmo de simulações de fenômenos naturais. O primeiro problema com o qual deparou-se foi o fato de que tais simulações requeriam uma enorme quantidade de cálculo e elevado número de iterações o que, mesmo para a tecnologia atual, toma tempo proibitivo caso se queiram resultados precisos.

Assim, surgiu inicialmente na forma de um computador capaz de resolver equações diferenciais em paralelo, projetado por Vanevaur Bush em 1920, a idéia do paralelismo. No entanto, o marco fundamental foi o surgimento de máquinas para processamento paralelo, como o ILLIAC IV, construído nos fins de 60 na universidade de Illinois composto de 64 processadores. Atualmente, pode-se contar com máquinas muito poderosas compostas de, até mesmo, milhares de processadores

em um único computador. Tais computadores porém tem custo muito elevado, sendo por vezes inviável a sua aquisição por instituições sem muitos recursos financeiros. Dessa forma, com a popularização do ambiente em rede local surgiram alternativas como a implantação de sistemas paralelos compostos de diversas máquinas, cada uma com UCP própria, realizando a comunicação entre si através de barramento *Ethernet*.

A seguir serão relacionados alguns conceitos importantes a respeito de paralelismo, onde são definidos alguns mecanismos básicos[Ben90], [Henn95] e [Kum94].

Processo (ou tarefa): Refere-se ao conjunto de instruções que são executadas seqüencialmente.

Processador: É o elemento de hardware capaz de executar seqüencialmente as instruções de um processo.

Programa concorrente: É o conjunto de processos que comunicam e cooperam entre si para resolver um determinado problema.

Máquina paralela: É o conjunto de elementos de processamento interligados de forma a poderem comunicar-se e cooperar entre si.

Multiprocessador: É o conjunto de processadores interligados de forma a poderem comunicar-se e cooperar entre si.

Concorrência (software): É quando se dá a coexistência de vários processos (ou tarefas) que se podem ser executados em paralelo.

Paralelismo (hardware): É quando se dá a coexistência de vários elementos de processamento que podem executar em paralelo vários processos.

Multiprogramação: mapeamento de N processos em P processadores, em que o número de processos N é maior que o número de processadores P.

Multiprocessamento: execução em paralelo de P processos em P processadores, sendo que o número de processadores P é maior do que a unidade.

Com essas definições podemos apresentar um problema fundamental da programação concorrente, que é exatamente o de mapear toda a concorrência em paralelismo.

2.2.1 Taxonomia das arquiteturas

A crescente demanda de processamento motivou a busca de alto desempenho, e com isso o surgimento de vários modelos de arquiteturas paralelas. Segundo Duncan [Dun90] arquiteturas paralelas são caracterizadas pela presença de múltiplos processadores que cooperam entre si. Diversas taxonomias para arquiteturas paralelas foram propostas.

Segundo Flynn [Flyn66], as arquiteturas de computadores podem ser divididos em quatro categorias, de acordo com o fluxo de dados e instruções a serem processadas simultaneamente, conforme mostrando na figura 2.1 a seguir.

Figura 2.1 Taxonomia de arquiteturas segundo Flynn

a) *Single Instruction, Single Data* (SISD) É a arquitetura convencional do tipo computador von Neumann, com fluxos de entrada, processamento e saída, executando uma instrução por ciclo de *clock*. Possui um único processador. Cada operação atua sobre um único elemento de dados. Como exemplo temos a maioria dos computadores que não são supercomputadores

b) *Multiple Instruction, Single Data* (MISD) Esta forma é incluída somente para completar a taxonomia, não existindo exemplos práticos. Porém, as máquinas com a arquitetura MISD são classificadas diferentemente segundo vários autores.

Para alguns, correspondem à processadores vetoriais. Outros descrevem-nas como variantes de máquinas SIMD. Para a maioria porém, tais máquinas não têm existência prática.

c) *Single Instruction, Multiple Data* (SIMD) Seu funcionamento é do tipo síncrono, onde cada instrução é executada simultaneamente em cada processador (*lock step*), e determinístico. Cada instrução pode manipular mais de um dado por vez. É uma forma conveniente de paralelismo a nível de instrução e/ou operação. Subdivide-se em duas categorias: SIMD Vetorial e SIMD Paralela. Entre os exemplos conhecidos podem-se citar os *Arrays* de processadores, onde todos os processadores executam a mesma operação sobre dados diferentes, que tem o exemplo prático nas máquinas *Connection Machine* CM-2. Outro exemplo são os vetores com *pipeline*, onde os processadores têm unidades funcionais diferentes que permitem *pipeline*. Os exemplos práticos são o IBM 9000 e o *Cray YMP*

d) *Multiple Instruction, Multiple Data* (MIMD) Seu funcionamento tanto pode ser do tipo síncrono ou assíncrono, sendo conveniente para paralelismo de blocos, "loops" e subrotinas. O código pode ser o mesmo para todos os processadores (SPMD - *Single Program Multiple Data*) ou diferente. Entre os exemplos do caso assíncrono podem-se citar o IBM ES/9000-900, o *Cray YMP*, e o *cluster* de RS/6000 usando PVM¹. Como exemplo do caso síncrono, pode-se citar o RS/6000 (com quatro instruções por ciclo).

Segundo [Sat96], visando abranger os computadores escaláveis, Gordon Bell [Bel94] propôs uma taxonomia para incluir essas arquiteturas mais recentes. Dessa forma, pode-se incluir sistemas compostos de estações de trabalho interconectadas por um chaveamento de alta velocidade, que são classificados como computadores escaláveis. Um exemplo de computadores escaláveis são as chaves ATM. A taxonomia proposta por Bell [Bel94] apresenta a seguinte estrutura para classificação das arquiteturas:

- Fluxo de instruções único
 - Fluxo de dados único

¹ PVM - *Parallel Virtual Machine* apresentado no Apêndice C

- CISC
- RISC
- *Superscalar* e VLIW RISC
- Processadores de sinais digitais
- "*Multithreaded*"
- Fluxos de dados múltiplos
 - Vetoriais
 - SIMD
- Fluxos de Instruções Múltiplos
 - Multiprocessador (memória compartilhada)
 - Acesso à memória não uniforme
 - Somente cache (COMA)
 - "*Memory coherent*"
 - Nenhum cache
 - Acesso à memória uniforme
 - "*Multithreaded*"
 - Supercomputadores e "*main frames*"
 - Múltiplos barramentos
 - Multicomputador (passagem de mensagem)
 - Chaveamento distribuído e centralizado
 - Granularidade fina homogênea
 - Granularidade média não homogênea
 - Estações interconectadas na rede
 - Rede distribuída

- Aglomerados
- Estações em rede local
- Estações em área externa

A seguir serão fornecidos maiores detalhes de cada tipo de arquitetura, segundo *Bell in* [Sat96]:.

a) Fluxo único de instruções - O que caracteriza esta categoria é a execução de um único fluxo de instruções em cada processador. Nesta categoria estão incluídos desde computadores CISC até computadores com palavras de instruções longas, tais como os super escalares e VLIW RISC, e computadores onde múltiplos "*threads*", processam o mesmo fluxo de instruções. Os processadores vetoriais SIMD, núcleo dos supercomputadores, são largamente utilizados em aplicações com grande uso de ponto flutuante e em aplicações de transformações gráficas.

b) Fluxos múltiplos de instruções - O que caracteriza esta categoria, denominada de "MIMD", a execução de múltiplos fluxos de instruções sobre fluxos distintos de dados, pelos seus múltiplos processadores. São classificados conforme a forma de comunicação entre os processadores: os multiprocessadores e os multicomputadores.

b.1) Multiprocessadores - Nesta taxonomia, de acordo com o tipo de acesso à memória, os multiprocessadores que apresentam memória compartilhada são classificados em duas categorias: acesso à memória uniforme (UMA); e acesso à memória não uniforme (NUMA). A memória "cache" é caracterizada pelo seu acesso rápido. Os multiprocessadores classificados nesta categoria como "somente cache" (COMA) são um tipo de NUMA, onde a memória local de cada nó de processamento é transformada em memória cache. Uma subclasse de NUMA, aqui classificada como "coerência de memória", é caracterizada por apresentar uma memória "cache" pequena, utilizada para acessos a dados presentes na memória de outro nó. Os sistemas DASH [Len92], e FLASH [Kus94] são exemplos dessa classe de sistemas. Esta hierarquia de acesso presente em multiprocessadores NUMA deve ser explorada pelos compiladores e pelo programador para reduzir o custo de comunicação.

b.2) Multicomputadores - Esta categoria se caracteriza por apresentar recursos de comunicação por passagem de mensagem. Estações de trabalho distribuídas utilizando chaves ATM [McD95], com interfaces de comunicação bem projetadas, podem apresentar um custo de comunicação da ordem dos multicomputadores como o CM5 [Bel94]. Por outro lado, estações de trabalho conectadas em uma rede local são uma boa forma de se obter computadores escaláveis, uma vez que podem ser utilizadas até algumas dezenas de estações para executar um único trabalho.

2.2.2 Considerações sobre projeto e performance

Neste item faremos uma breve revisão da análise dos fatores que caracterizam e influenciam o desempenho de um computador, e das métricas normalmente usadas na sua medição. As métricas² podem ser realizadas sob a perspectiva do construtor de sistemas de computação que pretende introduzir melhorias no desempenho dos seus sistemas, e a do consumidor de informática que pretende efetuar comparações entre sistemas de diversos fornecedores, incluindo sistemas baseados em processadores distintos.

Inicialmente a pergunta que podemos fazer é: o que medir em um computador? A resposta pode conduzir a diversas possibilidades. A primeira é o tempo de execução de um programa, chamada de latência. Outra possibilidade é o número de tarefas executadas por unidade de tempo, chamada de *throughput*. Pode-se ainda relacionar a medida do tempo de execução de um programa. De certa forma, abstraindo-se a qualidade do programa ou algoritmo e compilador, deve-se ainda incluir: o tempo designado por uso total da UCP (que é a soma do tempo de uso da UCP pelo usuário e pelo sistema), a execução do código da aplicação na UCP, acessos à memória (para instruções e dados), acessos a periféricos externos

² Na seleção das métricas mais adequadas para uma comparação eficiente entre sistemas de diversos fornecedores, sugere-se a consulta na Internet das home-pages das instituições governamentais mais conceituadas: a SPEC (<http://www.spec.org>) e a TPC (<http://www.tpc.org>).

(incluindo discos), tempo de espera de I/O, tempo de execução de outros programas, e desempenho do sistema.

A seguir serão revistos alguns conceitos fundamentais a respeito da medição do desempenho em computadores e máquinas paralelas.

2.2.2.1 *Speed-up* e eficiência

O *speedup* (S_p) obtido por um algoritmo paralelo sendo executado sobre p processadores é a razão entre o tempo levado por aquele computador executando o algoritmo serial mais rápido (t_s) e o tempo levado pelo mesmo computador executando o algoritmo paralelo usando p processadores (t_p), ou seja,

$$S_p = \frac{T_s}{T_p}$$

onde S = tempo de execução seqüencial/tempo de execução paralela se existem N processadores executando $1/N$ do processamento. Então $S_{max} = N$.

Pode-se considerar alguns fatores que afetam o *speed-up*, entre eles a sincronização, que é a coordenação temporal das tarefas e que implica na espera por duas ou mais tarefas em um determinado ponto (ponto de sincronismo) para continuar a execução. Os processadores podem ficar parados esperando que outras tarefas finalizem a execução em outros processadores. Outro fator é o *overhead* de coordenação, que é o tempo utilizado para coordenar as tarefas, que não é utilizado no processamento do programa (tempo de iniciar e finalizar uma tarefa, tempo de sincronização).

Pode-se definir também o fator eficiência (E), que é a razão entre o *speedup* obtido com a execução com p processadores. Esta medida mostra o quanto o paralelismo foi explorado no algoritmo. Quanto maior a fração inerentemente seqüencial, menor será a eficiência.

A Lei de Amdahl determina o potencial de aumento de velocidade a partir da porcentagem paralelizável do programa. Pela Lei de Amdahl, tem-se que, se f é a

fração inerentemente seqüencial de uma computação a ser resolvida por p processadores, então o *speedup* S é limitado de acordo com a fórmula a seguir:

$$Sp = \frac{1}{1 - \%paralelizavel}$$

Segundo Amdahl, em um programa que não seja paralelizavel, ou seja um programa totalmente seqüencial, o valor de % paralelizável do programa é igual a zero, que leva a um valor de *speedup* igual a 1, ou seja, não existe aumento na velocidade de processamento. Em um programa totalmente paralelo, no qual ocorra paralelização total, o valor de % paralelizável do programa é igual a 1, e que leva a um valor de *speedup* ilimitado.

2.2.2.2 Granularidade

A granularidade é uma medida usada para indicar a relação entre o tamanho de cada tarefa e o tamanho total do programa, ou seja, é a razão entre computação e comunicação. Pode-se também entender a granularidade como sendo o "grau de divisão" do dado de um problema entre as UCPs que participarão da solução. Assim, a granularidade está diretamente relacionada com o tempo de processamento total (quanto maior ela for, menor será o tempo de processamento) e com o tempo de comunicação entre as UCPs (uma vez que quanto mais UCPs participarem da tarefa, maior será o tempo de comunicação).

O nível de granularidade varia de fina ("fine-grained", ou pouco processamento) a grossa ("coarse-grained", quando há muito processamento).

Granularidade Fina - "fine-grained" (figura 2.2): Possuem uma ou poucas instruções. Provoca uma maior frequência nas comunicações, e facilita o balanceamento de carga. Neste caso o processamento é menor do que a comunicação e tende a diminuir a desempenho do programa levando a um alto custo de sincronização.

Figura 2.2 - Granularidade fina

Granularidade Grossa - "coarse-grained" (figura 2.3): As tarefas possuem muitas instruções, tais como uma subrotina. Pode permitir melhorar o desempenho do programa, mas dificulta o balanceamento de carga. Neste caso, ocorre uma menor frequência nas comunicações, e o processamento é maior do que a comunicação, o que leva a um menor custo de sincronização.

Figura 2.3 - Granularidade grossa

A melhor granularidade depende do grau de dependência entre os processos. É importante definir o grau de controle (sincronização) necessário. Finalmente, a própria natureza do problema pode limitar a granularidade possível.

Um sistema paralelo é dito escalonável³ ("*scalable*") quando a inclusão de mais processadores leva a um aumento proporcional do *speedup*. Isso depende de

³ Escalabilidade É a propriedade de um sistema paralelo de aumentar o *speedup* a medida em que aumenta-se o número de processadores

alguns fatores que incluem o hardware, o algoritmo utilizado na paralelização, e o código gerado. Sob o aspecto de hardware, um dos maiores gargalos é o tráfego de informação na rede que interliga os processadores. O aumento do número de processadores pode degradar a capacidade de comunicação da rede diminuindo o desempenho do programa. Sobre o aspecto de software, alguns algoritmos trabalham melhor em certas escalas de paralelismo. O aumento do número de processadores pode acarretar uma queda na eficiência durante a execução do programa.

2.2.2.3 Balanceamento de carga e distribuição de processos

É a distribuição otimizada das tarefas, ou alocação de processos entre os processadores, de modo a assegurar o tempo mais eficiente de execução dos processos em paralelo. O ideal é que a distribuição dos processos que compõem a aplicação entre as máquinas minimize o custo de execução. Se as tarefas não são distribuídas de maneira equilibrada, uma tarefa pode ficar esperando por outra. O desempenho pode ser aumentado se o trabalho for distribuído de maneira uniforme. O usuário submete um processo ao software de balanceamento de carga, que irá fazer a distribuição do processo ao processador que estiver mais disponível no momento. O balanceamento de carga pode fazer com que um processo fique em fila de espera aguardando um processador para ser executado. Existem alguns softwares que executam o balanceamento de carga como o *LoadLeveler* da IBM, descrito resumidamente no apêndice C, e o software gratuito (*freeware*) DQS.

A distribuição deve maximizar o uso dos recursos de processamento mas minimizar os custos de comunicação. Se as tarefas são de tamanhos diferentes, pode ser mais eficiente manter um "pool" de tarefas, que serão distribuídas para cada processador à medida que eles acabem o processamento de uma tarefa. O balanceamento de carga em um sistema paralelo é um dos mais importantes problemas a serem resolvidos para que o mesmo seja eficiente. O balanceamento pode ser mais importante em alguns ambientes do que em outros. Uma comparação que pode ser feita é aquela que leva em consideração um ambiente heterogêneo e carga definida pelo usuário, contra um ambiente com processadores idênticos

rodando um *job* por processador. Quanto ao tipo, a distribuição pode ser estática ou dinâmica.

A depuração de um programa paralelo é bem mais complicada do que a de um programa serial, embora depuradores paralelos comecem a estar disponíveis para uso. Devemos observar que os detalhes de comunicação têm que ser tão bem estudados quanto os de processamento. Além disso ferramentas para análise e monitoramento de programas paralelos já existem.

2.2.2.4 Problemas que podem ocorrer na paralelização

É importante discutir também quais são os principais obstáculos que ocorrem no Paralelismo. Entre eles está a sincronização entre os processos, onde as tarefas executadas em paralelo, num determinado instante, aguardam a finalização mútua para coordenar os resultados ou trocar dados e reiniciar novas tarefas em paralelo. É necessário que haja a coordenação dos processos e da comunicação entre eles para evitar que a comunicação seja maior do que o processamento e que, conseqüentemente, haja uma queda no desempenho dos processos em execução. Outros obstáculos são os compiladores e ferramentas que encontram-se em amadurecimento. Existem poucos compiladores e ferramentas prontas para paralelização automática que resolvam definitivamente o problema do paralelismo.

Podemos relacionar como obstáculo também a conversão de algoritmos seqüenciais em paralelos, pois o programador gasta um tempo considerável em analisar o código fonte para paralelizar e recodificar este código. Basicamente é necessário rever o programa seqüencial, avaliar como será particionado, quais os pontos de sincronização, quais as partições que poderão ser executadas em paralelo e qual a forma de comunicação que será empregada entre as tarefas paralelas. Além disso, nem tudo é paralelizável. É necessário que haja uma análise do algoritmo a ser paralelizado para que seja possível a paralelização nos dados ou nas funções do mesmo, levando sempre em consideração a quantidade de comunicação em relação ao processamento. Como exemplo de problema que pode ser paralelizado pode-se

citar: calcular a energia potencial de milhares de estruturas moleculares diferentes e achar a de menor energia

No entanto pode-se reduzir o tempo total de processamento. Devemos considerar que, por ser uma tecnologia recente, há dificuldades em se desenvolver aplicações usando paralelismo. Um problema é a perda de Portabilidade, pois a programação paralela utiliza multiprocessadores com arquiteturas de comunicação entre os processadores baseados em memória compartilhada (*shared memory*) e memória distribuída (*distributed memory*). Os programas adaptados aos processadores baseados em memória compartilhada não podem ser executados em uma máquina com memória distribuída, pois não utilizam os mecanismos de troca de mensagens, impossibilitando assim a portabilidade. No caso inverso, memória distribuída para memória compartilhada, seria possível a portabilidade; porém o mecanismo de troca de mensagens seria ignorado, pois todos os dados já poderiam ser acessados pelos processadores através da memória compartilhada. Isto apenas traria um fluxo desnecessário de comunicação.

Como obstáculo podemos ainda citar o problema na depuração (Debug) de código. Os processos são distribuídos e executados em vários processadores simultaneamente, entretanto não existe uma forma eficiente de acompanhar passo-a-passo a alteração das variáveis durante a execução do processamento das diversas tarefas paralelas. Uma alternativa que está sendo empregada é a utilização de arquivos de rastro de execução, chamados *tracefiles*, que contém um conjunto de dados gerados durante a execução para que sejam posteriormente analisados pelo usuário. Existem programas com interface gráfica que permitem ao usuário acompanhar a evolução do desempenho durante o processamento das tarefas, como por exemplo, o Xmpi para MPI LAM, Paragraph, Pablo, Paradyn, PE e VT.

Finalmente podemos relacionar os casos que não são paralelizáveis, como o seguinte problema, estritamente sequencial (exemplo em FORTRAN):

```
DO 500 J=1,50 A(J)=A(J-1) *2.0
```

```
500 CONTINUE
```

2.2.2.5 Modelos de acesso à memória

A comunicação entre os processadores depende da arquitetura de memória utilizada no ambiente. Para entendermos como os processadores se comunicam é fundamental compreender esses tipos de arquitetura de memória. Entre os modelos de acesso à memória encontramos o acesso à memória compartilhada, e o acesso à memória distribuída, que serão descritos a seguir.

Memória compartilhada: Neste caso, múltiplos processadores acessam a mesma memória, sendo que somente um processador acessa um endereço na memória por vez, e a sincronização é feita pelo controle das operações de leitura e escrita executadas pelas tarefas. Sua implementação é feita em hardware. Uma localização de memória não pode ter seu valor mudado por uma tarefa enquanto outra tarefa estiver acessando a mesma localização. O compartilhamento de dados entre tarefas é rápido, e é feito na velocidade de acesso à memória. Uma das desvantagens neste caso é que, se o número de processadores aumenta, os caminhos de acesso à memória podem formar um gargalo no sistema. O usuário é responsável por especificar a sincronização, utilizando *locks* (travas). Outra desvantagem é o fato de que essas máquinas são computadores extremamente caros, e há um limite no número de processadores. Além disso são necessárias técnicas de sincronização para a leitura e gravação especiais. Entre os exemplos podemos citar o Cray Y-MP.

Memória distribuída (*Distributed Memory*): A memória é fisicamente distribuída entre os processadores, ou seja, os processadores operam independentemente de onde cada um possui sua própria memória. O acesso à memória local é feito sem interferência. Assim, cada memória local só é acessada pelo seu próprio processador, e os dados são compartilhados através da rede usando um mecanismo de troca de mensagens (*Message Passing*). A sincronização é realizada através da comunicação entre os processadores. Uma das maiores preocupações é como fazer a decomposição dos dados, ou seja, como dividir *arrays* entre as UCPs de modo a minimizar a comunicação. Não existe limite teórico para o número de processadores. Os exemplos deste modelo são as máquinas hipercubo o *cluster* de estações com PVM, o IBM Risc/6000 SP2, etc. Nestes casos, a memória é fisicamente distribuída, mas parece para o usuário como uma única memória

compartilhada, com espaço de endereçamento único. Entre as desvantagens que este modelo apresenta está a dificuldade para mapear as informações, sendo que o usuário é responsável pelo sincronismo e recebimento de dados. Outra desvantagem é o elevado *overhead* devido à comunicação.

2.3 Programação Paralela

Para obter resultados mais rápidos em tarefas grandes e complexas uma técnica que pode ser utilizada é a exploração do paralelismo. Esta técnica consiste em dividir-se as grandes tarefas em tarefas pequenas que serão distribuídas em vários processadores para serem executadas simultaneamente. Esses processadores comunicam-se entre si para que haja coordenação (sincronização) na execução das diversas tarefas em paralelo. O principal objetivo do paralelismo é aumentar o desempenho (reduzindo o tempo de execução) no processamento. Deve-se dizer que no próprio paralelismo existem complexidades pertinentes, e muitas vezes os resultados podem não corresponder ao esforço empregado em programação.

Podemos ter diversos tipos de paralelismo, e dentre as várias formas de classificá-los que foram encontradas na literatura, levou-se em consideração o objeto paralelizado, como segue abaixo [Fos95], [Bau92]:

Paralelismo de Dados- O processador executa as mesmas instruções sobre dados diferentes. É aplicado, por exemplo, em programas que utilizam matrizes imensas e para cálculos de elementos finitos. Entre os exemplos pode-se citar a resolução de sistemas de equações, a resolução de problemas de elementos finitos, a integração numérica. Nesta técnica, uma grande quantidade de dados é submetida paralelamente a um processamento idêntico ou similar em diferentes UCP.

Paralelismo Funcional- O processador executa instruções diferentes que podem ou não operar sobre o mesmo conjunto de dados. É aplicado em programas dinâmicos e modulares onde cada tarefa será um programa diferente. Os exemplos deste tipo de paralelismo podem ser o clássico problema do Paradigma Produtor-Consumidor, simulação, rotinas específicas para tratamento de dados (imagens).

Paralelismo de Objetos- Este é o modelo mais recente, que se utiliza do conceito de objetos distribuídos por uma rede, como por exemplo a Internet, capazes

de serem acessados por métodos, ou funções, em diferentes processadores para uma determinada finalidade. Entre os exemplos encontra-se o MPP⁴ (Processamento Massivamente Paralelo), compreendendo o conceito de "massivamente distribuído".

Algoritmo Relaxado- Neste algoritmo, os processos paralelos rodam de maneira auto-suficiente, cada um com a sua porção de dados sem qualquer sincronização ou comunicação entre eles.

Interação Síncrona- Cada processador realiza o mesmo trabalho em diferentes porções de dados. Neste algoritmo, entretanto, é necessária uma sincronização e atualização dos dados a fim de usar-se o resultado de uma interação nas interações subseqüentes. Muitas simulações, quando convertidas para o ambiente paralelo, tomam a forma deste algoritmo.

Ao desenvolver um algoritmo paralelo devemos estudar o que se quer paralelizar, determinar o custo de processamento de cada tarefa, e saber como dividir as tarefas entre as UCP., neste caso quanta carga cada tarefa vai exigir da UCP, e qual a arquitetura de máquinas que possuímos. Assim, ao se dividir um problema, temos que notar a importância de balancear a divisão dos processos de modo que uma UCP mais poderosa não fique sub-utilizada ou que uma com menos poder de processamento não seja sobrecarregada.

Sem dúvida, a comunicação é o maior gargalo da execução de programas em paralelo. Assim, deve-se procurar minimizá-la ao máximo, afim de evitarmos uma queda de performance na execução. A maneira mais óbvia de minimizar a comunicação é diminuir o número de processadores envolvidos na solução do problema.

De maneira geral, as partes do código paralelizáveis consistem de funções do programa que não dependem umas das outras e que podem ser executadas simultaneamente. Um exemplo típico de código paralelizável seria o problema de calcular a integral de uma função pelo método de somatória simples. A forma mais direta de resolver este programa em paralelo seria dividir a função em partições e passar para diferentes UCPs cada uma dessas partições de modo que cada UCP

⁴ Ver no Apêndice C

calcule a integral em sua pequena região. Ao final do cálculo, cada UCP passaria o seu resultado para uma outra UCP que se encarregaria de somar os valores, obtendo assim a integral.

Uma última consideração importante é a sincronização entres as tarefas. Uma vez que as UCPs geralmente poderão possuir carga e poder de processamento diversos, há a possibilidade de que uma UCP termine a sua tarefa antes de outra e tenha que passar dados para aquela que ainda está trabalhando. Assim, é importante que o programador tenha uma boa noção da ordem de execução dos processos para que não deixe um processo esperando por muito tempo a resposta de outra UCP, o que certamente atrasa a execução .

2.3.1 Paralelização de Programas

A divisão do problema grande em problemas pequenos ainda é uma tarefa artesanal. Devido a grande diversidade existente de arquiteturas paralelas, foram propostas diversas linguagens, vários compiladores e criadas muitas bibliotecas especiais, visando explorar o máximo de paralelismo possível. Existem esforços para se automatizar esse processo mas as coisas não são tão fáceis. As principais alternativas para programação paralela são:

a) Compiladores Paralelizantes. O paralelismo em um programa pode ser explorado implicitamente através de compiladores paralelizantes [Sat96], como o Parafrase2 [Pol88] e o Suif , que paralelizam automaticamente, ou explicitamente, programas escritos em C. A utilização de um sistema paralelizante minimiza dificuldades naturais das linguagens paralelas, e permite também o aproveitamento de programas sequenciais já implementados. Alguns compiladores paralelizantes encontram-se em fase experimental e podem ser conseguidos através da Internet, como pode ser visto na Tabela 2.1.

b) Linguagens Paralelas. Na programação paralela a linguagem expressa o paralelismo através de comandos, permitindo explorar fontes de paralelismo que não sejam passíveis de detecção por um sistema paralelizante. Em outras palavras, pode-se ir muito além do que compiladores paralelizantes. Com isso podem ser

exploradas as aplicações onde formas de paralelismo específicas requisitadas não podem ser paralelizadas automaticamente.

c) **Bibliotecas especiais.** Muitas bibliotecas específicas têm oferecido uma forma de implementar programas de forma paralela. Um conjunto de bibliotecas específicas para multiprocessadores vem sendo disponibilizadas, tais como a fornecida pelo sistema *Balance* da *Sequent* [Tha88], e muitas outras voltadas para diversas plataformas, como a biblioteca do sistema PVM (*Parallel Virtual Machine*) [Gra92], e as bibliotecas do sistema *Quarks* [Uta95], e ainda do sistema *Treadmarks* [Kel93] que oferecem um modelo de memória distribuída compartilhada.

c) **Diretivas de compilação** Compiladores como o *Power C* [Bau92] oferecem diretivas especiais para utilizar o paralelismo. A seguir é apresentado um programa escrito em *Power C Language*, segundo exemplo de [Sat96]:

```
#define MAXRET 1000000
double total_pi = 0.0;
void main()
{
    int i;
    double x, largura, local_pi;
    largura = 1/MAXRET;
    #pragma parallel
    #pragma shared (total_pi, largura)
    #pragma local(i, x, local_pi)
    {
        #pragma pfor iterate (i=0; MAXRET; 1)
        for (i = 0; i < MAXRET; i++)
        {
            x = ((i-0.5) * largura);          /* calcula x */
            local_pi = local_pi + (4.0 / ( 1.0 + x * x));
        }
        local_pi = local_pi * largura;
    }
    #pragma critical
    {
        total_pi = total_pi + local_pi;
    }
}
printf("valor de pi = %d\n", total_pi);
}
```

Tabela 2.1 Softwares disponíveis na Internet para paralelização

Software	Descrição	Endereço (URL)
PVM	Software que permite que uma rede heterogênea possa ser usada como um único computador paralelo, possibilitando a distribuição de tarefas entre processos executados em computadores de arquiteturas distintas.	Oak Ridge National Laboratory (ORNL) http://www.ornl.gov/ PVM em Cornell Theory Center http://www.tc.cornell.edu/UserDoc/Software/PTools/pvm/
XPVM	Ferramenta p/ visualização de aplicações paralelas desenvolvidas com PVM.	Oak Ridge National Laboratory (ORNL) http://www.ornl.gov/
PVMe	<i>Parallel Virtual Machine Extended</i> - Versão do PVM específico para Switch-SP2.	PARALLEL VIRTUAL MACHINE EXTENDED (PVMe) http://lscftp.kgn.ibm.com/ppsvibm/show/products/pvme.html PVMe em Cornell Theory Center http://www.tc.cornell.edu/UserDoc/Software/PTools/pvme/
PE	<i>Parallel Environment</i> -Ambiente de desenvolvimento e execução de aplicações paralelas em C, C ++ ou Fortran.	PARALLEL ENVIRONMENT (PE) http://lscftp.kgn.ibm.com/ppsvibm/show/products/pe.html PE em Cornell Theory Center http://www.tc.cornell.edu/UserDoc/Software/PTools/pe
POE	<i>Parallel Operation Environment</i> , parte integrante do ambiente PE.	PARALLEL OPERATION ENVIRONMENT (POE) http://www.tc.cornell.edu/UserDoc/Software/PTools/pe
VT	Ferramenta para visualização e monitoração de desempenho (performance) de programas paralelos. Parte integrante do ambiente PE.	VISUALIZATION TOOL (VT) http://www.tc.cornell.edu/UserDoc/Software/PTools/vt/
<i>Parallel Debugger</i> (xpdbx)	Ferramenta para depuração de programas paralelos.	PARALLEL DEBUGGER (XPDBX) http://www.tc.cornell.edu/UserDoc/Software/PTools/pdbx/

MPL	<i>Message Passing Library</i> , parte integrante do PE.	MESSAGE PASSING LIBRARY (MPL) http://www.tc.cornell.edu/UserDoc/Software/PTools/mpl/
MPI-CH	Implementação do padrão MPI (<i>Message Passing Interface</i>) da Universidade de Argonne, Illinois, EUA que define rotinas que possibilitam a distribuição de tarefas entre processos, permitindo a paralelização de aplicações, através de troca de mensagens.	MPI-CH http://www.mcs.anl.gov/mpi/mpich MPI em <i>Cornell Theory Center</i> http://www.tc.cornell.edu/UserDoc/Software/PTools/mpi/MPI_intro/
MPI-LAM	Implementação do padrão MPI (<i>Message Passing Interface</i>) pelo Centro de Supercomputação de Ohio, EUA que define rotinas que possibilitam a distribuição de tarefas entre processos, permitindo a paralelização de aplicações, através de troca de mensagens.	MPI-LAM http://www.osc.edu/OLD/lam.html
<i>Adaptor</i>	Paraleliza programas em Fortran	Não disponível para <i>download</i>
<i>Paragraph</i>	Software para visualização de desempenho de programas paralelos	Não disponível para <i>download</i>
Java PVM	Ambiente PVM - Java	Não disponível para <i>download</i>

O paralelismo expresso pelo usuário pode, portanto, ser muito maior do que aquele especificado por um programa paralelo através da expressão por comandos específicos de uma linguagem de programação paralela. O paralelismo pode também ser explorado pelo usuário, através de chamadas a rotinas de uma biblioteca que administram os recursos de paralelismo da máquina. E, finalmente, pode ser explorado através de diretivas do compilador. Entre estas formas, a utilização de uma linguagem de programação paralela é a que oferece um ambiente mais adequado de programação.

Um programa sequencial especifica a execução de uma lista de comandos em ordem rígida e linear. Um processo pode ser definido como a execução de um

programa seqüencial. A comunicação entre processos que podem ser atribuídos aos múltiplos processadores de um computador com arquitetura paralela, pode ser efetuada através de variáveis compartilhadas ou por passagem de mensagem. Linguagens de programação paralela, baseadas em passagem de mensagem, como a *Concurrent C* [Geh86] [Geh88][Geh92] e a *Ada* [Geh88] foram propostas, assim como bibliotecas que oferecem rotinas para a passagem de mensagem [Tur93], como a biblioteca do sistema PVM (*Parallel Virtual Machine*) [Gra92].

Os modelos de programação baseados em variáveis compartilhadas permitem implementações com menor complexidade em relação aos modelos com passagem de mensagem. Entretanto, as leituras e escritas dessas variáveis devem ser feitas considerando algumas restrições. A leitura e escrita, ou múltiplas escritas, de variáveis não podem ser executadas de forma simultânea. Isto exige a utilização de uma seção crítica, que envolve o acesso à variáveis compartilhadas. O mecanismo de exclusão mútua é uma forma que implementa a seção crítica, garantindo que uma seqüência de comandos seja executada de forma exclusiva por um processo.

2.3.2 Formas de Paralelizar um Programa

Como já foi descrito, existem duas formas de se paralelizar um programa. Uma das formas é a automática, onde alguns compiladores tentam fazer alguma paralelização automática, especialmente em "DO loops", mas que ainda se encontram em um estado inicial. A outra forma é a manual, onde o programador gasta tempo para paralelizar. Ele deve seguir os seguintes passos:

1. Se já existe um programa serial, deve-se depurar o código serial completamente.
2. Identificar as partes do programa que podem ser executadas concorrentemente: este fato exige que haja uma total compreensão do algoritmo. Pode ser necessário uma total reestruturação do programa ou algoritmo, ou ainda um algoritmo totalmente novo.
3. Identificar as dependências de dados: um *loop* possui dependência de dados quando uma mesma localização da memória é escrita em diferentes iterações. Ele não pode ser paralelizado porque a ordem de execução das iterações é importante

para o resultado final. Um *loop* possui independência de dados, quando uma mesma localização de memória não é utilizada em iterações diferentes. Neste caso, pode ser paralelizado.

4. Decompor o programa utilizando alguma forma de paralelismo: entre as possíveis decomposições temos o paralelismo funcional (que é a decomposição e distribuição do processamento a ser realizado por cada tarefa, com programas diferentes para cada processador), o paralelismo de dados (que é a decomposição dos dados em partições e posterior distribuição). Deve-se distribuir os programas de forma igual em cada processador.

5. Compilação, depuração e testes.

6. Otimização de medidas de desempenho e identificação de áreas problemáticas.

Fisicamente, a paralelização de programas pode ser feita através de diversos meios. A forma mais comum é utilizar uma rede de processadores, onde não existe controle centralizado e cada processador executa um programa diferente. Outra opção é usar uma rede com computador mestre e computadores escravos, onde os programas são diferentes para o processo mestre e para os processos escravos. O processo mestre centraliza o controle, e inicia os escravos, distribui o trabalho, sincroniza a comunicação e executa operações de entrada e saída. O computador mestre pode ou não contribuir para a execução do programa a ser processado. Tipicamente, os escravos executam todo o processamento necessário. E, finalmente, a paralelização pode ser feita usando computadores com arquitetura SPMD (*Single Program Multiple Data*). Neste caso, não existe um controle centralizado, e todos os processadores executam o mesmo programa, sendo que todos os processadores contribuem para o processamento. Este modelo implica que os cálculos e as comunicações entre os processadores sejam coordenados em determinados pontos.

2.3.3 - Linguagens de Programação

Segundo [Sat96] as linguagens podem ser classificadas em dois tipos básicos: as linguagens imperativas e as linguagens declarativas. As linguagens imperativas

são caracterizadas por modificar um estado do programa através de comandos. Fazem parte desta classe as linguagens procedurais e as linguagens orientadas a objetos. As linguagens declarativas são caracterizadas por uma programação realizada através de expressões. Fazem parte desta classe as linguagens funcionais e lógicas. A seguir serão apresentados maiores detalhes destes tipos de linguagem.

a) Linguagens procedurais. As linguagens procedurais são caracterizadas por subdividir o programa em subprogramas, ou também como são chamados em rotinas, subrotinas ou procedimentos. Os exemplos mais conhecidos dessa classe são as linguagens C e Pascal.

Com os efeitos colaterais que podem ser gerados, como a alteração em um procedimento com dados que são lidos em outros procedimentos, a análise de dependência de dados torna-se uma tarefa complexa. Existe uma dependência de dados entre dois comandos quando um deles altera um dado que é lido ou escrito pelo outro. Neste caso, os dois comandos não podem ser executados simultaneamente. Na programação através de uma linguagem que permite que o usuário expresse o paralelismo, este deve efetuar uma análise de dependência de dados. No caso de existir uma dependência de dados, mecanismos de exclusão mútua devem ser utilizados, envolvendo o acesso ao dado.

Devido ao fato de que as linguagens procedurais serem muito utilizadas, existem extensões dessas linguagens, entre as quais a linguagem C e o Fortran. Entre as extensões da linguagem C que foram propostas, podem ser citadas: a *Concurrent C* [Geh86] [Geh92], a *Jade* [Rin93] e a *CPAR* [Sat96]. Existem também propostas algumas extensões para a linguagem Fortran, entre as quais Fortran D [Fox92] e a *HPF (High Performance Fortran)* [Lov93].

b) Linguagens orientadas a objetos. Sobre o ponto de vista da programação paralela, o paradigma de orientação a objetos apresenta alguns conceitos atraentes, que são o encapsulamento das informações em objetos, e o seu acesso apenas através de chamadas a seus métodos. O encapsulamento garante a inexistência de dependência de dados entre objetos, existindo assim um paralelismo implícito entre eles. Os paralelismos existentes entre métodos de um mesmo objeto podem ser

explorados. Alguns exemplos de linguagens para arquiteturas paralelas com memória compartilhada ou distribuída foram propostas, entre as quais podem ser citadas as seguintes: COOL [Cha90] [Cha94], a Mentat [Gri93], a Orca .

c) Linguagens funcionais. As funções de programas que são definidos segundo uma linguagem funcional comportam-se como funções matemáticas, e seu o resultado obtido depende apenas de seus argumentos [Bal89a]. Não existem efeitos colaterais, visto não ser permitido que a função interagir em a execuções de outras funções, através de variáveis globais ou de ponteiros. Do ponto de vista do paralelismo a ausência de efeitos colaterais é uma característica atraente, visto não haver dependência entre as funções, o que proporciona que seja explorado o paralelismo implícito entre elas. Quando porém uma função utiliza o resultado retornado por uma outra função, esta função é obrigada a esperar esse resultado, causando uma queda no desempenho do programa. Alguns cuidados devem ser tomados, pois embora possa existir uma grande possibilidade de exploração do paralelismo, podem haver muitas paralelizações cujas granularidades sejam muito finas, porém, sabe-se que as granularidades maiores possuem um desempenho melhor.

d) Linguagens lógicas. Os programas construídos utilizando o paradigma da programação lógica são expressos como regras lógicas. Uma propriedade das linguagens de programação lógica que é particularmente atraente para o ponto de vista do paralelismo, e que está presente em sua semântica, é a independência da ordem em que diferentes operações são processadas durante a execução do programa. Esta independência permite que estas operações sejam executadas paralelamente [Pon94]. Alguns exemplos de propostas de linguagens lógicas para processamento paralelo foram apresentadas, explorando implicitamente o que ficou conhecido como AND-paralelismo e o OR-paralelismo [Pon94].

2.4 Considerações finais

No Apêndice C são apresentadas de forma resumida as linguagens e aplicações paralelas encontradas comercialmente ou ainda em fase de estudo. O que se pretende mostrar neste Apêndice é um breve relato dos principais aspectos das

linguagens e aplicações existentes, buscando sempre destacar os aspectos mais relevantes de cada exemplo. Finalmente deve-se dizer que devido ao enorme número de implementações encontradas, apenas as mais importantes e com certo interesse estão relacionadas. Algumas serão apenas referenciadas, e sempre que possível haverá alguma referência bibliográfica, ou mesmo um endereço eletrônico da Internet. Sempre que possível serão colocados exemplos de comandos ou comportamentos paralelos das linguagens, com o intuito de exemplificar as mesmas.

Entre as principais ferramentas de programação encontramos as já citadas bibliotecas para passagem de mensagens, que na realidade são novas funções construídas para serem ligadas através de *links* (linkadas) a programas de linguagens já existentes como C, FORTRAN e C++. Entre as principais, que serão apresentadas a seguir, encontramos o PVM e MPI

As linguagens de programação permitem verificação de corretude e otimizações maiores, porém isso nos remete a incorporação de um novo modelo de programação nas linguagens. As principais que surgem quando falamos de linguagem de programação distribuída, nos levam a fatores como a comunicação entre processos, com referência a memória compartilhada e troca de mensagem. Outros fatores são quanto a criação de processos e o suporte ao paralelismo, e ainda a forma como são feitos os mapeamentos de processos a processadores, na forma estática ou dinâmica. Entre as principais que estão relacionadas encontramos HeNCE, Linda, HPF, CC++, SR, C*.

Capítulo 3

A Linguagem Java

Resumo Aqui são apresentados os aspectos mais importantes da ferramenta de programação¹ JAVA, e é analisada sua estrutura. Neste capítulo serão revistos os conceitos da ferramenta Java e as especificações da linguagem Java. É apresentada uma sinopse das ferramentas para Java encontradas na literatura.

3.1 Introdução

A ferramenta Java além de ser uma linguagem de programação é também uma plataforma de programação. Como linguagem de programação, Java é dita de alto nível e normalmente é descrita por seus desenvolvedores como: simples, independente de arquitetura, orientada para objeto, portátil, distribuída, de alto desempenho, interpretada, *multithreaded*, robusta, dinâmica, segura [Ces98], [Gos96].

3.1.1 Conceitos básicos

Um programa Java é compilado e interpretado, e isso faz com que a ferramenta Java seja incomum. Atuando como um compilador, pode-se realizar a tradução de um programa Java em uma linguagem intermediária. Esse código intermediário é denominado *bytecode* Java, e é um código independente de plataforma. Este código será

¹ Usaremos neste capítulo o termo ferramenta Java no sentido que irá englobar não só a linguagem, mas toda a plataforma, envolvendo a máquina virtual, o interpretador, o compilador, etc.

interpretado pelo interpretador Java. Com um interpretador, são analisadas gramaticalmente as instruções do *bytecode* Java e essas são executadas no computador hospedeiro, porque este arquivo é binário e específico para cada plataforma. A compilação ocorre apenas uma vez; a interpretação ocorre cada vez que o programa é executado. A figura 3.1 a seguir, ilustra como isto ocorre.

Figura 3.1-Fluxo de execução de um programa Java

Pode-se pensar nos *bytecodes* Java como sendo as instruções de código de máquina para uma máquina virtual Java (JVM). Todo o conjunto de ferramentas Java, seja um interpretador Java, seja um conjunto de desenvolvimento Java ou um *browser* de Rede que pode executar *applets*² Java, são uma implementação da máquina virtual (JVM). A JVM também pode ser implementada em *hardware*.

Os *bytecodes* Java buscam cumprir a promessa "escreva uma vez, execute em qualquer lugar"³. Pode-se compilar um programa Java em *bytecodes* em qualquer plataforma que tenha um compilador Java, sendo que hoje em dia as principais plataformas já possuem este tipo de implementação. Os *bytecodes* podem ser executados

² Os *Applets* são um tipo especial de programa Java que é executado em um browser de rede

³ *Write once, run anywhere*

então em qualquer implementação da JVM. Por exemplo, o mesmo programa Java pode ser executado em Windows NT, Solaris, e Macintosh, como pode ser visto na figura 3.2.

Figura 3.2 - A independência de plataforma de um programa Java

Como um ambiente independente de plataforma, a ferramenta Java pode ter um desempenho mais lento que o de um código próprio da máquina hospedeira. Porém, compiladores inteligentes, interpretadores bem desenvolvidos, e compiladores *just-in-time* de *bytecode* podem tornar o desempenho de aplicações Java próximos do desempenho do código nativo, sem ameaçar a portabilidade.

A maioria dos programas em Java contém um ou mais objetos ou unidades de compilação. Cada unidade contém declarações de pacote e importação, além de declarações de classes e interface. Embora possam existir várias classes e interfaces por unidade, cada unidade pode ter no máximo uma interface e uma classe públicas, porque é dessa maneira que a unidade será utilizada por outros objetos do sistema. O restante deve ser privado, e pode ser declarado explicitamente como privado na declaração de classe, ou aceito como *default*.

Ao utilizar o compilador Java, *Javac*, o usuário estará compilando código fonte Java em *bytecodes*. Esses *bytecodes* contêm instruções independentes de máquina. Em seguida, o interpretador Java, também chamado Java (parte do ambiente de execução Java), interpreta esses *bytecodes*. Ao criar uma unidade de compilação Java, cada programa deverá ter um sufixo *.Java*.

3.1.2 Histórico

A ferramenta Java é uma linguagem e um ambiente de programação, projetada para resolver vários problemas práticos de programação moderna. A Java começou como parte de um projeto maior da Sun Microsystems, Inc, cuja intenção inicial era desenvolver *softwares* avançados para consumidores de produtos de eletrônica. A missão inicial era desenvolver aplicativos complexos e avançados para aplicação em pequenos dispositivos eletrônicos, com foco principal nas televisões interativas. Estes dispositivos deveriam ser pequenos, confiáveis, portáteis, atuando de forma distribuída. As primeiras tentativas de criação de tais dispositivos receberam nomes como *Digital Assistance e Portable Data Assistentes* (PDAs).

Quando esse projeto começou, a intenção era usar C++, mas foram encontrados vários problemas. Inicialmente estes eram apenas os apresentados pela tecnologia de compiladores, mas com o tempo passaram a surgir outros problemas e, assim, a melhor forma de resolvê-los foi propor uma nova linguagem. A idéia era criar um *software* para controle de televisões, mas por volta de 1994 a *Sun* percebeu que nem o mercado nem a

tecnologia estavam prontos para isso, e decidiram direcionar o projeto para a Internet. Desde o seu início, em 1992, surgiram muitas outras aplicações para a ferramenta Java.

A Java foi lançada em 1995 como versão alfa. O pacote consistia do *browser* (navegador) *HotJava*⁴, dos compiladores Java, das bibliotecas e todo o ambiente necessário para desenvolver programas em Java. A versão beta, lançada posteriormente, introduziu mudanças estruturais tornando-a incompatível com a versão anterior.

No início de 1996, a Sun Microsystems criou a *JavaSoft*, lançada em janeiro de 96, na versão 1.0 do ambiente de desenvolvimento Java (JDK 1.0). O kit contém todas as ferramentas necessárias para escrever qualquer programa ou *applet* (pequenos programas) usando Java. Contém um *Applet Viewer*, que permite testar e visualizar *applets*, o compilador Java, um *debugger* de protótipo e a máquina virtual Java. O pacote Java tem a capacidade de criar programas auxiliares, que podem ser usados para montar documentos HTML (*Hypertext Markup Language*).

A partir daí a evolução não parou. As maiores empresas de *software* do mundo, inclusive a Microsoft, licenciaram a Java. Várias já têm projetos para desenvolver aplicativos de grande porte, em um futuro próximo, usando Java. Já existe uma interface independente para acesso a bancos de dados suportada pelas maiores empresas do ramo.

3.2 O Paradigma Java

Embora o termo Java normalmente se refira à linguagem Java, o fenômeno Java está de fato baseado em quatro componentes inter-relacionados do qual a linguagem de programação é apenas uma parte. O arquivo de classes Java, o API Java, e a máquina virtual Java trabalham junto com o programa em linguagem Java para formar o ambiente Java completo.

⁴ *HotJava™ Browser* - HotJava é um browser de Rede World-Wide Web, e é uma das primeiras aplicações que foram escritas usando a plataforma Java®. O HotJava foi o primeiro browser a carregar e executar dinamicamente o código Java de maneira segura, em qualquer lugar na Internet.

3.2.1 O Programa Java

Um programa em Java é um programa em linguagem orientada para objetos, e os programas Java são construídos definindo-se hierarquias de classes e interfaces. Uma interface é como uma classe que pode encapsular somente constantes e métodos abstratos. A linguagem Java só suporta herança de classes simples, mas permite herança múltipla de interfaces.

A sintaxe da linguagem Java se assemelha à da C++, mas muitas das características mais complexas de C++ estão ausentes em Java. Por exemplo, não há nenhum ponteiro em Java. Isto impede que uma categoria inteira de erros surja, e torna a linguagem Java mais robusta. A Java é implementada para eliminar toda memória ociosa (*garbage-collected*). Assim, não tem necessidade de (e não pode) alocar memória, nem a memória pode ser liberada diretamente pelo programador. Exceções e *multithreading* são construídas dentro da linguagem. Ao contrario de C++, Java é uma linguagem completamente orientada para objetos. Tudo em Java é uma classe ou interface. No Apêndice E pode-se verificar algumas diferenças entre as linguagens C++ e Java.

Os programas Java são escritos em arquivos fonte com a extensão *.Java. Embora um arquivo Java possa conter qualquer número de classes e interfaces, geralmente é uma boa prática colocar cada classe e interface em seu próprio arquivo. Ao arquivo pode ser dado o mesmo nome da classe ou interface que a contém. Os arquivos fonte Java são uma sucessão de caracteres *Unicode* de 16 bits. Assim podem ser escritos programas Java em quase qualquer tipo de editor de linguagens. Devido à maioria dos editores gerarem caracteres ASCII de 8 bits, os compiladores Java podem converter o ASCII em *Unicode* durante a compilação.

3.2.2 O Arquivo de Classes Java

Um compilador Java recebe os arquivos fonte escritos pelo programador e verifica sua validade. As classes que passam na inspeção são traduzidas em um formato

binário. O formato binário é definido pela Sun Microsystems® e é chamado arquivo de classes Java. O Arquivo de Classes Java não é o único formato binário permitido no ambiente Java. Outros formatos binários podem ser criados seguindo certas regras de compatibilidade binária especificadas pela Sun Microsystems®. As regras de compatibilidade binária indicam as mudanças que podem ser feitas em uma classe ou interface que não forcem uma recompilação de outras classes ou interfaces que se referiram à classe mudada. Porém, o Arquivo de Classes Java é provavelmente o formato binário predominante para compilação Java.

Cada arquivo de Classes Java contém dados binários que expressam completamente uma classe ou interface de um programa Java. Os Arquivos de Classe Java levam o nome da classe ou interface que eles contêm e usam uma extensão **.class*. O compilador converte cada método de uma classe em um fluxo de instruções para a máquina virtual Java. As instruções são chamadas *bytecodes* Java. Os *bytecodes* para cada método são colocados no arquivo de classe junto com outras informações sobre a classe ou interface.

3.2.3 O API Java

Uma vez que um programa Java foi compilado em arquivos de classe, está pronto para ser executado. Os programas Java são executados dentro de um ambiente de execução Java, que é composto de uma máquina virtual Java e de um conjunto de bibliotecas de execução padrão. As bibliotecas de execução são os arquivos de classe que representam as classes e interfaces que formam o Java API. Essas classes e interfaces Java API tem a disponibilidade garantida em qualquer ambiente Java. As classes e interfaces garantem aos programadores um modo de acesso independente de plataforma quanto aos recursos de sistemas do computador hospedeiro. A máquina virtual e as bibliotecas de execução Java devem ser suportadas por uma plataforma que possa ser hospedeira de programas Java.

O API Java é composto de um conjunto de pacotes⁵ Java. O pacote `Java.lang` fornece serviços de importação básica para a operação da máquina virtual Java. `Java.io` oferece fluxos de entrada e saída. O pacote `Java.util` é uma coleção de classes que implementam estruturas de dados úteis. O `Java.net` provê serviços de rede. O pacote `Java.awt` fornece uma interface gráfica para os programas (*graphical user interface*), e o `Java.applet` possibilita a execução de um tipo especial de programa Java que é executado em um browser de rede, chamado *applet*. As principais bibliotecas que compõem a API de Java são listadas na tabela 3.1.

Tabela 3.1 - Bibliotecas que compõem a API de Java

Biblioteca	Principais Classes	Descrição
<code>java.applet</code>	<code>Applet</code> ,...	Classes para implementação de applets.
<code>java.awt</code>	<code>Component</code> , <code>MenuComponent</code> , <code>Window</code> , <code>Frame</code> , <code>Panel</code> , <code>MenuItem</code> , <code>Scrollbar</code> , <code>Button</code> , <code>TextArea</code> , <code>Dialog</code> ,...	<i>Abstract Window Toolkit</i> : Biblioteca de classes para gráficos, texto, janelas e construção de uma GUI independente de plataforma.
<code>java.awt.image</code>	<code>PixelGrabber</code> , <code>RGBImageFilter</code> , <code>CropImageFilter</code> ,...	Classes para processamento de imagens.
<code>java.awt.peer</code>	<code>WindowPeer</code> , <code>PanelPeer</code> , <code>MenuComponentPeer</code> ,...	Interfaces para uma GUI independente de plataforma.
<code>java.io</code>	<code>DataOutputStream</code> , <code>PrintStream</code> , <code>DataInputStream</code> , <code>ByteArrayOutputStream</code> ,...	Classes para todo tipo de operação de entrada e saída.
<code>java.lang</code>	<code>Object</code> , <code>Class</code> , <code>Integer</code> , <code>Long</code> , <code>Float</code> , <code>Double</code> , <code>Character</code> , <code>Math</code> , <code>Thread</code> ,...	Classes fundamentais da linguagem Java: objetos para encapsular tipos primitivos, classes com funções matemáticas.
<code>java.net</code>	<code>Socket</code> , <code>URL</code> , <code>InetAddress</code> , <code>DatagramPacket</code> ,...	Classes para operações em rede.
<code>java.util</code>	<code>Hashtable</code> , <code>Vector</code> , <code>Date</code> , <code>Enumeration</code> , <code>Random</code> ,...	Classes utilitárias.

3.2.4 A Máquina Virtual Java

A máquina virtual Java (ou JVM) executa programas Java. A JVM carrega as classes, incluindo as classes do programa e as classes da biblioteca de execução, por

⁵ Pacotes são grupos de classes e interfaces Java®.

intermédio de um carregador de classes. Este carregador verifica, de acordo com certos critérios que asseguram a execução dessas classes, que os *bytecodes* contidos dentro dessas classes, não violem a integridade da máquina virtual Java. Todos os programas Java são ligados dinamicamente⁶ durante o processo de carregamento. Classes e interfaces se referem um ao outro usando um qualificador completo que é único para toda classe ou interface. Um qualificador completo é o nome da classe ou interface do pacote, onde a identificação de seus elementos é separada através de pontos. Por exemplo, o nome da classe Objeto é " Java.lang.Object ".

A máquina virtual Java é um computador abstrato. A Sun Microsystems especifica certas características que ela deve ter, mas deixa muitas escolhas para a criatividade do projetista da JVM. A máquina virtual Java é frequentemente implementada em software. No entanto, também poderia ser implementada fisicamente em um *chip* de silício. A máquina virtual Java deve executar *bytecodes*, mas pode usar diversas técnicas. As formas de execução podem ter o *bytecode* interpretado, compilado por um compilador *just-in-time*, executado em código nativo no *chip*, ou executado por qualquer outra técnica disponível no repertório do projetista da JVM.

A operação da máquina virtual Java é entrelaçada com as classes dos pacotes de API Java, especialmente o pacote Java.lang. Por exemplo, a JVM cria uma instância da classe Java.lang.Class para cada classe carregada em tempo de execução. Os programadores podem obter qualquer informação da classe para qualquer objeto invocando o método *getClass()* para esse objeto. (*getClass()* é um método da classe Java.lang.Object do qual todos os objetos Java descendem. Todo objeto herda o método *getClass()*.) Os programadores podem alterar a maneira pela qual a JVM carrega as classes através de Java.lang.ClassLoader. A política de segurança da JVM pode ser padronizada pelo programador Java através da *Java.lang.SecurityManager*.

⁶ linked dynamically

3.2.5 A Rede

Embora não seja requerida uma rede de computadores para a execução de um programa Java, a tecnologia Java é normalmente associada com redes. A rede *www*⁷ lançou a Java por boas razões. A primeira é que as redes geralmente conectam computadores heterogêneos. Os arquivos de classe Java, cujo conteúdo é executável, são apropriados para serem distribuídos em redes, porque um programa Java não tem a necessidade de ser recompilado para a plataforma do computador hospedeiro. O programa é compilado uma vez, e os arquivos de classe resultantes podem ser carregados através da rede e podem ser executados em qualquer tipo de computador hospedeiro que suporta o ambiente Java. Os mecanismos de segurança do ambiente Java emitem avisos de que o código carregado inadvertidamente da rede pode danificar ou roubar dados, ou interferir com outros processos que são executados no computador cliente. O pacote *Java.net* do API Java fornece um conjunto de serviços de rede que estão disponíveis em todos os programas Java. A rede é o paradigma central do programa Java, e foi a chave do sucesso inicial da linguagem Java.

3.3 A Plataforma Java

Uma plataforma é o hardware ou ambiente de *software* no qual são executados os programas. Os aplicativos são executados em qualquer plataforma de forma independente da plataforma original. São executados *bytecodes* gerados em uma plataforma, que não são específicos de qualquer máquina física, mas sim instruções para uma máquina virtual. Um programa escrito na linguagem Java compila um arquivo de *bytecode* que pode ser executado onde quer que a Plataforma Java esteja presente, em qualquer sistema operacional. Em outras palavras, o mesmo arquivo pode se executado em qualquer sistema operacional onde possa ser executada a Plataforma Java. Esta portabilidade é possível porque o núcleo da Plataforma Java é uma Máquina Virtual Java.

⁷ World Wide Web

A plataforma Java difere da maioria das outras plataformas porque é uma plataforma de *software*, que é executada sobre outras plataformas baseadas em *hardware*. A maioria das outras plataformas é descrita como uma combinação de hardware e sistema operacional. A plataforma Java tem dois componentes: a máquina virtual Java (JVM) que será descrita em detalhes nos próximos itens, e a Aplicação de Interface de Programa Java (Java API).

A Java API é uma grande coleção de componentes de software já prontos que fornecem muitas facilidades, tais como a interface gráfica de usuário (GUI). A Java API se agrupa em bibliotecas (pacotes) de componentes relacionados. A figura 3.3 descreve um programa Java, com uma aplicação ou *applet* que está sendo executada na plataforma Java. Como mostra a figura, a Java API e a JVM separam a programação Java das dependências de hardware.

O *framework* API Java é aberto e extensível. As especificações para cada interface estão sendo desenvolvidas por especialistas em cada área. Deverão ser publicadas especificações e estas estarão abertas para a revisão. As implementações das especificações de API estarão disponíveis a partir do Java *Soft* e de outros fabricantes. No ambiente atual, de rápidas inovações, o *framework* API Java permite que as inovações existam como extensões à Plataforma Java.

A API é organizada em grupos ou em conjuntos. Cada um dos conjuntos API pode ser implementado como um ou mais pacotes (*namespaces*). Cada pacote se agrupa em um conjunto de classes e interfaces que definem um conjunto de campos relacionados, construtores e métodos.

3.3.1 A Plataforma Java Básica

A Plataforma Java Básica é uma plataforma mínima Java que os projetistas podem admitir que estará presente para executar as aplicações e os *applets* Java. Esta plataforma se aplica a computadores em rede, computadores de mesa (*desktops*), e

workstations. Ela contém a mesma Máquina Virtual Java antes mencionada, mas com um conjunto mínimo de API exigido para executar os *applets* e aplicações básicas. Este conjunto mínimo é conhecido como Java Core API, e também como Java Applet API ou Java API Básico. Os projetistas que usam este conjunto mínimo podem estar seguros que seus programas serão executados em qualquer local sem a necessidade de bibliotecas de classe adicionais.

Para manter a compatibilidade, a licença para o uso da Plataforma Java Básica exige a existência do Java Core API em qualquer implementação particular. Como mais bibliotecas de classe serão desenvolvidas, a Plataforma Java Básica crescerá, e estas adições migrarão para a Plataforma Java Básica que deverá estar presente na licença de uso.

Um outro conjunto de APIs, chamado *Extension Standard API* (extensão padrão API), vem sendo definido como *JavaSoft*, em sociedade com as principais companhias de software, para estender a funcionalidade do conjunto básico. Com o passar do tempo, os subconjuntos da *Extension Standard API* poderão migrar para a Plataforma Java Básica.

3.3.2 Visão interna da plataforma Java

A Plataforma Java tem duas partes principais, a Máquina Virtual Java e a Java API, como mostrado na Figura 3.3 e descritas em seguida. Nesta figura, a Plataforma Básica Java é a parte mostrada em negro, inclusive os blocos chamados Adaptadores. A Java API inclui o Núcleo de Java API e a Extensão Padrão Java API. As classes são a implementação destas APIs. A máquina virtual Java está no núcleo da plataforma. A Interface portátil reside entre a máquina virtual Java e o sistema operacional ou sobre o *browser*. Esta interface portátil tem uma parte independente de plataforma (mostrada em negro) e uma parte dependente de plataforma, mostrada como Adaptadores. O Sistema Operacional e JavaOS provêem a funcionalidade de rede. Assim, máquinas diferentes podem ser conectadas por uma rede, como mostrado.

Figura 3.3 - Esquema da plataforma básica Java em diversos sistemas operacionais

A Máquina Virtual Java - A máquina virtual Java é um computador "*soft*" que pode ser implementado em *software* ou *hardware*. É uma máquina abstrata projetada para ser implementada sobre os processadores existentes. As interfaces portáteis e os adaptadores permitem que sejam facilmente portados em sistemas operacionais novos sem ter que ser completamente reescritos.

Java API – A API Java forma uma interface padrão para os *applets* e aplicações, independentemente do sistema operacional. A Java API é um suporte essencial para o desenvolvimento das aplicações. A API especifica um conjunto de interfaces essenciais em um número crescente de áreas chaves que os desenvolvedores usarão para construir as suas aplicações em Java.

Java Core API - (Núcleo API) Provê a linguagem básica, utilitários, entrada e saída (I/O), rede, GUI, e serviços para *applet*. As companhias de Sistemas Operacionais que licenciaram a Java devem inclui-la em qualquer Plataforma Java que desenvolvam.

Java Padrão Extension API (Extensão Padrão API Java)- Estende as capacidades Java, além do núcleo Java API (*Java Core API*). Algumas destas extensões migrarão eventualmente para o núcleo Java API. Outras extensões, que não sejam padrão API, podem ser criadas por *applet*, aplicativos, ou pelo sistema operacional. Como cada extensão nova especificada de API é divulgada, ela estará disponível para revisão pelos usuários e, assim, terá uma realimentação antes de ser finalizada.

3.3.3 Visão da JVM

A máquina virtual Java é chave para a independência com relação ao sistema operacional, sendo uma plataforma que esconde o sistema operacional dos *applets* e das aplicações Java. É muito fácil transportar a máquina virtual para um browser ou para um outro sistema operacional.

Além disso, a máquina virtual define um formato independente de máquina para arquivos binários cujo formato de arquivo é chamado classe (*.class). Este arquivo contém instruções para um computador virtual na forma de *bytecodes*. A representação de um *bytecode*, em qualquer programa de linguagem Java, é simbólica. Isto significa que a indexação dos métodos não é constante, mas, ao invés disto, são dados nomes simbólicos em forma de cadeias de caracteres (*strings*). A primeira vez em que um método é chamado, o mesmo é procurado através de seu nome nos arquivos de classes, e o valor de acesso será determinado naquele momento. Assim, pode-se obter um acesso mais rápido nas chamadas subseqüentes. Então, quaisquer novos métodos, ou cancelamento dos métodos, podem ser feitos em qualquer hora na execução da estrutura de classe, referindo-se de forma simbólica, e ter acesso correto sem alterar o código.

Os *bytecodes* são uma representação em alto nível do programa, de forma que a otimização e a geração de código de máquina (por um compilador *just-in-time*⁸) pode ser executado em nível mais alto. Além disso, a remoção de valores não utilizados⁹ pode ocorrer dentro da máquina virtual, desde que as variáveis sejam mantidas em pilhas no espaço de endereçamento da Plataforma Java.

3.3.4 Plataformas para Java

Como já foi afirmado na introdução, a plataforma Java busca seguir a filosofia "escreva uma vez, execute em qualquer lugar". A primeira metade desta promessa é possibilitada pela linguagem de programação Java, que atende às necessidades de tarefas modernas de computação como *threading* e *garbage collect*. Os pacotes Java padrões são até mesmo parte do apoio de JDK (*Java Development Kit*) mas com classes de rede poderosas, interfaces de GUI ricas e classes úteis de IO.

A segunda metade da promessa é garantida pelo Ambiente de Aplicação Java, o componente que permite que um *applet* Java ou aplicativo seja executado em um sistema específico. Um ambiente de aplicação Java inclui uma máquina Java virtual e os pacotes de apoio e arquivos necessários. Eles são interligados no Ambiente de Aplicação Java para fornecer uma boa interação entre o programa em linguagem Java e o sistema hospedeiro.

Desde o princípio, a plataforma Java foi projetada para ser escalonável. Para alcançar esta meta, são necessários ambientes de aplicação para uma variedade de sistemas diferentes. A plataforma Java foi implementada primeiro para *browsers* de Internet, e então para Ambientes de Aplicação para sistemas como Windows 95/NT e UNIX. Agora, como um Ambiente de Aplicação, a Java está encontrando seu lugar em computadores de rede, dispositivos de usuário e sistemas integrados, e até mesmo em cartões inteligentes onde o ambiente Java usa uma quantidade minúscula de memória.

⁸ Ver no item 3.5- O Compilador JIT (*Just-in-time*)

⁹ *Collect Garbage*

3.3.4.1 Aplicativos em hospedeiros (*hosted*)

Hoje em dia os principais browsers de Internet suportam Java. Um *browser* é um exemplo de um ambiente hospedeiro de aplicações Java. Alguns o denominam uma aplicação "*container*"¹⁰ porque contém um ambiente de aplicação Java que executa um código Java. Alguns browsers têm um ambiente de aplicação Java embutido enquanto outros usam o ambiente de aplicação Java disponível no sistema operacional que o suporta. Outro exemplo de uma aplicação tipo *container* é um servidor de rede que suporta *servlets*¹¹ Java.

3.3.4.2 Sistema operacional hospedeiro

A maioria dos sistemas operacionais comuns oferece um ambiente de aplicação Java que pode executar aplicativos Java. Um papel importante de qualquer ambiente de aplicação Java é mapear os serviços dos sistemas e recursos para a parte independente de sistema do ambiente de aplicação Java. Por exemplo, se um sistema operacional tem um sistema de janelas nativo ou uma biblioteca com *threads*, o ambiente de aplicação Java pode usá-los de forma direta. O acesso fornecido para estes recursos sem afetar essencialmente a portabilidade do código Java é uma parte importante do desenvolvimento de um ambiente de aplicação Java.

O grau com que o ambiente de aplicação Java é integrado no sistema operacional, que lhe dá suporte, varia. No início, os esforços para incluir o Java em sistemas operacionais proporcionaram uma integração mínima. Isto provocou problemas com a instalação e a execução de aplicativos, e também com o acesso a recursos do sistema operacional. Mais recentemente o ambiente de aplicação Java para sistemas operacionais *desktop* garantiram uma integração melhor, de forma que os aplicativos

¹⁰ Recipiente

¹¹ *Servlets* são módulos que estendem as funções de requisição e resposta de ações, de servidores de rede Java. Por exemplo, um *servlet* poderia ser responsável por transmitir dados em um formulário de entrada HTML e aplicar a lógica usada para atualizar o banco de dados em uma empresa.

Java se comportem de forma mais similar aos aplicativos em código nativo. O ambiente de execução Java é um exemplo de um ambiente de aplicação Java hospedeiro¹².

3.3.4.3 O Sistema Operacional JavaOS™

O JavaOS é um sistema operacional centrado em Java, projetado com um conjunto de características e arquitetura para fornecer um ambiente Java completo para computadores ligados em rede. Os *network computers* diferem significativamente de computadores pessoais convencionais porque eles trabalham dentro de um ambiente cliente-servidor. A meta principal do modelo cliente-servidor é dividir tarefas, concentrando a interface do usuário no lado do cliente e a administração de dados no lado do servidor. O sistema JavaOS fornece os requisitos de um sistema operacional que são necessários para executar *applets* Java no sistema ambiente do cliente.

Por serem projetados para suportar apenas a parte da interface dos usuários de uma aplicação cliente-servidor, os computadores em rede oferecem a oportunidade de se dispor de dispositivos de computação baratos, com a administração do sistema simplificada para o cliente. Estas duas vantagens reduzem significativamente o custo total para usuários finais.

O JavaOS suporta apenas *applets* **100% Pure Java**¹³, que são executados por um aplicativo baseado em Java, como o *browser* HotJava. Os mesmos *applets* Java que são executados em um *browser* de rede em um PC podem ser executados em JavaOS em um computador em rede. Estes *applets* comunicam-se através de uma rede com aplicativos tais como sistemas de administração de banco de dados, que são executados em um servidor.

¹² . O ambiente de execução Java pode ser obtido no site <http://www.javasoft.com> para uso em sistemas Solaris ou Win32.

¹³ 100% Pure Java é um código fonte Java que é universalmente portátil e não explora funções de um sistema operacional específico.

A Sun Microsystems licenciou o uso do JavaOS para vários fabricantes que construirão computadores de rede baseados em JavaOS, utilizando várias UCPs e periféricos diferentes. Em versões futuras, os computadores em rede poderão usar os processadores Java.

3.3.4.4 JavaPC

O JavaPC é uma versão do sistema operacional Java OS voltada para ambientes do tipo PC. É um produto de software que transforma um PC em um computador de rede. Isto é útil para companhias com uma base instalada de PC que desejam migrar para uma arquitetura em rede baseada em JavaOS.

3.3.4.5 *PersonalJava e EmbeddedJava*

PersonalJava e EmbeddedJava são ambientes de aplicação Java projetados para serem executados em sistemas operacionais de tempo real. O PersonalJava é orientado para produtos centrados em comunicação, tais como dispositivos de *webtop* e *webphones*. O EmbeddedJava é projetado para suportar um grande volume de produtos embutidos que têm várias limitações de recursos. O PersonalJava e o EmbeddedJava demonstram a grande escalabilidade da Java, fornecendo subconjuntos do núcleo das APIs Java.

3.3.4.6 Cartão Java (Java Card)

O conceito Java Card (ou Cartão Java) traz a filosofia Java para cartões inteligentes. Estes dispositivos são embutidos em microprocessadores de tamanho minúsculo. Os cartões inteligentes fornecem um dispositivo seguro e programável para habilitar o comércio digital. O Cartão Java inclui uma pequena máquina virtual Java e um conjunto limitado de pacotes.

3.3.4.7 Processadores Java

O desempenho Java melhora dramaticamente em aplicações embutidas. O núcleo de UCP *picoJava™* é uma tecnologia licenciada que elimina a necessidade de um interpretador Java ou de um compilador JIT, executando os *bytecodes* Java diretamente no *chip* do processador. Isto simplifica o projeto de sistemas, melhora o desempenho, e reduz exigências de memória.

O núcleo do processador *picoJava™* é aperfeiçoado para o ambiente Java e suas necessidades, tais como *multithreading* e *garbage collect*. O processador *microJava™-701* da *Sun Microelectronics* é um protótipo de um processador *picoJava™* que possibilita que projetistas licenciados possam produzir os seus próprios microprocessadores baseados em Java. A avaliação do processador *picoJava II*, baseado no 701 estará disponível em breve para demonstração e projetos.

O núcleo do processador *picoJava™* compreende a parte central de um componente microprocessador ou controlador. Os processadores baseados no núcleo de *picoJava™* são desenvolvidos para consumidores de produtos eletrônicos que executam aplicativos em Java. A combinação do tamanho pequeno do núcleo do processador com o aumento de desempenho, pela execução direta de instruções de código Java, permitirão dispositivos de tamanhos menores, mas com desempenhos melhores, como telefones celulares, televisões e outros eletrodomésticos que podem ser desenvolvidos para criar novas opções de comércio.

Cinco grandes corporações já licenciaram o núcleo do processador *picoJava™* para uso em seus produtos em próximas gerações, representando um importante reforço no valor de processadores Java no mercado das principais indústrias. As licenças para *picoJava* foram obtidas pelas empresas Fujitsu Limited, NEC Corporation, IBM Corporation, Rockwell Collins, Inc., e LG Semicon Co Ltd.

3.4 O Compilador Java e Ambientes de Execução

Um ambiente de desenvolvimento para a linguagem Java inclui o compilador e os ambientes de execução, como mostrado na Figura 3.4. A plataforma Java é representada pelo ambiente de execução. O desenvolvedor escreve código fonte na linguagem Java (arquivos *.Java) e isto é compilado para *bytecodes* (arquivos *.class). Os *bytecodes* são instruções para a máquina virtual Java. Para criar um *applet*, primeiro o desenvolvedor armazena estes arquivos de *bytecode* em um servidor de HTTP, e coloca um comando `<applet code>` em uma página de *web* que dá nome a este arquivo de *bytecode*.

Quando um usuário final visita a página, o `<applet>` faz com que o arquivo de *bytecodes* seja transportado pela rede através do servidor para o *browser* do usuário final na plataforma Java. Neste ponto, os *bytecodes* são carregados na memória e então verificados em termos de segurança antes entrarem na máquina virtual.

Uma vez na máquina virtual, os *bytecodes* são interpretados pelo interpretador, ou opcionalmente, são transformados em código de máquina pelo compilador *just-in-time*. O interpretador e o compilador JIT operam no contexto de execução do sistema operacional (*threads*, memória, e outros recursos de sistemas). Qualquer classe das Bibliotecas de Classe Java (API) é dinamicamente carregada quando for requisitada pelo *applet*.

Figura 3.4- Compilação de código fonte gerando e executando *bytecodes*.

3.5 O Compilador JIT (*Just-in-time*)

Segundo [Chuc_98], a linguagem Java é conhecida como uma linguagem interpretada, porém, isto não é totalmente verdade. O código fonte Java, na prática é compilado em uma forma intermediária conhecida como *bytecode* Java ou código-J. Neste sentido, um compilador Java não é diferente dos demais compiladores tais como C ou C++. A diferença é que os compiladores que não são Java produzem tipicamente um código compilado em uma forma intermediária que é proprietária deste compilador, e que é então transformada no código objeto em uma compilação final. O código objeto mais tarde é linkeditado produzindo bibliotecas ou programas executáveis. A linguagem Java, por outro lado, define um código intermediário que é favorável à interpretação imediata, e este código por sua vez, é interpretado pela máquina virtual Java.

Vários fatores podem contribuir para um código ser amigável à interpretação, porém as áreas em que o Java é especificamente dirigido para a interpretação estão relacionadas ao formato de estrutura de arquivo, o formato classe de arquivo.

As classes de arquivos são projetadas com tamanhos e organizações de dados exatamente definidos. Esta especificação de precisões binárias e ordenamento de bytes assegura que qualquer máquina virtual que seja construída corretamente poderá ler e então executar o *J-code*. Posteriormente, as partes do arquivo serão todas indexadas por uma única estrutura de dados chamada de "*Constant pool*" no início do arquivo. Esta estrutura de dados é usada em métodos de referência, dentro da classe, por ponteiros que são preenchidos quando o arquivo de classes é carregado. Uma vez carregado, o conjunto de arquivos de classe que satisfaz o aplicativo Java corrente forma um gráfico de estruturas de classe interconectado com as suas "*constant pools*" contendo as referências relacionadas entre si (referências que contêm ponteiros para métodos em outras classes).

As classes de arquivos Java também podem formar um razoável ponto intermediário para outras linguagens. Duas linguagens, Ada95 e NetREXX, têm disponível um compilador que pode gerar arquivos *j-code* como saída.

Uma vez que os arquivos de classe estejam carregados e as referências nas "*constant pools*" prontas, algum método deve ser usado para colocar a máquina virtual Java em ação. Para aplicações é o método estático "*main*" na primeira classe que foi carregada, em "*applets*" é o método "*init*" na *applet* da classe principal. Em qualquer dos eventos, a máquina virtual inicia seu interpretador apontando o vetor do ambiente de execução para o primeiro código de byte do primeiro método e executando este *bytecode*. Se temos o código fonte do Java, a execução começa quando a máquina virtual chama a função *ExecuteJava* no módulo *interpreter.c*.

Então, para resumir o que acontece quando se executa um programa Java, o fluxo é algo como descrito a seguir:

Um aplicativo qualquer, que contém uma cópia da máquina virtual Java carrega um arquivo de classe inicial. No caso de um *browser* de rede, o JVM está em um objeto compartilhado ou DLL que é carregado com o *browser*, no caso do *Sun Java Developers Kit* (JDK), o JVM é parte do aplicativo. Quando carregado, o arquivo de classe é

esquadrinhado para referências a outras classes, que também são carregadas. Isto ocorre recursivamente, até que nenhuma referência carregada permaneça não resolvida. Então a aplicação inicia a classe chamando o interpretador em um método previamente especificado (*main()* ou *init()*). Durante a execução, novas referências às classes desconhecidas são resolvidas. A execução termina quando não existem mais usuários que se conectam ao sistema.

A introdução de uma nova capacidade no sistema é facilmente efetuada na fase em que a classe é carregada. Durante o carregamento da classe, o sistema tem *bytecodes* não processados e está no processo de criação de uma estrutura a partir da qual o interpretador executará esses *bytecodes*. É aí que é chamado o compilador JIT típico, quando a classe está para ser carregada, para substituir os códigos de byte por uma versão mais rápida desses códigos.

Agora que o código foi compilado, quais são as vantagens que a compilação JIT oferece? A resposta para esta pergunta tem duas partes. A primeira consiste em ter um compilador regular em que muitas otimizações são feitas no código intermediário e não no código fonte. A segunda é que a diferença entre um código que emula uma instrução e um código que a executa é significativa. Enquanto é possível fazer apenas otimizações, a maioria dos compiladores de JIT faz ambas as coisas, a otimização e a substituição do código indiscriminadamente.

3.6 Considerações Finais

Maiores detalhes do funcionamento do compilador Java e detalhes de seu ambiente de execução pode ser visto o Apêndice D. Neste apêndice são descritos maiores detalhes do compilador JIT, [Chu98]. São também descritos os conceitos de *Applets* e aplicações, e ainda uma descrição do Java Script. Apresenta-se também informações mais detalhadas sobre os *bytecode* e como funcionam os *threads* em linguagem Java.

Um conjunto de sessões descrevem as ferramentas e componentes Java tais como geradores de compiladores para Java, ambientes de desenvolvimento, montadores para

Java, descompiladores Java e alguns aspectos da Linguagem Java, como as diferenças entre Java e C/C++. O Apêndice E descreve aspectos de implementação da linguagem Java, ou seja a construção léxica de Java e a construção Sintática de Java.

Capítulo 4

Propostas JAVA em Arquiteturas paralelas

Resumo Neste capítulo serão apresentadas as propostas já existentes de exploração dos aspectos paralelos para a ferramenta Java. Uma abordagem geral levando em consideração os aspectos que podem ser paralelizados, tanto no ambiente, como na linguagem é apresentada. Assim, com intuito de apresentar o caminho a ser seguido no capítulo subsequente, estas propostas são analisadas criando o panorama onde serão propostas as soluções que foram implementadas neste trabalho e descritas no capítulo 5.

4.1 Introdução

Pudemos verificar no segundo capítulo as diferentes possibilidades de paralelização oferecidas em uma arquitetura paralela envolvendo linguagens tradicionais. De acordo com o que foi visto no referido capítulo, podemos explorar o paralelismo em diferentes níveis de granularidade, que variam desde a mais fina, que inclui comandos da linguagem, até a mais grossa, que é feita com a comunicação entre processos. Uma das propostas desse trabalho envolve a análise de granularidade mais fina, ou seja dentro dos comandos da linguagem.

No terceiro capítulo estudamos os principais aspectos da linguagem Java, seus comandos e sua filosofia. Ainda neste capítulo foram apresentadas as diversas ferramentas encontradas na literatura e disponíveis no mercado.

No presente capítulo será apresentada uma visão que busca somar às características do ambiente Java, aquelas que permitam explorar um maior paralelismo do ambiente em questão. Na sequência serão examinadas as possibilidades de paralelização que poderão ser exploradas, analisando as oportunidades oferecidas pela linguagem Java e, conseqüentemente, elaborando o conjunto de conceitos que fazem parte da proposta deste trabalho. A seguir serão relacionados alguns casos encontrados na literatura e de forma comparativa será apresentada a proposta do ambiente paralelo.

4.2 Possibilidades de Paralelismo em JAVA

Podemos dividir, de forma simplificada, a atuação de um compilador em seis fases: análise léxica, análise sintática, análise semântica, otimização global, geração de código e otimização local. Além disso, fazem parte do compilador a tabela de palavras reservadas que contém os caracteres especiais e palavras reservadas da linguagem, a tabela de símbolos, e as rotinas de tratamento de erros. A figura 4.1 mostra um diagrama com esse esquema.

Figura 4.1 Funcionamento básico de um compilador.

De forma simplificada, um ambiente Java pode ser claramente entendido como sendo o compilador da Figura 4.1, segmentado em duas partes. Com a ajuda do ambiente Java, através de um editor, escreve-se o programa fonte Java. A primeira parte seria então o compilador *javac*, que irá proceder a análise deste programa fonte e gerar os *bytecodes*.

Desta forma o código seria construído como é feito na geração de código intermediário relocável. A figura 4.2 representa este procedimento. A segunda parte seria aquela que executaria a geração de código para uma segunda máquina hospedeira, correspondendo à máquina virtual Java. Este conceito é representado na Figura 4.3

Figura 4.2 O ambiente Java simplificado – o javac

Figura 4.3 O ambiente Java simplificado – a máquina virtual Java.

Assim, pode-se dizer que a nova idéia associada aos compiladores Java é executar em duas fases o procedimento que, anteriormente, os compiladores executavam de forma

integrada. A separação do corpo do compilador em duas etapas aumenta as opções nas quais o código intermediário pode ser gerado em uma máquina qualquer, em seguida ser transmitido pela rede, e executado em uma outra máquina distante, com a geração de código sendo produzida num último instante para aquela máquina específica que recebeu o código pela rede. É importante destacar que o *bytecode* gerado deve ser especificado por um padrão universal, pois assim qualquer máquina que disponha de uma máquina virtual Java poderá receber esse código e transformá-lo em um código do seu conjunto de instruções de máquina específicos. Qualquer tentativa de paralelização, deve manter esse *bytecode* padrão sem alterações.

Os compiladores, que até o presente, sempre obtinham como entrada um código especificado em uma linguagem de alto nível, e convertiam esta entrada em uma linguagem de baixo nível, que correspondia à linguagem do computador hospedeiro, passaram a ser independentes de máquina. Podemos visualizar o conceito dos *bytecodes* da linguagem Java, de forma simplificada, se considerarmos que os *bytecodes* sejam um código intermediário gerado logo após a fase de análise. Na sequência, ao invés do compilador original continuar sua tradução até o nível da linguagem de máquina do hospedeiro inicial, o código gerado é transferido para outra máquina, através da transmissão em rede por exemplo. Uma vez que este código intermediário seja recebido por outra máquina, um programa especial (uma máquina virtual Java) irá terminar a conversão feita pelo compilador original, de forma a transformá-lo para a linguagem de máquina desta segunda máquina. Isso também poderia ser feito transmitindo esse código intermediário (*bytecode*) para um número qualquer de máquinas, desde que cada uma delas possuísse o programa apropriado, ou seja uma máquina virtual Java, que se encarregaria de fazer a decodificação final para cada uma das máquinas.

São muitas as possibilidades de se explorar paralelismo na linguagem Java. A possibilidade de paralelismo já explorada pela linguagem Java são os *threads*, que na verdade dependem do ambiente em que a linguagem está implementada. Isso não é um problema, visto que a maioria dos Sistemas Operacionais hoje existentes já dispõe deste recurso, por exemplo os sistemas *System 8 (Apple)*, *Win98 (PC)*, ou *Solaris (SUN)*. Com

base na Figura 4.4 faremos um resumo das possibilidades de exploração de paralelismo da linguagem Java.

Nesta figura vê-se primeiramente, assinalado com o número 1, a paralelização de *bytecodes*. Este processo, já explorado pelo projeto DARPA ([Bik97a], [Bik97b], [Bik97c]) será visto no item 4.2.1, que discute o paralelismo implícito existente em *loops* dos *bytecodes* procedentes de um compilador Java. O processo funciona como uma otimização de código intermediário de compiladores. Para compreendê-lo melhor pode-se observar a Figura 4.1, que descreve o funcionamento interno de um compilador. O bloco que representa a otimização pode ser dividido em outros dois: a otimização global e a otimização local. Na etapa da otimização global são efetuadas as manipulações do código intermediário, sendo independente de máquina. As operações mais comuns entre os compiladores são as otimizações de expressões, a eliminação de sub-expressões comuns, a fatoração do cálculo de sub-expressões de uso freqüente, as otimizações de construção iterativas. Na otimização local, a máquina na qual vai ser executado o programa influi na otimização a ser feita. É feita a otimização do uso de registradores, do uso do conjunto de instruções de máquina, do tempo de execução do programa. A geração de códigos é a fase intermediária entre a otimização global e a otimização local. Podemos ver esta situação na Figura 4.5.

Figura 4.4 Possibilidades de explorar paralelismo em Java

Figura 4.5 Detalhamento da fase de otimização de um compilador

O funcionamento da ferramenta *javab* é então uma otimização do código Java, sendo que esta ferramenta atua como um otimizador/paralelizador de código intermediário. Deve-se observar que o *bytecode* que é gerado deve ser compatível com o padrão definido para a máquina Java, mantendo assim a portabilidade deste código para qualquer máquina.

Uma segunda possibilidade de paralelização, assinalada pelo número 2 da figura 4.4, seria o uso de bibliotecas especiais paralelas para Java. Na literatura encontraram-se alguns esforços neste sentido, entre os quais podemos citar JPVM [Lea97] e JMPI [Fox97]. Este conceito explora a passagem de mensagens discutidos, e são implementados os conceitos de PVM e MPI para Java. Maiores detalhes destas aplicações serão apresentados no item 4.2.2.

Uma terceira possibilidade de exploração do paralelismo, assinalada pelo número 3 na figura 4.4, já pode ser encontrada naturalmente no ambiente Java e refere-se aos *multithreads*. Esta capacidade de paralelismo é, na verdade, fundamentada na concorrência de código, já que o verdadeiro paralelismo só poderia ser explorado no caso de um computador possuir mais de um processador. Desta forma os *multithreads* são gerenciados como processos num sistema operacional, onde a memória é compartilhada, e os processos devem, na verdade disputar regiões de conflito. O projeto *javar* [Bik97b] é um exemplo dessa possibilidade. Infelizmente trata-se de um protótipo em fase de construção, em fase ainda rudimentar, não sendo possível avaliar sua contribuição de forma mais profunda.

Uma quarta forma de se explorar o paralelismo, assinalado pelo número 4 da Figura 4.4, seria a exploração de uma arquitetura paralela, utilizando o conjunto de instruções de máquina desta arquitetura. Esta possibilidade motivou o início deste trabalho, apresentando-se um direcionamento para estudo de arquiteturas alternativas tais como as de fluxo de dados, onde a proposta fosse uma linguagem paralela para uma dessas arquiteturas. Logo no início dos trabalhos, a linguagem Java surgiu como uma forte promessa de linguagem para processamento em rede, e isso nos atraiu de forma positiva.

Finalmente chega-se a quinta opção, que também não teve nenhuma tentativa de exploração encontrada nas comunidades científicas e ou comerciais: **uma linguagem Java Paralela**. Esta é a proposta a ser desenvolvida neste trabalho. Pode-se observar na figura 4.4 um tracejado que destaca uma área do diagrama que compõe a proposta feita: a construção de um conversor automático de Java para **Japar**, e um compilador **Japar** que substitua o compilador *javac*.

Como foi dito a respeito de arquiteturas paralelas, não foram encontradas na literatura disponível, aplicações de linguagem Java paralela nessas arquiteturas especiais. Por isso a linguagem desenvolvida neste trabalho foi preparada para, num futuro, também poder ser explorada dentro de um ambiente paralelo. Neste caso, a proposta deste trabalho incluiu também uma máquina virtual Java que pudesse ser desenvolvida em quaisquer arquiteturas paralelas disponíveis. Em outras palavras, a máquina virtual Java desenvolvida poderá, com algum esforço, ser adaptada a qualquer conjunto de instruções de arquiteturas paralelas. Dessa forma poder-se-ia explorar todo paralelismo oferecido pela linguagem **Japar**. Este é um tema interessante para ser explorado em um futuro trabalho, ou seja, a adaptação de uma JVM para alguma arquitetura paralela, explorando assim todo o potencial deste tipo de arquitetura.

Nos itens seguintes serão apresentados alguns trabalhos encontrados na literatura, alguns na forma de projetos, outros ainda em estudo, mas que representam o estado da arte na pesquisa de ambientes paralelos para Java. Os detalhes da proposta deste trabalho serão vistos no item 4.3, que descreve a proposta para uma linguagem paralela.

4.2.1 Paralelização Automática de Loops do *Bytecode*

Uma ferramenta de pesquisa denominada **javab** [Bik97c], foi proposta para explorar o paralelismo implícito de *loops* em *bytecodes*, sendo uma ferramenta que transforma *bytecodes* sequenciais em paralelos. Esta ferramenta, escrita originalmente em linguagem C, suporta a detecção automática e exploração automática do paralelismo de *loop* implícito fazendo isso em *bytecodes*, usando assim as instruções de arquitetura independente de máquina da JVM. A seguir, na figura 4.6, é feita uma descrição mais detalhada da ferramenta.

Figura 4.6 Mecanismo de paralelização automática de *bytecodes* pela ferramenta **javab**

Como pode ser visto na figura 4.6 a ferramenta executa a transformação sobre os *bytecodes*, onde o paralelismo do *loop* implícito é transformado em explícito por meio de um mecanismo *multithreading* já existente na JVM [Ven98].[Lin97] Um arquivo binário *MyClass.class*, no formato de arquivo classe, é usado como entrada do **javab**. A seguir o compilador transforma o programa de entrada em uma forma que usa o mecanismo *multithreading* existente na JVM para transformar todo o paralelismo implícito em explícito. Como o próprio paralelismo é expresso em *bytecode* JVM, o programa transformado ainda pode ser executado por qualquer implementação da JVM. Na figura isso é feito pelo compilador Java denominado **javac**., ou ainda por um interpretador que esteja inserido no *browser* ou até mesmo um visualizador de *applets*. É importante observar que um aumento de velocidade (*speedup*) pode ser obtido em plataformas que suportam a execução paralela de *threads*. Todas as transformações podem ser feitas

independente do programa fonte e plataforma em que o *bytecode* foi gerado, e independente da plataforma em que o *bytecode* eventualmente será executado. Assim, esta abordagem permite a otimização de *bytecodes* que foram gerados pelo compilador *javac*, *bytecodes* que foram obtidos de uma rede de computadores, *bytecodes* gerados por compiladores que produzem *bytecodes* para outras linguagens de alto nível, ou *bytecodes* produzidos por um *assembler* JVM, como o Jasmin [Mey98]. Como os nomes de arquivos são essenciais em Java, o programa transformado é armazenado novamente no arquivo *MyClass.class*, depois que o programa original foi salvo em *MyClass.old*. Algumas classes auxiliares, suportes para a execução paralela dos *loops*, podem também ser geradas pela ferramenta de paralelização de *bytecode* *javab*. Desta forma a paralelização do *bytecode* é feita imediatamente antes da execução, ou seja *just-in-time*.

A ferramenta *javab* pode ser ativada como segue:

javab [- diretivas] [nome do arquivo a ser paralelizado]

Um número arbitrário de arquivos de classe podem ser fornecidos para a ferramenta (por ex. '*javab Class1.class Class2.class*' ou '*javab *.class*'), e as opções podem ser dadas em qualquer ordem (por ex., '*javab -op MyClass.class, 'javab -o -p MyClass.class*' ou '*javab -o MyClass.class -p*').

As diretivas podem ser as seguintes, para todos os arquivos do tipo classe (*class*) especificados:

- -b ***javab*** mostra informações sobre os blocos básicos no *bytecode* da JVM.
- -c: ***javab*** sumariza o conteúdo das constantes *pool*.
- -d: ***javab*** imprime as instruções do *bytecode* da JVM para cada método.
- -l: ***javab*** preserva semântica de manuseio exata em exceções de *linking*.
- -n: ***javab*** habilita a paralelização de *loops* isolados.
- -o: ***javab*** escreve o nome de cada arquivo de classe com extensão **.old*.
- -p: ***javab*** detecta automaticamente os *loops* paralelos no *bytecode* da JVM.
- -q: : ***javab*** habilita a interação com o usuário no caso em que informações interprocedurais são requeridas.
- -s: ***javab*** mostra a pilha de estados associados com cada instrução do *bytecode*.

- -v: **javab** imprime mensagens sobre as suas atividades.

Uma saída diferente de zero é retornada pelo **javab** se um erro for encontrado em qualquer um dos arquivos de classe especificados, e uma saída igual a zero caso contrário. Todos os erros, mensagens, solicitações do usuário, e saídas forçadas pelas opções ‘-bcdsv’ são escritas para o arquivo erro padrão.

Com a opção ‘-o’, o **javab** renomeia cada arquivo de classe *MyClass.class* para *MyClass.old* e gera a saída para este arquivo em *MyClass.class* novamente. Se o **javab** não puder renomear o arquivo *MyClass.class* como *MyClass.old* porque este último arquivo já existe, o **javab** assumirá que a conversão já foi feita. Neste caso, o programador deve primeiro renomear *MyClass.old* como *MyClass.class*, se o primeiro arquivo de fato contém o programa com o código *bytecode* original, ou, caso contrário, o programador deverá primeiro lidar com o arquivo conflitante *MyClass.old* de alguma outra forma. Se nenhum arquivo é especificado, então o **javab** lê um arquivo de classe a partir da entrada padrão e, se a opção ‘-o’ é também fornecida, escreve de volta o arquivo classe para a saída padrão, o que habilita o uso do **javab** em *pipes* Unix ou redirecionamentos entrada-saída (por ex., **javab -op <filein> fileout**). Claramente, a opção ‘-o’ só é útil quando combinada com a opção de paralelização ‘-p’.

Com apenas a opção ‘-p’, o **javab** irá tentar detectar o paralelismo de *loop* implícito sem realmente transformar os arquivos classe de entrada. Então, o **javab** examina o *bytecode* JVM de cada método em cada arquivo de classe para detectar e explorar automaticamente o paralelismo implícito nos *loops* naturais que são definidos pelos *back edges* nos diagramas de fluxo [Aho86] [Wol96]. O procedimento padrão adotado pelo **javab** é preservar o estado visível ao usuário depois de aplicadas as exceções [Yel96] tanto quanto possível, sem ser muito restritivo em relação às transformações. E então transformar somente as regiões de código para as quais o compilador pode provar que as exceções em tempo de execução (*run-time exceptions*- ver Apêndice E) não podem ocorrer, e a seguir permitir alterações semânticas com respeito ao manuseio de erros da JVM e exceções no momento da ligação (*linking*). A chave do compilador ‘-l’ pode ser usada para preservar a semântica exata das exceções manuseada no processo de ligação (*linking*).

Em alguns casos, a ferramenta pode detectar completa e automaticamente o paralelismo de *loop* implícito. Por exemplo, na Figura 4.7 mostra-se um pequeno fragmento em Java junto com o correspondente *bytecode* JVM, como representado por essa ferramenta. Primeiro, as *flags* de acesso do método, nome e descritor de método [Yel96], são mostradas ('0x08 loop1(I)[D]'). O próximo descritor $w \rightarrow w'$ (no exemplo $1 \rightarrow 1'$) denota o número correspondente de palavras que foram mudadas da pilha de operandos do processo que faz a chamada para os operandos locais do chamado, e colocadas de volta na pilha de operandos do processo que faz a chamada pelo chamado, respectivamente. Finalmente, é apresentado o *bytecode*, onde blocos básicos diferentes são separados por linhas horizontais. Para cada instrução, o *byte offset* é mostrado junto com o número de palavras que residem na pilha antes da execução desta instrução entre colchetes.

```

static double[] loop1(int n) {
    double[] a = new double[n];
    for (int i = 0; i < n; i++)
        a[i] = 1.0;
    return a;
}

*** method 0x08 loop1(I)[D] 1->1
    max_stack = 4
    max_locals = 3
    code_length = 23
-----
[ 0] 0: iload_0
*[ 1] 1: newarray double
-----
[ 1] 3: astore_1
[ 0] 4: iconst_0
[ 1] 5: istore_2
[ 0] 6: goto 16
-----
[ 0] 9: aload_1
[ 1] 10: iload_2
[ 2] 11: dconst_1
*[ 4] 12: dastore
-----
[ 0] 13: iinc 2 1
-----
[ 0] 16: iload_2
[ 1] 17: iload_0
[ 2] 18: if_icmplt 9
-----
[ 0] 21: aload_1
[ 1] 22: areturn
-----

```

Figura 4.7 Fragmento de Java com seu *bytecode* correspondente produzido por *javab*

Cada instrução que pode gerar uma exceção de tempo de execução ou de montagem (ignorando erros JVM) é marcada com um '*'. Para este *loop* particular, o *javab* pode determinar que a saída de *loop* anormal da instrução no endereço 12 não será executada, porque uma referência de arranjo *non-null* é usada, e o índice está sempre dentro dos limites do arranjo declarado. Além disto, o *javab* pode determinar que todas as iterações do

loop do *bytecode* correspondentes ao *i-loop* são completamente independentes. Aí o protótipo continua com a paralelização deste *loop*.

Se o método *loop1()* aparece em um arquivo de classe *MyClass.class* que define uma classe com o nome totalmente qualificado *package.MyClass*, então o **javab** constrói posteriormente um novo arquivo de classe *MyClass-loop1-x.class*, que define uma classe auxiliar *package.MyClass-loop1-x* (onde '-x' denota um número gerado pelo compilador), e coloca este novo arquivo de classe no mesmo diretório em que residia o arquivo de classe original. Desta forma, a classe auxiliar pertence ao mesmo *package* que a classe original. A classe auxiliar é uma sub-classe de '*java.lang.Thread*', e contém um *run()-method* que pode ser usado para que diferentes *threads* executem o *loop-body* do *loop* paralelo de forma simultânea. O número de *threads* *T* (onde $16 \geq T \geq 2$) que pode ser usado para execução paralela pode ser definido pela variável de ambiente '*JAVAB-THREADS*' (o valor *default* é 4). O *loop* original em *MyClass.class* é substituído por um construtor que inicializa estes *T threads*. Depois que todos os *loops* no arquivo original foram examinados, a classe (possivelmente) modificada é escrita de volta no arquivo *MyClass.class*, depois que uma cópia do programa original foi salva no arquivo *MyClass.old*.

Em outros casos, pode ser necessária a interação com o usuário. Por exemplo, na Figura 4.8, mostra-se outro pequeno fragmento de um programa Java juntamente com o *bytecode* JVM correspondente:

```

static void loop2(double a[], int n) {
    for (int i = 0; i <= n+30; i++)
        a[i] = 1.0;
}

*** method 0x08 loop2([DI)V 2->0
    max_stack  = 4
    max_locals  = 3
    code_length = 21
-----
[ 0] 0: iconst_0
[ 1] 1: istore_2
[ 0] 2: goto 12
-----
[ 0] 5: aload_0
[ 1] 6: iload_2
[ 2] 7: dconst_1
*[ 4] 8: dastore
-----
[ 0] 9: iinc 2 1
-----
[ 0] 12: iload_2
[ 1] 13: iload_1
[ 2] 14: bipush 30
[ 3] 16: iadd
[ 2] 17: if_icmple 5
-----
[ 0] 20: return
-----

```

Figura 4.8 Outro fragmento Java com *bytecode* correspondente produzido por *javab*

Aqui, apesar do *javab* poder determinar que as iterações no *bytecode* do *loop* correspondentes ao *i-loop* são independentes, sem uma análise interprocedural, o compilador não pode determinar se pode ocorrer uma saída anormal de *loop* devida a uma exceção em tempo de execução provocada pelo *bytecode dastore* no endereço 8. Nestes casos, ou são feitas hipóteses que assumem a pior situação, optando por manter o *loop* como serial (o *default*), ou, se a opção do compilador ‘-q’ foi fornecida, é iniciada a interação com o usuário para obter a análise interprocedural, como segue:

```

***MyClass.class::Queries for method 'loop([DI)V'
>Queries for the 1-dimensional array parameter '[D' in local 0
    *dim-1: is range [0,31+n) valid where n=parameter in local 1?
    (y/n/q) =>

```

As seguintes respostas podem ser usadas:

- ‘y’: confirma a solicitação
- ‘n’: nega a solicitação (o *loop* será mantido como serial)
- ‘q’: nega a solicitação e desabilita interação posterior com o usuário

Além disso, para impedir dependências de dados *loop-carried* não previstas, para cada parâmetro de *array* que está envolvido em uma instrução <t>*astore* dentro do *loop* paralelo candidato, o compilador também pergunta ao usuário se este parâmetro irá

sempre referenciar os (sub) *arrays* não sobrepostos na memória com respeito a todos os outros parâmetros de *array* que foram referenciados dentro do *loop*.

Considere-se, por exemplo, o seguinte método estático Java *query()*:

```
static void query (int [] [] a, int [] b, int x [], int k, int l) {  
    for (int i=0; i<=k+100; i++)  
        for (int j=0; j<l-25; j++)  
            b[i] +=a[i] [j] * x[j];  
}
```

Se a interação com o usuário foi habilitada pela *flag* '-q', então eventualmente as seguintes solicitações serão feitas ao programador:

```
***ManQuery.class::Queries for method 'query([I[I][III)V'  
    > Queries for the 2-dimensional array parameter '[I' in local 0  
        *dim-1: is range [0,101+n) valid where n=parameter in local 3?  
        *dim-2: is range [0,-25+n) valid where n=parameter in local 4?  
    > Queries for the 1-dimensional array parameter '[I' in local 2  
        *dim-1: is range [0,-25+n) valid where n=parameter in local 4?  
    > Queries for the 1-dimensional array parameter '[I' in local 1  
        *dim-1: is range [0,101+n) valid where n=parameter in local 3?  
Is storage of array parameters 0 '[I' and 1 '[I' non-overlapping?  
Is storage of array parameters 2 '[I' and 1 '[I' non-overlapping?
```

Percebe-se que recorrer à interação assim que a informações interprocedurais são solicitadas pode colocar uma carga pesada sobre o usuário, especialmente se o código fonte não está disponível. Contudo, esses problemas são aliviados pela incorporação da análise interprocedural em ambientes em que os programas de *bytecode* completos estão disponíveis.

Para uma apresentação mais detalhada sobre as técnicas utilizadas pelo protótipo da ferramenta de paralelização em *bytecode*, *javab*, ver Bika [Bik97a] e [Bik97b]. Observe que o *javab* é ainda um protótipo de ferramenta e não um produto comercial acabado. Os autores esperam que esta ferramenta de paralelização em *bytecode* forneça um ambiente com suficiente funcionalidade para permitir o rápido desenvolvimento de novos métodos automáticos de paralelização em *bytecode* para JVM. Os mesmos incentivam os usuários a modificar ou estender o *javab* para posterior experimentação com paralelização *bytecode*.

E com a opção '-v', o *javab* fornece informações sobre os *loops* particulares que não podem ser paralelizados, o que revela-se muito útil para pesquisas. Acreditamos que o uso desta opção possa fornecer idéias sobre os tipos de melhoramentos necessários para aumentar o número de *loops* em *bytecodes* JVM que possam ser paralelizados automaticamente.

4.2.3 Paralelismo em bibliotecas especiais

Uma outra possibilidade de se explorar o paralelismo é utilizar bibliotecas paralelas, tais como o MPI, MPL, PVM, descritas no Apêndice C. O uso de sistemas de rede de computação heterogênea interconectada como uma única fonte de computação lógica tornou-se uma abordagem difundida na computação de alto desempenho. Neste caso o paralelismo é explorado como uma interface com o sistema operacional, utilizando principalmente as passagens de parâmetros entre os diferentes processos dos programas. Essas bibliotecas paralelas implementam programas de forma paralela, através de um conjunto de bibliotecas específicas para computadores multiprocessados sendo que podem ser definidas para diversas plataformas, e podem oferecer inclusive memória distribuída compartilhada.

Uma biblioteca especial pode ser escrita para implementar os recursos de paralelismo que são extraídos na interação com o sistema operacional. Como exemplo deste tipo de implementação temos o PVM (Paralell Virtual Machine), que é um sistema de passagem de mensagens que interliga uma variedade heterogênea de computadores executando *softwares* como se estes estivessem em um computador distribuído de grande memória. Desta forma, o PVM oferece a capacidade de resolver grandes problemas computacionais usando memória agregada e o poder de computação de muitos computadores. Um programa PVM é implementado como uma coleção de tarefas cooperativas, ou unidades paralelas, com acesso a recursos de uma máquina virtual paralela, através de uma biblioteca de rotinas de interface padrão (PVM). Estas rotinas de interface permitem a criação e término de tarefas por agrupamentos heterogêneos de computadores, como também comunicação e sincronização entre tarefas. As bibliotecas de

PVM, inclusive aquelas usadas para estas experiências, são escritas em C e devem ser compiladas para cada arquitetura de máquina.

A linguagem Java possui algumas características que são ferramentas promissoras para a resolução de alguns dos problemas inerentes associados à programação em redes paralelas. Por exemplo, de uma perspectiva dos aplicativos, Java fornece uma interface portátil e uniforme para *threads*. O uso de *threads* ao invés dos processos tradicionais mostrou um grande potencial para o aumento da tolerância à latência e possibilidade de computações de granularidade fina, e alcança um bom desempenho em ambientes de processamento paralelo de memória distribuída [Ada98]. Da perspectiva da implementação de sistemas, Java suporta um alto grau de portabilidade do código e uma API uniforme para serviços de operação do sistema tais como comunicações em rede.

Vários sistemas foram desenvolvidos para suportar programação de redes paralelas em Java. Uma abordagem comum para este problema é o uso de sistemas baseados em “voluntários” tais como Bayanihan [Sar97] e Javelin [Cap97]. Estes sistemas são baseados na decomposição de programas em *applets* que são *downloaded* por clientes voluntários, executam no contexto de um *browser* de rede, e resolvem partes de um problema maior. Estas abordagens são atrativas na medida em que elas podem empregar potencialmente o vasto arranjo de potência de processamento disponível na forma de máquinas cliente conectadas à rede. A dificuldade primária destas abordagens é a restrição significativa inserida nas comunicações pelo modelo de segurança Java para *applets*, que potencialmente elimina aplicativos eficientes para redes paralelas. De forma oposta a essas implementações tem-se o JPVM, que é baseado em aplicações Java *stand-alone*. Na literatura foi possível encontrar algumas destas implementações que buscam explorar as capacidades da linguagem Java através de bibliotecas especiais. A seguir será descrito de forma resumida o sistema JPVM, descrevendo seus modelos de programação e interface, de sua interface interativa, e sua implementação

4.2.3.1 JVMP

A Máquina Virtual Paralela Java (JPVM) busca explorar a computação de Redes paralelas em Java. Sistemas de rede de computação paralela permitem que aplicativos

individuais controlem conjuntos de recursos disponíveis para muitos usuários, cada vez mais potentes, bem conectados, heterogêneos e freqüentemente sub-utilizados [And95] [Ada98]. Os resultados obtidos com sistemas de redes de computação paralela são satisfatórios [And98], mas a utilização de recursos compartilhados, heterogêneos, distribuídos, conectados por redes comerciais como um único computador paralelo apresenta sérios problemas tanto do ponto de vista do aplicativo quanto dos programadores que desenvolvem softwares de sistemas. Por exemplo, da perspectiva do aplicativo, verificou-se que sucessivos programas de rede paralela exibirão quase sempre granularidade média e grossa e serão tolerantes com a latência da rede, mas estes atributos podem ser difíceis de alcançar em alguns aplicativos. Da perspectiva de implementação do sistema, a heterogeneidade resulta em problemas difíceis, tais como as tarefas de combinação da plataforma e portabilidade do sistema.

A biblioteca JPVM [Ada98] é um sistema de *software* para transmissão explícita de mensagens baseado na programação paralela Java de memória distribuída MIMD. A biblioteca suporta uma interface similar às interfaces C e Fortran fornecidas pelo sistema PVM (*Parallel Virtual Machine*) [Gei94], mas com sintaxe e semântica fornecidas pela linguagem Java e mais adequadas aos estilos de programação desta linguagem. A similaridade entre a JPVM e o sistema PVM permite uma curva de rápida aprendizagem para programadores com experiência em PVM, tornando o sistema JPVM acessível e de baixo investimento para a migração de aplicativos paralelos para o sistema Java. Ao mesmo tempo, o JPVM oferece características novas, não encontradas no padrão PVM, tais como *threads* seguros, múltiplas comunicações em terminais por tarefa, e roteamento de mensagens direto no caso *default*. O JPVM é implementado em Java, sendo portanto portátil entre plataformas que suportam alguma versão da máquina virtual Java. Esta característica abre a possibilidade de utilização de recursos geralmente excluídos dos sistemas de rede de computação paralela, tais como as *workstations* baseadas em Macintosh e Windows.

A interface de programação fornecida pelo sistema JPVM é similar àquela suportada pelo PVM, com o acréscimo de melhorias para explorar os benefícios potenciais da Java como linguagem para a programação de sistemas de rede paralelos. Como em

PVM, o programador decompõe o problema a ser resolvido em um conjunto de implementações de tarefas sequenciais complementares. Estas tarefas sequenciais são executadas em um conjunto de processadores disponíveis e chamam rotinas de bibliotecas especiais para controlar a criação de tarefas adicionais e transmitir mensagens entre as tarefas. Em JPVM, as implementações das tarefas são codificadas em Java, como aplicativos com um método público estático *main()*, e o suporte para a criação de tarefas e transmissão de mensagens é fornecida por uma biblioteca de classes JPVM.

A interface na qual ocorre a maior parte da interação JPVM é exportada por uma classe denominada *jpvmEnvironment*, que são instâncias declaradas pelas tarefas JPVM para conectar e interagir com o sistema JPVM e outras tarefas. Objetos desta classe representam comunicações terminais dentro do sistema, e são identificados pelos identificadores característicos de sistema do tipo *jpvmTaskId*, que é análogo ao identificador de tarefas PVM. A JPVM permite que as tarefas mantenham um número ilimitado de conexões de comunicação, simplesmente alocando instâncias múltiplas de *jpvmEnvironment*. Esta característica simplifica o processo de desenvolvimento de módulos separados que podem ser ligados e que precisam executar as comunicações.

A tarefa de criação em JPVM é suportada pelo método *pvm_spawn()*, que toma um parâmetro *string* indicando o nome de uma classe Java visível na variável de ambiente *CLASSPATH*, bem como o número de tarefas para serem disparadas. Cada tarefa criada em *pvm_spawn()* é executada em sua própria instância da máquina virtual Java, evitando problemas como o uso conflitante dos serviços do sistema entre as tarefas. Estas instâncias criadas da máquina virtual Java são colocadas no conjunto de processadores disponíveis para o JPVM, e cada um executa um objeto da classe Java especificada.

A transferência de mensagens em JPVM é executada usando os métodos *pvm_send()* e *pvm_recv()* da classe *jpvmEnvironment*. Antes dos dados serem enviados, eles são reunidos em um objeto *jpvm_Buffer*. Análogo aos *buffers* PVM, os objetos *jpvm_Buffer* são os reservatórios de mensagens do JPVM. A interface *jpvm_Buffer* contém dois grupos básicos de métodos: aqueles que colocam os dados em um *buffer* e aqueles que extraem os dados do *buffer*. Onde possível, é usado um carregamento direto na interface

para simplificar a aplicação do código. Operações de empacotamento e desempacotamento escalares e vetoriais são fornecidas para todos os tipos Java básicos, assim como objetos *String* e *jpvmTaskId*. A interface *buffer* JPVM difere do padrão PVM em seu uso explícito de objetos do *buffer* (quando comparado ao uso implícito de *buffers* de envio e recebimento no PVM). Este *design* é um exemplo das alterações do JPVM em relação ao padrão PVM para melhor suportar a programação *multithreaded*.

Depois que o conteúdo de uma mensagem foi organizado dentro de um objeto *jpvm_Buffer*, o *buffer* pode ser enviado para qualquer outra tarefa no sistema JPVM usando o método assíncrono *pvm_recv()* da classe *jpvmEnvironment*. Parâmetros para *pvm-send()* especificam o *buffer* a ser enviado, a identidade da tarefa para a qual a mensagem deve ser enviada, e um número inteiro de identificação para a mensagem, chamado *tag* da mensagem.

Para receber mensagens, as tarefas chamam o método *pvm_recv()* da classe *jpvmEnvironment*. Usando as várias versões de *pvm_recv*, as tarefas podem requisitar mensagens baseadas na identidade do emissor da mensagem, no número de identificação (*tag*) da mensagem, em ambos, ou em nenhum deles, isto é, recebe qualquer mensagem. Operações de recebimento ficam bloqueadas até que uma mensagem do tipo requisitado esteja disponível, neste ponto uma estrutura *jpvmMessage* contendo a mensagem recebida é retornada. Além de conter um *jpvmBuffer* com o conteúdo das mensagens, um objeto recebido *jpvmMessage* contém a identidade do emissor da mensagem, bem como o número do identificador da mensagem.

Apesar dos ambientes de execução atuais para Java limitarem significativamente o desempenho, o uso de Java como uma linguagem de implementação de sistemas fornece algumas vantagens, entre as quais está a habilidade para empregar recursos tais como *threads* e *sockets* em uma interface portátil para um grande número de plataformas. A implementação de transmissão de mensagens JPVM é baseada na comunicação em sockets TCP. Observou-se que o uso de conexões TCP diretas, tarefa por tarefa, melhora o desempenho da antiga implementação PVM de transmissão de mensagens (*UDP-based daemon routed*). Cada instância *jpvmEnvironment* cria um *socket* servidor durante a

inicialização e fixa o nome do hospedeiro e o número da porta necessários para conectá-lo dentro da identidade *jpvmTaskId*. Internamente, o JPVM usa *threads* para administrar conexões e envio de mensagens. Cada *jpvmEnvironment* cria um *thread* dedicado para verificar e aceitar conexões *peer*, como mostrado na figura 4.9. Quando uma conexão é aceita por este *thread* dedicado de administração de conexões, ele cria um novo receptor de mensagens dedicado à nova conexão. Quando o *thread* receptor recebe mensagens, ele os enfileira em uma fila de mensagens internas (Figura 4.9). Então, quando o *thread* usuário executa operações *pvm_recv()*, ao invés de executar diretamente qualquer entrada de rede, ele simplesmente consulta a fila de mensagens interna. Um atributo importante da implementação de comunicações JPVM é a segurança dos *threads*. Como visto na figura 4.9, *threads* de múltiplos usuários podem executar simultaneamente operações *pvm_recv()* – a estrutura de filas de mensagens internas é um tipo que pode ser manipulado com segurança por qualquer número de *threads* leitores (usuário) e escritores (receptores internos). As operações *pvm_send()* são também sincronizadas para suportar *threads* com segurança.

Figura 4.9 Implementação da comunicação de JPVM

A habilidade de disparar novas tarefas, para determinar e estender a configuração do sistema, e para determinar quais tarefas estão sendo executadas pelo sistema é baseada em serviços fornecidos por um conjunto de processos *daemon* JPVM executados em cada

sistema hospedeiro. A implementação *daemon* está baseada no mecanismo de comunicação padrão JPVM, isto é, a implementação simplesmente engloba um programa JPVM normal usando a interface descrita anteriormente. O *daemon* utiliza uma organização de estilo servidor, repetidamente bloqueando e atendendo novas solicitações. Para a maior parte das solicitações do cliente (por ex., tarefas de status, informações sobre a configuração do sistema), o *daemon* pode responder imediatamente. Contudo, para solicitações de criação de tarefas, o *daemon* cria um *thread* para começar o novo processo local, evitando resposta lenta para outras solicitações do cliente enquanto as tarefas estão sendo criadas.

Para avaliar o desempenho da implementação JPVM, os autores [Fox97] realizaram um conjunto de experimentos para quantificar os custos das bibliotecas básicas primitivas e para medir o desempenho real do aplicativo. Foram usados para estes experimentos um conjunto de 5 hospedeiros com processador dual Pentium Pro conectados por *ethernet* de 10 *megabits*. Cada hospedeiro no sistema continha 2 processadores Pentium Pro de 200 MHz e 128 MB de memória, e estava executando o Linux 2.0. A versão da Java usada foi a Linux JDK 1.1.3. Desde que cada hospedeiro é uma máquina com processador dual, a diferença em tempo para criar um ou mais hospedeiros é desprezível. O custo da tarefa de criação não é pequeno, quando é criada mais de uma tarefa, são requeridos aproximadamente 0.5 segundo por tarefa no hospedeiro único e entre 0.5 e 1.5 segundos por tarefa no sistema multi-hospedeiros. O disparo é um pouco menos dispendioso para números de tarefas pequenos no sistema de um único hospedeiro porque a comunicação requerida é menor. Para um maior número de tarefas, o sistema multi-hospedeiro começa a apresentar melhor desempenho, já que pode amortizar custos de comunicações, paralelizando a criação de um conjunto de tarefas.

Um dos testes realizados pelos autores mostram os tempos consumidos em mensagens *round-trip* entre duas tarefas para vários tamanhos de mensagens. Estas medidas foram executadas com mensagens contendo inteiros e *bytes raw* para examinar o *overhead* associado com a necessidade de mascarar as diferenças de *bytes ordering*. Como seria esperado, os *overheads* de transmissão de mensagens introduzidos pelo sistema Java JPVM

implementado são muito altos, uma ordem de grandeza de latência é acrescentada, e somente cerca de 1/8 da largura de banda de comunicação disponível é utilizada.

Devido aos altos custos associados com os primitivos básicos JPVM, os autores concluíram que o *overhead* introduzido pela estratégia de usar um sistema implementado em Java como suporte em tempo de execução para aplicativos paralelos Java superaram os eventuais benefícios. Concluíram também que, os altos custos destas operações essencialmente impediriam o uso do JPVM como uma plataforma para aplicativos de rede de uso intensivo. Contudo, para aplicativos que exibem granularidade média para grossa, o JPVM pode fornecer serviços suficientemente eficientes para permitir um bom aumento de velocidade. Para verificar esta afirmativa, mediram o desempenho de um algoritmo paralelo de multiplicação de matrizes implementado em JPVM e compararam com uma versão seqüencial do programa também implementado em Java. Os autores mostram os resultados destes experimentos para 3 tamanhos de problemas. Os tempos são apresentados para ambos, o programa completo, incluindo o tempo de criação, e para o algoritmo de multiplicação sozinho (excluindo o tempo de criação). Para o menor tamanho de problema, foi obtido um aumento de velocidade pequeno, e foi observado um retardamento para o programa como um todo. Este tamanho de problema tem a granularidade muito fina dados os custos básicos associados ao JPVM. No problema de tamanho médio, os autores observaram que o algoritmo de multiplicação começa com um bom aumento de velocidade, mas o programa como um todo tem um desempenho pobre devido ao alto custo da tarefa de criação. Finalmente, para o maior tamanho de problema, tanto o algoritmo de multiplicação quanto o programa como um todo apresentaram um bom aumento de velocidade. Como esperado, maiores aumentos de velocidade foram obtidos quando os custos associados com as tarefas de criação são excluídos. Apesar desta parecer uma comparação pouco razoável, ela demonstra que para programas JPVM com tempos de execução longos, os custos das tarefas de criação podem ser amortizados, sendo possível obter boa eficiência para as fases de computação/comunicação do programa.

4.2.3.2 Outros exemplos de bibliotecas especiais

Um sistema similar ao JPVM em sua interface de programação e modelo é a biblioteca JavaPVM [Thu97]. Este sistema também fornece uma interface como a PVM para aplicativos Java. A diferença principal entre a JavaPVM e a JPVM está na implementação. Ao contrário de JPVM, que é implementada inteiramente em Java, a JavaPVM é baseada nos mecanismos *Java Native Methods*, fornecendo envoltórios (*wrappers*) de métodos nativos ao redor de rotinas padrão PVM existentes. Embora esta abordagem ofereça vantagens em termos de desempenho (ao menos a curto prazo), e interoperabilidade, pelo uso de métodos nativos, o sistema é restrito em sua portabilidade. De fato, o JavaPVM pode ser usado apenas em plataformas onde o padrão PVM esteja disponível. Além disto, sendo um simples envoltório ao redor de uma implementação com padrão PVM, o JavaPVM é limitado por características sintáticas e semânticas do PVM.

Outro pacote de *software* para computação em rede paralela em Java é o sistema IceT [Gra97]. Este sistema é inerentemente colaborativo, com ambiente multi-usuário. O IceT fornece mecanismos pelos quais recursos podem ser disponibilizados para o sistema para uso por usuários que não tenham privilégios de *log-on*. Os atrativos desta abordagem baseada em meta-computação incluem melhor utilização de bases de recursos maiores, bem como um ambiente colaborativo melhorado para computação de alta desempenho. Entre os desafios introduzidos por esta idéia estão as numerosas questões de segurança introduzidas pelas capacidades de *code-upload* para usos sem privilégios.

A combinação da linguagem Java como uma linguagem de programação de aplicativos e linguagens de implementação de sistemas permite o suporte de várias características poderosas em JPVM, incluindo a segurança de *threads* e comunicações terminais múltiplas por processo. Os experimentos iniciais de análise de desempenho indicam que os custos associados com os primitivos básicos JPVM são bastante altos, mas isto é esperado dado o uso de Java para a implementação do sistema. Contudo, os autores verificaram que o sistema pode suportar um bom aumento de velocidade para aplicativos que exibam um nível de granularidade apropriadamente grossa, tolerância à latência de rede, e habilidade para amortizar custos associados à tarefas de criação.

Capítulo 5

Proposta e Implementação do Ambiente Paralelo

Resumo Neste capítulo, com base nas possibilidades de paralelismo apresentadas no capítulo anterior, é apresentada uma proposta de criação de um ambiente paralelo para Java. A seguir serão apresentadas as implementações que foram realizadas. É feito então, com o uso do ambiente **Japar** construído, explorando-se as características paralelas deste ambiente e considerando-se o estudo realizado anteriormente, uma série de verificações testes e avaliações desta proposta.

5.1 Introdução

Considerando-se a representação descrita na Figura 4.4, que explora as possibilidades de paralelismo em Java, apresenta-se agora a proposta do ambiente **Japar**. A proposta visa oferecer uma linguagem paralela, com características Java, que permita ao usuário tanto explorar diretamente os recursos paralelos da linguagem, quanto explorar, se assim lhe convier, os recursos de otimização/paralelização automática oferecidos pelo conversor **ConJava**.

Algumas observações importantes devem ser feitas a respeito do desenvolvimento e da implementação do presente trabalho. Inicialmente deve-se dizer que este trabalho buscou, dentro do possível, a maior compatibilidade em relação às especificações originais

publicadas pela SUN MICROSYSTEMS®. Isso foi possível devido a essa empresa permitir o acesso a algumas rotinas de desenvolvimento do código fonte do ambiente Java. Este código é disponibilizado pela SUN® a desenvolvedores de sistemas, sendo vetado qualquer tipo de uso comercial. Por isso pode-se utilizar todos os protótipos de classes, especificações de variáveis, detalhamentos de arquivos *header*, e demais definições originais. Muitas das rotinas auxiliares desenvolvidas pela SUN® também puderam ser agregadas à implementação dos arquivos que fazem parte do objetivo deste trabalho. Isso permitiu que grande parte do ambiente Java pudesse ser construído. Devido à complexidade do trabalho, alguns aspectos foram priorizados, e outros considerados secundários em relação ao objetivo da proposta deste trabalho. Ainda assim, em implementações futuras, um extenso desenvolvimento deve ser executado, aliando diversos outros trabalhos de mesmo porte para cumprir a totalidade da proposta. Neste sentido o projeto pôde alcançar um razoável nível de implementação, onde uma importante aplicação foi explorar os aspectos de paralelismo até então não observados.

5.2 Proposta

A seguir serão descritos quais são os itens que compõe a proposta realizada neste trabalho, que serviu de base para a implementação realizada.

5.2.1 Características do ambiente paralelo proposto

O conjunto de ferramentas implementado compõe-se dos seguintes módulos:

- ✓ Um conversor (filtro) da linguagem Java para a linguagem **Japar**, denominado **ConJava**,
- ✓ O **Japar**, que é um compilador completo da linguagem Java, e que pode substituir o compilador *javac*. Este módulo inclui também as especificações BNF para a linguagem **Japar**, que podem ser vistas no Apêndice B,
- ✓ Um conjunto de classes que implementa o paralelismo da linguagem **Japar**, cuidando principalmente dos conceitos de passagem de mensagens. Entre essas classes encontram-se o arquivo de classes **Japar.par.para**. e as rotinas de

biblioteca para explorar os recursos de paralelismo, entre elas o conjunto de classes (pacote) **Japar.par**

- ✓ O carregador de classes para a máquina virtual Java, **Japar.par.ClassLoaderPar()**
- ✓ Uma máquina virtual simplificada (porém completa) para futuras explorações do paralelismo oferecido pela linguagem **Japar**,
- ✓ Um conjunto de rotinas de teste.

Os principais módulos, o conversor, a linguagem e a máquina virtual serão descritos de forma geral, dando suporte a essas três principais implementações que permitem a criação deste ambiente, ou seja: a linguagem **Japar**, o conversor Java para a linguagem **Japar** (**ConJava**) e a máquina virtual com capacidade de trabalhar com a linguagem **Japar**, denominada **MVJP**.

Um programa **Japar** é definido com procedimentos locais e procedimentos globais. Para seguir uma lógica da tradição em linguagens de programação, os comandos serão mantidos em língua inglesa, a qual se convencionou como padrão. A linguagem prevê o uso desde comandos puramente paralelos de uma linguagem distribuída até comandos que tratam de primitivas de passagens de mensagens e parâmetros, que usem mecanismos para uma programação que utilize memória compartilhada.

As palavras-chave que são adicionadas à linguagem **Japar**, e que tem sua estrutura em forma de BNF no Apêndice B, são as seguintes:

Async	Endwhile	Parfor	Select
Body	Fork	Parwhile	Spec
Create	Join	Priority	Terminate
Delay	Parbegin	Process	Transaction
Enddowhile	ParDowhile	Processor	Within
Endfor	Parend	Resource	

A seguir essas palavras reservadas são descritas com maiores detalhes. Um programa **Japar** usa uma estrutura geral que prevê definições de variáveis locais e globais, e também procedimentos locais e globais. Uma estrutura geral, que envolva as rotinas locais e o corpo principal da linguagem, é então descrita como uma sequência de comandos onde cada componente possui um local específico para ser declarado, seguindo a estrutura a seguir:

Resource nome

Especificações externas (importadas)

Operações e declarações de tipos

body nome (formais)

variáveis e outras declarações locais

inicialização de código

“procedures” e processos (inicializações e usos de classes e objetos)

finalização de código

end nome

Diversos comandos, declarações e um pacote foram acrescentados à linguagem Java. A seguir esses conceitos são apresentados de forma simplificada.

a) Deve-se considerar que o paralelismo envolve declarações, rotinas (procedimentos ou funções), e processos.

a.1) **Declarações** – Entre as declarações consideramos os comandos *for*, *while* e *do-while*. Como exemplo, nas declarações apresentamos o comando *for*:

parfor (múltiplas iterações de um *loop* em paralelo),

parfor i=1 to 10

declarações

parendfor

Entre as declarações, o usuário pode desejar explicitar também que uma sequência seja forçada a ser paralelizada. Desta forma ele pode explicitar o início e o final, definindo assim um bloco a ser paralelizado.

parbegin (lista de declarações em paralelo)

<i>parbegin</i>
declaração1
declaração2
declaração3
<i>parend</i>

a.2) **Rotinas e procedimentos** – Com o objetivo de definir o início e o fim de procedimentos (objetos), estas declarações tem apenas a função de marcar o início e o fim, definindo assim o escopo de seqüências, facilitando para o compilador a construção e manipulação de sua pilha interna de blocos locais ou globais.

Resource, Body

a.3) **Processos** – Neste caso, têm-se comandos que dividem os processos e criam automaticamente novos processos, que podem ser executados paralelamente, e ao término deles re-sincronizados. Esses comandos são importantes no caso de criação de *threads*, e otimizam a execução de processos em ambientes onde existem processos concorrentes. Esses comandos devem ser explicitados pelo usuário na chamada de um processo, que deve utilizar:

fork, join

b) Em termos de comunicação por passagem de mensagens, a linguagem **Japar** prevê a existência de comandos, que devem ser explorados apenas em versões futuras do ambiente **Japar**,

Sendmesageto(X) , receivemessagefrom(Y),

Lock(), unlock()

Estes elementos são explorados em um pacote (*package*) que é carregado em um conjunto de classes paralelas (ver 5.3.3 e 5.3.4)

c) Em termos de elementos paralelos definidos para transações entre processos, tipos de processos, especificações de processadores, criação de processos, especificação de prioridades, tem-se:

Create, Transaction, Async, Body, Create, Delay, Priority, Process, Processor, Select, Spec, Terminate, Transaction

d) Pacote – Para o completo uso dos conceitos acrescidos na linguagem **Japar**, o usuário, além dos comandos relacionados que podem ser usados, deverá incluir um pacote (*package*) especial de classes denominado `japar.par`, que contém as classes que implementam especificações paralelas da linguagem **Japar**. Este pacote deve ser importado através do comando:

*import japar.par.**

A discussão a respeito deste pacote é apresentada no item 5.4 (Rotinas de biblioteca para os recursos de paralelismo: `japar.par`).

A seguir, cada módulo da proposta é descrito de forma detalhada, servindo de referência para a implementação.

5.2.2 O conversor para reestruturar Java

O conversor **ConJava** foi desenvolvido para qualquer usuário que deseje fazer uso da linguagem **Japar**, sem contudo ter que aprender detalhes de sua implementação. Desta forma o usuário comum, sem nenhuma intimidade com recursos avançados de paralelismo, pode dispor desses recursos de forma prática. Deve-se porém ressaltar as possibilidades limitadas do uso do conversor. Pelo fato de muitas das possibilidades de paralelismo necessitarem de uma declaração explícita do programador, o conversor não poderá aproveitar todas as condições de paralelismo possíveis.

O conversor partirá de um código originalmente sequencial Java, atuando como um dispositivo tradutor, e irá gerar uma saída definida em termos de estrutura **Japar**. Lembramos que um dispositivo paralelizador é um otimizador de código que pode ser entendido como um conversor de alto nível, que irá detectar as estruturas que podem ser

substituídas por outras mais eficientes, neste caso, estruturas paralelas. Este dispositivo buscará encontrar as possíveis fontes, de certa forma já consagradas, de paralelização e transformará esse código em uma ação paralela, com isso rescrevendo o código seqüencial em um novo código paralelo.

Um conversor atua como um dispositivo tradutor especial conhecido por filtro. Exemplos de dispositivos deste tipo são os conversores de texto, e os filtros de alto nível, para converter linguagens de alto nível entre padrões. Seu modo de operação resume-se a descobrir estruturas seqüenciais com potencial para serem paralelizadas, ou seja, estruturas bem definidas, e substituí-las por outras paralelas. Seu funcionamento é semelhante também a certos otimizadores de código, neste caso operando apenas em estruturas seqüenciais. No caso do conversor **ConJava** o algoritmo é também seqüencial, e apenas os potenciais consagrados de paralelização podem ser explorados, sendo reescritos de forma a serem compilados pelo compilador **Japar**.

O usuário pode também escrever um programa em linguagem Java tradicional e, utilizando-se do conversor **ConJava** obter automaticamente um código paralelo para utilizar a Máquina Virtual Java Paralela (MVJP), utilizando-se do compilador **Japar**, ou simplesmente compilar seu código de forma seqüencial e utilizar a mesma máquina paralela, sem código paralelo. Mas por se tratar de uma compilação automática, esta nunca será melhor do que o desenvolvimento realizado por programadores treinados na implementação paralela. Em sua essência, o conversor irá reestruturar comandos tradicionais em comandos explícitos paralelos.

5.2.2.1 Um filtro especial: ConJava

O conversor construído aqui é um tipo especial de tradutor ao qual denominamos filtro. Chamamos filtro a um programa onde a linguagem-objeto a ser produzida também é uma linguagem de alto nível. Para melhor entendermos isso podemos, por exemplo, citar o caso em que o texto-fonte esteja redigido em FORTRAN, e que um tradutor aceite tal texto e o converta para alguma linguagem de baixo nível (linguagem de montagem, ou linguagem de máquina) ou então para alguma outra linguagem de alto nível, tal como

PASCAL ou BASIC. Um tradutor que efetue conversões entre duas linguagens de alto nível é denominado filtro, se a linguagem objeto for muito semelhante à linguagem fonte.

Muitas vezes, programas que efetuam traduções entre dois dialetos de mesma linguagem, ou que permitem converter para uma forma padronizada um texto que inclua extensões de uma linguagem disponível, são denominados pré-processadores. Basicamente, a função dos pré-processadores, neste caso, consiste na eliminação de construções de nível mais alto que o da linguagem-base, e na correspondente inserção de textos equivalentes, compostos exclusivamente de construções da linguagem-base. No presente trabalho, o conversor **ConJava** se utilizou deste conceito. Assim o conversor **ConJava** limita-se a reescrever o texto-básico Java em outro arquivo na linguagem **Japar**.

A reescrita do código seqüencial Java em uma estrutura paralelizante é abordada aqui como uma forma de otimização de código de alto nível. Nesta fase o conversor **ConJava** limitou-se a explorar apenas a transformação do código Java no aspecto das construções de blocos repetitivos, ou seja, as estruturas de laços. Porém, como demonstra [Caf_81], os programas gastam grande parte da execução em estruturas de laços, e chamadas de rotinas, de tal forma que estas estruturas representam grande interesse de otimização ou paralelização. Em suas palavras: “Quanto às técnicas de otimização, observou-se que a maior parte do tempo de execução é gasto em uma parte relativamente pequena do programa, ou seja, 90% do tempo é gasto em 10% do código (*rule 90-10*). Então, a parte mais percorrida de um programa, os laços internos, são um alvo óbvio para a otimização. Otimizações de laços típicas são a remoção de computações invariantes de laço e a eliminação de variáveis de indução”. Este aspecto demonstra que a paralelização de estruturas de laços já oferecem uma grande possibilidade de melhoria de código de alto nível.

5.2.2.2 A BNF e o YACC

Portanto, para construirmos o conversor de linguagem Java (alto nível) para a linguagem **Japar** (alto nível) utilizamos um filtro. A melhor forma que encontramos para

realizar a construção deste tradutor foi utilizando o gerador de compiladores YACC¹, a partir de uma BNF específica como uma de suas entradas.

A implementação utilizou-se de uma gramática semelhante à desenvolvida por Dmitri Bronnikov [Bron-98], sendo essa uma gramática LARL, de forma que não inclui a linguagem Java de forma completa. O estado em que a sintaxe da linguagem se encontra desenvolvida e testada apresenta-se em nível quase completo, apesar de ainda necessitar de alguns aperfeiçoamentos. Existem, entretanto, alguns conflitos gramaticalmente solucionados. Entre esses conflitos, encontram-se os de *Shift/Reduce*² que foram resolvidos às custas de uma gramática definida por um subconjunto da linguagem. Contudo algumas ações ainda devem ser executadas para completar a verificação de sintaxe de programa, sendo elas:

¹ O YACC (*Another Yet Compiler Compiler*) é um interpretador de linha de comando, que cria automaticamente um reconhecedor de linguagem de comando para uma dada sintaxe, tendo ainda um *software* semelhante, e principalmente compatível, o BISON. Sua semelhança com o YACC é muito grande, e no presente trabalho foi utilizada uma versão do BISON. Esta ferramenta pode ser obtida livremente (*freeware*) no endereço eletrônico da GNU, gnu.ist.utl.pt/software/pspp/, e neste trabalho nos referiremos a ela pelo nome genérico YACC. Esta ferramenta converte uma gramática livre de contexto em um conjunto de tabelas para uma automação simples que executa um algoritmo LR(1). Apesar da definição da gramática permitir a ocorrência de alguma ambigüidade, são utilizadas regras de precedência para eliminar ambigüidades que possam surgir. No presente trabalho a gramática será LARL, que é uma gramática um pouco menos poderosa, sendo que uma ambigüidade que não é resolvida é o caso clássico do *else* pendente.

O YACC processa como entrada um arquivo BNF com as regras gramaticais de uma linguagem especificada, produzindo como saída uma implementação em linguagem C que deverá ser compilada para obtermos o reconhecedor da linguagem especificada. A linguagem especificada poderá ser, por exemplo, Pascal ou um subconjunto de outra linguagem de comandos. O código gerado poderá ser compilado em qualquer máquina que possua o compilador C, mesmo que não seja em ambiente UNIX, porque é independente da máquina, uma vez que o YACC fornece todas as rotinas de entrada e saída.

O YACC é considerado uma ferramenta de CASE (*Computer Aided Software Engineering*) bastante flexível, que permite que os arquitetos de linguagem concentrem-se no projeto da linguagem e não na sua implementação. A operação se resume à definição da linguagem usando BNF, obtendo o código de trabalho como saída. Esta ferramenta usa como entrada um arquivo ("y.tab.h") onde são especificadas e definidas as cadeias (*tokens*) da gramática da linguagem. Na saída é gerado um arquivo em linguagem C ("y.tab.c") Para maiores detalhes dessa ferramenta pode-se consultar o trabalho precedente a esse, onde esta ferramenta é utilizada para o desenvolvimento de um compilador Pascal para a arquitetura RISC-LIE [Tra94].

² O YACC fornece uma ferramenta de apoio que permite um bom controle da eliminação ou diminuição de conflitos de ambigüidades. Dentro deste aspecto devemos considerar dois tipos de conflitos importantes:

1 - Os conflitos aceitáveis, de forma a tornar a gramática razoavelmente consistente, usualmente denominados conflitos *Shift/Reduce*,

2- Os conflitos que não podem ser aceitos de maneira alguma, sob pena da gramática tornar-se ambígua, usualmente denominados conflitos *Reduce/Reduce*.

- a) Verificação dos modificadores aplicados a uma classe, *interface*, campo ou *constructor* (permitidas como *interface*, campo, ou *constructor*) em respectivamente uma classe, uma *interface*, um campo ou uma declaração de *constructor*. Por exemplo, uma declaração de classe não deveria permitir outros modificadores tais como *abstract*, *final* e *public*.
- b) Para uma declaração de expressão, verificar se trata-se de um incremento, ou decremento, ou associatividade de tarefa.
- c) Verificar o tipo de expressão em um operador *cast*, e se este indica um tipo. Alguns dos compiladores disponíveis permitirão uso simultâneo idêntico para um tipo e uma variável na mesma extensão que depende de contexto.
- d) Alterar a definição léxica de '[' seguido imediatamente por qualquer número de caracter de branco ou espaço seguido por ']' para símbolo de OP_DIM. O símbolo é definido como $[\backslash [] \{white_space\} * [\backslash]]$ no léxico.

Entretanto, na versão atual, alguns conflitos de *shift/reduce* não foram ainda resolvidos, entre os principais podemos citar o caso do *else* pendente em *if-then-else* ³.

A BNF completa utilizada para definir a estrutura gramatical do conversor ConJava pode ser vista no Apêndice A.1. Uma discussão mais detalhada a respeito desta gramática é

³ **Conflitos *Shift/Reduce*** : As linguagens consideram comandos *if-then* e comandos *if-then-else*, com um par de regras como:

```
if_cmd : IF expr THEN cmd      (1ª regra)
        | IF expr THEN cmd ELSE cmd (2ª regra) ;
```

Quando o *token* ELSE é lido e torna-se o próximo *token* a ser interpretado gramaticalmente, os conteúdos da pilha (assumindo que a entrada é válida) são adequados para redução (*reduction*) pela primeira regra. Entretanto, também é válido substituir (*shift*) o *token* ELSE, porque isto levaria a uma eventual redução pela segunda regra. Esta situação, onde tanto um *shift* quanto um *reduction* seriam válidos, é denominada conflito *shift/reduce*. O YACC é projetado para resolver estes conflitos escolhendo o caminho *shift*, a menos que existam declarações de precedência contrárias especificadas pelo operador. Para compreender a razão desta escolha, vamos compará-la com a outra alternativa. Se o analisador escolhe substituir (pelo caminho *shift*) o ELSE, conforme o exemplo abaixo:

IF x THEN IF y THEN vence; ELSE perde;

o resultado é anexar a declaração *else* ao comando *if* mais interno, tornando as duas entradas, de substituição pela primeira ou pela segunda regras equivalentes.

feita no item 5.3.2, visto que ela foi utilizada também na construção da estrutura do compilador Japar.

5.2.3 O compilador paralelo

Outra possibilidade de se explorar o paralelismo que foi proposta, é a que busca incluir na linguagem Java um conjunto de comandos usados por linguagens paralelas. Estes comandos poderiam substituir comandos já existentes ou simplesmente serem acrescentados como uma opção a mais dentro do *set* de instruções desta linguagem. Neste caso surge uma proposta de uma linguagem paralela, que convencionamos chamar **Japar**, ou seja, Java Paralelo. A linguagem **Japar** inclui diversos comandos semelhantes àqueles usados por linguagens de programação já consagradas, como *Concurrent C*, *SR (Synchronizing Resources)*, *HPF (High Performance Fortran)*, entre outras.

Deve-se notar que para o programador explorar ao máximo as características paralelas da linguagem **Japar**, ele deverá possuir um bom nível de conhecimentos sobre paralelismo, e dominar os conceitos explorados no item 2.3 sobre programação paralela. Assim, conceitos sobre paralelização de programas, as formas de paralelizar-se um programa, bem como alguma linguagem de programação paralela são pré-requisitos para possibilitar a utilização de todos os recursos dessa linguagem.

Considerando-se esses aspectos chega-se então a condição de sugerir-se um conjunto de abordagens paralelas dentro do ambiente Java proposto pela SUN®. A estrutura proposta segue rigorosamente toda estrutura Java já definida e acrescenta aspectos de linguagens paralelas. Em outras palavras, a linguagem proposta aceita qualquer comando original da linguagem Java, podendo ainda processar novos conceitos paralelos que foram acrescentados. Isso permite total compatibilidade com os códigos Java já escritos anteriormente, fazendo assim que esses possam ser reaproveitados sem nenhuma perda. A seguir serão descritos os detalhes dos conceitos e a implementação realizada.

5.2.3.1 O compilador Japar

O compilador aqui implementado é um compilador que processa um subconjunto de comandos da linguagem Java. Sua função é a de substituir o compilador **javac** que acompanha a maioria dos ambientes de Java disponíveis. A implementação do compilador recebeu uma nova denominação para ser diferenciado do compilador **javac**, e foi batizado de **Japar**, sugerindo assim um compilador Java com aplicações paralela.

5.2.3.2 A BNF utilizada

Parte das especificações BNF para a linguagem **Japar**, que correspondem a linguagem Java, são as mesmas adotadas para a construção do conversor **ConJava**. Uma breve discussão sobre essa BNF pode ser vista no item anterior (5.2.2) e sua estrutura encontra-se no apêndice A.1. Porém a BNF para linguagem **Japar** recebe um conjunto de novos comandos. É necessário dizer que este compilador é compatível com o compilador **javac**, com apenas algumas restrições que não foram implementadas devido ao grande número de atividades existentes em um compilador deste porte. A BNF complementar da linguagem Java para a linguagem **Japar** acrescenta diversos elementos, como os discutidos a seguir.

A primeira alteração na BNF é aquela que inclui os novos elementos na sessão de declarações, aqui destacados em negrito:

```
statement :  
    variable_declaration  
    | ( expression ';' )  
    | ( statement_block )  
    | ( if_statement )  
    | (do_statement_par)  
    | ( do_statement )  
    | (while_statement_par)  
    | ( while_statement )  
    | (for_statement_par)  
    | ( for_statement )  
    | ( try_statement )  
    | ( switch_statement )  
    | ( 'synchronized' '(' expression ')' statement )
```

```

| ( 'return' [ expression ] ';' )
| ( 'throw' expression ';' )
| ( identifier ':' statement )
| ( 'break' [ identifier ] ';' )
| ( 'continue' [ identifier ] ';' )
| ( ';' )
;

```

Inicialmente serão descritos os componentes de laços, onde a estrutura de elementos repetitivos em *loops* são as principais. O comando *for* tem como estrutura a seguinte BNF, definida para a linguagem **Japar**:

```

for_statement_par :
    'parfor' '(' ( variable_declaration | ( expression ';' ) | ';' )
    [ expression ] ';' [ expression ] ';' ')' statement 'parend'
;

```

O comando *while* paralelo é definido pela seguinte BNF:

```

while_statement_par :
    'parwhile' '(' expression ')' statement
;

```

O comando *do* pode ser visto a seguir:

```

do_statement_par :
    'pardo' statement 'parwhile' '(' expression ')' ';'
;

```

Para o caso de uma classe ser declarada como sendo paralela, ou seja como pertencente a um bloco a ser paralelizado, e definida como um tipo de declaração,

```

type_declaration :
    [ doc_comment ] ( class_declaration | interface_declaration ) ';'
    | [ doc_comment ] ( class_declaration_par | interface_declaration ) ';'
;

```

e a estrutura dessa classe fica assim definida:

```
class_declaration_par :  
    { modifier } "resource" 'class' identifier  
    [ 'extends' class_name ] 'body' 'identifier'  
    [ 'implements' interface_name { ',' interface_name } ]  
    '{' { field_declaration } '}' 'parend'  
;  
;
```

Para o uso dos comandos *fork* e *join*, que irão criar dois processos iguais, a BNF é a seguinte:

```
compilation_unit :  
    [ package_statement ]  
    { import_statement }  
    { type_declaration }  
    [ type_declaration_par ]  
;
```

A definição para a declaração do processo é a seguinte:

```
type_declaration_par :  
    [ doc_comment ] ( 'fork' class_declaration | interface_declaration ) ';' 'join'  
;
```

A BNF que corresponde aos elementos definidos para transações entre processos, tipos de processos, especificações de processadores, criação de processos, especificação de prioridades, deve considerar alterações em diversos pontos da BNF original. A seguir é apresentada a reunião destas alterações em uma única sequência:

```
// Expressões  
expression_par:  
    within expression ? tc-primary ( expression-list ) : 1 expression  
    | create identifier ( expression-list ) processor-clause priority-clause  
;  
primary:
```



```

        tc-primary ( expression-list)
        ;
tc.primary:
    primary.name
    | primary-> name
    ( *expression )
    ;
processor-clause:
    processor ( expression )
    ;
priority-clause:
    priority ( expression )
    ;
// Declarações

type-specifier:
    process spec identifier ( parameter-decl-list) transaction-part
    | process anytype
    | process identifier
    ;
parameter-decl-list:
    parameter.type-decl
    | parameter-type-decl, parameter-decl-list
    ;
transaction.part:
    { trans-decl-list }
    ;
parameter-type-decl:
    type-name
    decl-specifier declarator
    ;
trans.decl.list:
    trans-decl
    trans-decl trans-decl-list
    ;
trans-decl:
    transaction trans-type-specifier trans-declarator-list ;
    ;
trans-type-specifier:
    async
    | type-specifier
    ;

```

```

trans-declarator-list:
    trans-declarator
    | trans-declarator, trans-declarator-list
    ;
trans-declarator:
    declarator( parameter.decl-list~i>
    | declarator
    ;
statement_par:
    select {guard-alternative-list }
    | delay expression ;
    ;
guard-alternative.list:
    guard alternative
    | guard alternative or guard-alternative-list
    ;
guard:
    ( expression )
    ;
alternative:
    statement.list
    | terminate;
    ;
// Definitions Externas

external-definition:
    process body identifier ( param-list) process-body
    ;
process-body:
    function-statement

```

A BNF complementar para todas as estruturas descritas acima, pode ser vista de forma completa no Apêndice B.1, que inclui todos elementos da linguagem **Japar**.

Visando manter a compatibilidade, todos os comandos da linguagem Java foram implementados. E para a compatibilidade da máquina virtual Java, foram considerados os duzentos e trinta (230) *opcodes* originais propostos pela SUN® e definidos por esta empresa, e foram acrescentados mais dez (10) novos.

5.2.4 A máquina virtual paralela simplificada para Japar: MVJP

Uma das formas de se explorar o paralelismo da linguagem **Japar** em arquiteturas paralelas será aquele em que se utiliza de um carregador de classes especiais. Portanto, a MVJP deverá possuir além do módulo do compilador JIT e do módulo interpretador de Java, um outro módulo especial que irá detectar se os *bytecodes* recebidos foram paralelizados pela linguagem Java. Com essa informação, caberá a esta MVJP explorar, através do conjunto de instruções característicos daquela plataforma, o melhor desempenho. Se a arquitetura for paralela, o trabalho da MVJP será otimizar ao máximo esse conjunto de instruções, combinando com essas os *bytecodes* paralelizados pela linguagem **Japar**.

A principal preocupação do desenvolvimento deste projeto foi alterar o funcionamento da máquina virtual Java, permitindo que esta pudesse aceitar características paralelas. Por isso, muitas rotinas adotadas foram simplificadas em relação ao código original Java desenvolvido pela SUN MICROSYSTEMS. Também, muitas rotinas originais permaneceram inalteradas, e assim, foram utilizadas em seu formato original. O motivo pelo qual isso ocorreu deve-se ao fato já descrito a respeito da compatibilidade e completeza do ambiente implementado. As rotinas mantidas originais foram utilizadas como complemento ao projeto, e não representam nenhuma influência ao código paralelo. A relação das principais rotinas reproduzidas (sempre com autorização especial para desenvolvedores) podem ser vistas no Apêndice B.

A máquina virtual escrita encontra-se em fase de desenvolvimento. Isso significa que apesar da máquina estar pronta para receber e executar código paralelo, ainda não foi desenvolvida para ser executada em nenhuma máquina paralela específica, devendo esta tarefa ser objetivo de um futuro trabalho.

No próximo item será descrita a máquina virtual Java, onde deve-se considerar dois aspectos importantes. O primeiro envolve a máquina hospedeira, e o segundo envolve o tratamento que pode ser dado ao código paralelo produzido pelo compilador **Japar**.

5.2.4.1 Conceitos da máquina virtual paralela

A máquina virtual paralela construída atendeu a dois objetivos principais. O primeiro é referente a necessidade de ter-se uma máquina que pudesse executar o código paralelo fornecido pelo compilador **Japar**. O outro objetivo visava preparar esta máquina virtual Java para uma arquitetura paralela em uma nova versão.

Quanto a preparar a máquina virtual Java para uma arquitetura paralela, a MVJP está pronta para ser portada para alguma arquitetura, sem grandes adaptações. Algumas dessas modificações envolvem escrever o arquivo `cpu.c` – este arquivo contém macros que condicionam vários aspectos para diferentes processadores. O arquivo de cabeçalho `cpu.h` define quais são as cpus que podem ser aceitas. No caso em que se deseja incluir uma nova cpu, por exemplo, de um ambiente paralelo, deve-se fazer uma alteração, para incluir essa cpu. Outra rotina que deve ser alterada é a que cuida de código nativo para o ambiente DOS, ou seja a rotina em *assembly* denominada `invokeNative_x86.asm`.

A versão para a máquina MVJP aceita uma série de *bytecodes* que foram colocados com o único propósito de prepará-la para o uso em arquiteturas paralelas. As novas versões do ambiente deverão explorar esses recursos, que serão melhor usados segundo os recursos da própria arquitetura.

Quanto ao aspecto referente a necessidade de ter-se uma máquina que pudesse executar o código paralelo do compilador **Japar**, deve ser dito que a MVJP é de fundamental importância para os testes que foram realizados. Isto porque a máquina virtual Java tradicional não pode operar com todos os *bytecodes* gerados pelo compilador **Japar**. Portanto para que os testes pudessem ser realizados a máquina virtual Java paralela (MVJP) deveria ser construída.

Uma última observação ainda deve ser feita a respeito dessa máquina: nem todos os *bytecodes* são operacionais. Isso por que alguns *bytecodes* propostos são fundamentalmente códigos para serem usados por máquinas paralelas. No caso dessa versão, esses *bytecodes* são substituídos pela instrução *nop*.

5.2.4.2 Tratamento do código paralelo

Um aspecto que é importante discutir dentro da máquina paralela é o de como o código irá ser tratado. O *bytecode* recebido tanto pode ser tratado por um interpretador como executado pelo compilador JIT.

Parte do código paralelo gerado é o resultado da otimização produzida pelo compilador **Japar** e parte é definido pela estrutura da MVJP, explorado em ambientes paralelos. Mesmo que na atual fase a MVJP não tenha sido testada em ambientes realmente paralelos, a máquina existente está estruturalmente preparada para que isso seja possível através de uma adaptação em versões posteriores desta máquina.

Um tratamento especial oferecido pela MVJP é ativado quando os *bytecodes* paralelos são recebidos. Para tanto foi proposta uma série de novos *bytecodes* e especiais para acrescentar as características paralelas. Na presente versão da MVJP, apenas dez *bytecodes* foram considerados. Os *bytecodes* aqui referenciados foram descritos no item sobre a implementação dessa máquina.

5.3 Implementações Realizadas

A seguir, com base nas propostas descritas no item 5.2, cada módulo é detalhado no aspecto da implementação realizada.

5.3.1 ConJava

Para implementar o conversor **ConJava** foram desenvolvidos diversos programas, listados no Apêndice A. A seguir, esses programas serão descritos de forma simplificada:

- a) ConJava.y - Arquivo de entrada para o processador YACC, contendo a relação de *tokens*, a gramática BNF para Java, as ações semânticas correspondentes (Apêndice A.2).

- b) Y_tab.c – Arquivo resultante da execução do YACC. Contém a implementação em linguagem C. Este arquivo foi renomeado para ConJava.c e compilado no projeto.
- c) Conv.h - Arquivo *header* – Contém todas as declarações de tipos de dados, e protótipos de funções que são usadas de forma global pelos demais programas.
- d) Code.c – Arquivo que contém a implementação das rotinas de tratamento para geração de código Japar. Entre as rotinas podemos observar as rotinas que mantêm o código inalterado, as rotinas de manipulação da pilha e gerenciamento da rotina polonesa, e as rotinas de tratamento específico para a conversão de código, entre as quais code_for(), code_while() e code_do().

No presente trabalho, devido ao uso do YACC, que ofereceu a possibilidade da criação do filtro ConJava, todo o código foi construído em linguagem C, visto que a saída produzida pelo YACC se dá nesta linguagem. O YACC possui uma estrutura onde se pode definir a gramática através de uma BNF, e as ações semânticas correspondentes a esta gramática, onde as ações semânticas receberam dois tratamentos. O primeiro tipo de tratamento foi dado quando o analisador sintático identifica as palavras reservadas e identificadores gerais e demais símbolos que não podiam ser paralelizados automaticamente. Neste caso a ação semântica limitou-se a copiá-los no arquivo de saída (denominado provisoriamente saída.jap). Uma chamada para importar o pacote japar.par.* é acrescentado no início do arquivo.

Como exemplo, podemos ver, a seguir, parte do código de entrada para o YACC, que representa as chamadas para esse tipo de tratamento:

```
SelectionStatement
: IF '(' Expression ')' Statement
    { manter_sequencia($1); manter_sequencia('(');
      manter_sequencia($1); manter_sequencia(')');
      manter_sequencia() }
| IF '(' Expression ')' Statement ELSE Statement
    { manter_sequencia($1); manter_sequencia('(');
      manter_sequencia($3); manter_sequencia(')');
      manter_sequencia($5); manter_sequencia($6);
      manter_sequencia($7) }
| SWITCH '(' Expression ')' Block
```

```

{ manter_sequencia($1); manter_sequencia('(');
  manter_sequencia($3); manter_sequencia(')');
  manter_sequencia($5) }
;

```

A outra forma de tratamento aplicava-se quando o analisador sintático reconhecia uma estrutura possível de ser paralelizada. Neste caso o tratamento do tradutor seria separar a estrutura buscando otimizá-la, isto é rescreve-la em formato paralelo. A estrutura semântica realizou este procedimento através da chamada de rotinas especiais que rescreviam as estruturas de laços. Parte deste código pode ser visto a seguir:

```

IterationStatement
: WHILE '(' Expression ')' Statement
  { code_while( ) }
| DO Statement WHILE '(' Expression ')' ';'
  { code_do( ) }
| FOR '(' ForInit ForExpr ForIncr ')' Statement
  { pol_opr(op_at);
    substitui_simb ($2, tipo , todovalor); }
| FOR '(' ForInit ForExpr ')' Statement
  { code_for( );
    pol_opr(op_at);
    substitui_simb ($2, tipo, todovalor);
  }
;

ForInit
: ExpressionStatements ';' { code_for( ) }
| LocalVariableDeclarationStatement { code_for(); }
| ';' { code_for(); }
;

ForExpr
: Expression ';' { code_for( ) }
| ';' { code_for( ) }
;

ForIncr
: ExpressionStatements { code_for( ) }
;

```

O arquivo completo denominado ConJava.y (entrada para o YACC) pode ser visto no Apêndice A.2. Os códigos dos procedimentos de paralelização das rotinas `code_for()`, `code_while()`, e `code_do()` são apresentados no Apêndice A.3.

5.3.2 Japar

Para implementar o compilador Japar foram desenvolvidos diversos programas. A seguir são descritos de forma simplificada esses programas:

- a) Japar.y - Arquivo de entrada para o processador YACC, contendo a relação de *tokens*, a gramática BNF para Java, as ações semânticas correspondentes, e mostra tanto os elementos da linguagem tradicional Java, como os novos elementos que foram acrescentados (Apêndice B.3)
- b) Y_tab.c – Arquivo resultante da execução do YACC. Contém a implementação em linguagem C. Este arquivo não representa nenhuma informação adicional, pois trata-se apenas de uma transcrição que corresponde exatamente à gramática descrita no arquivo Javar.y traduzido em linguagem C. Este arquivo foi renomeado para Japar.c e compilador no projeto (*“project”* com os demais arquivos)
- c) Bytecodes.c – Programa que contém todas as especificações do conjunto de *bytecodes* que serão processados pela Máquina Virtual Java Paralela (MVJP).
- d) Bytecodes.h – Arquivo de cabeçalho que possui todas as especificações de protótipos de rotinas e declarações de variáveis que serão utilizados pelo compilador **Japar**, de forma global para todos os programas, considerando os aspectos dos *bytecodes*.
- e) Opcode.c – Programa que contém todas as especificações do conjunto de *opcodes* que serão processados pela Máquina Virtual Java Paralela (MVJP).
- f) Opcode.h – Arquivo de cabeçalho que possui todas as especificações de protótipos de rotinas e declarações de variáveis que serão utilizados pelo compilador **Japar**, de forma global para todos os programas.
- g) Flowgraph.c – A rotina que explora o laço encontrado, determinado seu grafo de fluxo. Esta rotina é quem permite a otimização do laço determinado. A partir da criação da estrutura do fluxo de instruções que é montado, é possível aplicar-se os algoritmos de paralelização/otimização disponíveis.

- h) **Dump.c** – Arquivo que escreve o arquivo de saída do compilador **Japar**. O arquivo formata a saída e cria um arquivo de *bytecodes* a serem processados pela MVJP
- i) **Class.c** – Arquivo que lê e verifica as interfaces, os atributos, os métodos, os campos e os arquivos de classe originais da linguagem Java
- j) **Class.h** – Arquivo de cabeçalho das definições usadas por **Class.c**
- k) **Parallel.c** – Arquivo que contém todas as rotinas de paralelização/otimização de código. Este arquivo define como as instruções podem ser reestruturadas para cada bloco a ser paralelizado. Os diferentes tratamentos envolvem a construção de seqüências para laços, onde a otimização é realizada, ou estruturas onde apenas a construção é baseada na disposição dos *bytecodes* que serão tratados posteriormente pela MVJP. Como exemplo deste tratamento estão as gerações de códigos para as instruções da linguagem **Japar** *fork, join, resource*, e outras.
- l) **Parallel.h** – Arquivo que contém as definições de cabeçalho utilizadas por **Parallel.c** e outras rotinas que se comunicam com esse programa.
- m) **Semanti.c** – Arquivo que contém algumas regras semânticas especiais do compilador
- n) **Symbol.c** – Rotina que cria, gerencia e atualiza a tabela interna de símbolos para o compilador **Japar**.
- o) **Lex.c** – Arquivo que contém o analisador léxico e os *tokens* definidos para a gramática **Japar**. Na verdade este arquivo foi absorvido pelo arquivo **Japar.y**, mas uma versão contendo apenas o analisar léxico está também disponível.

A seguir iremos descrever alguns aspectos importantes que foram considerados na implementação do compilador **Japar**.

No arquivo **Bytecodes.c**, uma série de dez novos *bytecodes* foram criados. Os *bytecodes* e os *opcodes* acrescentados utilizaram as posições ainda não usadas na tabela de *opcodes*, e ocuparam um local a partir da posição 210 até 220. Entre os *bytecodes* especiais que foram criados encontramos os seguintes:

- ✓ 210 (0xd2) *bk_ini*
- ✓ 211 (0xd3) *bk_end*
- ✓ 212 (0xd4) *fork_f*
- ✓ 213 (0xd5) *join_f*
- ✓ 214 (0xd6) *fork_w*
- ✓ 215 (0xd7) *join_w*
- ✓ 216 (0xd8) *join_dw*
- ✓ 217 (0xd9) *join_dw*
- ✓ 218 (0xda) *resource*
- ✓ 219 (0xdb) *body_end*
- ✓ 220 (0xdc) *status*

Como esses *bytecodes* são especificamente gerados pelo compilador **Japar**, não são reconhecidos pela máquina Java comum. Por isso, para serem executados devem ser necessariamente submetidos a MVJP.

O *bytecode* *bk_ini* indica para a MJVP que um bloco paralelo esta sendo iniciado. Para a versão atual da MJVP isso é interpretado da mesma forma que um *bytecode* do tipo *nop*, pois ainda está sendo executada em uma máquina seqüencial. Nas implementações que seguirão, esta instrução tem a função de comunicar-se com os processadores solicitando que um deles execute uma sessão em paralelo com o processador dedicado a ela. Fica claro, contudo, que a forma como esse processo irá ser implementado dependerá da arquitetura em que a MJVP em questão estará sendo executada. O *bytecode* que realiza a comunicação de fechamento do bloco paralelo é *bk_end*.

Os *bytecodes* relacionados de 212 até 217 são *bytecodes* especiais para o tratamento dado aos laços (*for*, *while* e *do_while*), onde os processos irão ser forçados a ser executados em paralelo nos diversos processadores disponíveis. O objetivo aqui é explorar o paralelismo em sua granulosidade mais fina, ou seja, estabelecida em nível de instruções. Como nesta versão a MVJP ainda é seqüencial, também esses *bytecodes* não oferecem

nenhuma vantagem para arquiteturas paralelas. Os *bytecodes* seguintes, *resource* e *body_end* serão responsáveis por criar mecanismos de gerenciamento de rotinas locais e globais, dentro de estruturas, onde o objetivo é uma granulosidade maior, ou seja, em nível de procedimentos.

Os bytecodes estão implementados no arquivo `bytecodes.c`, e podem ser vistos a seguir de forma resumida:

```

} bytecode[] = {
/* ***-----> opcode
   *****-----> mnemônico
               **-----> #operandos (em bytes)
                   **-----> pilha pre (em palavras)
                       **-----> pilha post (em palavras)
                           **-----> exceções
                               **-----> desvio */
/* *** ***** ** ** ** ** */
{ 0, "nop", 0, 0, 0, 0, 0 },
/* *** ***** ** ** ** ** */
{ 1, "aconst_null", 0, 0, 1, 0, 0 },
.
.
.
* *** ***** ** ** ** ** */

/* opcodes utilizados como especiais => a partir do 210 */
{ 203, "???", 0, 0, 0, 0, 0 },
{ 204, "???", 0, 0, 0, 0, 0 },
{ 205, "???", 0, 0, 0, 0, 0 },
{ 206, "???", 0, 0, 0, 0, 0 },
{ 207, "???", 0, 0, 0, 0, 0 },
{ 208, "???", 0, 0, 0, 0, 0 },
{ 209, "???", 0, 0, 0, 0, 0 },
{ 210, "bk_ini ", 1, 6, 7, 0, 3 },
{ 211, "bk_end", 1, 6, 7, 0, 3 },
{ 212, "fork_f", 1, 6, 7, 0, 3 },
{ 213, "join_f", 1, 6, 7, 0, 3 },
{ 214, "fork_w", 1, 6, 7, 0, 3 },
{ 215, "join_w", 1, 6, 7, 0, 3 },
{ 216, "fork_dw", 1, 6, 7, 0, 3 },
{ 217, "join_dw", 1, 6, 7, 0, 3 },
{ 218, "resouce", 1, 6, 7, 0, 3 },
{ 219, "body", 1, 6, 7, 0, 3 },
{ 220, "status", 1, 6, 7, 0, 3 },
{ 221, "???", 0, 0, 0, 0, 0 },

```

A estrutura do arquivo *paralel.c* pode ser vista no Apêndice B.5.

5.3.3 Rotinas de biblioteca para os recursos de paralelismo: **japar.par**

Os comandos paralelos que permitirão a exploração do paralelismo implementado serão acionados pelo comando *import.japar.par*, ou seja, com a inclusão dessa biblioteca, a linguagem passa a oferecer recursos paralelos. Para o caso de implementar-se o paralelismo entre processos, o compilador **Japar** considerou a possibilidade de controle por passagem de parâmetros, cuidando dessa forma principalmente dos conceitos de passagem de mensagens. Entre essas classes encontra-se o arquivo de classes **japar.par.para**.

Em termos de comunicação por passagem de mensagens são usados os comandos *Send message to X*, *receive message from Y*, e *lock*, *unlock* que devem ser explicitadas pelo usuário.

O comando *Send message* especifica que um objeto está enviando uma mensagem para outro objeto (ou processo), criando assim um estado crítico que deve ser gerenciado pela MVJP. O mesmo se dá quando o comando *receive message from* é explicitado. Para os casos em que se estabelece uma sessão crítica, os comandos *lock* (trava) e *unlock* (destrava) são extremamente importantes quando devem ser feitas escrita e leitura de variáveis, considerando-se as restrições de que variáveis não podem ser acessadas de forma simultânea.

Entre os principais elementos que podem ser utilizado no pacote **japar.par**, podemos relacionar:

- ✓ Japar.par.Critica – Estabelece uma sessão crítica prioritária definida pelo usuário
- ✓ Japar.par.Lock – Cria uma trava de acesso a uma determinada sessão crítica ou a uma variável.
- ✓ Japar.par.Unlock – Desativa uma trava criada pelo elemento Lock
- ✓ Japar.par.Critica_GC – Gerencia o uso da sessão critica definida pelo usuário, liberando a área quando o usuário sai da sessão
- ✓ Japar.par.Pilha – Cria uma pilha de valores (objetos) para serem gerenciados na sessão crítica.

- ✓ Japar.par.Polonesa – Gerencia a pilha criada.
- ✓ Japar.par.Testa – Testa a sessão crítica, para acesso a esse espaço.
- ✓ Japar.par.ClassLoaderPar – Este elemento será melhor descrito no item 5.5.

Os elementos relacionados acima foram definidos com o intuito de propiciar a exploração pela linguagem **Japar** do aspecto da utilização de bibliotecas especiais, deixando assim aberta também a oportunidade em explorar-se essa tecnologia. Contudo a biblioteca **japar.par** apenas identifica os comandos dando-lhes um tratamento sequencial, devendo ser este aspecto explorado em uma versão futura do ambiente paralelo proposto. Assim definimos a exploração de bibliotecas paralelas para um trabalho futuro que pode ser explorado usando o mesmo compilador **Japar** desenvolvido no presente trabalho, sem que nenhuma alteração deva ser feita neste compilador. Apenas os elementos acima relatados foram implementados, mas fica claro que para um tratamento completo de bibliotecas especiais um conjunto muito maior deve ser explorado.

5.3.4 O carregador de classes VM Java: Japar.par.ClassLoaderPar()

Uma ferramenta importante para a operação da MJVP é um carregador de classes que considere as diferentes implementações que foram desenvolvidas no compilador **Japar**. Um carregador de classe é um objeto responsável por carregar classes. Dado o nome de uma classe, o carregador de classes deveria tentar localizar ou gerar dados que constituem uma definição para essa classe. Uma estratégia típica é transformar o nome da classe em um nome de arquivo e então ler um arquivo de classe daquele nome de um sistema de arquivo. No caso do carregador de classes paralelas, ele deve atender tanto a requisição do interpretador da MVJP quanto seu JIT. O carregador aqui considerado é semelhante ao carregador definido pela SUN®, com algumas alterações. Este carregador de classes faz parte do pacote Japar.par citado anteriormente.

Todo objeto de Classe contém uma referência ao ClassLoader que o define. Sempre que código de Java executável precisa usar uma classe que não foi carregada, o método de `loadClass` é invocado para o carregador de classe desta classe que contém o código em

questão. O uso paralelo de uma classe que esteja fazendo acesso a objetos paralelos deve, em tempo de execução, considerar um carregamento paralelo, explorando assim todo desempenho paralelo.

As classes de objetos não são criadas por carregadores de classe, mas sim criadas automaticamente quando requeridas em tempo de execução da máquina Java. Para que o carregamento seja eficiente é necessário que as cargas dos diferentes objetos sejam distribuídas e executadas pelos processadores disponíveis. Os carregadores de classe podem ainda serem usados por gerentes de segurança para indicar domínios de segurança: duas classes podem ser consideradas "amigáveis" ou relacionadas, mas somente se elas estivessem definidas pelo mesmo carregador de classe. Isso significa que pode existir uma comunicação entre os processos, havendo ainda troca de mensagens entre as mesmas. Por isso o carregador de classes deve prever esta interação, e com o ponto de vista de passagens de mensagens explorar este aspecto.

A seguir podemos ver um pequeno trecho de código onde são considerados alguns aspectos paralelos.

```
// código original definido pela SUN para o carregador java.lang.ClassLoader

public abstract class ClassLoader {
    protected ClassLoader() throws SecurityException;
    protected abstract Class loadClass(String name, boolean resolve)

        throws ClassNotFoundException;
    protected final Class defineClass(byte data[],
        int offset, int length)

        throws NullPointerException, IndexOutOfBoundsException,
        ClassFormatError;
    protected final void resolveClass(Class c)

        throws NullPointerException;
    protected final Class findSystemClass(String name)

        throws ClassNotFoundException;
}

//fim do código original

/* Sessão que considera a possibilidade de paralelismo do carregador */
```

```
throws ClassParalela;  
throws SessaoCriticaDeClasses;  
throws GerenciaCarregamentoParalelo;  
throws RequisitaProcessadores
```

```
}
```

5.3.5 A máquina virtual Java

O conjunto de programas original do ambiente Java é construído por cerca de 721 arquivos em linguagem C, incluindo as versões para Solaris e para ambiente win32⁴. Existem ainda cerca de 592 arquivos header para solaris e win32. Os arquivos são especificados para serem montados através de arquivos makefile para a ferramenta GCC GNU, ou arquivos de *project* para ambiente PC. A seguir serão descritas algumas dessas rotinas que foram utilizadas, e quando houver, as modificações que foram inseridas nestas rotinas. Algumas rotinas que são apenas utilizadas sem maiores interesses serão apenas referenciadas.

Para implementar a máquina virtual Java paralela MVJP foram desenvolvidos diversos programas, e também reutilizados vários programas originais, como descrito anteriormente. Os principais programas e rotinas da máquina virtual Java paralela (MVJP) são descritos de forma simplificada a seguir:

- a) bytes.c – Arquivo contendo as rotinas de manipulação de *bytes* para várias arquiteturas. Esta rotina deve ser alterada para poder receber qualquer tipo de *byte* da arquitetura a ser implementada.
- b) bytes.h - Arquivo de cabeçalho que possui todas as especificações de protótipos de rotinas e declarações de variáveis que serão utilizados pelo programa bytes.c.
- c) jit.c – Arquivo que implementa o compilador “just in time” (JIT). Este compilador não foi preparado para trabalhar com código paralelo, sendo que quando ele encontra qualquer formação paralela ele simplesmente trata como seqüencial. Porém esse compilador teve que ser preparado para poder aceitar os

códigos *bytecode* gerados pelo compilador **Japar** sem provocar nenhum travamento. Com isso o compilador teve que ser alterado e adaptado para funcionar com essas modificações. O tratamento do compilador JIT com abordagem paralela deverá ser elaborado em uma futura fase na continuidade deste projeto. A versão do compilador deveria ser atualizada sempre que haja mudanças na interface do interpretador e do compilador JIT.

- d) `cpu.c` – Este arquivo contém macros que condicionam vários aspectos para ambientes diferentes. As CPUs que são suportadas atualmente pelo arquivo estão relacionadas no quadro abaixo. Observação: os arquivos podem não executar nas versões mais recentes. A rotina `cpu.h` define quais são as cpus que podem ser aceitas; neste caso devemos proceder uma alteração para incluir a nova cpu desejada, no caso de estarmos implementando uma nova arquitetura paralela. Na Tabela 5.1, a seguir são relacionados os parâmetros e CPU's já implementadas.

Tabela 5.1 – CPU's implementadas.

Versão de referência	Nome da diretiva	Aplicação
CpuMacClassic	CpuMacClassi c	Qualquer Macintosh com cpu 68000 ou mais maior, e com Classic Quickdraw: 512KE, MacPlus, Classic Mac, Portable, 100
MacII	CpuMacII	Macintosh com cpu 68020 ou maior, com Color Quickdraw :Mac II, IIx, IIc, IIci, IIcx, IIfx, IIlc, Iisi, 140, 170, 700, 900, Classic II
29000	cpu 29000	The GC 824 card
Newton	CpuNewton	ARM em um Newton
Intel	CpuIntel	The Intel 380386
Printer	CpuPrinter	PostScript executada a partir de um caminho definido de "download"
Power	cpuPower	Processador 601 PowerPC

⁴ Ambiente Microsoft Windows para ambiente PC.

Neste arquivo são feitas as futuras atualizações para as arquiteturas a serem estudadas, como por exemplo a arquitetura Data Flow, uma arquitetura vetorial, etc.

- e) `jvm.c` – O programa `jvm` implementa a máquina virtual Java que foi preparada para receber os códigos de classes processados pelo compilador Japar.
- f) `ClassLoader` – Programa que implementa o carregador de classes para a máquina virtual Java.
- g) `ClassLoad.c` – Rotinas por carregar e solucionar definições de classe.

A seguir são relacionadas algumas rotinas originais definidas pela SUN® no desenvolvimento do ambiente:

- a) `Check_code` – Verifica se o código em um bloco de um método não explora nenhuma falha de segurança.
- b) `gc.c` – Programa que efetua a administração de memória para o ambiente Java e realiza a coleção de lixo (*garbage collection*)
- c) `Check_format.c` – Este arquivo contém as rotinas que verificam formatos de classes, e de tipos de dados, retorna valor 0 no caso de sucesso, -1 para o caso de alguma verificação ter determinado um valor fora do espaço de memória, -2 no caso de erro no formato da classe, -3 no caso de haver incompatibilidade de versão, -4 no caso de uso inadequado de classe. Este programa também preenche a estrutura `class_size_info` que contém informação do tamanho de vários componentes de classe. Este programa foi aplicado parcialmente, visto que a verificação de tipos unicode não foram implementadas na presente versão do compilador.
- d) `check_version.c` – Programa que verifica se a versão correta está sendo utilizada. Para a versão atual, o parâmetro `JNI_VERSION_1_2` utilizado para definir as máquinas que utilizam a versão JDK 1.2.1 foi alterado para `JNI_VERSION_PAR_1`, que identifica o ambiente para o compilador paralelo Japar.

- e) `Class.c` – Implementação original para a classe `Class`
- f) `typecodes.h` – Arquivo de cabeçalho que define o sistema para tipos de código (*typecode*) da representação interna dos valores de escalares. Este sistema de *typecode* permite representar o tipo de escalares de um modo uniforme. Por exemplo, todos os tipos inteiros têm bits em comum, e são distinguidos por um campo interno de tamanho. Tipos sem tamanhos múltiplos não têm um campo de tamanho. Os escalares somente podem ter tamanhos que são potência de 2.
- g) `monitor_cache_md.c` – Código para administrar o monitor da cache (*pool*). Esta estrutura é característica do ambiente Windows de 32 bits. O monitor *pool* usa uma estrutura de dados especial de lista ligada, devido ao fato de que uma vez que a cache se expande, não se contrai, embora isso seja possível.
- h) `threadruntime.c` – Arquivo para o tratamento de *threads*.
- i) `invokeNative_x86.asm` - Arquivo *assembly* que contém as informações a respeito da função que traduz a chamada de Java para a rotina `_stdcall` usada em métodos nativos. A máquina virtual Java passa todos os argumentos de sua pilha, e espera que os resultados sejam colocadas nesta pilha. Deve-se copiar os argumentos da pilha definidas na linguagem C, e colocar esses valores retornados na pilha da linguagem Java.

5.4 Resultados Obtidos

Os testes que foram realizados tratam da avaliação da eficiência do compilador **Japar**. A seguir serão descritos os testes e as avaliações realizados para cada um dos módulos implementados.

5.4.1 ConJava

Com o objetivo simples de demonstrar como um pequeno arquivo Java seqüencial foi convertido pelo conversor **ConJava** mostra-se a seguir um arquivo denominado `teste.java` e a seguir sua conversão `saida.jap`.

```

/*****
*   Arquivo Java teste.java
*   Versao 2.5
*   Este programa sera submetido ao conversor ConJava, que
*   deve converter o mesmo para um programa paralelo
*****/

class for1 {
    double[] a;

    void obtem_N(int N, double escala) {
        double[] b = new double[N];
        double[] c = new double[N];

        for (int i = 10; i < N; i += 4) {
            int local = 3;
            b[i] = a[i] * escala + local;
        }
    }
}

```

E o resultado da conversão é o arquivo `saida.jap`:

```

/*****
*   Arquivo de conversao para Java Paralelo
*   Versao 2.5
*   Junho 1999
*   Autor Antonio Fernando Traina
*   Direito autorais (C)
*   1998, 1999 Antonio Fernando Traina,
*   todos os direitos reservados.
*   ESTE CONVERSOR É FORNECIDO SEM QUALQUER GARANTIA EXPRESSA
*   OU INCLUÍDA, PORÉM SEU USO NÃO É LIMITADO, PODENDO SER
*   USADO PARA QUALQUER PROPÓSITO PARTICULAR, SEM MODIFICADO
*   DESDE QUE REFERENCIADA
*   contato: ftraina@linkway.com.br
*   O programa ConJava deve ser usado para converter
*   um programa *.java sequencial em um arquivo
*   *.jap paralelo
*****/

import japar.par.*

class for1 {
    double[] a;

    void obtem_N(int N, double escala) {
        double[] b = new double[N];
        double[] c = new double[N];

        /*for paralelo , com N de 1 ate 100.000 */
        parfor (int i = 10; i < N; i += 4) {
            int local = 3;

```

```

        b[i] = a[i] * scal + local;
    } endfor;
}

/*****
*   Fim do arquivo convertido
*   Sucesso na conversao
*****/

```

5.4.2 Japar e a máquina virtual Java paralela (MVJP)

A MVJP construída tem a função de interpretar e executar os *opcodes* especiais do compilador Japar. Na presente versão, nem todos os *bytecodes* encontram-se funcionais. E para uma melhor medida do desempenho do código gerado, os *bytecodes* sem aplicação, que inclusive tinham a função semelhante a *bytecodes nop* foram retirados.

Para uma avaliação do compilador e da máquina virtual foram utilizados diversos programas, baseados em um algoritmo construído para realizar o cálculo do valor de Pi (π). Neste programa, o cálculo de Pi poderá ser mais preciso ou menos preciso, em função do número de interações desejada pelo usuário. Um algoritmo simplificado pode ser visto a seguir:

```

Programa Calcula Pi
inicio
    iniciar variaveis
    enquanto leitura do valor for diferente de zero
    inicio
        mensagem (Entre numero de intervalos para cálculo, e 0 para
                    finalizar)
        inicio da contagem do tempo
        se valor do número de intervalos for igual a zero
            interromper
        senão
            cálculo (iterativo) do valor de Pi
            mensagem( O valor de Pi é aproximadamente... com erro de ...)
            final da contagem do tempo
            mensagem(tempo gasto para o cálculo)
        fim do se
    fim do enquanto
fim do programa

```

Este algoritmo teve algumas versões implementadas, que podem ser observadas no Apêndice B.4. A relação de suas versões é listada na Tabela 5.2. Para a avaliação do

comportamento da execução em paralelo desse programa, foi também utilizado um programa MPI, cuja versão foi publicada em novembro de 1999 para Windows NT em sua edição MPICH.NT.1.2.0, que pode ser obtida pela Internet no endereço eletrônico:

<http://www-unix.mcs.anl.gov/~ashton/mpichbeta.html>

Tabela 5.2- Relação dos programas de teste.

Nome do Programa	Linguagem	Uso de MPI
cpi.c	C	Não
cpimpi.c	C	Sim
cpijava.java	Java	Sim
cpipar.java	Japar	Sim

O programa cpi.c implementa o algoritmo do cálculo de Pi usando comandos em linguagem C, e é executado de forma local. O programa cpimpi.c usa exatamente o mesmo algoritmo onde comandos MPI são inseridos para que ele possa ser executado em diversos computadores, através da rede local. A medida do tempo de execução destes dois programas é feita usando uma função já existente em MPI, chamada MPI_Wtime(), que pode ser acessada através do *header* mpi.h.

O programa cpipar.java é uma construção em linguagem Japar para ser executada por esse compilador, onde o comando *for* é descrito de forma a forçar a paralelização. De forma a explicitar esse comando, o programa ConJava não foi utilizado, e o programa foi escrito manualmente. Com esse programa paralelo objetivou-se enfocar dois aspectos: a melhoria com a otimização de seus *bytecodes* e a exploração de *threads* em ambiente paralelo.

Os testes foram efetuados em quatro computadores. Os computadores utilizados encontram-se na Tabela 5.3.

Tabela 5.3 – Relação dos computadores utilizados

Nome da Máquina	CPU	<i>Time clock</i> (MHz)	RAM (Mbytes)	Número de processadores	Sistema Operacional
Atm5 ^(*)	Intel Pentium Pro	200	64	2	NT 4.0
Atm7	Pentium II	233	64	1	NT 4.0
Atm2	Pentium II	233	64	1	NT 4.0
Blackbird	Pentium II	300	64	1	Windows 98

^(*) para efetuar alguns testes, o número de processadores variou entre um e dois.

Apenas o computador Blackbird não se encontra ligado em rede com os demais computadores, sendo que neste computador foram efetuados apenas testes de desempenho local. Entre os computadores Atm2, Atm5 e Atm7, conectados por rede TCP/IP, foi estabelecida uma comunicação em ambiente MPI . Os testes efetuados visaram observar a possível melhoria da execução do código paralelo em relação ao código sequencial.

Para o conhecimento do comportamento do ambiente de teste, algumas verificações foram realizadas, a fim de reconhecermos quais fatores influenciam na medida. Cada medida foi obtida executando-se o programa ao menos dez vezes, e nestes casos foi obtida a média desses valores, aceitos como resultados finais. Inicialmente buscou-se aferir as diferenças entre o programa cpi.c e o programa cpimpi.c, e os resultados podem ser vistos na Tabela 5.4.

Na Tabela 5.4 devemos observar que o tempo está expresso em segundos. Desta forma, pode-se notar por esta tabela, que o comportamento do programa executado em um ambiente de uma mesma máquina (Atm7-local), mesmo quando se varia o número de processos através do MPI, permanece constante, sem perdas, nem qualquer tipo de ganho.

Tabela 5.4 – Comparação entre os programas cpi.c e cpimpi.c

Número de iterações	Tempo em Segundos (precisão em milisegundos)			
	Cpi.c Atm7	Cpimpi.c Atm7 (MPI) 1 processo	Cpimpi.c Atm7 (MPI) 2 processos	Cpimpi.c Atm7 (MPI) 3 processos
10 ⁶	0,28	0,28	0,28	0,28
10 ⁷	2,72	2,71	2,72	2,72
10 ⁸	27,10	27,12	27,14	27,15
10 ⁹	271,2	270,7	270,5	270,7

O arquivo *batch* host1.bat que executa o ambiente MPI, e que possui as diretivas MPI possui o seguinte formato para execução na máquina Atm7:

```
MPICH Job 1.1.0
exe \\atm7\C_ATM7\cpimpi.exe
hosts
atm7 1\
```

Ao ser executado nesta configuração, a Tabela 5.4 que o comportamento dos programas cpimpi.c e cpi.c são idênticos, em termos de tempo de execução. E isso significa que o processo de execução do programa MPI não acrescenta nenhum tipo de degradação ao sistema.

Um outro teste efetuado considerou os aspectos em que o programa cpimpi.c foi executado em diversos computadores, sendo executados e distribuídos sempre a partir da máquina Atm7. O resultados podem ser vistos nas Tabelas 5.5a e 5.5b.

Primeiramente foi executado o *batch* para dois computadores, usando as máquinas Atm7 e Atm2. Observa-se que estas máquinas possuem configurações idênticas, tanto de *hardware* quanto de *software*, inclusive sendo mantidos todos os programas no ambiente de forma igual. O arquivo *batch* host2.bat para esse caso possui o seguinte formato:

```
MPICH Job 1.1.0
exe \\atm7\C_ATM7\cpimpi.exe
hosts
atm7 1\
atm2 1\
```

Nesta configuração, como pode-se ver na Tabela 5.5a, o tempo foi reduzido aproximadamente à metade de seu valor inicial, numa melhoria média de 48%. Observou-se que seu comportamento mostrou-se bastante linear.

Tabela 5.5a – Comparação com o programa cpimpi.c usando vários computadores

Número de iterações	Tempo em Segundos (precisão em milisegundos)		
	cpimpi.c Atm7 (MPI) 1 processo	Cpimpi.c Atm7 (MPI) 1 processo Atm2 (MPI) 1 processo	Cpimpi.c Atm7 (MPI) 1 processo Atm2 (MPI) 1 processo Atm5 (MPI) 1 processo (um processador)
10 ⁶	0,28	0,15	0,12
10 ⁷	2,71	1,39	1,07
10 ⁸	27,12	13,82	10,63
10 ⁹	270,7	135,3	106,20

A última coluna da Tabela 5.5a mostra o resultado da conexão de três máquinas, sendo que o arquivo *batch* host3.bat executado tinha o seguinte formato:

```
MPICH Job 1.1.0
exe \\atm7\C_ATM7\cpimpi.exe
hosts
atm7 1\
atm2 1\
atm5 1\
```


O resultado em relação à execução em uma única máquina foi uma melhoria de aproximadamente 60%.

Tabela 5.5b – Comparação com o programa cpimpi.c usando vários computadores

Número de iterações	Tempo em segundos (precisão em milisegundos)			
	Cpimpi.c Atm7 (MPI) 2 processos	Cpimpi.c Atm7 (MPI) 1 processos	Cpimpi.c Atm7 (MPI) 1 processo	Cpimpi.c Atm7 (MPI) 1 processo
	Atm2 (MPI) 1 processo	Atm2 (MPI) 1 processo	Atm2 (MPI) 1 processo	Atm2 (MPI) 1 processo
			Atm5 (MPI) 2 processos (um processador)	Atm5 (MPI) 2 processos (dois processadores)
10 ⁶	0,19	0,15	0,17	0,09
10 ⁷	1,85	1,39	1,6	0,81
10 ⁸	18,1	13,82	15,9	7,95
10 ⁹	180,05	135,3	161,0	79,62

Os dados da Tabela 5.5b referem-se aos testes feitos em várias máquinas e com variação no número de processos em cada máquina. Na primeira coluna pode ser visto o resultado da execução de três processos simultâneos entre as máquinas Atm7 e Atm2. Neste caso solicitamos que dois processos fossem executados localmente na máquina Atm7 e um processo na máquina Atm2. Pode-se notar que houve uma perda em relação a execução de apenas dois processos (um em cada máquina), que podem ter sido provocados por problemas de comunicação e sincronização entre os processos. Executou-se, também, quatro processos em três máquinas, e comparando-se com três processos em três máquinas foram notadas algumas alterações quanto ao tempo. Na terceira coluna observam-se os tempos da execução de quatro processos sendo dois na máquina Atm5, quando está

dispunha de apenas um processador. O tempo novamente aumentou em comparação ao processamento visto na terceira coluna da Tabela 5.5a, quando haviam três processos em três máquinas. Quando executou-se o programa usando quatro processos, sendo dois processo na máquina Atm5 com dois processadores, este tempos mostraram uma melhora de aproximadamente 25% em relação ao processamento em três máquinas e três processos.

Uma outra avaliação buscou analisar o desempenho do mesmo algoritmo de cálculo de Pi escrito em linguagem Java. Este teste é apresentado na Tabela 5.6 e pode-se notar que a precisão da medida do tempo sofreu uma perda em relação à implementação em linguagem C. Essa alteração deve-se ao fato de não ser possível usar a rotina do conjunto MPI. Desta forma, a medida de tempo em linguagem Java foi feita utilizando-se o *package* java.util.Date onde pode-se encontrar as funções do objeto Date getHours(), (s.getMinutes() e getSeconds(). A função getTimeInMillis() encontrada no objeto Calendar do *package* java.util.*, ao ser executada, apresentou problemas na execução da máquina Java (JVM), por isso não pôde ser utilizada. Utilizou-se então apenas a precisão em segundos.

Tabela 5.6 – Comparação com o programa em linguagem Java cpijava.java.

Número de iterações	Tempo em segundos**			
	Cpijava.java Atm7 1 processo	cpijava.java Atm7 (MPI) 2 processos e 1 processador	Cpijava.java Atm7 1 processo Sem Uso do compilador JIT	Cpi.c Atm7
10 ⁶	0 ~1,0	0~1,0	2.0	0,28
10 ⁷	2,0	2,0	21.0	2,72
10 ⁸	17,0	17,0	205.0	27,10
10 ⁹	171,0	171,0	2044	271,2

** Precisão num programa java em segundos, precisão em milisegundos não é possível.

A avaliação do programa foi feita, portanto apenas de forma local, visto que a linguagem, no atual estágio não pode ser executada com parâmetros MPI. Desta forma, os programas foram executados localmente, e seu desempenho foi rigorosamente igual, nos casos em que utilizou-se ou não o ambiente MPI. Isso atesta o funcionamento do ambiente MPI de forma a não influenciar o desempenho do programa. A execução do programa usando apenas uma máquina foi feito carregando-se o ambiente Java a partir do arquivo *batch* hostjava.bat:

```
MPICH Job 1.1.0
exe \\atm7\C_ATM7\java.exe cpijava.
hosts
atm7 1\
```

A medida do tempo de execução do programa Java foi feita utilizando-se o compilador JIT, e apresentou resultados bastante superiores quando comparado com o programa em linguagem C comum. É possível efetuar-se a compilação do programa com a opção `Djava.compiler=NONE`, que irá executar o programa utilizando o interpretador, ou seja sem o compilador JIT. É importante destacar que o compilador JIT não está disponível em algumas versões de máquinas Java, como por exemplo Java para UNIX. Quanto ao desempenho da máquina Java sem o compilador JIT, existe uma notável diferença, que pode ser observada na Tabela 5.6.

A primeira coluna da Tabela 5.6 mostra os tempos de execução para o programa Java sendo executado de forma não paralela. Pela coluna dois, pode-se observar que, como esperado, não há nenhuma alteração quando este mesmo programa é executado em um processador, com dois processos. A terceira coluna mostra os tempos quando este mesmo programa é executado sem o compilador JIT. Pode-se observar um aumento em até 12 vezes do tempo em relação ao compilador JIT.

Outro teste importante realizado buscou verificar o comportamento da máquina *Atm5*, em relação ao número de processadores. Os resultados deste teste podem ser vistos na Tabela 5.7a. O programa para o cálculo de Pi foi executado em um processo e em dois

processos, com um e com dois processadores. O resultado comprovou o observado anteriormente, com melhoria quase ideal de desempenho.

Tabela 5.7a – Testes de execução do programa cpijava.java no computador Atm5

Número de iterações	Tempo em segundos			
	cpijava.java Atm5 (MPI) 1 processo um processador	cpijava.java Atm5 (MPI) 2 processos 1 processador	cpijava.java Atm5 (MPI) 1 processos 2 processadores	cpijava.java Atm5 (MPI) 2 processos 2 processadores
10 ⁶	0,32	0,32	0,32	0,16
10 ⁷	3,19	3,19	3,17	1,59
10 ⁸	31,9	31,8	31,7	15,94
10 ⁹	318,6	317,91	316,95	158,90

Na Tabela 5.7b são vistos os testes semelhantes aos da Tabela 5.7a, levando em consideração o programa em Java, com e sem compilador JIT. Novamente o resultado da compilação utilizando o compilador JIT superou a execução sem esse compilador na ordem de doze vezes.

Pode-se comparar o desempenho das máquinas Atm7 e Atm5 através das Tabelas 5.6 e 5.7b. A Tabela 5.6 apresenta os valores de tempo de execução do programa cpijava.java sendo executado na máquina Atm7, enquanto que a Tabela 5.7b mostra o tempo de execução para o mesmo programa na máquina Atm5. Ao comparar-se as duas primeiras colunas das tabelas pode-se verificar que a máquina Atm7 é entre 15 a 21% mais rápida do que a máquina Atm5.

Tabela 5.7b– Testes de execução do programa cpijava.java no computador Atm5

Número de iterações	Tempo em segundos							
	Cpijava.java Atm5 (MPI) 1 processo um processador		Cpijava.java Atm5 (MPI) 2 processos 1 processador		cpijava.java Atm5 (MPI) 1 processos 2 processadores		Cpijava.java Atm5 (MPI) 2 processos 2 processadores	
	Com JIT	Sem JIT	Com JIT	Sem JIT	Com JIT	Sem JIT	Com JIT	Sem JIT
10 ⁶	0,0	2,0~3,0	0,0	2,0~3,0	0,0	2,0~3,0	0,0	2,0~3,0
10 ⁷	2,0	25,0	2,0~3,0	25,0	2,0	25,0	2,0	25,0
10 ⁸	20,0	252,0	20,0	251,0	20,0	248,0	20,0	248,0
10 ⁹	201,0	2511,0	202,0	2511,0	200,0	2485,0	200,0	2485,0

Finalmente, na Tabela 5.8 são vistos os resultados dos testes do programa cpijava.jav, executado pelo compilador Japar e executado na máquina MVJP sendo executada na máquina Atm7.

Tabela 5.8 – Teste com o programa cpijava.java (paralelo) em ambiente sem e com MPI

Número de iterações	Tempo em segundos			
	cpijava.java Atm7	cpijava.java Atm7 (MPI) 1 processo	cpijava.java Atm7 (MPI) 2 processos	cpijava.java Atm7 (MPI) 3 processos
10 ⁶	0,0	0,0	0,0	0,0
10 ⁷	1,0	1,0	1,0	1,0
10 ⁸	15,0	15,0	15,0	15,0
10 ⁹	153,0	154,0	154,0	154,0

Ao comparar-se a Tabela 5.6 com a Tabela 5.8 pode-se notar que o desempenho do programa cpijava.java é maior, na ordem de 10 a 11% mais rápido para o programa paralelizado em relação não paralelizado. Novamente o processamento do programa utilizando-se de vários processos com apenas um único processador não sofreram nenhuma alteração devido a influência do programa MPI, visto que os resultados mantiveram-se constantes.

A Tabela 5.9 apresenta os valores obtidos pela execução do programa cpijava.java executado pelo compilador Japar e executado na máquina MVJP baseada em Atm5, com um e dois processadores. A primeira e a segunda coluna mostram os valores do programa sendo executado por apenas um processador. Os valores, mesmo com o MPI foram iguais, como era esperado, e foram 11 à 15% melhores do que quando a execução do programa em máquina Java fornecida pela SUN, que podem ser vistos na Tabela 5.7b. Quando executados em dois processadores o resultado foi melhor, apresentando uma melhoria de até 48% em relação ao mesmo processo em apenas um processador.

Tabela 5.9 – Testes de execução do programa paralelo cpijava.java no computador Atm5

Número de iterações	Tempo em segundos	
	cpijava.java Atm5 1 processo 1 processador	cpijava.java Atm5 (MPI) 2 processos 2 processadores
10 ⁶	0,0	0,0
10 ⁷	1,0	0,0
10 ⁸	17,0	9,0
10 ⁹	173,0	89,0

Atualmente esta máquina encontra-se ainda em fase de testes, sendo que busca-se realizar uma comparação do programa `cpijava.java` paralelo usando-se vários computadores.

Também foram realizados alguns testes usando a máquina Blackbird de forma a testar o funcionamento durante o desenvolvimento da máquina MVJP, sendo esse o principal ambiente utilizado para o desenvolvimento e parte dos testes do compilador **Japar**. A avaliação do resultado neste computador foi feita utilizando um conjunto de rotinas que utilizavam dois tipos de conceitos nos algoritmos: um conjunto de programas sequenciais e um conjunto de programas correspondentes com conceitos paralelos. Alguns desses programas usados para testar o compilador podem ser visto no Apêndice B.5, e fazem parte da lista a seguir:

- ✓ Programa `for_simples.java` – programa contendo um simples laço *for*
- ✓ Programa `while_simples.java` – programa contendo um simples laço *while*
- ✓ Programa `for_duplo.java` – programa contendo um laço duplo *for*
- ✓ Programa `while_duplo.java` – programa contendo um laço duplo *while*
- ✓ Programa `for_varios.java` – Programa contendo vários laços *for* sequenciais
- ✓ Programa `while_varios.java` – Programa contendo vários *while* sequenciais
- ✓ Programa `quick_sort.java` – Programa Quick Sort
- ✓ Programa `varios.java` – Programa contendo diversos dos testes anteriores juntos em um arquivo
- ✓ Programas `todos.java` – Programa contendo teste para verificar diversas características do compilador

Como esta máquina executava apenas programas em ambiente sequenciais, a avaliação dos resultados considerou o ganho conseguido apenas com a otimização efetuada pelo compilador Japar. Foram medidos os tempos que um determinado programa utilizava executado utilizando algoritmo simples (Ts) e o mesmo programa utilizando um algoritmo

otimizado (To). O programa simples foi executado em uma máquina Java tradicional enquanto que o programa otimizado Java foi executado na MVJP.

Uma importante observação deve ser feita quanto ao aspecto de otimizado. Como a MVJP foi executada em um ambiente monoprocessador, o fator medido reporta-se à uma medida de ganho de otimização, já que o potencial paralelo não pode ser explorado realmente em ambientes deste tipo. Os ganhos observados no gráfico analisado a seguir, referem-se a otimização. Apesar de muitos dados terem sido gerados, principalmente na fase de desenvolvimento, selecionou-se apenas alguns mais representativos, com o objetivo de ilustrar esses dados.

Entre os diversos testes realizados obteve-se resultados em: execução de laço simples (comando *for*), execução de laço simples (comando *while*), execução de laço duplo (comando *for*), execução de laço duplo (comando *while*), execução de vários laços (comando *for*), execução de vários laços (comando *while*), execução do programa Quick Sort, execução do programa Vários, e finalmente execução do programa Todos

O teste mostrado na Figura 5.1 refere-se ao programa com apenas um comando *for*, apresentou uma otimização com o laço *parfor* de aproximadamente 22% do programa otimizado (To) em relação ao programa sequencial (Ts). Foi feito também um teste para o mesmo caso aplicando o laço *parwhile*, sendo que o ganho conseguido com esse laço foi de 17%. Este fato pode indicar um algoritmo de paralelização inferior, ou que o programa aplicado permite uma menor otimização. O comportamento geral da otimização dos dois laços mostrou bastante regularidade, onde o ganho seguiu um perfil semelhante para os dois casos. Nos testes usando o comando *parfor* duplo e usando o comando *parwhile* duplo, pode-se notar a diferença na otimização foi ainda maior em favor ao algoritmo usando *parfor*. O primeiro representou um ganho de 25%, enquanto que o segundo representou um ganho que variou de 11% a 17%.

Figura 5.1 – Resultados da execução do programa *for* simples.

Os testes com laços aninhados o ganho para o comando *parfor* foi de 21% e o ganho do comando *parwhile* foi de 24%. Este fato pode indicar que o algoritmo usado com vários laços aninhados foi melhor explorado pelo algoritmo de paralelização.

O programa QuickSort, com um gráfico visto na Figura 5.2, mostrou uma melhoria de até 30% com relação ao programa não otimizado. O comportamento do algoritmo acompanhou o resultado apresentado pelos programas anteriores.

No teste do programa Vários, o comportamento levemente oscilante do gráfico pode indicar que alguns efeitos foram melhor ou pior aproveitados pelo otimizador, sendo que o ganho foi em média entre 27 e 30%.

Figura 5.2 – Resultados da execução do programa *quicksort*

Deve-se ainda ressaltar que os demais programas de teste que foram realizados, apresentaram resultados similares aos vistos nas figuras 5.1 e 5.2, e portanto não foram relacionados neste capítulo.

5.5 Observações Finais

Ainda um outro breve teste para verificar o funcionamento da rede foi realizado de forma simplificada. Para realizar os testes do ambiente em rede remota, foi então necessário criar uma página eletrônica onde o acesso permitisse ao usuário efetuar um possível “*download*” das ferramentas necessárias para efetuar o uso do potencial paralelo. Para que o acesso a este *site* fosse uniforme, ao conectar-se ao endereço eletrônico, o usuário dispunha de duas opções: (1) Realizar o acesso ao site tradicional, sem uso da máquina virtual paralela, e neste caso o *applet* que era enviado era sequencial; (2) Acesso ao código Java Paralelo, neste caso a opção para efetuar *download* da máquina virtual (MVJP), do carregador de classes especial e demais ferramentas era oferecido ao usuário. Esta máquina deve substituir a máquina Java existente.

Desta forma, acredita-se, o *site* mantém sua compatibilidade, permitindo assim que qualquer tipo de usuário pudesse acessar indistintamente um ou outro local da página eletrônica. As páginas desenvolvidas foram escritas em *html*. Somente após o pacote de elementos paralelos ter sido transferido e instalado o usuário podia acessar a seção de código paralelo.

Então um pequeno *applet* paralelo estava disponível para ser executado. A verificação desejada neste caso não incluiu nenhum tipo de avaliação de desempenho da rede, nem avaliação de performance de qualquer outro tipo, mas sim de funcionamento do mecanismo de transferência e uso da MVJP em ambientes seqüenciais. Um estudo que envolvesse a taxa de transmissão ou performance da rede está fora dos propósitos deste trabalho.

Capítulo 6

Conclusões e Perspectivas Futuras

6.1 Conclusões

A proposta deste trabalho nasceu com o estudo de ambientes paralelos e linguagens orientadas para objeto. Num primeiro instante objetivou-se estreitar as possibilidades de uso dessas duas disciplinas da computação. Surgiu daí a idéia de se aproveitar alguma linguagem orientada para objetos que também oferecesse características de ambientes paralelos. Nesta ocasião a linguagem Java havia sido lançada recentemente e com grande aceitação, e foi tomada como referência para o desenvolvimento que se propunha neste trabalho.

A busca por soluções utilizando a linguagem Java em ambientes paralelos encontrou algumas tentativas de projetos em andamento na literatura. Exemplos disso são o *javab* e o *javar*, apoiados pelo projeto DARPA e desenvolvidos na Universidade de Indiana, o projeto utilizando Java baseado em PVM, da Universidade de Alabama. Este fato indicou um caminho a ser seguido, visto que mais pesquisadores acreditaram na idéia de utilizar-se os conceitos de paralelismo em Java.

Os resultados deste trabalho indicaram que as soluções propostas podem representar uma alternativa para os gargalos do tráfego em rede. O compilador **Japar**, desenvolvido

neste trabalho apresenta um conjunto de recursos em termos de comandos que, se bem aproveitados podem apresentar um rendimento satisfatório. Observa-se, porém, que um bom conhecimento sobre o uso de paralelismo é fundamental para uma melhor utilização dos recursos propostos.

Os primeiros resultados foram conseguidos ainda durante a fase de desenvolvimento do ambiente paralelo, que foi feito em uma máquina com apenas um processador, em um ambiente sequencial. Foram medidos os tempos de execução para vários programas, considerando-se sempre a possibilidade de paralelismo, mas ainda nesta fase medindo-se apenas os ganhos conseguidos pela otimização. Usando a máquina MVJP, conseguiu-se um ganho de aproximadamente 22 % do programa otimizado (T_o) em relação ao programa sequencial (T_s) para a execução com o laço *parfor*. Com o laço *parwhile*, obteve-se um ganho semelhante, em torno de 17%. Nos testes usando o comando *parfor* duplo e usando o comando *parwhile* duplo, o primeiro representou um ganho de 25%, enquanto que o segundo apresentou ganhos que variaram de 11% a 17%. Esses resultados apontam para o fato de que cuidados no desenvolvimento de compiladores, no aspecto de otimização, nos oferecem boas possibilidades de melhoria do código gerado. Um caso que atesta este aspecto é o uso de compilador JIT, utilizado neste trabalho, que mostrou-se muito mais eficiente do que o Interpretador, o que nos leva a considerar que muitas otimizações ainda podem ser feitas e podem levar a um ganho satisfatório.

Os resultados, utilizando ambientes MPI, nos ofereceram uma medida de avaliação importante para que tivéssemos certeza do rendimento da máquina virtual proposta. A execução em um ambiente controlado e estável permitiu que separássemos os principais efeitos a serem medidos. Segundo os testes realizados na máquina Atm5, o *speedup* foi na ordem de 1.94, na relação entre um e dois processadores nesta máquina. Este valor é muito próximo ao valor conseguido nesta máquina com o uso do MPI, que apresentou um *speedup* de aproximadamente 2.00. Este fato indica que algumas melhorias ainda podem ser conseguidas pelo ambiente Japar/MVJP. Entre as possíveis melhorias encontram-se um aprimoramento no algoritmo de resincronização dos processos, e talvez, uma forma de melhor distribuir-se os processos. A máquina MVJP encontra-se em fase de teste, atualmente, em mais de um computador.

Um dos produtos do presente trabalho foi a construção de um pré-compilador ou conversor denominado **ConJava** que, por se tratar de uma ferramenta automatizada, pode oferecer algumas vantagens para usuários pouco experientes em construções paralelas. Assim, esse recurso pode oferecer a possibilidade de mais pessoas utilizarem um recurso, por vezes, restrito a especialistas. Apesar do principal enfoque não ter sido dado a esta ferramenta, ela surge como uma possível aliada no treinamento de usuários do ambiente Java paralelo.

Criou-se um conjunto de novas instruções agregadas à linguagem Java com vários objetivos. Um deles refere-se a utilizar todos os recursos oferecidos pela linguagem Java, já testados e aprovados, e acrescentar a esses, novas potencialidades. A inserção de um novo comando aumenta o poder de decisão do programador, fazendo com que ele atue em níveis onde as ferramentas automatizadas não podem alcançar. A escolha ou não de uma sessão paralela faz com que o programador decida quais os caminhos que ele deseja priorizar em face do potencial de paralelização. Por exemplo, usar comandos tradicionais como *for* ou comandos paralelos *parfor* permite que o programador escolha quais os mecanismos que ele deseja que sejam paralelizados. Isso oferece, maior flexibilidade na construção do algoritmo.

O principal produto deste trabalho foi a construção do compilador Japar, juntamente com o desenvolvimento da MJVP. A construção deste compilador foi realizada com o apoio da ferramenta YACC. Isso garantiu que o compilador fosse desenvolvido de forma sólida e confiável, pois aliando-se aos recursos disponíveis pelo YACC, o tempo de desenvolvimento ficou bastante reduzido. O uso dessa ferramenta permitiu que fossem verificados alguns possíveis aspectos de ambigüidade, aspectos esses que sempre causam grandes preocupações em arquitetos de compiladores. A existência de algumas ferramentas específicas como o Jlex, testados durante o desenvolvimento, mostram como o uso dessas ferramentas tem crescido e se tornado comuns na geração de compiladores.

Um aspecto importante na implementação desta proposta foi o fato de termos tido acesso a informações a nível de código original do desenvolvimento da linguagem Java. Neste sentido, o projeto pôde alcançar um razoável nível de implementação, onde uma importante contribuição foi explorar, dentro de um ambiente já sólido, os aspectos de

paralelismo não observados ainda. Isso permitiu uma abordagem de implementação até então não tentada.

Acreditamos que este trabalho tenha oferecido algumas contribuições, entre as quais enfocar a abordagem paralela em uma linguagem relativamente recente e que vem ganhando muitos defensores. Se logo neste início for possível implantar um conceito paralelo na linguagem Java, isso deve no futuro demonstrar sua utilidade e com isso propiciar que um grande número de usuários possam fazer uso desses recursos.

6.3 Perspectivas Futuras

De forma geral, considera-se que o projeto proposto atingiu plenamente as expectativas iniciais, fornecendo assim uma promissora linha de pesquisa para o desenvolvimento de trabalhos futuros. Os desafios iniciais de desenvolvimento do compilador e da máquina virtual foram superados e assim, uma nova fase de refinamentos pode ser iniciada. Observa-se, porém, que muitas avaliações ainda devem ser feitas até que a proposta possa ser considerada definitivamente viável. A máquina virtual construída encontra-se pronta para que sejam efetuados outros estudos, usando outras arquiteturas paralelas.

Algumas possibilidades de paralelismo foram analisadas e utilizadas em profundidade e fizeram parte da proposta original deste trabalho. Outras porém não foram consideradas. Um trabalho futuro deveria considerar as demais possibilidades e propor novas soluções que não foram observadas no presente trabalho, e também melhorias dessas propostas. Não foi considerada também nenhuma abordagem orientada para objeto além daquelas já previstas pela linguagem Java. Um trabalho interessante a ser desenvolvido poderia explorar as implicações específicas desses conceitos, com a atenção voltada para o processamento distribuído com objetos [Gri93]

Um fator que pode ser considerado em trabalhos futuros seria estudar uma proposta para a programação multiparadigma. A programação multiparadigma é a integração de vários paradigmas de programação em um simples modelo. Isto permite o entendimento de várias formas de computação como partes de um fenômeno simples. A programação

multiparadigma engloba a codificação de algoritmos em um estilo natural, podendo ser o paradigma funcional, lógico ou o orientado a objeto, e, dependendo de como as funções são avaliadas, parte da informação é usada, ou uma manipulação de estado é necessária.

O desempenho em computadores paralelos é igualmente importante de ser verificado. Como os critérios de avaliação, podem ser considerados os acessos à disco, memória, atividades de E/S, tempo de compilação, uso do sistema operacional e tempo de CPU.

Muitos itens não foram ainda implementados na atual versão do projeto, entre os quais o conjunto de caracteres *Unicode* no compilador Japar. Os aspectos de segurança de Java não previstos no compilador Japar são outros aspectos a serem analisados, ainda que este fator deva ser considerado junto aos conjunto de ferramentas oferecidas pelo desenvolvedor original Java (SUN).

Apesar de alguns testes já terem sido realizados, fica claro que é necessário explorar a MVJP em outros ambientes multiprocessados, com mais processadores, e mais interligações, e ainda outros tipos de arquiteturas. Deve fazer parte ainda da avaliação da presente ferramenta, efetuar um conjunto de testes mais amplo para o compilador **japar**, e ainda melhorar o conversor **conjjava**.

Finalmente, uma caminho a ser seguido, é o estudo do comportamento deste ambiente em redes especiais, tais como a rede ATM, que já está disponível no grupo onde este trabalho foi desenvolvido.

Bibliografia

- [Ada98] Adam, J. F., Sunderam, V.S. “*Multiparadigm Distributed Computing with TPVM*”, Concurrency, Practice and Experience, disponível em <http://www.cs.virginia.edu/~ajf2/jpvm.ht>, 1998.
- [And98] Adam, J. F., “*JPVM: Network Parallel Computing in Java*”, disponível por ferrari@cs.virginia.edu. University of Virginia, Charlottesville, VA 22903, USA, 1998.
- [And95] Anderson T.E., Culler, D.E. and Patterson, D.A., “*A Case for NOW (networks of Workstations)*,” *IEEE Micro*, vol.15, no. 1, pp. 54-64, February, 1995.
- [Aho86] Aho, A. V., Sethi, R.; Ullman, J.D. “*Compilers Principles, Techniques and Tools*”, Addison-Wesley, 1986.
- [Alg94] Al Geist and Beguelin, A. “*Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*”, The MIT Press, 1994.
- [App97] Appel, A.W. “*Modem Compiler Implementation in C*”. Cambridge University Press, NewYork, 1997.

- [Bal89a] Bal D., Steiner, C., Tanenbaum A., "*Programming Languages for Distributed Systems*". Computer Surveys, 1989.
- [Bal89b] Bal, H.E. et alii., "*Programming Languages for Distributed Computing Systems*" ACM Computing Surveys, 21(3): 261-322, 1989.
- [Bal92] Bal, H.E. et alii., "*ORCA: A Language for Parallel Programming for Distributed Systems*". IEEE Trans. on Software Engineering, 18(3), 1992.
- [Ban97] Banerjee, U. "*Dependence Analysis*". Kluwer, A Book Series on Loop Transformations for Restructuring Compilers, Boston, 1997.
- [Bau92] Bauer, B.E. "*Practical Parallel Programming*", Academic Press, San Diego, 1992.
- [Bel94] Bell, G. "*Scalable Parallel Computers: Alternative, Issues and Challenges*". Intl. Journal of Parallel Programming, 22(1):3-47, 1994.
- [Ben90] Ben-Ari, M. "*Principles of Concurrent and Distributed Programming*". Cambridge, Prentice-Hall, 1990.
- [Ber91] Bershad, B.N. & Zekauskas, M.J. "*Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Shared Memory Multiprocessors*", Technical report CMU-CS-91-170, School of Computer Science, Carnegie-Mellon University, 1991.
- [Berg94] Berguelin, A.; Dongarra, J. et al. "*HeNCE: A User's Guide - Version 2.0*", disponível por: hence@cs.utk.edu, 1994.
- [Bik97a] Bik, A.J-C. and Gannon, D.B., "*Automatically Exploiting Implicit Parallelism in Java*". Concurrency, Practice and Experience, 9(6):579A19, 1997.
- [Bik97b] Bik, A.J-C. and Gannon, D.B., "*A Prototype Bytecode Parallelization Tool*". Technical Report 489, Computer Science Department, Indiana University, disponível em: <http://www.extreme.indiana.edu/hpjava/>, 1997.

- [Bik97c] Bik, A.J-C. and Gannon, D.B., "*Javar - A Prototype Java Restructuring Compiler*". Technical Report 487, Computer Science Department, Indiana University, disponível em: <http://www.extreme.indiana.edu/hpjava/>), 1997.
- [Bud97] Budimlic, Z. and Kennedy, K., "*Optimizing Java - Theory and Practice*". Concurrency, Practice and Experience, 9(6):445-463, 1997.
- [Cap97] Cappello, P., Christiansen, B.O., Ionescu, M.F., Neary, M.O., Schauser, K.O. and Wu, D., "-Javelin: Internet-Based Parallel Computing Using Java". Department of Computer Science University of California, Santa Barbara, 1997.
- [Car90] Carriero, N. and Gelernter, D., "*How to Write Parallel Programs: A First Course*". The MIT Press, 1990.
- [Car96] Cardoso, C., "*HTML - Programação de Home Pages na Internet*". Axcel Books do Brasil, Rio de Janeiro, 1996.
- [Car97] Carpenter, B. et alii., "*Experiments with 'HP Java'*". Concurrency, Practice and Experience, 9(6):633-648, 1997.
- [Ces98] Cesta, A.A., "*Tutorial: A Linguagem de Programação Java*", disponível em <http://www.dcc.unicamp.br/~cmrubira/aacesta/java/javatut.html>, 1998.
- [Cha90] Chandra, J.B. et al., "*COOL: a Language for Parallel Programming*". In: Programming Languages and Compilers for Parallel Computing, MIT Press, England, 1990.
- [Cha94] Chandra, J.B. et alii., "*COOL: An Object-based Language for Parallel Programming*". IEEE Computer. p.13-26, 1994.
- [Cho94] Cho, S. and Bergmark, D.; "*HeNCE 1.4 - Lab Exercises*"; LNCC/DCE, disponível em: <http://www.lncc.br/hence/hence.html>, 1994.
- [Chuc98] McManis, Chuck "*Just - In Time Compilation*" - artigo para a Digital Cat, LLC 953 Industrial Avenue, Suite 125 - Palo Alto, California 94303 disponível através do e-mail webmaster@Javacats.com.

- [Cie97] Cierniak, M. and Li, W.A., "*Optimizing Java Bytecodes*". *Concurrency, Practice and Experience*, 9(6):427-444, 1997.
- [Cor98] Cornell, G.; Horstmann, C.S., "*Core Java*", Makron Books, 1998.
- [Duk93] Duke, D.W; Elias, D.; Livny, M. and Turcotte, L., "*Clustered Workstations Environments*". In: *Supercomputing, Portland, Tutoriais*. Los Alamitos, IEEE Computer Society Press, 1993.
- [Dun90] Duncan, R., "*A Survey of Parallel Computer Architectures*". *IEEE Computer*, 23(2):5-16, 1990.
- [Fac98] Facco M. D., "*Linguagens de Programação*", disponível em: <http://dinf.unicruz.tche.br/~facco>, 1998.
- [Fis88] Fischer, C.N. and LeBlanc, R.J., "*Crafting a Compiler*". Benjamin-Cummings, Menlo Park, California, 1988.
- [Fly66] Flynn, M.J., "*Very high-speed computing system*". *Proc. IEEE* 54(12):1901-1909, 1966.
- [Fos95] Foster, I., "*Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*". Addison Wesley, 1995.
- [Fox92] Fox, G. et alii., "*Fortran D Language Specification*". Department of Computer Science, Rice University, 1992.
- [Fox97] Fox, G.C. and Furmanski, W., "*Java for Parallel Computing and as a General Language for Scientific and Engineering Simulation and Modeling*". *Concurrency, Practice and Experience*, 9(6):415-425, 1997.
- [Geh86] Gehani, N.H.; Roome, W.D., "*Concurrent C*". *Software Practice and Experience*, 16(9):821-44., 1986.
- [Geh88] Gehani, N.H.; Roome, W.D. "*Rendezvous Facilities: Concurrent C and Ada Language*". *IEEE Trans. on Software Engineering*, vol. 14, no.11, 1988.
- [Geh92] Gehani, N.H.; Roome, W.D., "*Implementing Concurrent C*". *Software Practice and Experience*, 22(3):267-85, 1992.

- [Gei94] Geist, A.; Beguelin, A.; Dongarra, J., "*PVM: Parallel Virtual Machine*," MIT Press, 1994.
- [Gel90] Gelernter, D. and Jagannathan, S., "*Programming Linguistics*". The MIT Press, 1990.
- [Gel91a] Gelernter, D., "*Mirror Worlds: or The Day Software Puts the Universe in a Shoebox... How it Will Happen and What it Will Mean*". Oxford University Press, 1991.
- [Gel91b] Nicolau, A., Gelernter, D., Gross, T. and Padua, D. (editors), "*Advances in Languages and Compilers for Parallel Computing*". The MIT Press, 1991.
- [Gos96] Gosling, J.; Joy, B.; Steele, G., "*Java Programming Language*". Addison-Wesley, Reading, Massachusetts, 1996.
- [Gra92] Grant, B.K.; Skellum, A., "*The PVM System: An In-depth Analysis and Documenting Study-concise Edition*". Numerical Mathematics Group, Lawrence Livermore National Laboratory, 1992.
- [Gra97] Gray, P.A.; V.S. Sunderam, "IceT: Distributed Computing and Java," 1997 disponível em <http://www.mathcs.emory.edu/~gray/abstract7.html>.
- [Gri93] Grimshaw A.S., "*Easy-to-use Object-oriented Parallel Processing with Mental*". IEEE Computer, v.26, n.5, p.39-51, 1993.
- [Gue94] Gueist, A. et alii., "*PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*". Cambridge, MIT Press, 1994.
- [Gus86] Guzzi, M.D., "*Multitasking Runtime Systems for the CEDAR Multiprocessor*". Technical Report CSRD n.604, Center for Supercomputing Research and Development, University of Illinois at Urbana Champaign, 1986.
- [Har91] Harrison, L.; Ammarguelat, Z., "*A Comparison of Automatic Versus Manual Parallelization of the Boyer-Moore Theorem Prover*". In: Languages and Compilers for Parallel Computing. Pitman Publishing, 1991.

- [Henn95] Hennessy J. L. and Patterson D. A., "*Computer Architecture: A Quantitative Approach*", 2nd Edition, pg. 124-161, 1995.
- [Hof96] Hoff, A. V.; Shaio, S.; Starbuck, O., "*Ligado em Java*". São Paulo, Makron Books, 1996.
- [Hol95] Holmes, J., "*Object_Oriented Compiler Construction*", Prentice Hall, 1995.
- [Kel92] Keleher, P. et alii., "*Lazy Release Consistency for Software Distributed Shared Memory*". Department of Computer Science, Rice University, 1992.
- [Kel93] Keleher, P. et alii., "*Treadmarks: Distributed Shared Memory on Standard Workstations and Operating Systems*". Technical Report Rice COMP TR-93-214. Department of Computer Science, Rice University, 1993.
- [Ken97] Kent, P. and Kent. J., "*JavaScript para Netscape: Guia de Programação para a Criação de Páginas Web Interativas*". Makron Books, São Paulo, 1997.
- [Ker84] Kernighan, B.W., "*The UNIX Programming Environment*". Prentice Hall, Englewood Cliffs, New Jersey, 1984.
- [Ker88] Kernighan, B.W. and Ritchie, D.M., "*The C Programming Language*". Prentice Hall, 1988.
- [Kof94] Kofuji, S.T., "*Considerações de Projeto e Análise do SPADE: um Multiprocessador de Larga Escala Baseado no Padrão ANSI/IEEE SCI*". Tese de doutorado. Escola Politécnica da Universidade de São Paulo, 1994.
- [Kum94] Kumar, V. et alii., "*Introduction to Parallel Computing*". The Benjamin Cummings Publishing Company Inc. (sinopse), 1994.
- [Kus94] Kuskin, J. et alii., "*The Stanford Flash Multiprocessor*". Computer Systems Laboratory, Stanford University, 1994.
- [Lea97] Lea, D., "*Concurrent Programming in Java*". Addison-Wesley, Reading, Massachusetts, 1997.
- [Len92] Lenoski, D., "*The Stanford DASH Multiprocessor*". IEEE Computer, 25(3):63-79, March 1992.

- [Lin96] Lindholm, T. and Yellin, F., "*The Java Virtual Machine Specification*". Addison-Wesley, Reading, Massachusetts, 1996.
- [Lin97] Lindholm, T. and Yellin, F., "*The Java Virtual Machine Specification*". The Java Series, Addison Wesley, 1997.
- [Lov93] Loverman, D.B., "*High Performance Fortran*". IEEE Parallel and Distributed Technology, Feb., 1993.
- [McDy-95] McDysan, David E.; Spohn, D.L., "*ATM Theory and Application*", McGraw-Hill, 1995
- [Mey97] Meyer, J. and Downing, T., "*Java Virtual Machine*". O'Reilly & Associates, Sebastopol, CA, 1997.
- [Mey98] Meyer, J., "*Jasmin: a Java Assembler Interface*", JVM bytecode assembler disponível em: <http://www.cat.nyu.edu/meyer/jasmin/>, 1998.
- [Mid87] Midkiff, S.P. and Padua, D.A. "*Compiler Algorithms for Synchronization*". IEEE Transactions on Computers, C-36:1485-1495, 1987.
- [Mid95] Midorikawa, E.T.; Sato, L.M., "*Integrando as Otimizações de Acesso a Dados e Paralelismo*". In: Simpósio Brasileiro de Arquitetura de Computadores - Processamento de Alto Desempenho, 7, Canela, RS. Anais, p.49-60, 1995.
- [Nit91] Nitzberg, B.; Lo, V., "*Distributed Shared Memory: A Survey of Issues and Algorithms*". IEEE Computer, pp. 52-60, August 1991.
- [Oak97] Oaks, S. and Wong, H., "*Java Threads*". O'Reilly & Associates, Sebastopol, CA, 1997.
- [Par92] Parsons, T.W., "*Introduction to Compiler Construction*". Computer Science Press, New York, 1992.
- [Patt96] Patterson, David A., "*Computer Architecture – A Quantitative Approach*", Second Edition, Morgan Kaufmann, pp 76-111, 1996.
- [Pol88] Polychronopoulos, C.D., "*Parallel Programming and Compilers*". Kluwer, Boston, 1988.

- [Pol88] Poychronopoulos, C.D., *"Parallel Programming and Compilers"*. Kluwer Academic Publishers, 1988.
- [Pon94] Pontelli, E. et alii., *"A High-Performance Parallel Prolog System"*. Laboratory for Logic and Databases, Department of Computer Science, New Mexico University, 1994.
- [Pug92] Pugh, W., *"The Omega Test: a Fast and Practical Integer Programming Algorithm for Dependence Analysis"*, Communications of the ACM, 1992.
- [Rin93] Rinard, M.C. et alii., *"JADE: a High-Level, Machine-independent Language for Parallel Programming"*. IEEE Computer, 26(6):28-38, 1993.
- [Roc98] Rocha L. S. H., *"Recursos Java"*, disponível em: http://www.dsc.ufpb.br/~helder/java/Cap_1.html#footnotes, 1998.
- [Sar97] Sarmenta, L. F. G., Hirano, S., Ward, S. A., *"Towards Bayanihan: Building anExtensible Framework for Volunteer Computing Using Java"*, ACM Workshop on Java for High-Performance Network Computing, 1997.
- [Sat92] Sato, L.M., *"Um Sistema de Programação e Processamento para Sistemas Multiprocessadores"*. In: Simpósio Brasileiro de Arquitetura de Computadores - Processamento de Alto Desempenho, 4, São Paulo, SP. Anais, p.95-107, 1992.
- [Sat96] Sato, L.M.; Midorikawa, E. T., *"Introdução à programação paralela e distribuída"*. Apostila disponível em: <http://www.lsi.usp.br/~liria/sat96/apost/tudo.html>, 1992.
- [Sca93] Scales, D.J.; Lam, M.S., *"A Flexible Shared Memory System for Distributed Memory Machines"*. Computer Systems Laboratory, Stanford University, 1993.
- [Tha88] Thakkar, S., *"The Balance Multiprocessor System"*. IEEE Micro, v.8, no.1, p.57-63.
- [Thu97] Thurman, D., *"JavaPVM, An Interface to Parallel Virtual Machine for Java"*- Georgia Intitute of Technology - disponível em: <http://www.isye.gatech.edu/chmsr/JavaPVM/>.

- [Tur93] Turcotte, L.H. *"A Survey of Software Environment for Exploiting Networked Computing Resources"*. Engineering Research Center for Computational Field Simulation, Mississippi State, EUA, 1993.
- [Tra94] Traina, A. F., "Proposta e Construção de um, Compilador Pascal para Arquitetura RISC-LIE". Dissertação de Mestrado, IFSC- USP, 1994.
- [Uta95] University of Utah. *"Application Programming with Quarks"*. Computer Systems Laboratory, 1995.
- [Van98] Van der Linden, P., *"Just Java"*, Makron Books, 1998.
- [Ven98] Venners, B., *"Inside the Java Virtual Machine"*. Java Masters, McGraw-Hill, 1998.
- [Yel96] Yellin, F. and Lindholm, T., *"The Java Virtual Machine Specification"*. Addison-Wesley, Reading, Massachusetts, 1996.
- [Wol96] Wolfe, M.J., *"High Performance Compilers for Parallel Computers"*. Addison-Wesley, Redwood City, California, 1996.
- [Zim90] Zima, H. and Chapman, B. *"Supercompilers for Parallel and Vector Computers"*, ACM Press, New York, 1990.

Endereços eletrônicos acessados

Sites: Computação Paralela

MPI MPI - Message Passing Interface

<http://www.erc.msstate.edu/labs/hpcl/projects/mpi/>

MPI - The Message Passing Interface Standard

<http://www-unix.mcs.anl.gov/mpi/index.html>

PVM PVM Parallel Virtual Machine

http://www.epm.ornl.gov/pvm/pvm_home.html

BSP Ravi Palepu

<Http://www.scs.carleton.ca/~papel/BSP.html>

FTP BSP (at ftp.comlab.ox.ac.uk)

ftp://ftp.comlab.ox.ac.uk/pub/Packages/BSP/bsplib_contrib/

The World Wide Web Virtual Library Concurr.

<http://archive.comlab.ox.ac.uk/concurrent.html>

Linguagens

Linda

<http://www.stat.ucla.edu/develop/web/languages/linda/>

<http://www.cs.yale.edu/HTML/YALE/CS/Linda/linda.html>

High Performance Fortran

<http://www.nsc.liu.se/~boein/f77to90/a8.html>

High Performance Fortran (HPF)

<http://www.netlib.org/hpf/>

High Performance Fortran Forum Home Page

<http://www.crpc.rice.edu/HPFF/home.html>

Compositional C++

http://csvax.cs.caltech.edu/comp/ccpp/lang_def/def.html

Fortran 95

<http://consult.cern.ch/cnl/222/node36.html>

ISO-IEC JTC1-SC22-WG5 (Fortran 95)

http://www.etrn.ox.sc.uk/WG%/IS1539-1_199x.html

Imagine1, Inc. Homepage - F

<http://kumo.swcp.com/imagine1/>

HPC++ Technologies for Object Parallel Progr.

<http://www.extreme.indiana.edu/hpc++/docs/arpa/hpccBB.html>

Europa Parallel C++ Archive

<http://www.lpac.ac.uk/europa/Articles/>

HPC Topics and Online Documentation

<http://kumo.swcp.com/imagine1/>

Simulação

an Execution-Driven Multiprocessor Simulation.

<http://galeb.etf.bg.ac.yu/~dav0r/limes/index.html>

Parallel Simulation Bookmarks

<http://www.cpsc.ucalgary.ca/~gomes/HTML/sim.html>

Internet Parallel Computing Archive

<http://www.hensa.ac.uk/parallel/index.html>

Parallel Cluster Algorithms

<http://www.npac.syr.edu/users/paulc/lectures/montecarlo/node74.html>

EuroPVM-MPI'97 Meeting

<http://www.cyf-kr.edu.pl/pvmpi97/>

Announcement Parallel Algorithms for Image E.

<http://www.ph.tn.tudelft.nl/PRInfo/reports/msg00098.html>

Parallel Programming Laboratory

<http://charm.cs.uiuc.edu/>

Workshop de Programação Paralela

<http://www.tc.cornell.edu/Edu/Workshop/>

Load balancing

http://www.uni-paderborn.de/fachbereich/AG/monien/RESEARCH/load_balancing.html

Sistema Condor

<http://www.cs.wisc.edu/condor/>

Parallel Virtual Machine

http://www.epm.ornl.gov/pvm/pvm_home.html

Message Passing Interface

<http://www.arc.unm.edu/workshop/mpi/mpi.html>

Upshot

<http://www.mcs.anl.gov/home/lusk/upshot/upshotman/upshot.html>

Pablo

<http://bugle.cs.uiuc.edu/Projects/Pablo/pablo.html>

PVMe:

<http://www.lncc.br/pvme/pvme.html>

Implementações disponíveis de MPI

<http://www.mcs.anl.gov/home/lusk/mpich/>

Link para a página do MPI - (The Maui Project/University of New Mexico)

<http://www.arc.unm.edu/workshop/mpi/mpi.html>

<http://www-unix.mcs.anl.gov/mpi/mpich/>

Sites relativos a Computação Paralela Distribuída

<http://eagle.cs.cuhk.hk/ors.html>

Algoritmos para Programação Paralela

<http://www.cs.cmu.edu/~scandal/cacm.html>

Programação Paralela

http://www.usi.utah.edu/user_guides/spug/spug/node44.html

10 razões para preferir MPI a PVM

http://www.osc.edu/Lam/mapi/mapi_top10.html

bibliografia sobre paralelismo

<http://www.inf.puc-rio.br/~noemi/victal/biblio.html#Linguagens>

Toba:

<http://www.cs.arizona.edu/sumatra/toba/doc/index.html>

Jasmine.

<http://members.tripod.com/~SourceTec/jasmine.htm>

DejaVu

<http://www.isg.de/OEW/Java/>

WingDis

<http://www.wingsoft.com/wingdis.shtml>

Sites para Mocha

<http://www.geocities.com/SiliconValley/Way/7588/>

<http://www.kcnet.com/~jbockh01>

<http://mats.gmd.de:8080/clemens/jacob/>

<http://www.Javasoftware.com>

Sun Microsystems. "The Java Platform".

<http://www.sun.com.br>

Tutorial Java, uma mudança no conceito de software. (1997).

<http://www.geocities.com/SiliconValley/Pines/8314/java.html>

Tutoriais.

<http://www.geocities.com/Yosemite/Gorge/4724/>, 1998.

Cesta, A.A. (1998). Tutorial: "A linguagem de programação Java".

<http://www.dcc.unicamp.br/~cmrubira/aacesta/java/javatut.html>

Sun Microsystems. (1998). The Java Platform.

<http://www.sun.com.br>

Rocha L.S. H. (1998). Recursos Java.

http://www.dsc.ufpb.br/~helder/java/Cap_1.html#footnotes

Tutoriais (1998)

<http://www.geocities.com/Yosemite/Gorge/4724/>

Facco M. D. (1998). Linguagens de Programação

<http://dinf.unicruz.tc.br/~facco>

Glossário Internet da Net DS. (1998)

<http://www.netds.com.br/portug/glossario.htm#client>

javab e javar:

<http://www.extreme.indiana.edu/~ajcbik/JAVAB/index.html>

Java

<http://home.com.kyungnam.ac.kr/~drkong/java/report/SmallJava/tools.html>

“Tutorial Java, Uma Mudança no Conceito de Software”.

<http://www.geocities.com/Silicon Valley/Pines/8314/java.htm>, 11 Jul 97.

Anexos

Apêndice A.1

Estrutura BNF da linguagem Java

```
arglist
 ::=
   expression { "," expression }
```

```
bit_expression
 ::=
   ( "~" expression )
 | ( expression
   ( "="
   | "<<"
   | ""
   | "" )
   expression )
```

```
casting_expression
 ::=
   "(" type ")" expression
```

```
character ::=
   "based on the unicode character set"
```

```
class_declaration
 ::=
   { modifier } "class" identifier
   [ "extends" class_name ]
   [ "implements" interface_name { "," interface_name } ]
   "{ " { field_declaration } "}"
```

```
class_name
 ::=
   identifier
   | ( package_name "." identifier )
```

```
compilation_unit ::=
    [ package_statement ]
    { import_statement }
    { type_declaration }
```

8.constructor_declaration

```
constructor_declaration
    ::=
    { modifier } identifier "(" [ parameter_list ] ")"
    statement_block
```

```
creating_expression
    ::=
    "new" ( ( classe_name "(" [ arglist ] ")" )
    | ( type_specifier [ "[" expression "]" ] { "[" "]" } )
    | ( "(" expression ")" ) )
```

```
decimal_digits
    ::=
    "0..9" { "0..9" }
```

```
doc_comment
    ::= "/*" "... text ..." "*/"
```

```
do_statement
    ::=
    "do" statement "while" "(" expression ")" ";"
```

```
exponent_part
    ::=
    "e" [ "+" | "-" ] decimal_digits
```

```
expression
    ::=
    numeric_expression
    | testing_expression
    | logical_expression
    | string_expression
    | bit_expression
    | casting_expression
    | creating_expression
    | literal_expression
    | "null"
    | "super"
    | "this"
    | identifier
    | ( "(" expression ")" )
```

```

| ( expression
| ( "(" [ arglist ] ")" )
| ( "[" expression "]" )
| ( "." expression )
| ( "," expression )
| ( "instanceof" ( class_name | interface_name ) )
) )

```

field_declaration

```

::=
( [ doc_comment ] ( method_declaration
| constructor_declaration
| variable_declaration ) )
| static_initializer
| ";"

```

float_literal

```

::=
( decimal_digits "." [ decimal_digits ] [ exponent_part ] [
float_type_suffix ] )
| ( "." decimal_digits [ exponent_part ] [ float_type_suffix ] )
| ( decimal_digits [ exponent_part ] [ float_type_suffix ] )

```

float_type_suffix

```

::=
"f" | "d"

```

for_statement

```

::=
"for" "(" ( variable_declaration | ( expression ";" ) | ";" )
[ expression ] ";"
[ expression ] ";"
")" statement

```

identifier

```

::= "a..z,$,_" { "a..z,$,_,0..9,unicode character over 00C0" }

```

if_statement

```

::=
"if" "(" expression ")" statement
[ "else" statement ]

```

```
import_statement
::=
"import" ( ( package_name "." "*" ";" )
| ( class_name | interface_name ) ) ";"
```

```
integer_literal
::=
( ( "1..9" { "0..9" } )
| { "0..7" }
| ( "0" "x" "0..9a..f" { "0..9a..f" } ) )
[ "l" ]
```

```
interface_declaration
::=
{ modifier } "interface" identifier
[ "extends" interface_name { "," interface_name } ]
"{ " { field_declaration } "}"
```

```
interface_name
::=
identifier
| ( package_name "." identifier )
```

```
literal_expression
::=
integer_literal
| float_literal
| string
| character
```

```
logical_expression
::=
( "!" expression )
| ( expression
( "ampersand"
| "ampersand="
| "|"
| "|="
| "^"
| "^="
| ( "ampersand" "ampersand" )
| "||="
```

```
| "?"  
| "%=" )  
expression )  
| ( expression "?" expression ":" expression )  
| "true"  
| "false"
```

```
method_declaration  
::=  
{ modifier } type identifier  
"(" [ parameter_list ] ")" { "[" "]" }  
( statement_block | ";" )
```

```
modifier  
::=  
"public"  
| "private"  
| "protected"  
| "static"  
| "final"  
| "native"  
| "synchronized"  
| "abstract"  
| "threadsafe"  
| "transient"
```

```
numeric_expression  
::=  
( ( "-"  
| "++"  
| "--" )  
expression )  
| ( expression  
( "++"  
| "--" ) )  
| ( expression  
( "+"  
| "+="  
| "-"  
| "-="  
| "*"  
| "*="  
| "/"  
| "/="  
| "%"  
| "%=" )  
expression )
```

```
package_name
    ::=
    identifier
    | ( package_name "." identifier )
```

```
package_statement
    ::=
    "package" package_name ";"
```

```
parameter
    ::=
    type identifier { "[" "]" }
```

```
parameter_list
    ::=
    parameter { "," parameter }
```

```
statement
    ::=
    variable_declaration
    | ( expression ";" )
    | ( statement_block )
    | ( if_statement )
    | ( do_statement )
    | ( while_statement )
    | ( for_statement )
    | ( try_statement )
    | ( switch_statement )
    | ( "synchronized" "(" expression ")" statement )
    | ( "return" [ expression ] ";" )
    | ( "throw" expression ";" )
    | ( identifier ":" statement )
    | ( "break" [ identifier ] ";" )
    | ( "continue" [ identifier ] ";" )
    | ( ";" )
```

```
statement_block
    ::= "{ { statement } }"
```

```
static_initializer
    ::=
        "static" statement_block

string
    ::= "'" { character } "'"

string_expression
    ::= ( expression
        ( "+"
        | "+=" )
        expression )

switch_statement
    ::=
        "switch" "(" expression ")" "{"
        { ( "case" expression ":" )
        | ( "default" ":" )
        | statement }
        "}"

testing_expression
    ::=
        ( expression
        ( ""
        | "<"
        | "="
        | "<="
        | "=="
        | "!=" )
        expression )

try_statement
    ::=
        "try" statement
        { "catch" "(" parameter ")" statement }
        [ "finally" statement ]
```

type

```
 ::=
type_specifier { "[" "]" }

type_declaration
 ::=
[ doc_comment ] ( class_declaration | interface_declaration ) ";"

type_specifier
 ::=
"boolean"
| "byte"
| "char"
| "short"
| "int"
| "float"
| "long"
| "double"
| class_name
| interface_name

variable_declaration
 ::=
{ modifier } type variable_declarator
{ "," variable_declarator } ";"

variable_declarator
 ::=
identifier { "[" "]" } [ "=" variable_initializer ]

variable_initializer
 ::=
expression
| ( "{" [ variable_initializer
{ "," variable_initializer } [ "," ] [ "]" ] )

while_statement
 ::=
"while" "(" expression ")" statement
```


Apêndice A.2

Arquivo de entrada para o YACC (ConJava)

```
%{
/* -----
|           Seccao de declaracoes em linguagem C           |
|----- */
#include "CONV.H"
#include <ctype.h>
#include <string.h>

valor_dado todovalor;
tipo_dado tipo;
int conta_linha;

%}
/* -----
|           Seccao de declaracoes do YACC           |
|----- */

/* -----
|           Pilha de elementos da Tabela de simbolos           |
|----- */
%union {
    /* tipo pilha */
    st_Symbol *sym; /* ponteiro para tabela de simbolos a serem
                    transcritos no arquivo de saida */
    int val;
}
/* -----
|           Declaracao de Tokens para o Analisador Lexico           |
|           Palavras Reservadas da linguagem Java           |
|----- */

%token ABSTRACT
```

```

%token BOOLEAN BREAK BYTE BYVALUE
%token CASE CAST CATCH CHAR CLASS CONST CONTINUE
%token DEFAULT DO DOUBLE
%token ELSE EXTENDS
%token FINAL FINALLY FLOAT FOR FUTURE
%token GENERIC GOTO
%token IF IMPLEMENTS IMPORT INNER INSTANCEOF INT INTERFACE
%token LONG
%token NATIVE NEW JNULL
%token OPERATOR OUTER
%token PACKAGE PRIVATE PROTECTED PUBLIC
%token REST RETURN
%token SHORT STATIC SUPER SWITCH SYNCHRONIZED
%token THIS THROW THROWS TRANSIENT TRY
%token VAR VOID VOLATILE
%token WHILE
%token OP_INC OP_DEC
%token OP_SHL OP_SHR OP_SHRR
%token OP_GE OP_LE OP_EQ OP_NE
%token OP_LAND OP_LOR
%token OP_DIM
%token ASS_MUL ASS_DIV ASS_MOD ASS_ADD ASS_SUB
%token ASS_SHL ASS_SHR ASS_SHRR ASS_AND ASS_XOR ASS_OR
%token IDENTIFIER LITERAL BOOLLIT

%start CompilationUnit

%%
/* -----
   |           Regras Sintaticas para o Conversor ConJava           |
   ----- */

TypeSpecifier
    : TypeName          { manter_sequencia($1) }
    | TypeName Dims      { manter_sequencia($1);
                          manter_sequencia($2)    }
    ;

TypeName
    : PrimitiveType     { manter_sequencia($1) }
    | QualifiedName     { manter_sequencia($1) }
    ;

ClassNameList
    : QualifiedName     { manter_sequencia($1) }
    | ClassNameList ',' QualifiedName { manter_sequencia($1);
                                      manter_sequencia(',');  manter_sequencia($3) }
    ;

PrimitiveType
    : BOOLEAN          { pol_var($1) }
    | CHAR              { pol_var($1) }
    | BYTE              { pol_var($1) }
    | SHORT             { pol_var($1) }
    | INT               { pol_var($1) }

```

```

| LONG      { pol_var($1) }
| FLOAT     { pol_var($1) }
| DOUBLE    { pol_var($1) }
| VOID      { pol_var($1) }
;

CompilationUnit
: ProgramFile { manter_sequencia($1) }
;

ProgramFile
: PackageStatement ImportStatements TypeDeclarations
  { manter_sequencia($1);manter_sequencia($2);
    manter_sequencia($3) }
| PackageStatement ImportStatements { manter_sequencia($1);
  manter_sequencia($2) }
| PackageStatement TypeDeclarations { manter_sequencia($1);
  manter_sequencia($2) }
| ImportStatements TypeDeclarations { manter_sequencia($1);
  manter_sequencia($2) }
| PackageStatement { manter_sequencia($1) }
| ImportStatements { manter_sequencia($1) }
| TypeDeclarations { manter_sequencia($1) }
;

PackageStatement
: PACKAGE QualifiedName ';' { manter_sequencia($1);
  manter_sequencia($2); manter_sequencia(";") }
;

TypeDeclarations
: TypeDeclaration { manter_sequencia($1) }
| TypeDeclarations TypeDeclaration { manter_sequencia($1);
  manter_sequencia($2) }
;

ImportStatements
: ImportStatement { manter_sequencia($1) }
| ImportStatements ImportStatement { manter_sequencia($1);
  manter_sequencia($2) }
;

ImportStatement
: IMPORT QualifiedName ';' { manter_sequencia($1);
  manter_sequencia(';') }
| IMPORT QualifiedName '.' '*' ';' { manter_sequencia($1);
  manter_sequencia($2);
  manter_sequencia('.');
  manter_sequencia('*');
  manter_sequencia(';') }
;

QualifiedName
: IDENTIFIER { manter_sequencia($1) }
| QualifiedName '.' IDENTIFIER { manter_sequencia($1);
  manter_sequencia('.');

```

```
manter_sequencia($3) }
```

```
;
```

TypeDeclaration

```
: ClassHeader '{' FieldDeclarations '}' { manter_sequencia($1);
                                         manter_sequencia($2);
                                         manter_sequencia($3);
                                         manter_sequencia($4) }
| ClassHeader '{' '}' { manter_sequencia($1);
                       manter_sequencia($2);
                       manter_sequencia($3) }
```

```
;
```

ClassHeader

```
: Modifiers ClassWord IDENTIFIER Extends Interfaces
  { manter_sequencia($1);
    manter_sequencia($2);
    manter_sequencia($3);
    manter_sequencia($4);
    manter_sequencia($5) }
| Modifiers ClassWord IDENTIFIER Extends
  { manter_sequencia($1);
    manter_sequencia($2);
    manter_sequencia($3);
    manter_sequencia($4) }
| Modifiers ClassWord IDENTIFIER Interfaces
  { manter_sequencia($1);
    manter_sequencia($2);
    manter_sequencia($3);
    manter_sequencia($4) }
| ClassWord IDENTIFIER Extends Interfaces
  { manter_sequencia($1);
    manter_sequencia($2);
    manter_sequencia($3);
    manter_sequencia($4) }
| Modifiers ClassWord IDENTIFIER
  { manter_sequencia($1);
    manter_sequencia($2);
    manter_sequencia($3) }
|
  ClassWord IDENTIFIER Extends
  { manter_sequencia($1);
    manter_sequencia($2);
    manter_sequencia($3) }
|
  ClassWord IDENTIFIER Interfaces
  { manter_sequencia($1);
    manter_sequencia($2);
    manter_sequencia($3) }
|
  ClassWord IDENTIFIER
  { manter_sequencia($1);
    manter_sequencia($2) }
```

```
;
```

Modifiers

```
: Modifier { manter_sequencia($1) }
| Modifiers Modifier { manter_sequencia($1);
```

```

                                manter_sequencia($2) }
;

Modifier
: ABSTRACT      { manter_sequencia($1) }
| FINAL         { manter_sequencia($1) }
| PUBLIC        { manter_sequencia($1) }
| PROTECTED     { manter_sequencia($1) }
| PRIVATE       { manter_sequencia($1) }
| STATIC        { manter_sequencia($1) }
| TRANSIENT     { manter_sequencia($1) }
| VOLATILE      { manter_sequencia($1) }
| NATIVE        { manter_sequencia($1) }
| SYNCHRONIZED  { manter_sequencia($1) }
;

ClassWord
: CLASS         { manter_sequencia($1) }
| INTERFACE     { manter_sequencia($1) }
;

Interfaces
: IMPLEMENTS ClassNameList { manter_sequencia($1);
                                manter_sequencia($2) }
;

FieldDeclarations
: FieldDeclaration          { manter_sequencia($1) }
| FieldDeclarations FieldDeclaration { manter_sequencia($1)
                                         manter_sequencia($2) }
;

FieldDeclaration
: FieldVariableDeclaration ';' { manter_sequencia($1);
                                manter_sequencia(';') }
| MethodDeclaration           { manter_sequencia($1) }
| ConstructorDeclaration      { manter_sequencia($1) }
| StaticInitializer           { manter_sequencia($1) }
| NonStaticInitializer        { manter_sequencia($1) }
| TypeDeclaration             { manter_sequencia($1) }
;

FieldVariableDeclaration
: Modifiers TypeSpecifier VariableDeclarators { manter_sequencia($1);
                                                manter_sequencia($2);
                                                manter_sequencia($3) }
}
|                                     TypeSpecifier VariableDeclarators { manter_sequencia($1);
                                                manter_sequencia($2) }
}
;

VariableDeclarators
: VariableDeclarator { manter_sequencia($1) }
| VariableDeclarators ',' VariableDeclarator { manter_sequencia($1);

```

```

manter_sequencia(',');
}
;

VariableDeclarator
    : DeclaratorName
    { manter_sequencia($1)
    }
    | DeclaratorName '=' VariableInitializer
    { manter_sequencia($1);
    }
manter_sequencia('=');
}
;

VariableInitializer
    : Expression
    {
    manter_sequencia($1)}
    | '{' '}'
    {
    manter_sequencia($1);
    }
manter_sequencia($2)}
    | '{' ArrayInitializers '}'
    {
    manter_sequencia($1);
    }
manter_sequencia($2);
manter_sequencia($3)}
;

ArrayInitializers
    : VariableInitializer
    { manter_sequencia($1)
    }
    | ArrayInitializers ',' VariableInitializer
    {
    manter_sequencia($1);
    }
manter_sequencia(',');
}
    | ArrayInitializers ','
    {
    manter_sequencia($1);
    }
manter_sequencia($2)}
;

MethodDeclaration
    : Modifiers TypeSpecifier MethodDeclarator Throws MethodBody
    { manter_sequencia($1); manter_sequencia($2);
      manter_sequencia($3); manter_sequencia($4);
      manter_sequencia($5) }
    | Modifiers TypeSpecifier MethodDeclarator MethodBody
    { manter_sequencia($1); manter_sequencia($2);
      manter_sequencia($3); manter_sequencia($4) }
    |
    TypeSpecifier MethodDeclarator Throws MethodBody

```

```

        { manter_sequencia($1); manter_sequencia($2);
          manter_sequencia($3); manter_sequencia($4) }
      |
        TypeSpecifier MethodDeclarator MethodBody
        { manter_sequencia($1); manter_sequencia($2);
          manter_sequencia($3) }
      ;

MethodDeclarator
  : DeclaratorName '(' ParameterList ')' { manter_sequencia($1)
    manter_sequencia('('); manter_sequencia($3);
    manter_sequencia(')') }
  | DeclaratorName '(' ')' { manter_sequencia($1);
    manter_sequencia('('); manter_sequencia(')') }
  | MethodDeclarator OP_DIM { manter_sequencia($1);
    manter_sequencia($2) }
  ;

ParameterList
  : Parameter { manter_sequencia($1) }
  | ParameterList ',' Parameter { manter_sequencia($1)
    manter_sequencia(','); manter_sequencia($3) }
  ;

Parameter
  : TypeSpecifier DeclaratorName { manter_sequencia($1);
    manter_sequencia($2) }
  ;

DeclaratorName
  : IDENTIFIER { manter_sequencia($1) }
  | DeclaratorName OP_DIM { manter_sequencia($1);
    manter_sequencia($2) }
  ;

Throws
  : THROWS ClassNameList { manter_sequencia($1);
    manter_sequencia($2) }
  ;

MethodBody
  : Block { manter_sequencia($1) }
  | ';' { manter_sequencia(';') }
  ;

ConstructorDeclaration
  : Modifiers ConstructorDeclarator Throws Block
    { manter_sequencia($1);
    manter_sequencia($2);
    manter_sequencia($3);
    manter_sequencia($4) }
  | Modifiers ConstructorDeclarator Block
    { manter_sequencia($1); manter_sequencia($2);
    manter_sequencia($3) }
  | ConstructorDeclarator Throws Block
    { manter_sequencia($1); manter_sequencia($2);

```

```

        manter_sequencia($3)}
    |
        ConstructorDeclarator Block
        { manter_sequencia($1); manter_sequencia($2) }
    ;

ConstructorDeclarator
    : IDENTIFIER '(' ParameterList ')' { manter_sequencia($1)
        manter_sequencia('('); manter_sequencia($3);
        manter_sequencia(')') }
    | IDENTIFIER '(' ')' { manter_sequencia($1);
        manter_sequencia('('); manter_sequencia(')') }
    ;

StaticInitializer
    : STATIC Block { manter_sequencia($1);
        manter_sequencia($2) }
    ;

NonStaticInitializer
    : Block { manter_sequencia($1) }
    ;

Extends
    : EXTENDS TypeName { manter_sequencia($1);
        manter_sequencia($2) }
    | Extends ',' TypeName { manter_sequencia($1);
        manter_sequencia(','); manter_sequencia($3) }
    ;

Block
    : '{' LocalVariableDeclarationsAndStatements '}'
        { manter_sequencia('{'); manter_sequencia($2);
        manter_sequencia('}') }
    | '{' '}' { manter_sequencia('{');
        manter_sequencia('}') }
    ;

LocalVariableDeclarationsAndStatements
    : LocalVariableDeclarationOrStatement
        { manter_sequencia($1) }
    | LocalVariableDeclarationsAndStatements
LocalVariableDeclarationOrStatement
    { manter_sequencia($1); manter_sequencia($2) }
    ;

LocalVariableDeclarationOrStatement
    : LocalVariableDeclarationStatement { manter_sequencia($1) }
    | Statement { manter_sequencia($1) }
    ;

LocalVariableDeclarationStatement
    : TypeSpecifier VariableDeclarators ';' { manter_sequencia($1);
        manter_sequencia($2); manter_sequencia(';') }
    ;

```



```

Statement
    : EmptyStatement          { manter_sequencia($1) }
    | LabeledStatement        { manter_sequencia($1) }
    | ExpressionStatement ';' { manter_sequencia($1);
                                manter_sequencia(';') }
    | SelectionStatement      { manter_sequencia($1) }
    | IterationStatement      { manter_sequencia($1) }
    | JumpStatement           { manter_sequencia($1) }
    | GuardingStatement       { manter_sequencia($1) }
    | Block                   { manter_sequencia($1) }
    ;

EmptyStatement
    : ';'                      { manter_sequencia(';') }
    ;

LabeledStatement
    : IDENTIFIER ':' LocalVariableDeclarationOrStatement
      { manter_sequencia($1); manter_sequencia(':');
        manter_sequencia($3) }
    | CASE ConstantExpression ':' LocalVariableDeclarationOrStatement
      { manter_sequencia($1); manter_sequencia($2);
        manter_sequencia(':'); manter_sequencia($4) }
    | DEFAULT ':' LocalVariableDeclarationOrStatement
      { manter_sequencia($1); manter_sequencia(':');
        manter_sequencia($3) }
    ;

ExpressionStatement
    : Expression              { manter_sequencia($1) }
    ;

SelectionStatement
    : IF '(' Expression ')' Statement
      { manter_sequencia($1); manter_sequencia('(');
        manter_sequencia($1); manter_sequencia(')');
        manter_sequencia() }
    | IF '(' Expression ')' Statement ELSE Statement
      { manter_sequencia($1); manter_sequencia('(');
        manter_sequencia($3); manter_sequencia(')');
        manter_sequencia($5); manter_sequencia($6);
        manter_sequencia($7) }
    | SWITCH '(' Expression ')' Block
      { manter_sequencia($1); manter_sequencia('(');
        manter_sequencia($3); manter_sequencia(')');
        manter_sequencia($5) }
    ;

/*      Parte do otimizador */

IterationStatement
    : WHILE '(' Expression ')' Statement
    | DO Statement WHILE '(' Expression ')' ';'
    | FOR '(' ForInit ForExpr ForIncr ')' Statement
    | FOR '(' ForInit ForExpr          ')' Statement
    ;

```

```

ForInit
    : ExpressionStatements ';'
    | LocalVariableDeclarationStatement
    | ';'
    ;

ForExpr
    : Expression ';'
    | ';'
    ;

ForIncr
    : ExpressionStatements
    ;

ExpressionStatements
    : ExpressionStatement { manter_sequencia($1) }
    | ExpressionStatements ',' ExpressionStatement
      { manter_sequencia($1); manter_sequencia(',');
        manter_sequencia($3) }
    ;

JumpStatement
    : BREAK IDENTIFIER ';'
      { manter_sequencia($1); manter_sequencia(';') }
    | BREAK ';'
      { manter_sequencia($1); manter_sequencia(';') }
    | CONTINUE IDENTIFIER ';'
      { manter_sequencia($1); manter_sequencia(';') }
    | CONTINUE ';'
      { manter_sequencia($1); manter_sequencia(';') }
    | RETURN Expression ';'
      { manter_sequencia($1); manter_sequencia(';') }
    | RETURN ';'
      { manter_sequencia($1); manter_sequencia(';') }
    | THROW Expression ';'
      { manter_sequencia($1); manter_sequencia($2);
        manter_sequencia(';') }
    ;

GuardingStatement
    : SYNCHRONIZED '(' Expression ')' Statement { manter_sequencia($1);
      manter_sequencia('('); manter_sequencia($3);
      manter_sequencia(')'); manter_sequencia($5) }
    | TRY Block Finally { manter_sequencia($1);
      manter_sequencia($2); manter_sequencia($3) }
    | TRY Block Catches { manter_sequencia($1);
      manter_sequencia($2); manter_sequencia($3) }
    | TRY Block Catches Finally { manter_sequencia($1);
      manter_sequencia($2); manter_sequencia($3);
      manter_sequencia($4) }
    ;

```

```

Catches
    : Catch                                { manter_sequencia($1) }
    | Catches Catch                        { manter_sequencia($1) }
    ;

Catch
    : CatchHeader Block                    { manter_sequencia($1) }
    ;

CatchHeader
    : CATCH '(' TypeSpecifier IDENTIFIER ')' { manter_sequencia($1);
      manter_sequencia('('); manter_sequencia($3)
      manter_sequencia($4); manter_sequencia(')') }
    | CATCH '(' TypeSpecifier ')'          { manter_sequencia($1);
      manter_sequencia('('); manter_sequencia($3);
      manter_sequencia(')') }
    ;

Finally
    : FINALLY Block { manter_sequencia($1); manter_sequencia($2) }
    ;

PrimaryExpression
    : QualifiedName { manter_sequencia($1) }
    | QualifiedName '.' NewAllocationExpression { manter_sequencia($1);
      manter_sequencia('.'); manter_sequencia($3) }
    | QualifiedName '.' THIS { manter_sequencia($1);
      manter_sequencia('.'); manter_sequencia($3) }
    | QualifiedName '.' CLASS { manter_sequencia($1);
      manter_sequencia('.'); manter_sequencia($3) }
    | PrimitiveType '.' CLASS { manter_sequencia($1);
      manter_sequencia('.'); manter_sequencia($3) }
    | NotJustName { manter_sequencia($1) }
    ;

NotJustName
    : SpecialName { manter_sequencia($1) }
    | NewAllocationExpression { manter_sequencia($1) }
    | ComplexPrimary { manter_sequencia($1) }
    ;

ComplexPrimary
    : '(' Expression ')' { manter_sequencia('(');
      manter_sequencia($2); manter_sequencia(')') }
    | ComplexPrimaryNoParenthesis { manter_sequencia($1) }
    ;

ComplexPrimaryNoParenthesis
    : LITERAL { manter_sequencia($1) }
    | BOOLLIT { manter_sequencia($1) }
    | ArrayAccess { manter_sequencia($1) }
    | FieldAccess { manter_sequencia($1) }
    | MethodCall { manter_sequencia($1) }
    ;

ArrayAccess

```

```

: QualifiedName '[' Expression ']' { manter_sequencia($1);
    manter_sequencia('['); manter_sequencia($3);
    manter_sequencia(']') }
| ComplexPrimary '[' Expression ']' { manter_sequencia($1);
    manter_sequencia('['); manter_sequencia($3);
    manter_sequencia(']') }
;

FieldAccess
: NotJustName '.' IDENTIFIER { manter_sequencia($1);
    manter_sequencia('.'); manter_sequencia($3) }
| RealPostfixExpression '.' IDENTIFIER { manter_sequencia($1);
    manter_sequencia('.'); manter_sequencia($3) }
;

MethodCall
: MethodAccess '(' ArgumentList ')' { manter_sequencia($1);
    manter_sequencia('('); manter_sequencia($1);
    manter_sequencia($3); manter_sequencia(')') }
| MethodAccess '(' ')' { manter_sequencia($1);
    manter_sequencia('('); manter_sequencia(')') }
;

MethodAccess
: ComplexPrimaryNoParenthesis { manter_sequencia($1) }
| SpecialName { manter_sequencia($1) }
| QualifiedName { manter_sequencia($1) }
;

SpecialName
: THIS { manter_sequencia($1) }
| SUPER { manter_sequencia($1) }
| JNULL { manter_sequencia($1) }
;

ArgumentList
: Expression { manter_sequencia($1) }
| ArgumentList ',' Expression { manter_sequencia($1);
    manter_sequencia(','); manter_sequencia($3) }
;

NewAllocationExpression
: ArrayAllocationExpression { manter_sequencia($1) }
| ClassAllocationExpression { manter_sequencia($1) }
| ArrayAllocationExpression '{' '}' { manter_sequencia($1);
    manter_sequencia('{'); manter_sequencia('}') }
| ClassAllocationExpression '{' '}' { manter_sequencia($1);
    manter_sequencia('{'); manter_sequencia('}') }
| ArrayAllocationExpression '{' ArrayInitializers '}'
    { manter_sequencia($1); manter_sequencia('{');
    manter_sequencia($3); manter_sequencia('}') }
| ClassAllocationExpression '{' FieldDeclarations '}'
    { manter_sequencia($1); manter_sequencia('{');
    manter_sequencia($3); manter_sequencia('}') }
;

```

```

ClassAllocationExpression
: NEW TypeName '(' ArgumentList ')' { manter_sequencia($1);
    manter_sequencia($2); manter_sequencia('(');
    manter_sequencia($4); manter_sequencia(')') }
| NEW TypeName '(' ')' { manter_sequencia($1);
manter_sequencia($2)
    manter_sequencia('('); manter_sequencia(')')
    }
;

ArrayAllocationExpression
: NEW TypeName DimExprs Dims { manter_sequencia($1);
    manter_sequencia($2); manter_sequencia($3);
    manter_sequencia($4) }
| NEW TypeName DimExprs { manter_sequencia($1);
    manter_sequencia($2); manter_sequencia($3) }
;

DimExprs
: DimExpr { manter_sequencia($1) }
| DimExprs DimExpr { manter_sequencia($1);
    manter_sequencia($2) }
;

DimExpr
: '[' Expression ']' { manter_sequencia('[');
    manter_sequencia($2);
    manter_sequencia(']') }
;

Dims
: OP_DIM { manter_sequencia($1) }
| Dims OP_DIM { manter_sequencia($1);
    manter_sequencia($2) }
;

PostfixExpression
: PrimaryExpression { manter_sequencia($1) }
| RealPostfixExpression { manter_sequencia($1) }
;

RealPostfixExpression
: PostfixExpression OP_INC { manter_sequencia($1);
    manter_sequencia($2) }
| PostfixExpression OP_DEC { manter_sequencia($2);
    manter_sequencia($2) }
;

UnaryExpression
: OP_INC UnaryExpression { manter_sequencia($1);
    manter_sequencia($2) }
| OP_DEC UnaryExpression { manter_sequencia($1);
    manter_sequencia($2) }
| ArithmeticUnaryOperator CastExpression { manter_sequencia($1);
    manter_sequencia($2) }
| LogicalUnaryExpression { manter_sequencia($1) }

```

```

;

LogicalUnaryExpression
    : PostfixExpression                { manter_sequencia($1) }
    | LogicalUnaryOperator UnaryExpression { manter_sequencia($1;
                                                manter_sequencia($2) }
;

LogicalUnaryOperator
    : '~'                { manter_sequencia('~') }
    | '!'                { manter_sequencia('!') }
;

ArithmeticUnaryOperator
    : '+'                { manter_sequencia('+') }
    | '-'                { manter_sequencia('-') }
;

CastExpression
    : UnaryExpression                { manter_sequencia($1) }
    | '(' PrimitiveTypeExpression ')' CastExpression
        { manter_sequencia('('); manter_sequencia($2);
          manter_sequencia(')'); manter_sequencia($4) }
    | '(' ClassTypeExpression ')' CastExpression
        { manter_sequencia('('); manter_sequencia($2);
          manter_sequencia(')'); manter_sequencia($4) }
    | '(' Expression ')' LogicalUnaryExpression
        { manter_sequencia('('); manter_sequencia($2);
          manter_sequencia(')'); manter_sequencia($4) }
;

PrimitiveTypeExpression
    : PrimitiveType                { manter_sequencia($1) }
    | PrimitiveType Dims            { manter_sequencia($1); manter_sequencia($2) }
;

ClassTypeExpression
    : QualifiedName Dims            { manter_sequencia($1) }
;

MultiplicativeExpression
    : CastExpression                { manter_sequencia($1) }
    | MultiplicativeExpression '*' CastExpression
        { manter_sequencia($1); manter_sequencia('*');
          manter_sequencia($3); }
    | MultiplicativeExpression '/' CastExpression
        { manter_sequencia($1); manter_sequencia('/');
          manter_sequencia($3) }
    | MultiplicativeExpression '%' CastExpression
        { manter_sequencia($1); manter_sequencia('%');
          manter_sequencia($3) }
;

AdditiveExpression
    : MultiplicativeExpression      { manter_sequencia($1) }

```

```

| AdditiveExpression '+' MultiplicativeExpression
    { manter_sequencia($1); manter_sequencia('+');
      manter_sequencia($3) }
| AdditiveExpression '-' MultiplicativeExpression
    { manter_sequencia($1); manter_sequencia('-');
      manter_sequencia($3) }
;

ShiftExpression
: AdditiveExpression      { manter_sequencia($1) }
| ShiftExpression OP_SHL AdditiveExpression
    { manter_sequencia($1); manter_sequencia($2);
      manter_sequencia($3) }
| ShiftExpression OP_SHR AdditiveExpression
    { manter_sequencia($1); manter_sequencia($2);
      manter_sequencia($3) }
| ShiftExpression OP_SHRR AdditiveExpression
    { manter_sequencia($1); manter_sequencia($2);
      manter_sequencia($3) }
;

RelationalExpression
: ShiftExpression      { manter_sequencia($1) }
| RelationalExpression '<' ShiftExpression
    { manter_sequencia($1); manter_sequencia('<');
      manter_sequencia($3) }
| RelationalExpression '>' ShiftExpression
    { manter_sequencia($1); manter_sequencia('>');
      manter_sequencia($3) }
| RelationalExpression OP_LE ShiftExpression
    { manter_sequencia($1); manter_sequencia($2);
      manter_sequencia($3) }
| RelationalExpression OP_GE ShiftExpression
    { manter_sequencia($1); manter_sequencia($2);
      manter_sequencia($3) }
| RelationalExpression INSTANCEOF TypeSpecifier
    { manter_sequencia($1); manter_sequencia($2);
      manter_sequencia($3) }
;

EqualityExpression
: RelationalExpression
    { manter_sequencia($1) }
| EqualityExpression OP_EQ RelationalExpression
    { manter_sequencia($1); manter_sequencia($2);
      manter_sequencia($3) }
| EqualityExpression OP_NE RelationalExpression
    { manter_sequencia($1); manter_sequencia($2);
      manter_sequencia($3) }
;

AndExpression
: EqualityExpression      { manter_sequencia($1) }
| AndExpression '&' EqualityExpression      { manter_sequencia($1);
      manter_sequencia('&'); manter_sequencia($3) }
;

```

```

ExclusiveOrExpression
    : AndExpression { manter_sequencia($1) }
    | ExclusiveOrExpression '^' AndExpression { manter_sequencia($1);
        manter_sequencia('^') }
    ;

InclusiveOrExpression
    : ExclusiveOrExpression { manter_sequencia($1) }
    | InclusiveOrExpression '|' ExclusiveOrExpression
        { manter_sequencia($1); manter_sequencia('|');
        manter_sequencia($3) }
    ;

ConditionalAndExpression
    : InclusiveOrExpression { manter_sequencia($1) }
    | ConditionalAndExpression OP_LAND InclusiveOrExpression
        { manter_sequencia($1); manter_sequencia($2);
        manter_sequencia($3) }
    ;

ConditionalOrExpression
    : ConditionalAndExpression { manter_sequencia($1) }
    | ConditionalOrExpression OP_LOR ConditionalAndExpression
        { manter_sequencia($1); manter_sequencia($2);
        manter_sequencia($3) }
    ;

ConditionalExpression
    : ConditionalOrExpression
        { manter_sequencia($1) }
    | ConditionalOrExpression '?' Expression ':' ConditionalExpression
        { manter_sequencia($1); manter_sequencia('?');
        manter_sequencia($3); manter_sequencia(':');
        manter_sequencia($5) }
    ;

AssignmentExpression
    : ConditionalExpression { manter_sequencia($1) }
    | UnaryExpression AssignmentOperator AssignmentExpression
        { manter_sequencia($1); manter_sequencia($2);
        manter_sequencia($3) }
    ;

AssignmentOperator
    : '=' { manter_sequencia($1) }
    | ASS_MUL { manter_sequencia($1) }
    | ASS_DIV { manter_sequencia($1) }
    | ASS_MOD { manter_sequencia($1) }
    | ASS_ADD { manter_sequencia($1) }
    | ASS_SUB { manter_sequencia($1) }
    | ASS_SHL { manter_sequencia($1) }
    | ASS_SHR { manter_sequencia($1) }
    | ASS_SHRR { manter_sequencia($1) }
    | ASS_AND { manter_sequencia($1) }

```



```

        | ASS_XOR  { manter_sequencia($1) }
        | ASS_OR   { manter_sequencia($1) }
    ;

Expression
    : AssignmentExpression      { manter_sequencia($1) }
    ;

ConstantExpression
    : ConditionalExpression     { manter_sequencia($1) }
    ;

%%

/* -----
   |                               |
   |                               | Fim da gramatica
   |                               |
   ----- */

#include <ctype.h>
#include <string.h>
char *programe; /* para mensagens de erro */
int  lineno = 1;
#include <signal.h>
#include <setjmp.h>
jmp_buf begin;
int  indef;
char *infile; /* nome do arquivo de entrada */
FILE *fin; /* ponteiro para entrada de arquivo */
char **gargv; /* argumento da lista global */
int  gargc;

int  c; /* global para uso de aviso de cuidado _ "warning()" */

/* -----
   |                               |
   |                               | Funcao do Analisador Lexico do Conversor ConJava
   |                               |
   ----- */

yylex() /* ConJava */
{
    int c;

    pega_token:
    while ((c=pegac(fin)) == ' ' || c == '\t' || c == '\n');
    if (c == EOF) return 0;
/* -----
   |                               |
   |                               | Calculo do valor real ou inteiro
   |                               |
   ----- */
    if (isdigit(c)) { /* E' uma Constante */
        int cbuf[100], *p = cbuf;
        int cont=0, nroponto=0, conti, a = 1, estapos;
        *p++ = c;
        while (a) {
            c= pegac(fin);
            if ((! isdigit(c) && c != '.') ){
                unpegac(c, fin); cont--;
            }
        }
    }
}

```

```

        a=0; }
        *p++ = c;
        if (c== '.') { nroponto++; estapos = cont+1; }
        cont++;
    }
    *p=0;
    if (nroponto > 1) {
        printf("\nERRO _ Constante invalida");
        goto pega_token; }
    if (nroponto == 1) { /* Real (float, double...) */
        double valr, valrv;
        valr = cbuf[0] _ 48;
        if ((estapos_1) >= 1)
            for (conti = 1; conti <= (estapos_1) ; conti++)
                valr = valr*10 + (cbuf[conti]_48);
        valrv = (cbuf[cont] _ 48)/10.0;
        if (((cont_estapos) < 10) &&((cont_estapos_1) >= 1)){
            for (conti = cont_1; conti >= (estapos+1); conti__ )
                valrv = (valrv + (cbuf[conti]_48))/10.0;
        }
        else if ((cont_estapos) > 10)
            { printf("\nEstouro na precisao apos a virgula");
              valrv = 0.0; }
        valr = valr + valrv;
        if ( valr > 1.7E+308){
            printf("Cuidado => estouro de valor");
            todovalor.c_integer = 0.0;
        }
        else
            todovalor.c_real = valr;
        yylval.sym= install((char *)cbuf, C_REAL, todovalor);
        return C_REAL;
    }
    else { /* Inteiro (int) */
        double val;
        val = cbuf[0] _ 48;
        if (cont > 1)
            for (conti = 1; conti <= cont; conti++)
                val = val*10 + (cbuf[conti]_48);
        if ( (val > 4294967296 ) && (val < 1)) {
            printf("\nCuidado => estouro de valor");
            todovalor.c_integer = 0;
        }
        else {
            todovalor.endereco = val;
        }

        yylval.sym= install("", C_INT, todovalor);
        return C_INT;
    }
}

/* -----
|                                     |
|                               Localizar um simbolo                               |
|-----*/

if (isalpha(c)){
```

```

st_Symbol *s;
char sbuf[100], *p = sbuf;
char mbuf[100], *m = mbuf;
do{
    *p++ = c;
    *m++ = toupper(c);
}while ((c = pegac(fin)) != EOF && isalnum(c));

unpegac(c, fin);

*p = '\0'; /* o simbolo foi lido em sbuf */
*m = '\0';

/*
|          Verificar se o simbolo e' reservado          |
|-----|-----|
if ( (!strcmp(mbuf, "ABSTRACT"))) { return ABSTRACT; }
if ( (!strcmp(mbuf, "BREAK"))) { return BREAK; }
if ( (!strcmp(mbuf, "BYTE"))) { return BYTE; }
if ( (!strcmp(mbuf, "CASE"))) { return CASE; }
if ( (!strcmp(mbuf, "CATCH"))) { return CATCH; }
if ( (!strcmp(mbuf, "CLASS"))) { return CLASS; }
if ( (!strcmp(mbuf, "CONST"))) { return CONST; }
if ( (!strcmp(mbuf, "CONTINUE"))) { return CONTINUE; }
if ( (!strcmp(mbuf, "DEFAULT"))) { return DEFAULT; }
if ( (!strcmp(mbuf, "DO"))) { return DO; }
if ( (!strcmp(mbuf, "DOUBLE"))) { return DOUBLE; }
if ( (!strcmp(mbuf, "ELSE"))) { return ELSE; }
if ( (!strcmp(mbuf, "EXTENDS"))) { return EXTENDS; }
if ( (!strcmp(mbuf, "FINAL"))) { return FINAL; }
if ( (!strcmp(mbuf, "FINALLY"))) { return FINALLY; }
if ( (!strcmp(mbuf, "FOR"))) { for)code(); return FOR; }
if ( (!strcmp(mbuf, "GOTO"))) { return GOTO; }
if ( (!strcmp(mbuf, "IMPLEMENTS"))) { return IMPLEMENTS; }
if ( (!strcmp(mbuf, "IF"))) { return IF; }
if ( (!strcmp(mbuf, "IMPORT"))) { return IMPORT; }
if ( (!strcmp(mbuf, "NATIVE"))) { return NATIVE; }
if ( (!strcmp(mbuf, "NEW"))) { return NEW; }
if ( (!strcmp(mbuf, "PACKAGE"))) { return PACKAGE; }
if ( (!strcmp(mbuf, "PRIVATE"))) { return PRIVATE; }
if ( (!strcmp(mbuf, "PROTECTED"))) { return PROTECTED; }
if ( (!strcmp(mbuf, "PUBLIC"))) { return PUBLIC; }
if ( (!strcmp(mbuf, "RETURN"))) { return RETURN; }
if ( (!strcmp(mbuf, "STATIC"))) { return STATIC; }
if ( (!strcmp(mbuf, "SUPER"))) { return SUPER; }
if ( (!strcmp(mbuf, "SWITCH"))) { return SWITCH; }
if ( (!strcmp(mbuf, "SYNCHRONIZED"))) { return SYNCHRONIZED; }
if ( (!strcmp(mbuf, "THIS"))) { return THIS; }
if ( (!strcmp(mbuf, "THROW"))) { return THROW; }
if ( (!strcmp(mbuf, "THROWS"))) { return THROWS; }
if ( (!strcmp(mbuf, "TRANSIENT"))) { return TRANSIENT; }
if ( (!strcmp(mbuf, "TRY"))) { return TRY; }
if ( (!strcmp(mbuf, "VOID"))) { return VOID; }
if ( (!strcmp(mbuf, "VOLATILE"))) { return VOLATILE; }
if ( (!strcmp(mbuf, "WHILE"))) { WHILE-CODE(); return WHILE; }
if ( (!strcmp(mbuf, "FLOAT"))) { return FLOAT; }

```

```

if ( (!strcmp(mbuf, "INT")))          { return INT;      }
if ( (!strcmp(mbuf, "CHAR")))         { return CHAR;     }
if ( (!strcmp(mbuf, "BOOLEAN")))      { return BOOLEAN;  }
if ( (!strcmp(mbuf, "BYTE")))          { return BYTE;     }
if ( (!strcmp(mbuf, "LONG")))          { return LONG;     }
if ( (!strcmp(mbuf, "DOUBLE")))        { return DOUBLE;   }
if ( (!strcmp(mbuf, "SHORT")))         { return SHORT;    }

/* -----
|      Caso contrario sera tratado como identificador      |
|----- */
if ((s=lookup(sbuf)) == 0) s = install(sbuf, undef, todovalor);
yylval.sym = s;
if (SINT) printf("\n__>yylval.sym {%s}(%d)", nome_simbolo(s),
    indice_tabela(s));
return IDENTIFICADOR;
}

/* -----
|      Tratamento para cadeia de caracteres                  |
|----- */
if (c=='"') { /* quoted string */
    char sbuf[100], *p, *emalloc();
    for (p = sbuf; (c=pegac(fin)) != '"'; p++) {
        if (c== '\n' || c == EOF) execerror("missing quote", "");
        if (p >= sbuf + sizeof(sbuf) - 1) {
            *p = '\0';
            execerror("string too long", sbuf);
        }
        *p = backslash(c);
    }
    *p = 0;
    yylval.sym = (st_Symbol *)emalloc(strlen(sbuf)+1);

    strcpy((char *)yylval.sym, (char *) sbuf);
    return VAR;
}

/* -----
|      Tratamento para Simbolos especiais eh                |
|      explorado diretamente pelo analisador                 |
|      sintatico.                                             |
|----- */
switch(c){
case '>' : return follow('=', OP_GE,OP_GT);
case '<' : {
    int c;
    c = pegac(fin);
    if (c == '>') return OP_NE;
    else if (c == '=') return OP_LE;
    else { unpegac(c,fin); return OP_LT; }
}
case '=' : return OP_EQ;
case '+' : return follow('+', OP_SOM, OP_INC)
case '\n' : lineno++; return '\n';
default : return c;
}

```

A.29

```

    if (argc==1)
        escreve(stdin, stdout);
    else
        /* lendo o arquivo */
        while(--argc>0)
            if ((fp=fopen(++argv,"r"))==NULL) {
                printf("Nao foi possivel abrir%s\n", *argv);
                return 1;
            } else {
                escreve(fp, stdout);
                fclose(fp);
            }
        return 0;
}

/* -----
| Funcao cabecalho inicia a escrita do arquivo convertido|
| e coloca um texto inicial para indicar versao, data e |
| autoria do conversor                                     |
|----- */

void cabecalho(FILE *oufp) {
    int cc;
    FILE *infp;
    infp = fopen("cabl.bin","r");
    while ((cc=getc(infp)) != EOF)
        putc(cc,oufp);
    fclose("cabl.bin");
}

/* -----
| Funcao fecha termina a escrita do arquivo convertido|
| e coloca um texto final para indicar que o arquivo foi |
| convenientemente convertido                             |
|----- */

void fecha(FILE *ufp) {
    int ca;
    FILE *nfp;
    nfp = fopen("fecha.bin","r");
    while ((ca=getc(nfp)) != EOF)
        putc(ca,ufp);
    fclose("fecha.bin");
}

void escreve(FILE *ifp, FILE *ofp) {
    int c;
    ofp = fopen("saida.jap","wb+");
    cabecalho(ofp);
    converte(ifp, ofp);
    fecha(ofp);
    fclose(ofp);
}

/* -----

```

```

|          Outras Funcoes de Apoio          |
----- */

warning(char *s, char *t) /* imprime mensagem de erro */
{
    fprintf(stderr, "%s: %s", progame, s);
    if (t) fprintf(stderr, " %s", t);
    if (infile) fprintf(stderr, " in %s", infile);
    fprintf(stderr, "\nerro de sintaxe na linha %d", conta_linha);
    while (c != '\n' && c != EOF) c = pegac(fin);
    /* if (c == '\n') lineneno++; */
    return;
}

static int ikkk=0;
char linha[120];

int unpegac (int c, FILE *arq) {
    ungetc (c, arq);
    _ikkk;
    return c;
}

int pegac(FILE *arq) {
    int c;

    c=linha[ikkk++]=getc(arq);
    if (c=='\n' || ikkk>79) {
        linha[ikkk]=0;
        conta_linha++;
        if (LEXI) printf("\n$ %d>  %s", conta_linha, linha);
        ikkk=0;
    }
    return c;
}

```

Apêndice B.1

A BNF complementar da linguagem Japar

```
SelectionStatement
: IF '(' Expression ')' Statement
    { manter_sequencia($1); manter_sequencia('(');
      manter_sequencia($1); manter_sequencia(')');
      manter_sequencia() }
| IF '(' Expression ')' Statement ELSE Statement
    { manter_sequencia($1); manter_sequencia('(');
      manter_sequencia($3); manter_sequencia(')');
      manter_sequencia($5); manter_sequencia($6);
      manter_sequencia($7) }
| SWITCH '(' Expression ')' Block
    { manter_sequencia($1); manter_sequencia('(');
      manter_sequencia($3); manter_sequencia(')');
      manter_sequencia($5) }
;

IterationStatement
: WHILE '(' Expression ')' Statement
    { code_while( ) }
| DO Statement WHILE '(' Expression ')' ';'
    { code_do( ) }
| FOR '(' ForInit ForExpr ForIncr ')' Statement
    { pol_opr(op_at);
      substitui_simb ($2, tipo , todovalor); }
| FOR '(' ForInit ForExpr ')' Statement
    { code_for( );
      pol_opr(op_at);
      substitui_simb ($2, tipo, todovalor);
    }
;

ForInit
: ExpressionStatements ';' { code_for( ) }
| LocalVariableDeclarationStatement { code_for(); }
| ';' { code_for(); }
;
```


ForExpr

```
: Expression ';' { code_for() }
| ';' { code_for() }
;
```

ForIncr

```
: ExpressionStatements { code_for() }
;
```

statement :

```
variable_declaration
| ( expression ';' )
| ( statement_block )
| ( if_statement )
| (do_statement_par)
| ( do_statement )
| (while_statement_par)
| ( while_statement )
| (for_statement_par)
| ( for_statement )
| ( try_statement )
| ( switch_statement )
| ( 'synchronized' '(' expression ')' statement )
| ( 'return' [ expression ] ';' )
| ( 'throw' expression ';' )
| ( identifier ':' statement )
| ( 'break' [ identifier ] ';' )
| ( 'continue' [ identifier ] ';' )
| ( ';' )
;
```

for_statement_par :

```
'parfor' '(' ( variable_declaration | ( expression ';' ) | ';' )
[ expression ] ';' [ expression ] ';' ')' statement 'parend'
```

while_statement_par :

```
'parwhile' '(' expression ')' statement
```

do_statement_par :

```
'pardo' statement 'parwhile' '(' expression ')' ';'
;
```

type_declaration :

```
[ doc_comment ] ( class_declaration | interface_declaration ) ';'
| [ doc_comment ] ( class_declaration_par | interface_declaration ) ';'
;
```

```

;

class_declaration_par :
    { modifier } "resource" 'class' identifier
    [ 'extends' class_name ] "body" 'identifier'
    [ 'implements' interface_name { ',' interface_name } ]
    '{' { field_declaration } '}' "parend"

;

compilation_unit :
    [ package_statement ]
    { import_statement }
    { type_declaration }
    [ type_declaration_par ]
    ;

type_declaration_par :
    [ doc_comment ] ( "fork" class_declaration | interface_declaration ) ';' "join"
;

```

// Expressões

```

expression_par:
    within expression ? tc-primary ( expression-list ) : 1 expression
    | create identifier ( expression-list ) processor-clause priority-clause
;

primary:
    tc-primary ( expression-list )
;

tc.primary:
    primary.name
    | primary-> name
    ( *expression )
;

processor-clause:
    processor ( expression )
;

priority-clause:
    priority ( expression )
;

```

// Declarações

```

type-specifier:
    process spec identifier ( parameter-decl-list ) transaction-part

```

```
| process anytype
| process identfier
;

parameter-decl-list:
    parameter.type-decl
    parameter-type-decl, parameter-decl-list
;

transaction.part:
    { trans-decl-list }
;

parameter-type-decl:
    type-name
    decl-specifier declarator
;

trans.decl.list:
    trans-decl
    trans-decl trans-decl-list
;

trans-decl:
    transaction trans-type-specifier trans-declarator-list ;
;

trans-type-specifier:
    async
    | type-specifier
;

trans-declarator-list:
    trans-declarator
    | trans-declarator, trans-declarator-list
;

trans-declarator:
    declarator( parameter.decl-list)
    | declarator
;

statement_par:
    select {guard-alternative-list }
    | delay expression ;
;
```

```
guard-alternative.list:
    guard alternative
    | guard alternative or guard-alternative-list
    ;

guard:
    ( expression )
    ;

alternative:
    statement.list
    | terminate;
    ;

// Definitions Externas

external-definition:
    process body identifier ( param-list) process-body
    ;

process-body:
```

Apêndice B.2

Opcodes Java

ATENCAO: Os *opcodes* aqui utilizados são propriedade intelectual da SUN MICROSYSTEMS, INC, e estão utilizados especificamente como autorização especial para desenvolvedores. Portanto não podem ser usados comercialmente, sendo seu uso permitido apenas sob concessão especial com registro junto a essa entidade. Portanto, respeitando esses limites declaramos:

Copyright 1994-1998 by Sun Microsystems, Inc.,

Os opcodes aqui reunidos são encontrados documentados na referencia "The Java Language Specification" dos autores James Gosling, Bill Joy e Guy Steele na edicao 1.0, e obtidos no site da SUN MICROSYSTEMS, publicados em livro no ano de 1996

1. nop
2. aconst_null
3. iconst_m1
4. iconst_0
5. iconst_1
6. iconst_2
7. iconst_3
8. iconst_4
9. iconst_5
10. lconst_0
11. lconst_1
12. fconst_0
13. fconst_1
14. fconst_2
15. dconst_0
16. dconst_1
17. bipush
18. sipush
19. ldc
20. ldc_w
21. ldc2_w
22. iload
23. lload
24. fload
25. dload
26. aload

27. i1oad_0
28. i1oad_1
29. i1oad_2
30. i1oad_3
31. l1oad_0
32. l1oad_1
33. l1oad_2
34. l1oad_3
35. f1oad_0
36. f1oad_1
37. f1oad_2
38. f1oad_3
39. d1oad_0
40. d1oad_1
41. d1oad_2
42. d1oad_3
43. a1oad_0
44. a1oad_1
45. a1oad_2
46. a1oad_3
47. i1a1oad
48. l1a1oad
49. f1a1oad
50. d1a1oad
51. a1a1oad
52. b1a1oad
53. c1a1oad
54. s1a1oad
55. i1store
56. l1store
57. f1store
58. d1store
59. a1store
60. i1store_0
61. i1store_1
62. i1store_2
63. i1store_3
64. l1store_0
65. l1store_1
66. l1store_2
67. l1store_3
68. f1store_0
69. f1store_1
70. f1store_2
71. f1store_3
72. d1store_0
73. d1store_1
74. d1store_2
75. d1store_3
76. a1store_0
77. a1store_1
78. a1store_2
79. a1store_3
80. i1a1store
81. l1a1store
82. f1a1store

83.	dastore
84.	aastore
85.	bastore
86.	castore
87.	sastore
88.	pop
89.	pop2
90.	dup
91.	dup_x1
92.	dup_x2
93.	dup2
94.	dup2_x1
95.	dup2_x2
96.	swap
97.	iadd
98.	ladd
99.	fadd
100.	dadd
101.	isub
102.	lsub
103.	fsub
104.	dsub
105.	imul
106.	lmul
107.	fmul
108.	dmul
109.	idiv
110.	ldiv
111.	fdiv
112.	ddiv
113.	irem
114.	lrem
115.	frem
116.	drem
117.	ineg
118.	lneg
119.	fneg
120.	dneg
121.	ishl
122.	lshl
123.	ishr
124.	lshr
125.	iushr
126.	lushr
127.	iand
128.	land
129.	ior
130.	lor
131.	ixor
132.	lxor
133.	iinc
134.	i2l
135.	i2f
136.	i2d
137.	l2i
138.	l2f

- 139. l2d
- 140. f2i
- 141. f2l
- 142. f2d
- 143. d2i
- 144. d2l
- 145. d2f
- 146. i2b
- 147. i2c
- 148. i2s
- 149. lcmp
- 150. fcmpl
- 151. fcmpg
- 152. dcmpl
- 153. dcmpg
- 154. ifeq
- 155. ifne
- 156. iflt
- 157. ifge
- 158. ifgt
- 159. ifle
- 160. if_icmpeq
- 161. if_icmpne
- 162. if_icmplt
- 163. if_icmpge
- 164. if_icmpgt
- 165. if_icmpie
- 166. if_acmpeq
- 167. if_acmpne
- 168. goto
- 169. jsr
- 170. ret
- 171. tableswitch
- 172. lookupswitch
- 173. ireturn
- 174. lreturn
- 175. freturn
- 176. dreturn
- 177. areturn
- 178. return
- 179. getstatic
- 180. putstatic
- 181. getfield
- 182. putfield
- 183. invokevirtual
- 184. invokespecial
- 185. invokestatic
- 186. invokeinterface
- 187. xxxunusedxxx
- 188. new
- 189. newarray
- 190. anewarray
- 191. arraylength
- 192. athrow
- 193. checkcast
- 194. instanceof

195. monitorenter
196. monitorexit
197. wide
198. multianewarray
199. ifnull
200. ifnonnull
201. goto_w
202. jsr_w
203. breakpoint
204. ldc_quick
205. ldc_w_quick
206. ldc2_w_quick
207. getfield_quick
208. putfield_quick
209. getfield2_quick
210. putfield2_quick
211. getstatic_quick
212. putstatic_quick
213. getstatic2_quick
214. putstatic2_quick
215. invokevirtual_quick
216. invokenonvirtual_quick
217. invokesuper_quick
218. invokestatic_quick
219. invokeinterface_quick
220. invokevirtualobject_quick
221. invokeignored_quick
222. new_quick
223. anewarray_quick
224. multianewarray_quick
225. checkcast_quick
226. instanceof_quick
227. invokevirtual_quick_w
228. getfield_quick_w
229. putfield_quick_w
230. nonnull_quick

Apêndice B.3

Mnemônicos dos Opcodes de Java

ATENCAO: Os *opcodes* aqui utilizados são propriedade intelectual da SUN MICROSYSTEMS, INC, e estão utilizados especificamente como autorização especial para desenvolvedores. Portanto não podem ser usados comercialmente, sendo seu uso permitido apenas sob concessão especial com registro junto a essa entidade. Portanto, respeitando esses limites declaramos:

Copyright 1994-1998 by Sun Microsystems, Inc.,

Os opcodes aqui reunidos são encontrados documentados na referencia "The Java™ Virtual Machine Specification" Second Edition, dos autores Tim Lindholm e Frank Yellin.

00 (0x00) *nop*
01 (0x01) *aconst_null*
02 (0x02) *iconst_m1*
03 (0x03) *iconst_0*
04 (0x04) *iconst_1*
05 (0x05) *iconst_2*
06 (0x06) *iconst_3*
07 (0x07) *iconst_4*
08 (0x08) *iconst_5*
09 (0x09) *lconst_0*
10 (0x0a) *lconst_1*
11 (0x0b) *fconst_0*
12 (0x0c) *fconst_1*
13 (0x0d) *fconst_2*

14 (0x0e) *dconst_0*
15 (0x0f) *dconst_1*
16 (0x10) *bipush*
17 (0x11) *sipush*
18 (0x12) *ldc*
19 (0x13) *ldc_w*
20 (0x14) *ldc2_w*
21 (0x15) *iload*
22 (0x16) *lload*
23 (0x17) *fload*
24 (0x18) *dload*
25 (0x19) *aload*
26 (0x1a) *iload_0*
27 (0x1b) *iload_1*
28 (0x1c) *iload_2*
29 (0x1d) *iload_3*
30 (0x1e) *lload_0*
31 (0x1f) *lload_1*
32 (0x20) *lload_2*
33 (0x21) *lload_3*
34 (0x22) *fload_0*
35 (0x23) *fload_1*
36 (0x24) *fload_2*
37 (0x25) *fload_3*
38 (0x26) *dload_0*
39 (0x27) *dload_1*
40 (0x28) *dload_2*
41 (0x29) *dload_3*
42 (0x2a) *aload_0*
43 (0x2b) *aload_1*
44 (0x2c) *aload_2*
45 (0x2d) *aload_3*
46 (0x2e) *iaload*

47 (0x2f) *laload*
48 (0x30) *faload*
49 (0x31) *daload*
50 (0x32) *aaload*
51 (0x33) *baload*
52 (0x34) *caload*
53 (0x35) *saload*
54 (0x36) *istore*
55 (0x37) *lstore*
56 (0x38) *fstore*
57 (0x39) *dstore*
58 (0x3a) *astore*
59 (0x3b) *istore_0*
60 (0x3c) *istore_1*
61 (0x3d) *istore_2*
62 (0x3e) *istore_3*
63 (0x3f) *lstore_0*
64 (0x40) *lstore_1*
65 (0x41) *lstore_2*
66 (0x42) *lstore_3*
67 (0x43) *fstore_0*
68 (0x44) *fstore_1*
69 (0x45) *fstore_2*
70 (0x46) *fstore_3*
71 (0x47) *dstore_0*
72 (0x48) *dstore_1*
73 (0x49) *dstore_2*
74 (0x4a) *dstore_3*
75 (0x4b) *astore_0*
76 (0x4c) *astore_1*
77 (0x4d) *astore_2*
78 (0x4e) *astore_3*
79 (0x4f) *iastore*

80 (0x50) *lastore*
81 (0x51) *fastore*
82 (0x52) *dastore*
83 (0x53) *aastore*
84 (0x54) *bastore*
85 (0x55) *castore*
86 (0x56) *sastore*
87 (0x57) *pop*
88 (0x58) *pop2*
089 (0x59) *dup*
090 (0x5a) *dup_x1*
091 (0x5b) *dup_x2*
092 (0x5c) *dup2*
093 (0x5d) *dup2_x1*
094 (0x5e) *dup2_x2*
095 (0x5f) *swap*
096 (0x60) *iadd*
097 (0x61) *ladd*
098 (0x62) *fadd*
099 (0x63) *dadd*
100 (0x64) *isub*
101 (0x65) *lsub*
102 (0x66) *fsub*
103 (0x67) *dsub*
104 (0x68) *imul*
105 (0x69) *lmul*
106 (0x6a) *fmul*
107 (0x6b) *dmul*
108 (0x6c) *idiv*
109 (0x6d) *ldiv*
110 (0x6e) *fdiv*
111 (0x6f) *ddiv*
112 (0x70) *irem*

113 (0x71) *lrem*
114 (0x72) *frem*
115 (0x73) *drem*
116 (0x74) *ineg*
117 (0x75) *lneg*
118 (0x76) *fneg*
119 (0x77) *dneg*
120 (0x78) *ishl*
121 (0x79) *lshl*
122 (0x7a) *ishr*
123 (0x7b) *lshr*
124 (0x7c) *iushr*
125 (0x7d) *lushr*
126 (0x7e) *iand*
127 (0x7f) *land*
128 (0x80) *ior*
129 (0x81) *lor*
130 (0x82) *ixor*
131 (0x83) *lxor*
132 (0x84) *iinc*
133 (0x85) *i2l*
134 (0x86) *i2f*
135 (0x87) *i2d*
136 (0x88) *l2i*
137 (0x89) *l2f*
138 (0x8a) *l2d*
139 (0x8b) *f2i*
140 (0x8c) *f2l*
141 (0x8d) *f2d*
142 (0x8e) *d2i*
143 (0x8f) *d2l*
144 (0x90) *d2f*
145 (0x91) *i2b*

146 (0x92) *i2c*
147 (0x93) *i2s*
148 (0x94) *lcmp*
149 (0x95) *fcmpl*
150 (0x96) *fcmpg*
151 (0x97) *dcmpl*
152 (0x98) *dcmpg*
153 (0x99) *ifeq*
154 (0x9a) *ifne*
155 (0x9b) *iflt*
156 (0x9c) *ifge*
157 (0x9d) *ifgt*
158 (0x9e) *ifle*
159 (0x9f) *if_icmpeq*
160 (0xa0) *if_icmpne*
161 (0xa1) *if_icmplt*
162 (0xa2) *if_icmpge*
163 (0xa3) *if_icmpgt*
164 (0xa4) *if_icmple*
165 (0xa5) *if_acmpeq*
166 (0xa6) *if_acmpne*
167 (0xa7) *goto*
168 (0xa8) *jsr*
169 (0xa9) *ret*
170 (0xaa) *tableswitch*
171 (0xab) *lookupswitch*
172 (0xac) *ireturn*
173 (0xad) *lreturn*
174 (0xae) *freturn*
175 (0xaf) *dreturn*
176 (0xb0) *areturn*
177 (0xb1) *return*
178 (0xb2) *getstatic*

179 (0xb3) *putstatic*
180 (0xb4) *getfield*
181 (0xb5) *putfield*
182 (0xb6) *invokevirtual*
183 (0xb7) *invokespecial*
184 (0xb8) *invokestatic*
185 (0xb9) *invokeinterface*
186 (0xba) *xxxunusedxxx*¹
187 (0xbb) *new*
188 (0xbc) *newarray*
189 (0xbd) *anewarray*
190 (0xbe) *arraylength*
191 (0xbf) *athrow*
192 (0xc0) *checkcast*
193 (0xc1) *instanceof*
194 (0xc2) *monitorenter*
195 (0xc3) *monitorexit*
196 (0xc4) *wide*
197 (0xc5) *multianewarray*
198 (0xc6) *ifnull*
199 (0xc7) *ifnonnull*
200 (0xc8) *goto_w*
201 (0xc9) *jsr_w*
Reserved opcodes:
202 (0xca) *breakpoint*
254 (0xfe) *impdep1*
255 (0xff) *impdep2*

Apêndice B.4

Programas usados em testes

Laço Simples

```
class Loop1 {
    double[] a;

    void instanceMethod(int N, double scal) {
        double[] b = new double[N];
        double[] c = new double[N];

        // generate a multi version code block-scheduled parallel loop
        // with threshold at 100,000 (requires passing of scal and b)

        /*par block, multi = 100000 */
        for (int i = 10; i < N; i += 4) {
            int local = 3;
            b[i] = a[i] * scal + local;
        }
    }
}
```

Laço Duplo

```
class Loop2 {

    static double[] a = new double[300];

    static void classMethod(int N) {

        double[] b = new double[300];
```

```
// GENERATE A PARALLEL LOOP WITH 8 THREADS

/*par threads = 8 */
for (int i = 0; i < N; i +=2)
i:      b[i] = 2.0 * a[i];

    // note that statement label is handled
    // differently, i.e. not normalized :-)
}
}
```

Laço

```
class LoopWorker_Loop2_classMethod_0 { }

// Some incorrect forms for parallel loops

class Loop3 {

    static int[] a;

    static {

        // PARALLEL LOOP IN A STATIC INITIALIZER NOT ALLOWED

        a = new int[100];

        /*par
        */
        for (int i = 0; i < 100; i+=2)
            a[i] = 1;
    }

    static void method() {

        /*par */
        for (int d = 0; d < 100.0; d += 1)    // incorrect type
            a[d] = 10;

        int i,j;

        /*par */                                // not declared in FOR
        for (i = 0; i < 100; i++)
            a[i] = 10;

        /*par */                                // both not declared in FOR
        for (i = 0, j = 0; i < 100; i++, j++)
            a[i] = 10;

        /*par */                                // two vars declared
```

```
for (int k = 0, l = 0; k < 100; k++, l++)
    a[k] = 10;

j = 0;
/*par */                                // double increment
for (int k = 0; k < 100; k++, j++)
    a[k] = 10;

j = 0;
/*par */                                // j is changed!
for (int k = 0; k < 100; k++) {
    a[k] = 10;
    j++;
}

/*par */                                // yes, this works!
// why bother with j anyway?
for (int k = 0; k < 100; k++)
    a[k] = 10;
}
```

Laço

```
class Loop4 {

    double[][] a;
    double[] b;

    Loop4(int N) {

        // IN THIS CASE N MUST BE PASSED TO RUN-METHOD()
        /*par */
        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
                a[i][j] = 3.0;

        // IN THIS CASE N DOES NOT HAVE TO BE PASSED
        /*par*/
        for (int i = 0; i < N; i++)
            b[i] = 3.0;
    }
}
```

Laço

```
class Loop5 {

    double[] a;
    double[] b;

    Loop5(int N) {

        // A cyclic-scheduled DOACROSS-loop
        // with busy-waiting random synchronization

        /*par sync = 5, cyclic, susp_wait */

        for (int i = 0; i < N-10; i++) {
            /*wait 0, i-10 */
            a[i] = 10.0f;
            b[i] = a[i+10];
            /*post 0 */
        }
    }
}
```

Laço

```
class Loop6 {

    static double[] a;
    static double[] b;

    double[] c;

    static void doit() {
        /*par guided, multi = 100 */
        for (int i = 500; i > 0; i -= 3)
            a[i] = 100;
    }

    static void method(int N, int notused) {
        int[] locarray = new int[300];
        int[] notusedarray = new int[300];

        /*par cyclic */
        for (int i = 0; 500 > i; i += 2)
            b[i] = 100 * N + locarray[i];
    }

    Loop6(int N) {
        /*par block*/
        for (int i = 5000; i > 0; i--)
            c[i] = 10.0f;
    }
}
```

```
}
```

Laço

```
class Loop7 {

    static void method(int len, int[] a, int[] b) {
        try {
            /*par (RuntimeException) */
            for (int i = 0; i < len; i++)
                a[i] = 1 / b[i];
        }
        catch(RuntimeException e) {
            System.out.println("a has not been initialized");
        }
    }

    public static void main(String arg[]) {
        int[] a = new int[100];
        int[] b = new int[100];

        for (int i = 0; i < 100; i++)
            b[i] = 10;
        b[52] = 0;

        method(100, a, b);
    }
}
```

QuickSort

```
import java.util.*;

class QuickAlt {

    // Quick-sorting

    final static void quick(int[] a, int q, int r) {

        if ((r - q) <= 20) {

            // Use Insertion-sorting for small arrays

            int i, j;
            for (i = q+1; i <= r; i++) {
                int v = a[i];
                for (j = i; q < j; j--)
                    if (a[j-1] > v)
                        a[j] = a[j-1];
                else
                    break;
            }
        }
    }
}
```

```

        a[j] = v;
    }
}
else {
    int t = a[q]; a[q] = a[(q+r)/2]; a[(q+r)/2] = t;

    int s = q;
    for (int i = q+1; i <= r; i++)
        if (a[i] <= a[q]) {
            t = a[++s]; a[s] = a[i]; a[i] = t;
        }
    t = a[q]; a[q] = a[s]; a[s] = t;

    /*par_inv */
    {
        quick(a, q, s-1);
        quick(a, s+1, r );
    }
}
}
}

```

Testar tudo

```

/* *****
***  TESTE Japar                      ***
***** */

class TestAll {

    static final int MAX = 1600;

    // Class variables

    static int[][] a = new int[MAX][MAX];

    // Instance variables

    int[][] b = new int[MAX][MAX];

    double[] d = new double[MAX];

    // Constructor with trivial DOACROSS-loop

    TestAll() {
        d[0] = 1.0;
        /*par cyclic, sync=1 */
        for (int i = 1; i < MAX; i++) {
            /*wait 0, i-1 */
            d[i] = d[i-1];
            /*post 0 */
        }
    }
}

```

```
}

public static void main(String arg[]) {
    long    t1, t2;

    // Test 1

    t1 = System.currentTimeMillis();
    TestAll o = new TestAll();
    t2 = System.currentTimeMillis();

    System.out.println("time TestAll() = " + (t2 - t1) / 1000.0);

    for (int i = 0; i < MAX; i++)
        if (o.d[i] != 1.0) {
            System.out.println("ERROR 1");
            System.exit(1);
        }

    // Test 2

    for (int k = 0; k <= MAX; k+=400) {

        for (int i = 0; i < MAX; i++)
            for (int j = 0; j < MAX; j++)
                a[i][j] = 2;

        t1 = System.currentTimeMillis();
        classMethod(k);
        t2 = System.currentTimeMillis();

        System.out.println("time classMethod(" + k + ") = " + (t2 - t1) /
1000.0);

        for (int i = 0; i < k; i++)
            for (int j = 0; j < k; j++)
                if (a[i][j] != -2) {
                    System.out.println("ERROR 2");
                    System.exit(1);
                }
        for (int i = k; i < MAX; i++)
            for (int j = k; j < MAX; j++)
                if (a[i][j] != 2) {
                    System.out.println("ERROR 2");
                    System.exit(1);
                }
    }

    // Test 3

    for (int k = 0; k <= MAX; k+=400) {

        for (int i = 0; i < MAX; i++)
            for (int j = 0; j < MAX; j++)
```

```

        o.b[i][j] = 3;

        t1 = System.currentTimeMillis();
        o.instanceMethod(k);
        t2 = System.currentTimeMillis();

        System.out.println("time instanceMethod(" + k + ") = " + (t2 - t1) /
1000.0);

        for (int i = 0; i < k; i++)
            for (int j = 0; j < k; j++)
                if (o.b[i][j] != -3) {
                    System.out.println("ERROR 3");
                    System.exit(1);
                }
        for (int i = k; i < MAX; i++)
            for (int j = k; j < MAX; j++)
                if (o.b[i][j] != 3) {
                    System.out.println("ERROR");
                    System.exit(1);
                }
    }

    // Test 4

    for (int k = 0; k <= MAX; k+=400) {

        for (int i = 0; i < MAX; i++)
            for (int j = 0; j < MAX; j++)
                o.b[i][j] = 4;

        t1 = System.currentTimeMillis();
        o.instanceMethod_stride(k);
        t2 = System.currentTimeMillis();

        System.out.println("time instanceMethod_stride(" + k + ") = " + (t2 - t1)
/ 1000.0);

        for (int i = 10; i <= k/2; i+=2)
            for (int j = 0; j < k; j++)
                if (o.b[i][j] != -4) {
                    System.out.println("ERROR 4");
                    System.exit(1);
                }
            else
                o.b[i][j] = 4;
        for (int i = 0; i < MAX; i++)
            for (int j = 0; j < MAX; j++)
                if (o.b[i][j] != 4) {
                    System.out.println("ERROR 4");
                    System.exit(1);
                }
    }

    // Test 5

```



```

for (int k = 0; k <= MAX; k+=400) {

    for (int i = 0; i < MAX; i++)
        for (int j = 0; j < MAX; j++)
            o.b[i][j] = 5;

    t1 = System.currentTimeMillis();
    o.instanceMethod_bnd(k);
    t2 = System.currentTimeMillis();

    System.out.println("time instanceMethod_bnd(" + k + ") = " + (t2 - t1) /
1000.0);

    for (int i = k-1; i > 17; i-=6)
        for (int j = 0; j < k; j++)
            if (o.b[i][j] != -5) {
                System.out.println("ERROR 5");
                System.exit(1);
            }
        else
            o.b[i][j] = 5;
    for (int i = 1; i < k; i++)
        for (int j = 0; j < k; j++)
            if (o.b[i][j] != 5) {
                System.out.println("ERROR 5");
                System.exit(1);
            }
}
// Test 6

System.out.println("\nThe following loops should print the integers
1..10");

/*par */
for (int i = 1; i <= 10; i++)
    System.out.print(" " + i );
System.out.println(" <<");
/*par */
for (int i = 0; i < 10; i++)
    System.out.print(" " + (i+1) );
System.out.println(" <<");
/*par */
for (int i = 10; i >= 1; i--)
    System.out.print(" " + i );
System.out.println(" <<");
/*par */
for (int i = 10; i > 0; i--)
    System.out.print(" " + i );
System.out.println(" <<");
/*par */
for (int i = 2; i <= 20; i += 2)
    System.out.print(" " + (i/2) );
System.out.println(" <<");

```

```

/*par */
for (int i = 2; i < 21; i += 2)
    System.out.print(" " + ( i/2 ) );
System.out.println(" <<");
/*par */
for (int i = 20; i >= 2; i -= 2)
    System.out.print(" " + ( i/2 ) );
System.out.println(" <<");
/*par */
for (int i = 20; i > 1; i -= 2)
    System.out.print(" " + ( i/2 ) );
System.out.println(" <<");
/*par */
for (int i = 3; i <= 30; i += 3)
    System.out.print(" " + ( i/3 ) );
System.out.println(" <<");
/*par */
for (int i = 3; i <= 31; i += 3)
    System.out.print(" " + ( i/3 ) );
System.out.println(" <<");
/*par */
for (int i = 3; i <= 32; i += 3)
    System.out.print(" " + ( i/3 ) );
System.out.println(" <<");
/*par */
for (int i = 3; i < 31; i += 3)
    System.out.print(" " + ( i/3 ) );
System.out.println(" <<");
/*par */
for (int i = 3; i < 32; i += 3)
    System.out.print(" " + ( i/3 ) );
System.out.println(" <<");
/*par */
for (int i = 3; i < 33; i += 3)
    System.out.print(" " + ( i/3 ) );
System.out.println(" <<");
/*par */
for (int i = 9; i < 99; i += 9)
    System.out.print(" " + ( i/9 ) );
System.out.println(" <<");
/*par */
for (int i = 11; i <= 93; i += 9)
    System.out.print(" " + ( (i-2)/9 ) );
System.out.println(" <<");
/*par */
for (int i = 11; i < 96; i += 9)
    System.out.print(" " + ( (i-2)/9 ) );
System.out.println(" <<");
int local = -10;
/*par */
for (int i = (5+local*2)+17; i < 17+(-local)*2-16; i += 2)
    System.out.print(" " + ( i/2 ) );
System.out.println(" <<");

```

```

    System.out.println("\nThe following three test3 should transfer exceptions
back");
    System.out.println("(but parallel versions may become undeterministic)\n");

    // Test 7

    for (int k = 0; k < 4; k++) {

    for (int i = 0; i < MAX; i++)
        a[i][0] = 7;

    try {
        classMethod_exc();
    }
    catch (Exception e) {
        int t = 0;
        for (int i = 0; i < MAX; i++)
            if (a[i][0] == -7) t++;
        System.out.println("Exception correctly transferred back from loop, "
+ t + " iterations done");
    }
    catch (Throwable e) {
        int t = 0;
        for (int i = 0; i < MAX; i++)
            if (a[i][0] == -7) t++;
        System.out.println("Throwable correctly transferred back from loop, "
+ t + " iterations done");
    }
    }

    // Test 8

    for (int k = 0; k < 4; k++) {

    for (int i = 0; i < MAX; i++)
        a[i][0] = 8;

    try {
        traversel(0, MAX-1);
    }
    catch (Exception e) {
        int t = 0;
        for (int i = 0; i < MAX; i++)
            if (a[i][0] == -8) t++;
        System.out.println("Exception correctly transferred back from traversel,
"
+ t + " elements set");
    }
    catch (Throwable e) {
        int t = 0;
        for (int i = 0; i < MAX; i++)
            if (a[i][0] == -8) t++;
        System.out.println("Throwable correctly transferred back from traversel,
"

```

```

        + t + " elements set");
    }
}

// Test 9

for (int k = 0; k < 4; k++) {

for (int i = 0; i < MAX; i++)
    a[i][0] = 9;

try {
    traverse2(0, MAX-1);
}
catch (Exception e) {
    int t = 0;
    for (int i = 0; i < MAX; i++)
        if (a[i][0] == -9) t++;
    System.out.println("Exception correctly transferred back from traverse2,
"
        + t + " elements set");
}
}

// Test 10

for (int k = 0; k < MAX; k+=400) {

    for (int i = 0; i < k; i++)
        a[0][i] = k-i-1;
    for (int i = k; i < MAX; i++)
        a[0][i] = 10;

    t1 = System.currentTimeMillis();
    quicksort(a[0], 0, k-1);
    t2 = System.currentTimeMillis();

    System.out.println("time quicksort(" + k + ") = " + (t2 - t1) / 1000.0);

    for (int i = 0; i < k; i++)
        if (a[0][i] != i) {
            System.out.println("ERROR 10");
            System.exit(1);
        }
    for (int i = k; i < MAX; i++)
        if (a[0][i] != 10) {
            System.out.println("ERROR 10");
            System.exit(1);
        }
}

// Test 11

Tree2 root = new Tree2(1.0);

```

```

    for (int k = 0; k < 2000; k++)
        root.insert( new Tree2(400.0) );
    System.out.println("seeing height " + Tree2.compHeight1(root));
    root.test(11);
    System.out.println("seeing height " + root.compHeight2());
    root.test(-11);

}

static void classMethod(int k) {
    /*par block, multi=400 */
    for (int i = 0; i < k; i++)
        for (int j = 0; j < k; j++)
            a[i][j] = -2;
}

void instanceMethod(int k) {
    /*par guided, multi=600 */
    for (int i = 0; i < k; i++)
        for (int j = 0; j < k; j++)
            b[i][j] = -3;
}

void instanceMethod_stride(int k) {
    /*par */
    for (int i = 10; i <= k/2; i+=2)
        for (int j = 0; j < k; j++)
            b[i][j] = -4;
}

void instanceMethod_bnd(int k) {
    /*par threads=16 */
    for (int i = k-1; i > 17; i-=6)
        for (int j = 0; j < k; j++)
            b[i][j] = -5;
}

static void classMethod_exc() throws Throwable {
    /*par block, (Exception) , (Throwable) */
    for (int i = 0; i < MAX; i++) {
        if (i == 500)
            throw new Exception();
        else if (i == 800)
            throw new Throwable();
        else
            a[i][0] = -7;
    }
}

final static void traversel(int low, int high) throws Throwable {

    if (low > high)
        return;
    else {

        int mid = (low+high)/2;

        if (mid == 500)

```

```

        throw new Exception();
    else if (mid == 800)
        throw new Throwable();

    a[mid][0] = -8;

    /*par_inv cut_depth=3, no_overhead, (Exception), (Throwable) */
    {
        traversel(low,mid-1);
        traversel(mid+1, high);
    }
}

final static void traverse2(int low, int high) throws Exception {
    if (low > high)
        return;
    else {
        int mid = (low+high)/2;

        if (mid == 500)
            throw new Exception();

        a[mid][0] = -9;

        /*par_inv cut_depth=6, overhead, (Exception) */
        {
            traverse2(low,mid-1);
            traverse2(mid+1, high);
        }
    }
}

private static void swap(int [] a, int i, int j) {
    int t = a[i]; a[i] = a[j]; a[j] = t;
}

final static void quicksort(int[] a, int q, int r) {
    if (q < r) {
        swap(a, q, (q+r)/2);
        int s = q;
        for (int i = q+1; i <= r; i++)
            if (a[i] <= a[q])
                swap(a, ++s, i);
        swap(a, q, s);
        /*par_inv*/
        {
            quicksort(a, q, s-1);
            quicksort(a, s+1, r );
        }
    }
}

class Tree2 {

```

```

double val;
Tree2 left;
Tree2 right;

Tree2(double newval) {
    val = newval;
    left = null;
    right = null;
}

// Class Method for Computing Height

final static int compHeight1(Tree2 t) {
    if (t == null)
        return 0;
    else {
        int l, r;

        t.val = 11;

        /*par_inv */
        {
            l = compHeight1(t.left);
            r = compHeight1(t.right);
        }
        return (l > r) ? (l+1) : (r+1);
    }
}

// Instance Method for Computing Height
// (expressed in rigid form required by JAVAR)

final int compHeight2() {
    int l = 0 , r = 0;

    val = -11;

    if ((left != null) && (right != null))
        /*par_inv */
        {
            l = left.compHeight2();
            r = right.compHeight2();
        }
    else if (left != null)
        l = left.compHeight2();
    else if (right != null)
        r = right.compHeight2();

    return (l > r) ? (l+1) : (r+1);
}

// Auxiliary Instance Methods

void test(double d) {

```

```
    if (val != d) {
        System.out.println("ERROR " + d);
        System.exit(1);
    }
    if (left != null)
        left.test(d);
    if (right != null)
        right.test(d);
}

void insert(Tree2 node) {
    if (left == null)
        left = node;
    else if (right == null)
        right = node;
    else if (val >= 0) {
        val = -400.0d;
        left.insert(node);
    }
    else {
        val = 400.0d;
        right.insert(node);
    }
}
}
```


Apêndice B.5

Programa de Otimização (parte)

```
/* Estudo do corpo do laço a ser otimizado */

w->l_len    = l->max_ad - l->min_ad + 1u;
w->loop_code = (u1_int *) make_mem(w->l_len * sizeof(u1_int));

for (i = 0u; i < w->l_len; i++)
    w->loop_code[i] = oc[i];

/* NOP código do laço antigo */

nop_loop(a, l, w->loop_code);

/* Copiando a Informação de carregamento de Type e Signature */

if (l->load_locs != ml)
    javab_out(-1, "informacoes locais perdidas de max em new_worker()");

w->load_ind = (u2_int *) make_mem(ml * sizeof(u2_int));
w->load_type = (u1_int *) make_mem(ml * sizeof(u1_int));
w->load_sig = (char **) make_mem(ml * sizeof(char *));

for (i = 0u; i < ml; i++) {

    w->load_ind[i] = 0u;
    w->load_type[i] = l->load_type[i];

    if (l->load_type[i] <= (u1_int) TP_2WORD)
```

```

        w->load_sig[i] = NULL;
    else if (l->load_type[i] < (u1_int) TP_ERROR) {

        if (l->load_sig[i]) {
            w->load_sig[i] = l->load_sig[i];
            l->load_sig[i] = NULL;          /* cadeia do laço */
        }
        else {
            javab_out(1, "tipo desconhecido local %u pode causar problemas", i);
            w->load_sig[i] = strdup("Ljava/lang/Object;");
        }

    }
    else
        javab_out(-1, "não foi possível paralelizar o laço com um tipo local desconhecido");
}

/* trocando o laço por rotinas de LoopWorker */

static void loop_replacement(loop_ptr l, u4_int clen) {

    u4_int i, j;
    u2_int voidret;
    u2_int method;
}

/* insercao de Bytecode */

now = 1u; /* first argument is `this' */

for (i = 0u; i < w->max_locals; i++) {
    if (w->load_sig[i]) {

        u1_int tp = w->load_type[i];

        add_byte(42u); /* aload_0 */

        if (now > U1MAX)
            javab_out(-1, "índice muito grande em add_constructor() (não pode ser aceito)");

        /* proximo spit_ls() sempre toma 2 bytes! */
    }
}

```

```

    spit_ls(tp, now, 0u); /* carrega */

    add_byte(181u); /* putfield */
    add_byte(HIGB_U2(w->load_ind[i]));
    add_byte(LOWB_U2(w->load_ind[i]));

    if ((tp == TP_LONG) || (tp == TP_DOUBLE))
        now += 2;
    else
        now++;
}
}

/* Construindo os atributos do codigo */

a->attribute_name_index = codea;

add_byte(HIGB_U2(ms));
add_byte(LOWB_U2(ms)); /* max_stack */

add_byte(HIGB_U2(ml));
add_byte(LOWB_U2(ml)); /* max_locals */

add_byte(0u);
add_byte(0u);
add_byte(0u);
add_byte(0u); /* code_length (tamanho do codigo) */

/* insercao de bytecode */

/* Get Values for Locals from Fields
   (reverse order -> destroys local 0 last!) */

{ u2_int mlx = w->max_locals;

  for (ii = 1u; ii <= mlx; ii++) {

    i = mlx - ii;

    if (w->load_sig[i]) {
      u1_int tp = w->load_type[i];

```

```

    add_byte(42u); /* aload_0 */

    add_byte(180u); /* getfield */
    add_byte( HIGB_U2(w -> load_ind[i]) );
    add_byte( LOWB_U2(w -> load_ind[i]) );

    spit_ls(tp, i, 1u); /* store */
}
}
}

{ s4_int off = (w -> entry_off + 3u);
  s2_int off2 = (s2_int) off;

  if (off == (s4_int) off2) /* fits in 2 bytes */
    spit_goto(w -> entry_off + 3u);
  else
    spit_goto(w -> entry_off + 5u);
}

/* copiando o corpo do laço */

for (i = 0u; i < w -> l_len; i++)
  add_byte(w -> loop_code[i]);

/* Change Stride
   ***** */

{ u4_int tb   = (new_pos - w -> l_len) + w -> ind_off;
  u2_int stride = w -> ind_step * my_not;

  if (w -> ind_is_w) {
    new_bytes[tb + 3u] = HIGB_U2(stride);
    new_bytes[tb + 4u] = LOWB_U2(stride);
  }
  else {
    if (stride > U1MAX)
      javab_out(-1, "stride too wide (should not occur)");
    new_bytes[tb + 2u] = (u1_int) stride;
  }
}
}

```

```
/* Paralelizacao real de laço */
```

```
void parallelize_loop(attribute_ptr a, loop_ptr l, u1_int *method) {
```

```
    u1_int *bytes    = a -> info, *new;
    u2_int max_stack = B2U2(bytes[0],bytes[1]);
    u2_int max_locals = B2U2(bytes[2],bytes[3]);
    u4_int code_length = B2U4(bytes[4],bytes[5],bytes[6],bytes[7]);
    u4_int exc_base   = code_length + 8u, i, j;
    u2_int exc_table_l = B2U2(bytes[exc_base],bytes[exc_base+1u]);
```

```
    javab_out(2, "    - SUCCESS: parallel loop %u-%u", l -> min_ad, l -> max_ad);
```

```
    if (a -> code_length != code_length)
        javab_out(2, "    - Subsequent Transformation");
```

```
    /* additional stack size: threads/parameters */
```

```
    stack_inc = 1u - (u2_int) my_not;
    /* Construindo novo código */
```

```
    new_worker(a, l, max_stack, max_locals, (char *) method);
    loop_replacement(l, code_length);
    /* Mudando o tamanho da pilha */
    if (stack_inc > max_stack) {
        max_stack = stack_inc;
        bytes[0] = HIGB_U2(max_stack);
        bytes[1] = LOWB_U2(max_stack);
    }
```

```
    /* Inserindo novo código */
```

```
    { u4_int lbase = l -> d -> low + 8u;
      s4_int off   = code_length - l -> d -> low;
      s2_int off2  = (s2_int) off;
```

```
    if (off == (s4_int) off2) { /* fits in 2 bytes */
        bytes[lbase++] = 167u; /* goto */
        bytes[lbase++] = HIGB_U2(off);
        bytes[lbase++] = LOWB_U2(off);
    }
    else {
        bytes[lbase++] = 200u; /* goto_w */
```

```

    bytes[lbase++] = HIGB_U4(off);
    bytes[lbase++] = LOWB_U4(off);
    bytes[lbase++] = HIGB_U2(off);
    bytes[lbase++] = LOWB_U2(off);
}
}

/* Mudando o tamanho do codigo*/

code_length += new_pos;

bytes[4] = HIGB_U4(code_length);
bytes[5] = LOWB_U4(code_length);
bytes[6] = HIGB_U2(code_length);
bytes[7] = LOWB_U2(code_length);

/* Mudando o tamanho da excessao (Exception) */

exc_table_l++;

/* Mudando o tamanho do atributo */

a->attribute_length = 12u + code_length + exc_table_l * 8u;

/* Construindo novo atributo de codigo */

new = (u1_int *) make_mem(a->attribute_length * sizeof(u1_int));

/* Copiando o codigo velho */

for (i = 0u; i < exc_base; i++)
    new[i] = bytes[i].

/* Copiando o novo codigo */

for (i = 0u; i < new_pos; i++)
    new[exc_base+i] = new_bytes[i];

/* velha Exception Handler e Nova Exception Handler
***** */

i = exc_base + new_pos; /* ponteiro em nova tabela */

new[i++] = HIGB_U2(exc_table_l);
new[i++] = LOWB_U2(exc_table_l);

```

```
for (j = 0u; j < (exc_table_l-1u) * 8u; j++)
    new[i++] = bytes[exc_base+2u+j];

save_handler += (exc_base - 8u);

if (save_handler > U2MAX)
    javab_out(0, "tamanho de codigo muito grande para handler_pc");

/* Proximos enderecos são calculados em relacao ao segmento de codigo */

new[i++] = HIGB_U2(exc_base - 8u);
new[i++] = LOWB_U2(exc_base - 8u); /* start_pc */
new[i++] = HIGB_U2(save_handler);
new[i++] = LOWB_U2(save_handler); /* end_pc */
new[i++] = HIGB_U2(save_handler);
new[i++] = LOWB_U2(save_handler); /* handler_pc */

add_cp_entry(CONSTANT_Utf8, "java/lang/InterruptedException", 0u, 0u);
add_cp_entry(CONSTANT_Class, "", constant_pool_count - 1u, 0u);

new[i++] = HIGB_U2(constant_pool_count - 1u);
new[i++] = LOWB_U2(constant_pool_count - 1u); /* catch_type */

/* Attribute Count */

new[i++] = 0u;
new[i++] = 0u;

if (i != a -> attribute_length)
    javab_out(-1, bytes perdidos em parallelize_loop());

a -> info = new;

free(bytes);
free_new_bytes();
}
```

Apêndice C

Exemplos de linguagens e aplicações paralelas

Neste Apêndice serão apresentadas as linguagens e aplicações paralelas encontradas comercialmente ou ainda em fase de estudo. O que se pretende mostrar é um breve relato dos principais aspectos das linguagens e aplicações existentes, buscando sempre destacar os aspectos mais relevantes de cada exemplo. Finalmente deve-se dizer que devido ao enorme número de implementações encontradas, apenas as mais importantes e com certo interesse estarão relacionadas. Algumas apenas serão referenciadas, e sempre que possível haverá alguma referencia bibliográfica, ou mesmo um endereço eletrônico da Internet. Sempre que possível serão colocados exemplos de comandos ou comportamentos paralelos das linguagens, com o intuito de exemplificar as mesmas.

Entre as principais ferramentas encontramos as já citadas bibliotecas para passagem de mensagens, que na realidade são novas funções construídas para serem ligadas através de *links* (linkadas) a programas de linguagens já existentes como C, FORTRAN e C++. Entre as principais que serão apresentadas a seguir encontramos PVM, MPI

Quanto às linguagens de programação, deve-se utilizar um novo compilador. As linguagens de programação permitem verificação de corretude e otimizações maiores, porém isso nos remete a incorporação de um novo modelo de programação nas linguagens. As principais que surgem quando falamos de linguagem de programação distribuída, nos levam a fatores como a comunicação entre processos, com referencia a memória compartilhada e troca de mensagem. Outros fatores são quanto a criação de processos e o suporte ao paralelismo, e ainda a forma como são feitos os mapeamentos de processos a

processadores, na forma estática ou x dinâmica. Entre as principais que serão apresentadas a seguir encontramos HeNCE, Linda, HPF, CC++, SR, C*.

a) A linguagem C*

A linguagem C* é uma extensão da linguagem de programação ANSI C e foi projetada para suportar a programação paralela baseada no paralelismo de dados. Os recursos extra incluídos na linguagem incluem recursos para *broadcasting*, redução e comunicação entre processadores.

Uma variável pode ser definida como sendo uma variável paralela. Associado a cada variável paralela está o seu molde (*shape*), que define como os elementos desta variável paralela é distribuída pelos processadores da máquina. Por exemplo, o comando ***shape* [16384] empregados;** é um comando que define um molde unidimensional com 16384 posições chamado empregados. Após um molde ser especificado, as variáveis paralelas podem ser especificadas de acordo com certo molde. Por exemplo, o comando ***unsigned int: empregados cadastro;*** especifica uma variável paralela cadastro como sendo uma variável do tipo ***unsigned int***. Uma operação com paralelismo de dados é especificado através do comando ***with***. Adotando o molde corrente, ***with*** é adotado para a maioria das operações sobre variáveis paralelas. Por exemplo,

```
shape [16384] empregados;  
  
unsigned int: empregados cadastro;  
  
with (empregados)  
  
cadastro = 0;
```

Outro recurso acrescido a C* são os operadores de redução. Por exemplo, no trecho abaixo a variável soma conterà a soma de todos os elementos ativos da variável paralela x.

```
int: numeros x;  
  
soma = (+=x);
```

b) A linguagem HeNCE

A linguagem **HeNCE** é uma ferramenta de programação paralela que não requer conhecimento de programação paralela por seus usuários. Na linguagem **HeNCE** o usuário especifica, explicitamente, paralelismo entre sub-rotinas desenhando um grafo onde os nós, no grafo, são sub-rotinas escritas em Fortran ou C e a linguagem **HeNCE** irá executar automaticamente, em paralelo, as sub-rotinas sempre que possível através de uma rede de máquinas heterogêneas. A linguagem **HeNCE** utiliza o PVM para comunicação entre processos.

A linguagem **HeNCE** permite ao usuário configurar máquinas virtuais e estimar os custos para cada par sub-rotina e máquina. Deste modo o programador pode fazer uso das capacidades especiais de cada máquina. Ele também permite ao usuário acompanhar a execução do programa (trace). Este exercício irá mostrar como editar um programa em C, construir uma máquina virtual, executar o programa demonstração em C e olhar o trace de sua execução. Neste laboratório o termo "clique" significa selecionar um botão ou campo com a tecla indicada do mouse.

c) A linguagem HPF (High Performance Fortran)

A linguagem HPF define um conjunto de extensões à linguagem Fortran90. Em HPF o mesmo código é executado sobre um subconjunto de dados pertencentes a uma mesma estrutura, como por exemplo uma estrutura de dados (*arrays*). Neste caso o compilador particiona os elementos dos *arrays* distribuindo-os pelos processadores para paralelizar as operações a serem executadas sobre eles.

Por exemplo operações sobre valores escalares podem ser aplicadas a *arrays*. Neste caso os *arrays* do tipo inteiro A, B e C podem ser operados como se fossem um único elemento de tal for que poder-se-ia obter o resultado em C dos vetores A e B simplesmente especificando-se $C=A+B$, ou ainda em C o valor de todos elementos de A fazendo-se $C=\text{SQRT}(A)$. Além disso novas operações intrínsecas a arrays foram definidas

Foram acrescentadas também novas instruções para especificar localidade, como **ALIGN**, com a função de alinhar arrays; a instrução **PROCESSORS**, com a função de criar um *array* de processadores abstratos; a instrução **DISTRIBUTE**, com a função de mapear *arrays* em processadores abstratos.

d) A linguagem OZ

A pesquisa visando o desenvolvimento e a implementação do OZ começou em 1990, quando Gert Smolka fundou o Laboratório de Programação de Sistemas no DFKI (German Research Center for Artificial Intelligence). A primeira implementação do OZ foi lançada em Setembro de 1992. Daí o OZ evoluiu rapidamente, até um ponto fixo que foi alcançado em Janeiro de 1995. Atualmente seu desenvolvimento está estável em forma e versão. A linguagem OZ foi desenvolvida como uma linguagem de alto nível para substituir linguagens sequenciais de alto nível, tais como Lisp, Prolog e Smalltalk.

É, portanto, uma linguagem concorrente que combina um sistema de objetos com aspectos avançados para processamento simbólico e solução de problemas. As primeiras aplicações do OZ incluem simulações, sistemas multi-agentes, processamento de linguagem natural, realidade virtual, interfaces de usuário gráficas, programações, utilização de tabelas e configurações. O projeto e implementação definiu-a como a primeira linguagem concorrente restrita com busca encapsulada.

A programação multi-paradigma empregada na linguagem OZ utiliza programação concorrente restrita (ccp) como fundamento. A concorrência é importante porque é a forma mais comum de controle e pode ser facilmente restringida aos requerimentos de qualquer estilo de programação. As restrições são importantes porque elas produzem uma maneira simples e uniforme de se expressar comunicação e sincronismo.

O Modelo de Programação OZ (OPM) é um modelo de programação concorrente resumindo programação funcional e programação orientada a objetos como partes de um modelo geral. Isto é particularmente interessante para programação concorrente orientada a objetos, para a qual não existia um modelo formal compreensivo até hoje. O modelo pode ser estendido, podendo expressar soluções encapsuladas de problemas, generalizando as capacidades de solução de problemas da programação lógica restrita.

O OPM tem sido desenvolvido junto com a linguagem de programação OZ, a qual é projetada para aplicações que requerem computação simbólica complexa, organização em agentes múltiplos, e controle em tempo real.

A computação no OPM é feita em um chamado espaço computacional, o qual hospeda um número de tarefas conectadas a um depósito compartilhado. A computação avança pela redução das tarefas, as quais podem manipular o depósito e criar novas tarefas. Quando uma tarefa é reduzida ela desaparece.

Entre as principais características da linguagem OZ encontram-se:

Variáveis Lógicas e Controle Concorrente: A linguagem OZ é baseada em variáveis lógicas e controle de concorrência. Variáveis podem ser usadas antes de terem valores associados a elas. Se um cálculo precisa do valor de uma variável que ainda não foi associada, o cálculo é suspenso e automaticamente retomado quando o valor da variável se torna disponível. Cálculos com informações parciais : OZ pode calcular com variáveis cujos valores estão só parcialmente especificados. Informações sobre os valores das variáveis são especificados por meio de restrições, as quais são automaticamente combinadas por um método de simplificação gerando unificação.

Linguagem de alto nível : Funções, procedimentos, objetos, classes, métodos, mensagens e módulos são criados dinamicamente e são nomeados como valores de primeira qualidade, os quais podem ser passados como argumentos, retornados como resultados, testados por igualdade, e gravados nos atributos dos objetos.

Máxima Composicionalidade : Cada cálculo no OZ é descrito como uma expressão. Expressões são definidas por composição concorrente, declaração local, e abstração procedimental. Já que a máxima composicionalidade no OZ não é obstruída por um tipo de instrução estática, os módulos paramétricos aparecem naturalmente.

Objetos Concorrentes e Herança Múltipla : Objetos são o conceito primário de estruturas concorrentes no OZ. Eles combinam encapsulamento de informações através de abstração procedimental com estados e exclusão mútua. Os objetos devem ser vistos como agentes provedores de serviços. Os serviços de um objeto são fornecidos através de métodos e podem ser solicitados por envio de mensagens ao objeto. Objetos são criados como instâncias de classes. As classes definem métodos, atributos e aspectos. A definição de uma classe pode envolver herança de outras classes. Objetos e classes são tratados como cidadãos de primeira classe. Eles são criados dinamicamente. Objetos podem ser gerados

como agentes concorrentes. A criação de agentes comunicadores no OZ é tão simples como escrever funções em Lisp.

Valores e Estruturas de Dados : Valores em OZ incluem *integers*, *floats*, *strings*, listas e registros. Vetores e dicionários são tratados como objetos. Estruturas de dados compostas combinam-se com variáveis lógicas e valores do mais alto nível. Estruturas de dados cíclicas e correntes infinitas são facilmente expressadas.

Nomes de Primeira Classe : No OZ, os nomes dos campos dos registros métodos e atributos são valores de primeira classe que podem ser especificados através de variáveis. Valores no OZ incluem uma classe de nomes primitivos que são criados dinamicamente, e que não podem ser violados. Em conjunção com escopo léxico, os nomes de primeira classe proporcionam aspectos que só são disponíveis nas linguagens tipadas : pode-se expressar tipos de dados abstratos assim como métodos privados e atributos de objetos.

Controle em Tempo Real : O OZ é bem apropriado para programação reativa com requerimentos em tempo real. O OZ emprega um escalonador de tarefas assíncrono, preemptivo com prioridades controladas. O acesso em tempo real é disponível, bem como o controle em intervalos de tempo de execução.

Pode-se entender que a programação multiparadigma é uma área de pesquisa atual, na qual ainda não foram realizados estudos ou projetos mais detalhados. Desta maneira, a linguagem OZ surge como pioneira, como uma breve introdução a esta interessante área de pesquisa. De acordo com seus criadores, a pesquisa futura no OZ se encaminhará para a extensão do modelo para programação distribuída, fato que interessa muito a comunidade científica, pois será a união de duas das mais importantes áreas de pesquisa da informática hoje em dia: programação multiparadigma e processamento paralelo e distribuído.

e) A linguagem Adaptador (Automatic Data Parallelism TranslatOR)

A linguagem **Adaptador** foi desenvolvida pelo GMD - *German National Research Center for Computer Science*, podendo ser usada de modo interativo no ambiente X-Windows ou na modalidade batch. O **Adaptador** é uma ferramenta para transformar programas de dados em paralelo, escritos em FORTRAN, em programas paralelos com passagem de mensagem explícita. A linguagem de entrada é um subconjunto do High

Performance Fortran (HPF) e Connection Machine Fortran (CMF). A ferramenta Adaptor possui uma ferramenta denominada *fadapt* que perfaz a transformação do programa de dados em paralelo para o programa paralelo com passagem de mensagem explícita. Ao usar a ferramenta *fadapt* o usuário escolhe, entre outras opções, qual o sistema de passagem de mensagens a ser utilizado para a execução do programa paralelo, entre as quais podem ser ferramentas PE (*Parallel Environmen*t da IBM (vista adiante) ou PVM 3.x – (*Parallel Virtual Machine*), de domínio público. Como resultado da transformação automática, o *fadapt* gera, além do programa paralelo com passagem de mensagem explícita, um Makefile correspondente para sua compilação e linkedição.

A compilação é feita pelo compilador xlf Fortran. 77 nativo. A linkedição é efetuada com a biblioteca do Adaptor - DALIB (*Distributed Array Library*) que contém as rotinas de comunicação e de operações com *arrays* necessárias à execução do programa paralelo.

A vantagem em se usar o **Adaptor** é que o usuário não precisa conhecer os comandos de passagem de mensagens como os do PE ou PVM. O usuário tem que entender de algumas das premissas do paralelismo, escrevendo o seu programa de dados em paralelo usando, no nosso caso, a linguagem Fortran do **Adaptor**. Assim sendo o esforço é na programação com os paradigmas do paralelismo. Mas é importante observar que **Adaptor** não transforma automaticamente código serial em paralelo, ou seja, não possui mecanismo para detectar se o código serial embutido em uma iteração pode ou não ser paralelizado.

f) A linguagem Compositional C++

É um conjunto de extensões a C++ para programação paralela, disponível em domínio público (p/UNIX). Instâncias de classes globais criam espaços de endereçamento independentes, dentro dos quais vários *threads* podem ser criados. A comunicação ocorre basicamente através de chamadas a métodos remotos. Os *threads* internos a um objeto podem ser comunicar através de variáveis do objeto. Tipos básicos e ponteiros globais podem ser transferidos como parâmetros de chamada remota sem interferência do programador. Sem a necessidade da definição de uma função de transferência.

Implementação de seis abstrações:

- 1.objeto processador: localidade do processo
- 2.ponteiro global: ligação entre processos
- 3.thread: execução concorrente
- 4.variável sync: sincronismo de threads, ou para uma sincronização mais fina.
- 5.função atomic: operações sobre mesmo objeto
- 6.função de transferência: acesso a estruturas por vários objetos processadores.

g) O ambiente PE

O Parallel Environment consiste em um ambiente para desenvolvimento e execução de programas paralelos em Fortran, C e C++. É constituído por ferramentas para desenvolvimento, execução, depuração e monitoramento de aplicações paralelas. O PE foi desenvolvido para funcionar sobre arquiteturas IBM RISC System/600 Scalable PowerParallel, como a do SP2, sobre clusters de máquinas RISC System/6000, ou uma combinação de ambos. Suporta os dois modelos básicos de programação paralela : SPMD, MPMD. Em programas **SPMD** (Single Program, Multiple Data), as tarefas que executam em paralelo são idênticas, mas trabalhando sobre diferentes conjuntos de dados. Programas **MPMD** trabalham com cada tarefa executando um programa diferente, sobre o mesmo conjunto de dados ou sobre dados diferentes..

É, também, um ambiente simples e amigável desenvolvido para facilitar a transição de uma aplicação serial para paralela, em outras palavras: PE permite o desenvolvimento e execução de programas paralelos usando os mesmos métodos e mecanismos de programas seriais. Este ambiente possui três conceitos principais chamados *Partition*, *Partition Manager* e *Resource Manager*. Estes conceitos são brevemente descritos a seguir:

Partition – É um grupo de nós processadores no qual se executa o programa

Partition Manager – é o programa que estabelece e controla sua partição, através de subrotinas que são ligadas a um programa e um *daemon* chamado **pmd**. Tem por responsabilidade: requisitar os nós necessários para o seu job; copiar o executável do nó

inicial para cada nó de sua partição; carregar o executável em cada nó de sua partição; e finalmente definir a entrada e saída padrões

Resource Manager - gerencia os nós que estão processando um job paralelo em um determinado momento. Quando requisitado pelo Partition Manager, o Resource Manager irá alocar os nós para utilização. O seu daemon **resd** é executado em cada nó.

Após o desenvolvimento do código-fonte da aplicação, o desenvolvedor passa para a fase de testes e modificações. Ele executa o programa usando o **POE (Parallel Operating Environment)**, a partir de qualquer um dos nós do **SP2**. **POE** é um ambiente de execução desenvolvido para mascarar aos usuários as diferenças entre execuções seriais e paralelas. Com ele, o usuário pode disparar sua aplicação em vários nós remotos simultaneamente. Quando o programa é disparado, o POE inicia o Partition Manager, que aloca os nós e inicializa o ambiente local. Dependendo da configuração, ele pode utilizar um arquivo de hosts, o gerenciador de recursos do SP2, ou ambos para alocar os nós. O arquivo de hosts deve conter uma lista explícita das máquinas que se deseja utilizar, enquanto que o gerenciador de recursos aloca automaticamente os nós disponíveis, baseado em sua avaliação. Utilizando o modelo SPMD, o Partition Manager inicializa o mesmo programa em todos os nós. Entretanto, para o modelo MPMD, deve ser informado que o programa irá executar em cada um dos nós. O **POE** é composto de :

- **Uma série de scripts de compilação paralela**, que permitem fazer a linkagem entre o programa do usuário e as bibliotecas, possibilitando a comunicação entre o nó principal e as tarefas executando em Hosts remotos.

Uma série de variáveis de ambiente, que podem ser utilizadas para alterar a forma de execução do POE, e conseqüentemente, da aplicação. Permite alterar vários aspectos, como alocação de nós, modelo de programação utilizado, e a forma de comunicação entre as tarefas;

- **Dois ferramentas gráficas de análise de desempenho :**
- **Program Marker Array** - Consiste em um vetor programável de pequenos *leds*, associados com cada tarefa. Esses *leds* variam suas cores de acordo com o *status* da tarefa, de modo que o usuário tenha, a cada instante um retorno da situação atual dos

diversos componentes de sua aplicação. Essa ferramenta é particularmente útil para identificar se um programa está com algum problema, como um *loop* infinito ou bloqueado.

System Status Array - Essa ferramenta fornece rapidamente o status da utilização de seus nós de processamento.

Parallel Debugging - Permite a depuração da aplicação. É composto pelo **pdbx** - depurador de modo texto - e pelo **xpdbx** - o correspondente em ambiente gráfico. Para executar o **Parallel Debugging** digite na linha de comando : **pedb**. Como esse programa utiliza interface gráfica, é necessário exportar o *display* para sua máquina local.

h) O programa LoadLeveler É um programa que administra e distribui *jobs* em um *cluster* ou em uma máquina multi-processadora (SP1). Tem como função decidir quando e onde um job irá rodar, baseado nas opções apresentadas pelo usuário e pelo administrador. A comunicação entre o usuário **LoadLeveler** é feita através de comandos bastante simples ou de uma interface gráfica denominada *xloadl*.

Os jobs são submetidos ao **LoadLeveler** através de um arquivo de comandos, que assemelha-se a um *script* do UNIX. Os *jobs* ficam retidos na fila até que o **LoadLeveler** possa alocar os recursos necessários à sua execução. Uma vez completado o job o usuário poderá ou não ser avisado pelo **LoadLeveler**. O **Loadleveler** tenta igualar o nível de processamento no SP1 ou no cluster.

URL: <http://www.lncc.br/loadl/loadl.html>

i) Programação Lógica- Projeto Opera

A Programação em Lógica possui dois tipos de paralelismo implícito, inerentes à estrutura de cláusulas do programa, que podem ser inferidos automaticamente:

- **Paralelismo OU:** consiste na execução em paralelo de cláusulas de um mesmo predicado, por exemplo, considere o predicado avô:

$\text{avô}(X,Y) :- \text{pai}(X,Z), \text{pai}(Z,Y).$

$\text{avô}(X,Y) :- \text{pai}(X,Z), \text{mãe}(Z,Y).$

Caso fosse feita uma consulta para descobrir o avô de Maria, na forma $avô(X, maria)$, as duas cláusulas poderiam ser disparadas em paralelo para descobrir o resultado através da variável X . Neste caso estaríamos explorando o paralelismo OU **implícito**, pois ao usuário ficaria transparente o modo como tal consulta seria efetuada.

- **Paralelismo E:** consiste na execução paralela de objetivos (literais) do corpo da cláusula, por exemplo, considere a cláusula hipotética abaixo:

$$a(X,Y) :- b(X), c(Y).$$

Ao fazer-se a consulta $a(X,Y)$, pode-se disparar os objetivos $b(X)$ e $c(Y)$ concorrentemente, e portanto explorar o paralelismo E implícito da linguagem. De fato, esse tipo de paralelismo pode ser dividido em dois subtipos: **paralelismo E independente**, no qual apenas objetivos que não compartilhem variáveis podem ser executadas em paralelo, e **paralelismo E dependente**, onde objetivos podem compartilhar variáveis, o que estabelece uma relação de produtor consumidor. Assim, o tema central do projeto OPERA é o estudo e a utilização das várias técnicas de exploração de paralelismo na programação em lógica.

Alguns dos objetivos atuais são: escalonamento distribuído de tarefas; escalonamento de tarefas para diversas arquiteturas; avaliação de granulosidade; utilização de informações de granulosidade para auxílio ao escalonamento; interpretação abstrata; integração dos modelos E e OU; paralelização de programas em lógica com restrições; otimização dos protótipos já desenvolvidos. Esses objetivos devem ser alcançados respeitando, o máximo possível, duas premissas básicas que são a exploração do paralelismo implícito, e a máxima exploração do paralelismo.

j) A linguagem CSP (Communicating Sequential Processes)

É uma linguagem de programação concorrente criada por C. A. R. Hoare em 1978. Baseada num conjunto mínimo de conceitos: processos sequenciais que comunicam por mensagens usando primitivas de input e output. Inspirou a linguagem OCCAM usada nos transputers. A comunicação entre processos baseia-se em comandos de *input* e *output*. A identificação dos processos é explícita tanto para emissão como para recepção. A comunicação é síncrona. Algumas de suas restrições impõem que: os processos têm que

ser disjuntos; cada processo só pode aceder às suas próprias variáveis; Todos os processos são executados de forma concorrente. O comando termina apenas quando todos os processos terminarem.

k) A Linguagem DPC++

É uma linguagem para programação distribuída orientada a objetos baseada em C++, ou seja Processamento Distribuído em C++. A linguagem DPC++ , é um projeto da UFRGS. Sua principal área de atuação são aplicações que necessitam de concorrência para melhorar seu desempenho, pois possui recursos de distribuição de tarefas. As características de orientação a objetos do DPC++ são as herdadas do C++. Inclusive, a sintaxe dos programas é a mesma. Porém em DPC++ foi introduzido um novo tipo de classe, a classe dos objetos distribuídos. A simplicidade da escrita de aplicações distribuídas é garantida pelo uso de recursos de manipulação do ambiente distribuído, provido pelo pré-processador DPC++. Este gera o código para ser submetido ao compilador C++.

O mecanismo de comunicação utilizado é o de *sockets*, com datagramas (UDP), a nível de transporte TCP/ IP, enviando as mensagens. Este mecanismo não oferece controle de fluxo, controle de erros e não é orientado à conexão. No entanto, o uso deste protocolo de comunicação é mais adequado ao modelo, além de permitir um desempenho melhor do sistema com confiabilidade aceitável se for empregado em redes locais. Uma das principais vantagens desta linguagem é que ela concilia as facilidades da programação orientada a objetos com a obtenção de melhores índices de desempenho na execução do programa, o que é possível através da execução distribuída da aplicação. Os conceitos básicos de orientação a objetos são aplicados ao modelo DPC++, onde encontram-se objetos que encapsulam todas as suas propriedades: dados (memória interna) e funções (métodos). A execução de programas é feita invocando-se métodos dos objetos, através do envio de mensagens. Ainda, no modelo DPC++ é explorada a concorrência de execução entre objetos, utilizando-os em sistemas distribuídos.

Um objeto distribuído é composto basicamente por três partes:

Métodos: o conjunto de métodos corresponde à implementação de todos os serviços oferecidos pelo objeto;

Estado Interno: o estado interno (ou memória) é composto pelo conjunto de dados manipulados pelo objeto. Estes dados não podem ser referenciados diretamente por outros objetos distribuídos, pois são considerados locais;

Interface de Comunicação: é através desta que o objeto recebe as mensagens de invocação dos seus métodos. A interface é ligada diretamente à rede de comunicação e é endereçada pelo identificador do objeto. Sua função é organizar as mensagens recebidas, por ordem de chegada, ativar os métodos correspondentes e, se necessário, enviar mensagens com respostas às solicitações recebidas.

1) - PVM (*Parallel Virtual Machine*)

O PVM é software de domínio público utilizado para desenvolvimento e execução de aplicações concorrentes e/ou paralelas. Foi desenvolvido para funcionar em uma série heterogênia de computadores, conectados de alguma forma, como por exemplo através de uma rede. Consiste na versão padronizada e gratuita, disponível na Internet. Possui diversas bibliotecas que podem ser incluídas nas aplicações dos usuários, de forma a incluir funções de gerenciamento, sincronização, etc. Possui ainda uma interface gráfica, o XPVM, também disponível, que possui todas as funções encontradas na console PVM, mais algumas ferramentas de visualização gráfica de vários aspectos da aplicação, como desempenho, intercomunicação, etc.

Na programação paralela divide-se um programa em vários módulos, a serem executados em diferentes estações paralelamente. Cada módulo, chamado de "tarefa" trabalha em cima da parte de dados que lhe é conferida e, de acordo com a estrutura do programa, troca os resultados com outras tarefas ou com uma tarefa mestre. Essa troca de resultados é feita através do que se conhece por *message passing*, que pode se dar entre CPU's de uma mesma máquina ou até mesmo entre máquinas independentes.

O acrônimo **PVM** é formado pelas palavras *Parallel Virtual Machine*, e foi Iniciado em 1989, no ORN. O **PVM** é um projeto atualmente desenvolvido por *Vaidy Sunderman* na Universidade de Emory, *Al Geist* em ORNL, entre outros que visa, principalmente o progresso na Ciência, sendo usado em várias instituições no mundo.

É importante, antes de falar-se de PVM em si, ver-se algumas definições e conceitos intrínsecos à este software. Um dos primeiros conceitos é o de *máquina virtual* que na realidade é o conjunto de máquinas, cada uma com sua CPU operando, através do PVM como uma máquina única. Esta *máquina virtual* é criada pelo programador que então dividirá o programa em processos para rodarem nas CPU's que a compõem.

Sob o **PVM** uma coleção de computadores seriais, paralelos e vetoriais, aparecem como um único computador de memória distribuída. Ele fornece as ferramentas para lançar tarefas na máquina virtual e permite que estas se comuniquem durante o processamento, afim de solucionar o problema. Um exemplo de implantação do **PVM** é o **PVM3**, sendo um software que permite, a partir de uma rede heterogênea de computadores *Unix*, chamada de *Máquina Virtual*, a distribuição e comunicação entre as diversas tarefas que compõem um programa paralelo.

O PVM, software de paralelismo instalado em rede, consiste de duas partes: a primeira parte, é um *daemon*, chamdo *pvmd3*, que deve estar instalado em todas as máquinas com as quais se pretende compor a máquina virtual. A segunda parte consiste de uma library, *pvm3.h* no caso do C e *fpvm3.h* no caso do FORTRAN, à ser incluída pelo usuário em seu programa, a qual implementa as funções do PVM que serão chamadas para lançar outros processos, trocar mensagens entre os processos e terminá-los ao fim do programa.

O primeiro passo, no PVM é definir que máquinas comporão a máquina virtual. Assim, através do programa *pvm* adiciona-se as máquinas uma a uma, com o comando *add hostname*, onde *hostname* será o nome das estações que participarão do problema (atomo, lepton, pion, etc.). Uma maneira mais simples consiste em criar-se um arquivo de configuração na conta do usuário que conterà as máquinas à serem usadas no programa paralelo. Uma vez criada a máquina virtual, o programador lança o seu processo, o qual, por sua vez lançará outros processos à medida do necessário em outra máquinas que compõem a máquina virtual.

No PVM cada processo, ao se iniciar, recebe um *Task ID* (TID) que será único e terá como função identificar diversos processos do programa afim de permitir a troca de

informações entre eles. Assim um processo enviará uma mensagem para um TID e não para outro processo em determinada máquina.

Podemos, de forma a resumir o conceito de PVM, relacionar os seguintes princípios:

1. PVM é uma Máquina Virtual configurável pelo usuário. A aplicação paralela é executada em um conjunto de máquinas definidas pelo próprio usuário
2. PVM permite um acesso seletivo ao *hardware*: O programa paralelo pode "ver" a máquina virtual como uma coleção de CPU's sem características especiais, ou pode aproveitar as vantagens da arquitetura de cada CPU da máquina virtual.
3. PVM oferece computação baseada em processos: A unidade de paralelismo no PVM é uma tarefa (*task*), que consiste em um código sequencial a ser executado em uma CPU da máquina virtual.
4. PVM opera segundo o modelo de "message passing" explícito. Os *tasks* que formam o programa paralelo dividem o trabalho através da troca de mensagens determinada pelo usuário a ser realizada pelos mesmos.

A organização das tarefas no PVM se dá segundo alguns paradigmas fundamentais a saber:

- Paradigma "Crowd Computing" - Consiste numa coleção de processos intimamente relacionados, tipicamente executando o mesmo código, que trabalharão em porções diferentes do código, trocando resultados intermediários entre si.
- Modelo *Master Slave* - Neste um processo é lançado pelo usuário, se encarregando de lançar outras tarefas que executarão o trabalho computacional, se encarregando de coletar os dados e apresentar os resultados.
- Modelo Nó á Nó - Neste modelo múltiplas instâncias de um único programa são executadas, com um processo cuidando do trabalho não computacional, além de contribuir eventualmente com o cálculo também.
- Paradigma de computação em árvore - Com este paradigma, os processos são lançados (usualmente no decorrer do cálculo) em uma estrutura em árvore estabelecendo

portanto uma relação pai - filho entre os processos, em oposição ao paradigma ``Crowd'', no qual há uma relação em forma de ``estrela". Este algoritmo é muito útil quando não se sabe *a priori* a carga total de trabalho à ser executado.

Entre as vantagens do PVM pode-se relacionar a possibilidade efetiva de utilização de computação *paralela*, ainda que em arquitetura não-paralela. Pode-se indicar ainda a Redução do tempo total de execução do programa, visto possibilita-se uma maior capacidade de paralelização escalável e dinâmica. A facilidade de instalação e uso, ser um Software de domínio público de grande difusão e aceitação também são fatores fortes para seu sucesso. Sua flexibilidade, com grande variedade de arquiteturas e redes de trabalho, recurso computacional facilmente expansível, e independência das aplicações, somando-se ao fato que é um software de fácil atualização, e ainda apresentar facilidade de Programação graças a bibliotecas para linguagens universalmente usadas (Fortran e C) fazem do **PVM** uma ferramenta poderosa e com grande utilização.

Apesar de não ser considerado ideal para computação de alto desempenho, o PVM consiste em uma ótima alternativa para a programação paralela aonde não se dispõe, ou não se pode dispor de arquitetura própria para o paralelismo. Assim, lançando-se mão de diversas CPU's em uma rede heterogênea, pode-se aproveitar as características de cada processador afim de obter um melhor resultado sem haver necessidade de adquirir máquinas mais poderosas, e conseqüentemente mais caras.

m) PVMe (*Parallel Virtual Machine Enhanced* -IBM)

O **PVMe** é um programa desenvolvido pela IBM que é compatível a nível de fonte com o pacote de software de domínio público PVM. A letra "e" abrevia a palavra "*enhanced*". Sua função é a mesma do PVM de domínio público, que consiste em permitir o desenvolvimento e gerenciamento de aplicações paralelas. Entretanto, como foi desenvolvido especificamente para as arquiteturas IBM, ele explora de forma muito mais eficiente os recursos disponíveis no SP2. Este software utiliza o protocolo LSP (*lightweight switch protocol*) que roda sobre a chave de comunicação de alta velocidade do SP1 (*high-speed switch*).

As diferenças entre o **PVM** e o **PVMe** é que o **PVM** pode ser utilizado em um conjunto heterogêneo de máquinas, enquanto que o **PVMe** só pode ser utilizado no SP1. No **PVM**, existe um processo *daemon* sendo executado em cada estação pertencente à máquina virtual; no **PVMe**, existe uma única *daemon*, rodando somente em uma máquina. Também no **PVM** podem existir várias tarefas em um determinado nó, ao passo que no **PVMe** só pode existir uma tarefa por nó. Embora tanto o **PVM** quanto o **PVMe** podem se comunicar pelo sistema de comunicação de alta velocidade (*high-speed switch*) do SP1, o **PVM** só pode fazê-lo utilizando o protocolo IP; o **PVMe** utiliza o *lightweight switch protocol*. Além disso no **PVM**, o usuário especifica as estações que formarão a máquina virtual através do arquivo *hostfile*; no **PVMe**, o usuário indica o número de nós desejados, e o gerenciador de alocação de nós (*Resource Manager*) se encarrega de selecionar os nós disponíveis. Existem ainda algumas diferenças na utilização do **PVMe** em relação ao **PVM** padrão, mas seu código é compatível com a versão **PVM** 3.2.5. Outras mudanças: existe um único processo *daemon* (**pvmd**) ; para cada tarefa existe um processo 'reader' ; a comunicação entre processos é feita por memória compartilhada .

Ainda entre as diferenças, quando se utiliza grupos dinâmicos no **PVM** é necessário se "linkar" a bibliotecas especiais; este "link" não é necessário para o **PVMe**. As bibliotecas de "run-time" são diferentes para o **PVM** e o **PVMe**. Então os programas devem ser "linkados" com bibliotecas diferentes para o **PVM** ou o **PVMe**, dependendo de qual deles vai ser utilizado.

n) MPI (*Message Passing Interface*)

O surgimento do MPI foi em Maio de 1994 quando a documentação do MPI foi apresentada (versão 1.0) e posteriormente atualizada em Junho de 1995 (versão 1.1) no Ridge National Laboratory. O motivo de sua criação foi a elaboração de um conjunto padronizado de rotinas para *Message Passing*, que pudesse ser implementado por diferentes fabricantes, obtendo o melhor desempenho possível.

O MPI consiste em um conjunto de bibliotecas, que podem ser utilizadas no desenvolvimento de aplicações paralelas. Elas implementam um sistema de troca de mensagens entre processos, que podem estar localizados em diferentes processadores, no caso de uma máquina paralela, ou em diferentes máquinas ligadas por uma rede de

interconexão.

Existe versão anterior ao MPI, denominada MPL, sendo também formada por uma série de bibliotecas, com a mesma função, e apresentando compatibilidade total com seu sucessor. Isto significa que aplicações desenvolvidas com MPL, podem ser utilizadas sem a necessidade de alteração no código fonte. O MPI veio apenas acrescentar novas funções às bibliotecas, mantendo a mesma funcionalidade da versão anterior.

Por se tratar de uma série de bibliotecas, torna-se difícil uma documentação que não seja a listagem das bibliotecas a serem incluídas nas aplicações e a descrição formal de cada uma das funções disponíveis. Entretanto, a descrição de funcionamento de cada uma das funções implementadas nas bibliotecas estão disponíveis para o acesso através dos manuais on-line a partir da própria conta do usuário do sistema SP2

As principais características do padrão MPI são sua eficiência, sendo cuidadosamente projetado para executar eficientemente em máquinas diferentes. Por exemplo: especifica somente o funcionamento lógico das operações. Deixa em aberto a implementação. Os desenvolvedores otimizam o código usando características específicas de cada máquina. Uma outra característica importante é sua facilidade, visto que esta ferramenta define uma interface não muito diferente dos padrões PVM, NX, Express, P4, etc. e acrescenta algumas extensões que permitem maior flexibilidade. A portabilidade, sendo compatível para sistemas de memória distribuída, *shared-memory*, *NOWs (network of workstations)* e uma combinação deles. Sua transparência permite que um programa seja executado em sistemas heterogêneos sem mudanças significativas. Um fator de segurança provê uma interface de comunicação confiável. O usuário não precisa se preocupar com falhas na comunicação. E finalmente uma importante característica que é a escalabilidade, visto que o MPI suporta escalabilidade sob diversas formas, por exemplo: uma aplicação pode criar subgrupos de processos que permitem operações de comunicação coletiva para melhorar o alcance dos processos.

Uma tendência que deve ser implementada no futuro, pelo *Oak Ridge National Laboratory e University of Tennessee*, é aquela que propõe viabilizar uma mistura entre as características do PVM e do MPI. Neste caso as funções seriam utilizar implementações específicas de hardware, quando disponíveis em máquinas multiprocessadoras, permitir

acesso à idéia de uma máquina virtual, com controle de recursos e tolerância a falhas, e finalmente usar a rede de comunicação PVM transparentemente para transferir dados entre diferentes implementações de MPI, permitindo a interoperabilidade. A característica final seria então um misto entre as funções do PVM, MPI, mantendo, tanto quanto possível, as identidades desses dois message-passing. Esta proposta foi chamada de PVMPI. E enquanto isso não se concretiza deve-se ponderar as características necessidade da aplicação para optar-se por MPI ou PVM. Além de verificar as configurações do ambiente, arquiteturas, para escolher o melhor processo. E finalmente analisar cautelosamente prós e contras do MPI e do PVM., como aqueles que estão relacionados na Tabela 1.2

Tabela 1.2 - Diferenças entre PVM e MPI

Portabilidade versus Interoperabilidade:	
PVM Nível de Interoperabilidade: Além da portabilidade (como o MPI), os programas permitem nível de interoperabilidade, tornando possível execuções em arquiteturas distintas sem quaisquer modificações.	MPI Nível de Portabilidade: Programas escritos para uma arquitetura podem ser compilados para uma outra arquitetura, compilados e executados sem quaisquer modificações.
Máquina Virtual:	
PVM Completa Abstração: <ul style="list-style-type: none">• O PVM permite que se veja a rede como uma coleção dinâmica de recursos computacionais (potencialmente heterogêneos), administrados como um computador paralelo único.• <i>Controle de processos:</i> capacidade de iniciar, interromper e controlar processos, em tempo de execução;• <i>Controle de recursos:</i> Bastante dinâmico - permite abstração, graças ao conceito da máquin virtual paralela;• <i>Topologia:</i> O PVM exige que o programador arranje manualmente tarefas em grupos, segundo a organização desejada.	MPI (Não suportado): <ul style="list-style-type: none">• O MPI não possui o conceito da máquina virtual paralela, centralizando-se no conceito de message-passing.• <i>Controle de processos:</i> Bastante restrito - permite somente o controle de grupos de tarefas;• <i>Controle de recursos:</i> Essencialmente estático - não suporta qualquer nível de abstração;• <i>Topologia:</i> Ainda que não possua o conceito de PVM, o MPI provê um alto nível de abstração, em termos de topologia.
Tolerância a Falhas:	
PVM Básico: Existem esquemas básicos de notificação de falha, para alguns casos. Porém, permite	MPI Rudimentar: As versões iniciais, não incluíam qualquer esquema de tolerância, mas a partir das

flexibilidade, de forma que, ainda em certas situações onde não existe resposta de uma máquina, uma aplicação receba resultados das outras.	últimas especificações, esquemas similares ao de PVM serão providos. Infelizmente, o modelo seguido é estático.
Segurança na Comunicação:	MPI
PVM Existente: Desde que o PVM possui um daemon mantendo a máquina virtual, ele pode usá-lo para criar rótulos únicos de contexto. Processos podem se comunicar com grupos, permitindo recuperação de falhas. As novas versões pretendem fazer uso de comunicadores, como os do MPI.	Excelente: O conceito de <i>communicators</i> no MPI, permite prover um elevado nível de segurança na comunicação, permitindo diferenciar mensagens de bibliotecas de mensagens de usuários.

o) SR - Linguagem Distribuída

É uma linguagem que se baseia em um conjunto de recursos, aonde um recurso é considerado padrão se instâncias desse próprio recurso são criadas dinamicamente, semelhante e classes em C++ . Uma criação de recurso sempre retorna uma *capability*, e cada programa tem um recurso principal diferenciado dos demais. Uma instância deste recurso é criada automaticamente quando é iniciado o programa. Os recursos e globais possuem uma parte de especificação (*spec*) e uma parte de implementação (*body*), que podem ser compilados separadamente. São usadas regras de compatibilidade estrutural para tipos, e a criação de novos processos pode ser implícito e explícito. No caso implícito, não é feito uma chamada a um comando para se fazer a criação de um novo processo. No caso explícito, é utilizado um comando para se criar um novo processo.

O conceito de máquina virtual, é diferente do conceito do PVM . Assim, uma maquina virtual provê um espaço de endereçamento próprio, podendo ter várias máquinas virtuais na mesma máquina e também ter máquinas virtuais em outras máquinas. Em sua comunicação e sincronização entre processos, os processos na mesma máquina virtual podem compartilhar variáveis, com semáforos, Em máquinas virtuais diferentes processos podem se comunicar com o uso de primitivas do tipo *send* e *receive*

O SR permite as quatros possibilidades de comunicação:

	Recebimento Síncrono	Recebimento Assíncrono
Envio Síncrono	Sim	Sim (<i>Call /Proc</i>)
Envio Assíncrono	Sim	Sim (<i>Send/Proc</i>)

p) A linguagem de programação paralela Linda

A linguagem Linda é concebida para a programação paralela com memória compartilhada. Ela é baseada no conceito de memória associativa compartilhada. Assim, tal como C e Fortran, Linda são as implementações compartilhadas. A linguagem utiliza espaço de tuplas que são implementados em memória distribuída, utilizando-se de primitivas, que são utilizadas para se fazer acesso as tuplas. Entre essas primitivas pode-se relacionar: OUT, tupla adicionada ao espaço de tuplas; IN, tupla satisfazendo *template* é retirada do espaço de tuplas, RD, tupla é lida mas permanece no espaço de tuplas; EVAL, com OUT, mas cria novo processo para avaliar a tupla. A sincronização é feita através de semáforos e Barreiras podem ser implementados com IN e OUT, a comunicação com IN e OUT também podem ser utilizada. A concorrência é feita com a utilização do EVAL para criar novos processos

Existe também a implementação Network Linda, que é a linguagem Linda implementado em memória distribuída. Em Linda o sistema escolhe máquina onde novo processo deve ser disparado, assim é possível o uso de informações de configuração e conhecimento de processos já disparados, porém o programador não tem como determinar máquina explicitamente.

Como consequência disso, a linguagem Linda é capaz de realizar um mapeamento de tarefas a processadores, através da distribuição de carga. Isto verifica-se importante para aplicações paralelas e para aplicações como sistemas operacionais distribuídos, pois vários estudos apontam uma alta porcentagem de tempo ocioso em redes de estações de trabalho, revelando que o compartilhamento faz com que máquinas sobrecarregadas tentem distribuir parte de sua carga para máquinas ociosas.

A linguagem Linda é capaz de efetuar um balanceamento de carga e aproveitamento de recursos, em outras palavras, realizar o melhor aproveitamento global de recursos, minimizar recursos de comunicação, modelar as tarefas e as máquinas disponíveis, e verificar a necessidade de migração para alcançar balanceamento. Com isso é possível realizar o compartilhamento de recursos, aproveitando recursos ociosos, e otimizar a alocação de recursos para aquela aplicação. Seu modelo de um sistema operacional com Balanceamento e Compartilhamento de Recursos utiliza um banco de processadores e estações pessoais, dispondo de dois tipos de alocação: a Estática e a dinâmica.

A alocação estática, que efetua a decisão sobre a alocação de cada processo é tomada antes do início da execução do programa utiliza grafos de precedência e grafos de interação. Os grafos de precedência, que são muitas vezes utilizados para alocação estática, descrevem as relações de precedência entre os vários processos. Os grafos de Interação, são usados quando não há relação de precedência clara entre as tarefas, sendo assim um grafo para descrever os padrões de comunicação. Neste caso, deve-se dizer que, algumas simplificações são usadas, como por exemplo as precedências são ignoradas, custo de processamento por tarefa é considerado igual em todos os processadores, possibilidade de superposição de comunicação e execução é ignorada. E as soluções utilizadas é usar algoritmos exatos, onde se faz a exploração de todas as possíveis soluções e, algoritmos Heurísticos, cujo comportamento é serem gulosos e iterativos.

A alocação dinâmica toma a decisão sobre a distribuição dos processos é baseada em informações obtidas em tempo real, possibilidade de utilizar migração de processos. Sua dificuldade maior é a de obter informação efetiva sobre o estado do sistema sem interferir sobre ele. Este método utiliza-se de indicadores de carga. A coleta de informação, cujo objetivo é manter as informações de carga atualizada sem sobrecarregar o sistema com troca de mensagens, é feita de forma voluntária, ou sob demanda. A coleta voluntária é feita quando cada máquina divulga seu estado a intervalos regulares de tempo ou quando ocorre alguma mudança de estado. E a coleta sob demanda é feita quando cada máquina divulga seu estado apenas quando este é requisitado.

Apêndice D

Conceitos e Ferramentas para o ambiente Java

D.1 O Compilador JIT (*Just-in-time*) [Chuc_98]

Considerando-se as descrições introdutórias do item 3.4.1, uma otimização consiste em rescrever os *bytecodes* para eliminar redundâncias. Por exemplo, o código de um *loop* poderia ser escrito:

```
for (int i = 0; i < someString.length ();  
    i++) { // do something here }
```

O *loop for* chamaria o método *length()* em *someString* em cada iteração do *loop*. Porém, como as *strings* em Java são imutáveis, o valor retornado por *length()*, é efetivamente uma constante. Isso significa que chamadas repetidas ao método são desnecessárias. Um otimizador, detectando esta condição, pode rescrever o *bytecode* para implementar o seguinte código:

```
Int temporaryValue = someString.length ();  
for (int i = 0; i < temporaryValue; i++) { // do something here }
```

O código acima é executado mais rapidamente porque o número de chamadas para invocação do método foi reduzido a um. Esta é uma otimização muito comum encontrada em compiladores existentes. Um otimização similar é converter o código:

```
for (int i = 0; i < 4; i++) { someArray[i]  
    = i; }
```

para algo como:

```
someArray[0] = 0; someArray[1] = 1; someArray[2]
= 2; someArray[3] = 3;
```

Esta técnica é conhecida como *loop unrolling*, e é primariamente uma otimização de velocidade. Ela elimina a inicialização e teste da variável *i*, reduzindo o número total de instruções executadas às custas de algum aumento no tamanho.

Outra otimização baseada em Java envolve a observação que para ser realmente orientada ao objeto, significa algumas vezes um *writing code* que não é muito eficiente. Considere uma classe *Foo* que tem o método *barValue()* e é implementado como segue:

```
class Foo {int private int bar = 4;
/* * return bar's value */ public int barValue ()
{ return bar; } }
```

Este pode parecer um exemplo fictício até você perceber que a classe *String* em Java tem um método *length()* que faz exatamente isto.

Agora uma referência no código que foi escrito como o seguinte código.

```
Foo SomeFoo = new Foo (); int z = someFoo.barValue ();
```

Poderia ser implementado como mostrado no código abaixo.

```
int z = someFoo.bar;
```

Esta versão seria mais rápida porque não existe referência de método, mas apenas seria válida se a variável exemplo *bar* não fosse privada. Se o compilador Java produzisse um código que referenciasse *bar* diretamente (como faz quando *flag* é usado com o comando *Javac*) então os *bytecodes* seriam rejeitados ao serem carregados pelo verificador de *bytecodes*, porque um objeto que não era *Foo* estava referenciando um campo *private* dentro de *Foo*, o que é estritamente proibido. Assim, é possível fazer algumas destas otimizações apenas depois que os objetos foram executados pelo verificador para uma checagem de integridade.

Mas quanta velocidade pode ser extraída destas otimizações? A resposta é alguma velocidade, mas não tanta quanto poderíamos desejar. Para uma comparação, o

compilador Java no Sun JDK pode ser instruído a fazer muitas destas otimizações passando pelo *-O flag*. Isto aumenta a velocidade do código em cerca de 15%, mas também gera um código que não passará em uma checagem rigorosa do verificador de *bytecodes*. Assim, o código compilado deste modo só é útil quando carregado a partir do disco local, onde o verificador não é executado, e não em *applets* onde uma classe *applet* pode ser rejeitada por não seguir as regras do linguagem.

É claro que, mesmo com estas otimizações, o código Java resultante ainda é cerca de 10 a 30 vezes mais lento que um código compilado em C++. Obviamente, o código que acessa as partes normalmente lentas de um sistema, como a interface como usuário, não é tão afetado quanto as partes baseadas em computações.

Para dar um exemplo de como as coisas podem ser alteradas, considere um dos *bytecodes* mais básicos do Java, *ALOAD*, que carrega uma referência de objeto de uma variável local e a coloca em uma pilha. A implementação real em C deste *opcode* é similar ao código mostrado abaixo.

```
/* load the stack */ optop[offset + OPTOP_OFFSET]
= vars[pc[1]]; pc += 2; optop += 1; continue
```

Posteriormente, este bit de código é encerrado em um comando *switch* cujas partes exteriores são mostradas no código abaixo:

```
while (1) { opcode = *pc; interruptor (opcode)
{... }}
```

A relação é então de aproximadamente sete instruções em C para cada *byte* de código simples em Java. Assim, se uma *string* de *bytecodes* *ALOAD* fosse substituída por códigos objetos equivalentes, a execução da instrução de *ALOAD* seria sete vezes mais rápida. Para *bytecodes* mais complicados, tais como o código de comparação *op*, o aumento pode ser ainda maior.

A idéia básica é então tomar o método, como definido pelos *bytecodes* Java, e substituí-lo por um código objeto compilado que executa a mesma função. Na sua maior simplicidade, o compilador JIT simplesmente substitui o *bytecode* por um código equivalente que executa a mesma função. Compiladores JIT mais complexos podem fazer

ambas as coisas – otimizar os *bytecodes* e então compila-los em um código objeto. Uma vez completo, o método aparece para o sistema como um método nativo ao invés de um método interpretado.

Existem atualmente vários compiladores JIT desenvolvidos, entre estes estão o Netscape JIT (produzido pela Borland), o compilador Symantec Cafe JIT, o Microsoft Internet Explorer JIT, e vários compiladores JIT desenvolvidos para ambientes do Macintosh.

É impossível relacionar exatamente o que estes compiladores JIT fazem sem o código fonte e isto causou alguma preocupação entre os especialistas em segurança. Por necessidade, estes compiladores atuam no código Java depois que ele passou pela verificação, porém qualquer código entre o *bytecode* Java e a execução real daquele código tem potencial para introduzir falhas na segurança. O risco é pequeno, porém o perigo é que um invasor maluco poderia descobrir uma maneira de corromper o compilador JIT usando legítimo *bytecode* Java. Como o compilador JIT está produzindo código que é executado fora dos limites de proteção do sistema interpretado, este código não é englobado pelos mecanismos de segurança. Uma vulnerabilidade no compilador JIT conduziria assim a um potencial comprometimento do tempo de execução completo do Java.

Uma alternativa, então é evitar completamente o segundo estágio da compilação e construir chips que executem diretamente os *bytecodes* do Java. A Sun Microsystems fez alguns progressos nesta área com o lançamento dos seus chips '*picoJava*'. Estes chips serão a base para os aplicativos baseados no Java e podem executar *bytecodes* Java de forma significativamente mais rápida que um interpretador. Futuramente, os chips Java certificados evitariam a preocupação com a segurança, já que a implementação dos chips pode ser verificada com os vetores de teste de chips.

Eventualmente, a primeira geração de compiladores JIT como aqueles disponíveis atualmente darão origem a uma segunda geração de compiladores. Esta primeira geração de compiladores JIT concentram-se em substituir o corpo de computação por corpos de métodos com equivalentes C e fazendo otimizações tais como as descritas acima. A segunda geração de compiladores se concentrará em algumas das partes mais complexas do sistema, tais como as chamadas de métodos, carregamento de classes, e as classes bases.

Os arquivos de classe suportam *tags*, conhecidas como atributos, que permitem que múltiplos corpos de método sejam armazenados em uma única classe Java. Estes corpos de método alternativos poderiam conter código de máquina altamente aperfeiçoado para plataformas de computação populares. Os *bytecodes* ainda estariam presentes, o que seria útil para a portabilidade, mas o código de máquina estaria disponível para execução rápida. Outras áreas de refinamento envolveriam a remoção ou substituição da máquina virtual corrente quando esta estivesse em execução de código nativo, aumentando o desempenho do código Java mais adiante.

A chave para o futuro é se lembrar que o código básico é portátil e facilmente mutável. Os arquivos de classe Java fornecem um código intermediário que pode ser diretamente interpretado, ou posteriormente compilado em um código de máquina, como é o caso em outras linguagens como C ou C++. Com os sistemas de segunda geração, seria possível compilar todas as classes exigidas por um *applet* ou aplicativo em um único objeto compartilhado ou executável. Neste ponto não há diferença entre a execução de um programa ou controle escrito em Java e um aplicativo ou controle escrito em C.

O *bytecodes* Java (*j-code*) são a representação intermediária de programas compilados. Apesar de serem normalmente escritos em Java, eles também poderiam ser escritos em qualquer outra linguagem cujo compilador compile para a forma Java intermediária. A primeira geração de compiladores "*just in time*" ou compiladores JIT operam substituindo o *J-code* pelo código de máquina que executa uma função equivalente. Esta substituição pode gerar um aumento de velocidade de 7 a 10 vezes para a maior parte dos códigos. Apesar de ainda não serem tão rápidas quanto os aplicativos desenvolvidos completamente em códigos nativos, outros fatores tais como os chips de microprocessadores Java e compiladores que produzem diretamente programas executáveis estão rapidamente preenchendo esta lacuna. Dentro dos próximos 12 a 18 meses deverá ser possível escrever um aplicativo em Java com um desempenho tão bom ou melhor do que um aplicativo similar escrito em C++. Quando isso acontecer, o Java será alçado ao seu lugar entre as outras linguagens de programação.

D.2 Applets e Aplicativos

A Plataforma Java permite ao desenvolvedor criar dois tipos diferentes de programas: os Applets e os aplicativos.

Os *Applets* são programas que exigem um browser para serem executados. A etiqueta `<applet>` é introduzida em uma página de Rede com o nome do programa a ser executado. Quando aquela página é acessada por um usuário da Internet ou de uma intranet, o *applet* é carregado automaticamente do servidor e executado na máquina de cliente. Os *applets*, tendem a ser pequenos e modulares, para evitar grandes demoras no carregamento.

Portanto, em vez da atividade acontecer no lado do servidor, como no caso da programação CGI (*Common Graphic Interface*), ela acontece no lado do cliente em um *browser* compatível com Java. Isso quer dizer que os *applets* não são restritos pela largura de banda ou velocidade do *modem* quando estão sendo executados. Os usuários podem ver e ouvir efeitos multimídia em páginas *Web* de forma eficaz.

Os aplicativos são programas que não requerem um browser para serem executados, e não têm nenhum mecanismo de carregamento embutido. Quando um aplicativo é chamado, ele é executado. Deste modo, os aplicativos são como programas em outras linguagens. Eles podem executar tarefas tradicionais, como as de um processador de texto, planilha eletrônica ou aplicação gráfica. Como os *applets*, um aplicativo requer uma Plataforma Java para ser executado. Porém a Plataforma pode estar disponível como um programa separado, pode ser embutida diretamente dentro do sistema operacional, ou pode ser embutida na própria aplicação.

Apesar dos *applets* e dos aplicativos serem chamados de formas diferentes, a maior parte deles fazem o mesmo uso das facilidades da linguagem Java. Por exemplo, um applet ou um aplicativo podem ter acesso a um banco de dados do hospedeiro, acessar os dados que precisa, usar os dados em processamento local, e armazenar os resultados no hospedeiro.

Um *applet* exige estar em rede para ser executado, enquanto um aplicativo não. Os aplicativos têm maior liberdade por ter acesso completo aos serviços do sistema. Por exemplo, um aplicativo, diferentemente de um *applet*, pode ter acesso normal de leitura e

escrita de arquivos em qualquer disco. Considerando que um *applet* pode ser carregado de uma página de Rede potencialmente não confiável, o acesso de leitura e escrita é restrito a qualquer sistema de arquivo, a menos do servidor do qual veio. Esta restrição poderá ser relaxada quando os *applets* forem marcados com assinaturas digitais, e permitirem ao usuário final ter assegurado que este *applet* foi carregado de forma inalterada, de uma fonte confiável. Onde é necessário o armazenamento de arquivos locais, normalmente um aplicativo é requerido.

D.3 Java Script

Originalmente chamada de *Livescript*, a *JavaScript* é uma linguagem *script* interpretada baseada em objetos e desenvolvida pela *Netscape*. Com a *JavaScript* é possível introduzir recursos de interatividade em uma página Web, fazer animações, imagens independentes de CGI¹ programas que usam *applets*. No entanto, as limitações do *JavaScript* são grandes.

A linguagem *JavaScript* não é compilada em *bytecodes*, mas interpretada bloco-a-bloco pelo navegador de uma página HTML. Cada *Script Java Script* consiste de um único módulo contínuo de código onde ficam todas as funções e variáveis utilizadas pelo programa. O código é posicionado no arquivo HTML entre descritores `<script>` e `</script>` em um arquivo HTM. Como um programa *JavaScript* só existe dentro de um arquivo HTML, não existem aplicativos *JavaScript* independentes de browser (navegador), como em Java.

Apesar das limitações, a *JavaScript* vem evoluindo e pode ser usada em um grande número de aplicativos sérios, principalmente aqueles destinados à Intranet e Internet e usados através de um *browser*.

¹ *Common Graphic Interface*

D.4 Bytecode

Uma das principais diferenças entre as linguagens Java e C/C++ é o processo de compilação e execução (ver 2.5.1). A compilação de um programa em C++ traduz o código fonte em instruções que são executadas pelo processador (*hardware*) do computador em que será executado. Já o processo de compilação de um programa em Java traduz o código fonte em instruções (chamadas de *bytecodes*), que são interpretadas por um sistema de execução que simula o processador de um computador virtual genérico (Máquina Virtual Java).

Portanto o compilador Java não gera código de máquina como as instruções nativas de hardware, mas sim gera *bytecodes*: um código de alto nível, independente de máquina para uma máquina hipotética que é processada pelo interpretador Java e sistema de *run-time*.

Um dos primeiros exemplos do uso de bytecode era o *UCSD P-System* que foi aplicado a uma variedade de arquiteturas de oito bits. Este *bytecode P-System* obteve grande popularidade no meio dos anos setenta e início dos anos oitenta, sendo muito difundido durante o auge das máquinas de oito bits. Atualmente, as arquiteturas podem utilizar o bytecode para a abordagem de software distribuído. São projetados *bytecodes* Java para serem fáceis de ser interpretados em quaisquer máquinas, ou para serem traduzidos dinamicamente, se preciso for, em código de máquina nativo.

A característica de ser independente da arquitetura (arquitetura neutra) não é apenas útil para aplicativos baseados em redes de computadores, mas também para distribuição de software em um sistema único. No mercado de software de hoje, os desenvolvedores de aplicativos têm que produzir versões dos aplicativos compatíveis com o IBM PC, Macintosh Apple, e cinquenta e sete tipos de *workstations* e arquiteturas disponíveis no fragmentado mercado do sistema operacional UNIX®.

No mercado de PC, com o Windows 95 e o Windows NT sendo executados sobre muitas arquiteturas de CPU, e a Apple aumentando o poder da família 68000 para o PowerPC, a produção de softwares para serem executados em todas as plataformas era

quase impossível até a presente data. Usando Java, junto com a ferramenta *Windows Toolkit*, a mesma versão de um aplicativo pode ser executada em todas as plataformas.

A máquina virtual é uma camada de software que interpreta os *bytecodes* e pode estar implantado sobre qualquer plataforma real. Por ocorrer a nível de *software*, a interpretação de *bytecodes* tende a ser bem mais lenta que a execução direta de linguagem de máquina nativa pelo processador. Apesar disso, devido à semelhança dos *bytecodes* com instruções de máquina, o desempenho é melhor em relação a outras linguagens interpretadas.

O código-fonte em Java também pode ser compilado diretamente para a linguagem de máquina nativa, embora assim, perca a sua característica principal, que é a independência de plataforma. Mesmo assim, não deixa de ser uma boa alternativa para garantir um maior desempenho em trechos de código que exigem muito do sistema.

Uma diferença importante é a ausência de um destrutor na classe Java. Não é necessário destruir objetos criados em Java, pois objetos não mais utilizados são automaticamente eliminados por um mecanismo de “coleta de lixo” e a memória ocupada por eles é automaticamente liberada.

Cada plataforma tem sua própria implementação da Máquina Virtual Java, porém apenas uma especificação de máquina virtual. Por isto, a Plataforma Java fornece um padrão de interface de programação uniforme para todos os *applets* e aplicativos para qualquer *hardware*. A Plataforma Java é então ideal para a Internet, onde um programa deveria ser capaz de funcionar em qualquer computador. Os compiladores *just-in-time* estão disponíveis para acelerar a execução convertendo *bytecodes* Java em linguagem de máquina. De dentro da linguagem Java, os desenvolvedores podem também escrever e chamar métodos nativos em C, C++ ou outra linguagem.

D.5 Threads

Os chamados *threads* são fluxos de execução que são executados dentro de um processo (aplicativo). Normalmente os *threads* compartilham regiões de memória, mas isto não ocorre necessariamente. Isso faz lembrar os de encapsulamento. Processos por assim

dizer, ‘avós’ dos *threads* permitem que o seu sistema operacional execute mais de um aplicativo ao mesmo tempo enquanto que os *threads* permitem que seu aplicativo execute mais de um método ao mesmo tempo.

Todos os programas feitos até agora tinham um único caminho. Nenhum deles executava duas coisas (duas partes de código) simultaneamente. Grande parte do software de qualidade escrito hoje faz uso de mais de uma linha de execução, ou seja, mais de um *thread*. Estes são os chamados programas *multithreaded*.

Um browser de hipertexto consegue fazer o *download* de vários arquivos ao mesmo tempo, gerenciando as diferentes velocidades de cada servidor e, ainda assim, permite que se continue interagindo, mudando de página no hipertexto enquanto o arquivo nem foi carregado totalmente. Isto não seria possível sem o uso de *threads*.

Um editor de textos permite que se edite o começo do arquivo, enquanto ele está sendo carregado do disco. A edição e o carregamento do disco são atividades que não podem ser intercaladas de maneira simples em uma parte de código. O editor está usando *threads*, e essas atividades estão sendo feitas em paralelo.

Se uma máquina só possui um processador, esse paralelismo é um falso paralelismo. O processador tem seu tempo dividido em pequenos intervalos, em cada intervalo ele executa uma das atividades e tem-se a sensação de que tudo está funcionando ao mesmo tempo, de forma simultânea. Em uma máquina que possui mais de um processador, então o ambiente será capaz de mapear *threads* em hardware e ter-se-á, realmente, um processamento paralelo.

Se alguém procurar a tradução de *threads* no dicionário é capaz de encontrar um desenho de um carretel de linha ou da rosca de um parafuso. É possível imaginar um programa que seja composto por várias linhas de execução que funcionam em paralelo, algumas vezes estas linhas podem se juntar, em outras se separar. Cada linha de execução cuida de uma tarefa: transferir um arquivo, tratar a entrada do usuário, mostrar sua janela na tela, etc.

Os *threads* são uma invenção recente, se comparada com o restante das linguagens. Algumas outras poucas linguagens fornecem facilidades para lidar com *threads*, por

exemplo a Modula-3. Os *threads* são um recurso extremamente difícil de se implementar, de modo que é possível dizer que, ou um ambiente de programação oferece facilidades para lidar com eles, ou o usuário não vai querer implementá-los, ou usá-los.

Uma forma onde o paralelismo é explorado em Java é através dos *threads*. O termo *multithreaded* significa um aplicativo com várias linhas de execução rodando simultaneamente em paralelo. Para entender esse conceito, vamos imaginar um Web browser, no qual várias coisas estão acontecendo ao mesmo tempo, como por exemplo um trecho de áudio está sendo tocado, ao mesmo tempo em que uma imagem está sendo recebida da rede, e que uma animação está sendo exibida, além disso a página está sendo rolada pelo usuário. A linguagem Java suporta o *multithread*, e isso significa que é possível criar aplicativos *multithreaded*. Na verdade, todos os programas Java são *multithreaded*. Por exemplo, o coletor de lixo é uma linha de execução que está sempre rodando na retaguarda.

As linhas de execução ou *threads* são às vezes chamadas de processos *lightweight*. Um importante benefício das múltiplas linhas de execução é que elas melhoram a performance interativa percebida pelo usuário em aplicativos gráficos. Em Java, o uso de *threads* é muito mais fácil, já que a linguagem tem originalmente todo o suporte necessário para a criação e o controle das *threads*.

A classe *Thread*, do package *Java.lang*, contém métodos para iniciar/parar *threads* e definir suas prioridades, entre outras coisas. Além disso, a própria sintaxe da linguagem oferece suporte direto para as *threads*, através da palavra-chave *synchronized*. Com essa palavra-chave é fácil marcar seções de código ou métodos inteiros que somente devem ser executados por uma única *thread* de cada vez. Por exemplo, um *applet* que execute uma animação, faz isso usando uma *thread*.

D.6 Ferramentas e componentes Java

Nesta seção serão revisados os ambientes para a ferramenta Java encontrados na literatura. Todos os componentes que puderam ser testados foram conseguidos em versão

livre, sendo eles *demos*, *freewares*, *sharewares*, ou *trial versions*². Muitas ferramentas apresentaram qualidade bastante aceitável, porém outras, até por que são softwares não comerciais e se tratam de versões em desenvolvimento, mostraram-se não muito satisfatórias. Em determinados momentos do texto que segue foram feitas algumas análises críticas dessas ferramentas.

Entre as numerosas ferramentas encontradas, alguns tipos puderam ser destacados. Existem ferramentas (Brewmaster, JavaDraw, ED, Dick Chase, etc.) que substituem ou apoiam algumas já existentes nos ambientes profissionais (SDK ou JDK), existem conversores da linguagem Java para a linguagem C (Toba, JACE), existem os descompiladores de classes binárias Java para código Java (Mocha, WingDis, DJava, Jasmin, JCD, JavaDis, Kimera) existem os compiladores de compiladores para definição da linguagem Java (Jlex), existem os visualizadores de classes para Java. Foram considerados aspectos do ponto de vista de desenvolvedores de softwares e não de usuários finais. Sempre que possível, foram colocadas as versões mais recentes (última pesquisa feita no mês de abril de 1999), apesar do fato de que muitas das análises foram feitas com versões anteriores. Mas este fato não inviabiliza a proposta inicial deste item que é a de fornecer um panorama das possibilidades encontradas na literatura.

D.6.1 Geradores de compiladores para Java

Um analisador léxico quebra um fluxo de caráter em símbolos (*tokens*). Escrever analisadores léxicos pode ser um processo tedioso, assim ferramentas de *software* foram desenvolvidas para facilitar esta tarefa. Talvez a mais conhecida destas ferramentas seja o Lex. O Lex é um gerador de analisador léxico para o sistema operacional UNIX, voltado à linguagem de programação C. O Lex transforma um arquivo de especificação especialmente formatado que contém os detalhes de um analisador léxico. Esta ferramenta cria um arquivo fonte C que é associado a um analisador construído por tabelas.

² Os nomes foram mantidos em original na língua inglesa por serem termos consagrados e a tradução em português não representar totalmente seu significado. As traduções poderiam ser: *demo*: uma simplificação da palavra demonstrativo; *shareware* e *freeware*: softwares livre para a comercialização; *trial version*: versão de teste.

O gerador JLex

O utilitário JLex é baseado no modelo de gerador de analisador léxico Lex. O JLex transforma um arquivo semelhante a especificação para Lex, e então cria um arquivo fonte Java para o analisador léxico correspondente.

Um arquivo de entrada JLex é organizado em três seções, separados através de diretivas duplas de por cento (```%% "`). Uma especificação própria de JLex tem o seguinte formato:

código de usuário

`%%`

Diretivas de **JLex**

`%%`

regras de expressão regulares

As diretivas ```%% "` distinguem seções do arquivo de entrada e devem ser colocadas no começo da linha desses arquivos. O resto da linha que contém o diretivas ```%% "` pode ser descartada e não deveria ser usada para agregar declarações adicionais ou código.

A seção de código de usuário, a primeira seção do arquivo de especificação, é copiada diretamente no arquivo resultante. Esta parte da especificação fornece espaço para a implementação de classes usadas ou tipos retornados.

A seção de diretivas JLex é a segunda parte do arquivo de entrada. Aqui, as definições de macros são declaradas e declarados nomes dos estados. A terceira seção contém as regras de análise léxica cada uma das quais consiste em três partes: uma lista opcional de estados, uma expressão regular, e uma ação.

O código de usuário precede o primeiro duplo por cento diretivo (```%% "`). Este código é copiado literalmente no arquivo fonte do analisador léxico que é produzido pelo JLex, no início do arquivo. Então, se o arquivo fonte do *lexer*³ precisa começar com uma declaração de pacote ou com a importação de uma classe externa, a seção de código de

³ Programa que dará origem ao analisador léxico

usuário deveria começar com essa declaração correspondente. Esta declaração será então copiada no início do arquivo fonte gerado.

A terceira parte da especificação de JLex consiste em uma série de regras para separar o fluxo de entrada em símbolos. Estas regras especificam expressões regulares, que estão associadas com ações que consistem em código fonte Java. As regras têm três partes distintas: a lista de estados opcional, as expressões regulares, e a ação associada. Este formato é representado como segue.

```
[<estados>] <expressão> {<ação>}
```

Se mais de uma regra emparelha uma *string* na entrada, o *lexer* gerado soluciona conflitos entre regras por uma escolha que busca economizar a regra que possui a *string* mais longa. Se mais de uma *string* da regra tem o mesmo comprimento, o *lexer* escolherá a regra que for primeiro escrita na especificação do JLex. Então, as regras que aparecem antes dentro da especificação obtêm uma prioridade mais alta, determinada no *lexer* gerado.

As regras dadas a uma especificação JLex deveria tentar utilizar todas as regras descritas. Porém, se o analisador léxico gerado receber uma entrada que não se encaixa em nenhuma de suas regras, um erro irá ocorrer.

O programa JLex transformará uma especificação corretamente descrita em um arquivo fonte Java para o analisador léxico correspondente. O analisador léxico gerado está na classe *Yylex*. Há dois construtores para esta classe, ambos requerem um único argumento: a cadeia de entrada para ser quebrada em *tokens*. A cadeia de entrada ou pode ser do tipo *Java.io.InputStream* ou do tipo *Java.io.Reader* (como *StringReader*). A função para acessar o *lexer* é *Yylex.yylex()*, que retorna o próximo *token* para a cadeia de acesso. O tipo retornado é *Ytoken* e a função é declarada como segue.

```
class Yylex { ...
    public Ytoken yylex () { ... }
```

O usuário tem que declarar o tipo de *Ytoken* e faz isto na primeira seção da especificação de JLex, a seção de código de usuário.

Um teste de avaliação foi feito, comparando o desempenho de um analisador léxico gerado por JLex (escrito em Java) e um analisador léxico escrito à mão. A comparação foi feita com analisadores léxicos de uma linguagem simples de programação.

O teste consistiu na execução, em cada analisador léxico, de dois arquivos fonte escritos em linguagem simples, medindo o tempo exigido para processar estes dois arquivos. Cada analisador léxico também foi chamado por um programa teste sem objetivo, escrito em Java. O analisador léxico gerado provou ser bastante rápido, como mostram os resultados mostrados na tabela a seguir:

Tabela D.1 – Comparação entre os resultados ...

Tamanho do Arquivo Fonte	Tempo de execução do programa gerado pelo JLex	Tempo de execução do analisador léxico escrito à mão
177 linhas	0.42 segundos	0.53 segundos
897 linhas	0.98 segundos	1.28 segundos

Uma das maiores reclamações relativas aos analisadores léxicos gerados por programas como JLex, é que eles não têm desempenho tão bom como os escritos à mão. Então, esta experiência é particularmente importante por demonstrar a velocidade relativa dos analisadores léxicos gerados pelo JLex.

A seguir pode-se ver uma lista, possivelmente incompleta, de características não implementadas do JLex.

- 1.O suporte a Unicode⁴ não foi implementado,
- 2.O operador de *lookahead* de expressão regular não foi implementado, e não foi incluída na lista de meta-caracteres de expressões regulares especiais.
- 3.O operador de início-de-linha (^)assume o comportamento sem seguir um padrão.

⁴ Em contraste com o tipo caracter de 8-bits mantido pela especificação Ansi para C, Java suporta um tipo de dados char de16-bits denominado conjunto de caracteres Unicode caráter jogo. Existe em Java uma classe *string* para manipulação destes caracteres Unicode. Junto com o tipo de char 16-bit, a Java possui um tipo primitivo 8-bit (byte), para poder manipular as informações do tipo caracter de 8-bit. Como quase todos sistemas operacionais suportam o tipo caracter de 8-bits ASCII, ao invés do caracter 16-bits Unicode, JLex assume que o arquivo texto - em particular a especificação JLex , e ainda o arquivo fonte do analisador léxico produzidos por JLex - são salvos em formato de ASCII.

D.6.2 Ambientes de desenvolvimento

A seguir serão analisadas algumas ferramentas de desenvolvimento encontradas *na world wide web*. As ferramentas foram analisadas segundo a facilidade de uso e confiabilidade, do ponto de vista do desenvolvedor. Algumas análises foram feitas com base em artigos escritos por diversos autores, os quais muitas vezes corroboravam os resultados da análise feita durante o estudo destas ferramentas neste trabalho.

a) Brewmaster (versão atual 2.0)

A interface GUI tem ícones minúsculos, e não têm menu com identificação *pop-up* ou outra auto-documentação, o que resulta em um quebra-cabeça para seu aprendizado. A combinação de browser/editor de classe é sólida e com boa apresentação e, uma vez que se acostume ao paradigma de navegação, é bem utilizável. São analisadas gramaticalmente todas as mudanças que são feitas e salvas na base, assim é impossível armazenar um programa sintaticamente incorreto no repositório. O problema maior dessa ferramenta é que o suporte para a compilação, depuração e execução de programas correntes são tediosos e muito limitados.

b) ED for Windows versão 3.72

O ED é uma extensão do editor/IDE já existente para Java. Também tem suporte para muitos outros tipos especializados de documentos. Se o interesse for trabalhar com muitos tipos de documentos e linguagens de programação de múltiplo, o ED é bastante indicado. Funcionam bem nesta ferramenta o editor e o IDE, com bastante possibilidade de ser caracterizado e que pode ser facilmente customizado. O principal problema de ED para um ambiente de Java é que falta especialização para Java.

c) *JavaDraw*: (versão atual 3.11)

O *JavaDraw* é um IDE convencional com alguns suplementos úteis. O suplemento mais útil observado é um pré-processador igual ao da linguagem C integrado. Porém é um software limitado. E todas as informações levam a crer que o *lavadraw* só trabalha com JDK 1.1. Muitas vezes o estado interno da base exige que o software seja reiniciado. Não é, portanto, um software confiável, além de não haver nenhuma avaliação segura sobre a natureza dos problemas encontrados nas ferramentas. Por exemplo, se a execução de um

programa aparentemente não fazer nada, nenhuma informação é retornada, e o programador fica sem saber que nada aconteceu. O editor é um pouco inamistoso, às vezes não executa as opções de *cut*, *paste*, ou a posição do cursor não é confiável. A fonte é visualmente ruim, com um espaçamento muito grande entre as letras e o destaque (*highlight*) é fraco.

d) Java Workshop (versão atual 2.0)

Esta versão ainda não está completa, mas o Java Workshop 2.0 é uma grande melhoria em relação a versão 1.0. Foram considerados aspectos de programação de IDE de forma bem cuidada e há um excelente tutorial sobre como usar os recursos da ferramenta. Ainda há problemas de depuração da interface GUI.

e) Dick Chase's Java Editor (versão atual: 2.08a)

O Editor Dick Chase's Java é muito rudimentar, sendo um editor Java sem nenhuma sofisticação, baseado em documentos de múltiplos *frames* e JDK. Sua virtude é ser completamente grátis, e o autor oferece até mesmo o código fonte. É muito pequeno e leve em termos de consumo de recursos, o que pode ser útil para configurações limitadas. Seu ponto alto são os ciclos de *compile/edit* básicos, e seu ponto mais fraco é não ter nenhuma interface de depuração para a verificação do que possa estar errado.

f) Javelin (versão atual 5.03.6)

O Javelin é um ambiente gráfico, que gera arquivos texto somente como um artifício para o compilador. O conceito no qual ele se baseia é bom e o GUI tem boa apresentação, mas a sua interface é óbvia e simples demais; em outras palavras, ainda há muitas falhas por todo o programa. O processo *import/export/compile* apresenta diferentes problemas; e o banco de dados interno pode ser corrompido. Os piores problemas foram corrigidos em versões posteriores à 5.02, que foi analisada, mas não se pode confiar que cada projeto novo apresente-se sem problemas, pelo menos durante algum tempo. Em geral, o Javelin mostra os sinais clássicos de um produto imaturo, insuficientemente testado, e comercializado apressadamente para aproveitar o mercado disponível. Entretanto muitas ferramentas funcionam e sua interface gráfica funciona bem. Além disso suas classes são apresentadas como caixas conectadas por linhas que representam herança. Os

conteúdos das caixas são dados e métodos da classe; podem ser definidos e os elementos podem ser editados em um estilo mais livre e agradável pelas declarações.

A Integração no ambiente Windows NT é fraca. O Javelin não usa nomes de arquivo longos e não cria diretórios de usuário. As mensagens de erro dos "*import files*" são mostradas por caixas de diálogo, e não são registradas permanentemente em nenhum lugar. Não há nenhuma correspondência significativa entre as mensagens de erro do compilador e a linha indicada. Não é possível clicar sobre uma mensagem de erro para saber algo a seu respeito. Não há nenhuma ferramenta de navegação de classe, diferente da interface gráfica. Não há nenhuma maneira de achar uma classe ou um método através de seus nomes. Não há nenhum apoio para JDB ou qualquer outra ferramenta de depuração. Finalmente, deve-se observar que a versão de avaliação grátis é pouco útil para uma avaliação significativa.

g) JPad e JPad Profissional (versão atual 3.6)

A JPad Pro é uma versão maior, melhor, e ligeiramente mais cara do Jpad. A versão demonstrativa disponível para *download* só permite a edição de arquivos minúsculos, o bastante para uma sondagem inicial, mas não o suficiente para qualquer trabalho sério. As melhorias principais do JPad Pro em relação ao Jpad são: os projetos, um Class Browser, as capacidades adicionais de edição de HTML e os arquivos JavaScript.

A capacidade de projeto de Jpad é mencionada porque é incomum. O JPad Pro coleta de forma automática e dinâmica todas as fontes para Java e as arquiva em uma hierarquia de um projeto. Isto pode ser conveniente, se a hierarquia não contém nenhum arquivo estranho, caso contrário pode-se ter problemas. A ferramenta JPad suporta JDK e Microsoft SDK, e está baseada em uma linguagem de *scripting* interna; todo o IDE pode ser customizado, e com algum esforço, é possível diagnosticar-se e consertar quaisquer problemas em sua interação com seu ambiente, permitindo adicionar seus próprios comandos. A ferramenta JPad Pro é sólida e segura, e com boa apresentação. Com exceção de alguns problemas secundários, todas as opções operam de forma segura, dispondo de ajuda on-line. Alguns problemas encontrados foram relativos à interação com a JDB pois essa apresenta-se de forma rudimentar, e a interação com o depurador da interface GUI Microsoft, que por vezes não funciona.

h) Kawa (versão atual 3.0)

O Kawa é baseado em uma editor de *frames* de múltiplos documentos. Sua implementação é bastante simples, mas sólida. O editor é simples, mas funcional. Inclui coloração de sintaxe, há uma boa capacidade para projetos, e arquivos no browser de projeto se expandem em um browser de classe para o arquivo. Suporta completamente o JDK 1.1 e fornece algum apoio para o *JavaBeans*. A ajuda on-line junta a documentação para HTML do JDK 1.1 com um apoio para o *Javadoc* do usuário. Possui um *Class browser* e o *debugger* é nativo, isto é, um depurador baseado diretamente no *debugger* Java para APIs em lugar do JDB. O depurador GUI é fraco em características, e às vezes não pode inspecionar objetos. O editor se atrapalha com linhas excessivamente longas.

i) Microsoft J++ Visual (1.1 *preview*, versão atual 6.0)

É importante observar que a versão que pode ser analisada é a *preview* 1.1. A ferramenta visual J++ é um ambiente Java integrado com o mesmo conceito de Microsoft Visual C++. A versão 1.1 é uma revisão quase completamente sem erros, com uma interface bastante boa. Por isso o ambiente de desenvolvimento é bastante amadurecido, bem caracterizado e familiar. Infelizmente, os elementos Java específicos do ambiente ainda são fracos. Deve-se considerar o fato de que, se existe uma companhia no mundo com os recursos para fazer esse um trabalho de ambiente, esta empresa é a Microsoft. Os melhores pontos observados foram: o ambiente de escrita de código básico é sólido, o compilador é muito rápido, a integração do *Compiler/Debugger* muito boa e rápida, sendo semelhante ao Visual C++. Mas o Compilador produz um código fraco em algumas circunstâncias. O compilador apresentou alguns problemas de linguagem onde o JDK opera de forma satisfatória. Extensões de J++ e melhorias para a linguagem Java ameaçam corromper a máquina Java.

j) MJE 0.98

É um mini editor de Java. O MJE é uma aplicação JAVA minúscula. São incluídas fontes. O autor está principalmente interessado na compatibilidade de plataforma para Java. Seus pontos fortes são a capacidade de construir projetos simples, o editor é rudimentar mas funcional, e a Interação com o compilador é boa. Seu ponto fraco é que o editor MJE

só apoia a fase *edit/compile*. Não há nenhum comando de execução. Não existe a possibilidade de salvar arquivos antes de sair

k) Mojo (versão 2.0)

A Mojo é um repositório baseado em IDE, com ênfase em ferramentas automatizadas para construir interfaces GUI. No anúncio feito no site, os desenvolvedores dizem ser possível importar arquivos Java, mas o que pôde ser verificado não correspondeu a essa afirmação de forma significativa. E segundo recomendações, a criação de uma nova classe deve ser feita em um dispositivo de definição (*Coder*), então deve-se usar um editor externo para escrever o texto e importá-lo com comandos de marcação e cópia, mas o seu resultado não é confiável. Pelo que pode ser observado, a Mojo não é a ferramenta adequada para um programador de propósito geral, mas a interface GUI do demonstrativo fornecido tem boa apresentação. Observa-se que a Mojo não tem nenhuma forma eficiente para importar código já existente, e este é o fator mais crítico.

l) OEW (versão atual 2.03)

O OEW é um editor de classe baseado em um repositório. Ao contrário de outros repositórios ou editores vistos até aqui, este realmente funciona, mas alguns problemas reduzem bastante a sua viabilidade. Na importação de arquivos fonte, a ferramenta OEW converte todos os nomes de arquivo para letra minúscula. Por exemplo, uma classe denominada *BaseObject.Java* se transforma em "*baseobject.java*". As consequências deste problema podem ser facilmente observadas. Mas a importação e exportação de arquivos opera de forma satisfatória, e é possível desenhar diagramas de conectividade entre as classes. Entre os problemas, além da dificuldade em se controlar os nomes das classes em arquivos importados, não há uma forma de se interromper a construção das classes em desenvolvimento. Outro problema detectado é que depois de certo tempo de uso, alguns menus deixam de funcionar. Por exemplo, o artigo de *Make/Options* trabalha em um ambiente perfeitamente limpo, mas não faz nada em algum ponto posterior. Em algumas vezes pode ocorrer, na geração de código, uma alteração na declaração do pacote (*package*), confundido com algum outro pacote. Por vezes pode haver problemas quando busca-se acrescentar ou remover uma classe nova. O OEW tem alguns problemas de

corrupção de memória interna que torna o software não confiável. As ferramentas de *cut* e *paste* no editor são incertas.

m) Supercede (versão atual 2.0).

A diferença entre esta ferramenta e as demais é que esta deixa o usuário modificar programas correntes, e incorpora as mudanças no programa corrente sem a necessidade de se reiniciar o mesmo. Para programas pequenos, esta é uma característica secundária; mas para programas maiores e mais complexos é uma capacidade valiosa. Além disso, a ferramenta é bastante familiar e permite editar fontes, execuções e um conjunto *breakpoints* através de uma interface gráfica. A Supercede trata arquivos fonte de uma maneira bastante convencional, e assim combina facilmente com outras ferramentas de Java. O ciclo básico de *compile/run/debug* opera de forma satisfatória. O *debugger* é completamente gráfico. Entre as características menos atraentes está o uso intensivo da memória. Existem também algumas falhas na depuração do código interno. Apesar do ambiente de depuração ter boas características, a apresentação e interface com o usuário é rudimentar.

n) IBM VisualAge for Java (versão 1.0 beta 1)

O VisualAge é um repositório baseado em ambiente Java, com um projeto realmente de impacto. Não existe nenhum botão para se executar a compilação, pois o VisualAge está compilando o código de maneira permanente, e mantém uma lista de problemas não resolvidos. Podem ser executadas quaisquer classes, ou executadas partes de código em uma janela. Não há nenhuma distinção entre executar, depurar, e editar código pois tudo é parte do mesmo processo integrado. A versão beta teste ainda apresentou alguns pequenos problemas. A navegação em um programa complexo não é fácil, a menos que se saiba exatamente onde se quer entrar na hierarquia *project/class method*. Não há nenhuma ferramenta para procurar um projeto, e nenhuma ferramenta de apoio para auxiliar essa busca. Os arquivos de texto exportados são muito diferentes dos originais para se efetuar uma comparação.

o) WingEditor WingEditor (versão atual 1.5.2)

A ferramenta WingEditor, conhecida como *JavaEditor*, *WingEditor*, é um IDE em desenvolvimento escrito completamente em Java. Com isso essa ferramenta se apoia em JDK, como a maioria dos IDEs parecem fazer. Várias características prometidas não são cumpridas na prática. As operações de *editor/compiler/debugger* funcionam conectadas, mas sua capacidade de projeto é rudimentar, e as interfaces de GUI conectam a JDB, necessitando do JDK 1.1 para ser executada. Outro problema é que a interface gráfica para JDB é confusa e pode causar erros.

p) ClassViewer (sem versões)

O Classviewer permite ver os conteúdos de um arquivo classe. Apresenta muitas informações para determinar as interfaces contidas na classe. O Classviewer é realmente básico, visualizando uma classe de cada vez, mas é uma ferramenta essencial para lidar com classes na ausência de código de fonte. Um problema que ocorre é que ele tem dificuldade em encontrar todos os componentes de classe, a menos que estas classes estejam no *classpath default*.

D.6.3 Montadores para Java

O Jasmin é um montador para a Máquina Virtual Java. Ele traduz descrições de Java escritos em ASCII, escritas em um montador similar de sintaxe simples, usando o conjunto de instruções da Máquina Virtual Java. O Jasmin converte a entrada em uma classe de arquivos Java binária, satisfatória para ser carregada em um sistema de execução Java. O montador Jasmin foi criado como complemento ao livro "Java Virtual Machine", escrito por Jon Meyer e Troy Downing e publicado pela O'Reilly Associates.

D.6.4 Descompiladores Java

Os descompiladores (*desassemblers*) foram escritos para uma grande variedade de aplicativos como o desenvolvimento dos primeiros compiladores. O primeiro descompilador foi escrito por Joel Donnelly em 1960, no *Naval Electronic Labs*, para

descompilar um código de máquina da linguagem Neliac em uma máquina *Remington Rand Univac M-460 Countess*. O projeto foi supervisionado pelo Professor Maurice Halstead, que trabalhou em descompiladores durante os anos sessenta, e publicou técnicas que formaram a base para os descompiladores de hoje.

Ao longo das últimas décadas, foram dados usos diferentes aos descompiladores. Nos anos sessenta, foram usados descompiladores para ajudar no processo de conversão de programas de computadores de segunda geração para computadores de terceira geração; deste modo, não seria usada força de trabalho na tarefa demorada de rescrever programas para as máquinas de terceira geração.

Durante os anos setenta, os descompiladores eram usados para portar programas, documentação, depuração, recriação de códigos fonte perdidos, e modificação de binários existentes. Nos anos noventa, os descompiladores transformaram-se em uma ferramenta de engenharia reversa, capaz de ajudar o usuário em tarefas como conferir softwares de códigos não confiáveis, conferir se um compilador gera o código correto, traduzir programas binários de uma máquina para outra, e entender a implementação de uma função de biblioteca particular.

Sobre esse assunto existem vastas discussões a respeito de ética, mas o objetivo de se utilizar descompiladores no presente trabalho transcende este tópico. O interesse aqui é utilizar um descompilador no sentido de se aprender mais a respeito da estrutura Java. A seguir algumas ferramentas para descompilar as classes de Java serão apresentadas.

a) DeJaVu (versão 1.0)

O DeJaVu é associado com OEW e é distribuído como uma versão grátis (*trial version*). A ferramenta produz uma saída que é quase final, para ser compilada, mas não deve ser compartilhada com Mocha por que o DeJaVu pode apresentar incompatibilidade com o JDK. A análise de fluxo de DeJaVu falha frequentemente, e nesses casos produz um código que está correto mas é quase impossível de se entender. Como um compilador de alto desempenho para Java, é um compilador que produz arquivos diretamente executáveis de fonte Java. Há algumas limitações que podem atrapalhar, mas a produção é executada rapidamente.

b) Mocha (Beta 1)

Em 1996, Hanpeter van Vliet lançou uma versão de beta de um descompilador Java chamado Mocha. O software causou certa controvérsia e sua distribuição pública foi temporariamente interrompida. É uma ferramenta para engenharia reversa de software, assim sendo o Mocha é um descompilador de classes completo, escrito em Java. Com uma classe fornecida, o Mocha produz um arquivo fonte compilável para a classe. O arquivo *.mocha resultante é semelhante ao arquivo *.Java original se for feita uma comparação com a classe original compilada. Porém o autor, Hanpeter, morreu de câncer no dia 31 de dezembro de 1996, com 34 anos, e por causa disso a distribuição original do descompilador desapareceu. A versão atual é um substituto do descompilador, e mantém as intenções do produto original. Os *softwares* produzidos por Hanpeter, ou seja Mocha e Crema, foram melhorados e integrados no *Jbuilder*, produto da Borland, que reclama os direitos sobre esse *software*, devido a um suposto acordo com o autor. Apesar disso o Mocha pode ser encontrado livremente em vários sites. Uma das versões que é uma evolução do Mocha é o *software* Jasmine.

c) WingDis (versão atual 2.12)

Há uma versão demonstrativa do WingDis, outro descompilador Java, mas é limitada e não é possível fazer uma análise realística de suas capacidades e falhas. Os produtores, Wingsoft, fornecem uma versão válida por 30 dias não limitada para testes. Em geral, vale a pena usar para desmontar uma classe; mas não há garantia de que uma classe desmontada venha a ser recompilada. Parece ser estruturalmente bem montado, é totalmente instalável e pode ser completamente executada. Não deve ser compartilhada com o Mocha por que pode haver conflito com o JDK. Podem ocorrer erros em algumas conversões fáceis, como por exemplo $x=x+1$ não ser convertida para $x++$, e ao invés disso ser traduzida como $x=this.x+1$. Pior do que isso, uma construção como $x++$ pode usar um valor original mas também como efeito secundário, um valor errado pode ser ocasionalmente utilizado. A ferramenta freqüentemente produz um código que pode ser compilado mas que não é funcionalmente equivalente ao original.

D.6.5 - Toba: Um tradutor de Java para C

A ferramenta Toba traduz arquivos de classes Java em código fonte C. Isto permite a construção de programas diretamente executáveis que evitam o *overhead* da interpretação. O Toba trata de aplicativos independentes, e não de *applets*. O Toba, na verdade, lê arquivos de classe Java, que são os arquivos de código objeto produzidos pelo compilador Java para a Máquina Java Virtual (JVM). O código gerado não é projetado para leitura humana (embora isso não seja impossível) mas sim para uma fácil otimização por um compilador de linguagem C razoavelmente inteligente. A tradução de um arquivo classe produz um arquivo *.c e um arquivo *.h. Cada *arquivo *.c possui referências para seu próprio arquivo *.h, e também o arquivo *.h de outras classes referenciadas pelos arquivos de classe.

A versão Toba 1.1 suporta os API da versão JDK 1.1, e pode ser executada na plataforma Solaris 2.5 sobre SPARC e na plataforma Linux 2.0 sobre a família x86 da Intel. Todas essas plataformas têm suporte para *threads* e AWT, e um compilador JIT está disponível especificamente para a plataforma de Linux/Intel.

Quanto à compatibilidade, a ferramenta Toba suporta o carregamento dinâmico em plataformas que têm um compilador JIT. Em outras plataformas, o Toba não suporta o carregamento dinâmico real, mas permite o carregamento de classes pré-compiladas de forma dinâmica. A maioria dos aplicativos Java não precisam de carregamento dinâmico e a função para o suporte de AWT atua de forma idêntica quando construído pela ferramenta Toba. Algumas diferenças podem aparecer quando é usado AWT baseado em pacotes BISS AWT. As diferenças adicionais são secundárias.

Quanto à interface de código nativa, as funções escritas em C podem ter ligações (*link*) com arquivos Java traduzidos pela ferramenta Toba. Porém, esta interface difere da interface de código nativa definida pela Sun

Quanto ao desempenho, este pode variar, segundo seus projetistas, de notável até espetacular, dependendo da aplicação. Os aplicativos de computações intensas têm ganhos maiores. Os arquivos Toba executáveis são aproximadamente quatro vezes mais rápidos do que a versão interpretada.

A implementação da ferramenta Toba consiste em um tradutor de arquivos de classe escrito em Java e uma interface codificada em C. O código de suporte inclui:

- ✓ Uma biblioteca de API Java (traduzido pela ferramenta Toba)
- ✓ Um coletor de lixo Boehm-Demers-Weiser
- ✓ Um pacote BISS AWT
- ✓ Uma grande biblioteca de Colin Plumb;
- ✓ Uma biblioteca de compressão zlib
- ✓ Bibliotecas padrões de sistema

Os tipos de dados são vistos no arquivo *toba.h*. Este arquivo define tipos gerais e estruturas de dados. Cada parâmetro do método é um dos *tipos Int, Long, Float, Double*, ou *Object*. Os tipos que são retornados podem ser quaisquer um desses, ou *Void*. Outros tipos integrais são passados como *Int*. Os ponteiros para objeto representam todos os tipos de referência, inclusive *arrays*. Todas as estruturas de objeto começam com um ponteiro de classe; para qualquer ponteiro de objeto.

As estruturas de dados específicas de uma classe particular são definidas no arquivo de *classname.h*. A estrutura de classe é uma variável global chamada *cl_classname*; e contém informações de classe, instância de ponteiros de função, e qualquer variável estática. A estrutura de um exemplo de objeto está definida através de *in_classname* de *struct*.

Cada arquivo de código nativo começa incluindo definições gerais e de classes específicas:

```
#include <toba.h>
#include "classname.h"
```

O arquivo de *classname.h* deve estar presente quando o código nativo é compilado. Isto é organizado automaticamente se os códigos Java e C são construídos juntos por um comando como *toba maincode.Java native.c*. Todo método nativo é uma função de C, e o código pode ser muito simples.

Toda classe deve ser inicializada antes de ser usada. De forma geral, isto é feito automaticamente pela ferramenta Toba. Porém, para o programador ter certeza que uma classe foi inicializada antes de que fosse feito o acesso ao código nativo, a classe deve ser inicializada chamando *initclass*. É possível chamar *initclass* mais de uma vez na mesma classe.

A biblioteca da ferramenta Toba contém vários *helps* escritos para apoiar o entendimento do código gerado e da biblioteca de API. Estas funções de ajuda estão disponíveis para uso através de métodos nativos. Um exemplo simples que ilustra a codificação de um método nativo pode ser visto no apêndice XXX.

A ferramenta Toba foi produzida por um grupo denominado Sumatra, do qual fazem parte Todd Proebsting, John Hartman, Gregg Townsend, Peter Bigot, Patrick Bridges, Tim Newsham, and Scott Watterson. O grupo está baseado no Departamento de Ciências da Computação da Universidade de Arizona. A primeira versão foi uma versão Beta 4 lançada em 17 de dezembro de 1996. A versão atual é a 1.1, e foi lançada em 31 de julho de 1998.

Apêndice E

Características da Linguagem Java

Neste apêndice serão apresentadas algumas características de Java, enquanto linguagem de programação, levando em conta comandos, tipos de dados e funções de biblioteca. Inicialmente faz-se uma comparação com a linguagem C++ devido a proximidade dos conceitos. Mas como a linguagem Java difere de C++, é necessário fazer-se uma distinção para evitar possíveis confusões.

E.1 Diferenças entre Java e C/C++

Para quem conhece a linguagem C++, à primeira vista a linguagem Java parece mais um novo dialeto do que uma nova linguagem. O programador de C/C++ imediatamente se sente familiarizado com a sua estrutura e palavras-chave. Esta familiaridade tem sido uma das principais responsáveis pelo grande sucesso da Java entre os programadores C/C++. Muitos já acreditam que a Java é a virtual sucessora do C++ como linguagem de programação preferida entre os desenvolvedores de aplicativos.

Mas as semelhanças param por aí. A Java é uma nova linguagem, muito mais robusta e muito mais simples que C/C++. Elimina uma gama de redundâncias e complicações do C/C++, e introduz novos recursos que facilitam o seu uso no

desenvolvimento de aplicativos distribuídos. A Java não tem aritmética de apontadores, estruturas, nem pré-processador. Possui um modelo de gerenciamento de memória otimizado que dispensa o programador da tarefa de liberar memória. Como resultado, programas escritos em Java tem menos bugs, não tem vazamentos de memória e não provocam GPFs. Diferentemente de C++, que é uma mera extensão do C com suporte à classes e objetos, a Java é uma verdadeira linguagem de programação orientada a objetos.

Uma das principais diferenças entre Java e C/C++ é o processo de compilação e execução. A compilação de um programa em C++ traduz o código-fonte em instruções que são interpretadas pelo processador do computador onde irá rodar. Já o processo de compilação de um programa em Java traduz o código fonte em instruções, como já visto, chamadas de *bytecodes*.

Em Java, toda classe implementa uma extensão à sua superclasse, herdando todos os métodos e variáveis não declarados *private*. Quando a palavra-chave “*extends*” não está presente em uma declaração de classe, assume-se que a mesma implementa uma extensão à classe *Object*, que não tem superclasse e está na raiz da hierarquia das classes em Java.

Uma diferença importante é a ausência de um destrutor na classe Java. Não é necessário destruir objetos criados em Java pois os objetos não mais utilizados são automaticamente destruídos por um mecanismo de “coleta de lixo” e a memória ocupada por eles é automaticamente liberada. Sendo assim, a classe não precisa de um método destrutor. Caso seja necessário realizar algo antes de destruir o objeto, é possível sobrepor o método *finalize()* (definido na classe *Object* e portanto herdado por todas as classes) com a finalização desejada. Expressões de controle de fluxo como *while*, *do... while*, *switch*, *for*, *if... else* tem em Java a mesma sintaxe que em C e C++ (ver item 2.5.2).

Em Java não existe nada parecido com os pré-processadores utilizados em C e C++. Todas as informações que definem uma classe, seus métodos, construtores e

variáveis são incluídos em um único arquivo. Pode-se implementar várias classes em um único arquivo fonte, mas o compilador criará cada classe em um arquivo *-bytecode* distinto.

Operações realizadas em C com *#define*, *#ifdef*, etc. podem ser implementadas em Java através do uso de constantes variáveis declaradas com o modificador “final”. Não existe *#include* em Java. As classes chamadas pelo programa são dinamicamente carregadas durante a execução.

Diferentemente de C++, a herança de métodos e variáveis de mais de uma superclasse não está implementada em Java, porém é possível utilizar constantes e métodos que foram implementados em classes diferentes através das *Interfaces*. As interfaces são um novo tipo de referência que especificam métodos e constantes, porém não definem qualquer especificação. Todos os métodos das interfaces são implicitamente abstratos e necessitam ser implementados nas classes que os utilizam.

A linguagem Java foi projetada com uma série de recursos para suportar a implementação de *multithreading* a nível de programação. Ou seja, pode-se definir diferentes trechos do programa que executarão paralelamente. A Java contém recursos para garantir a integridade dos dados e fazer da divisão de uma aplicação em *threads* uma tarefa simples. Primitivas da linguagem controlam a sincronização automática e o travamento de métodos e estruturas críticas. Qualquer método declarado *synchronized* recebe uma trava que impede seu uso por duas ou mais linhas de execução ao mesmo tempo.

Em Java, uma *string* não é uma cadeia de caracteres terminada em \0000, mas sim um objeto. A expressão:

```
String s;
```

declara a variável nome como sendo do tipo *string*. Neste ponto, nome é apenas uma referência que aponta para um objeto inexistente. Para efetivamente criar o objeto, é necessário criar uma instância da classe *String*:

```
nome = new String("João");
```

Um novo objeto tipo *string* é então criado, contendo “João”.

Apesar dos tipos primitivos possuírem nomes semelhantes aos do C++, seu tamanho e a forma como são tratados em Java são diferentes. Java introduz um tipo *boolean* que pode assumir os valores *true* ou *false*. Esse tipo é obrigatório em expressões condicionais. Todos os tipos primitivos em Java têm tamanho fixo e independem de plataforma. Em C ou C++, se uma variável não-inicializada é usada, pode gerar resultados imprevisíveis já que o seu conteúdo é desconhecido. Em Java, se uma variável não for inicializada, um valor *default* é utilizado. A tabela 2.2 relaciona o tamanho e os valores *default* dos tipos primitivos de Java.

Tabela 2.2: Tipos primitivos em Java

<i>Tipo</i>	<i>Tamanho</i>	<i>Default</i>
Boolean	-	false
char	16 bit	\u0000
byte	8 bit	0
short	16 bit	0
int	32 bit	0
long	64 bit	0L
float	32 bit	0.0f
double	64 bit	0.0d

Observe que o tipo *char* em Java possui 16 bits. Isto é porque *char* segue o padrão *Unicode*. Caracteres *Unicode* podem aparecer em qualquer lugar de um programa Java, podendo inclusive ser usados em nomes de variáveis e classes. Em Java, *null* é uma palavra-chave e não corresponde a 0 ou -1, mas à ausência de um valor. Objetos e *arrays* vazios contêm *null*.

O controle de condições excepcionais não é característica exclusiva de Java. Seu uso é permitido em C++. A diferença é que, em Java, o controle de exceções é obrigatório. Se você chama um método que provoca uma exceção, a Java não deixa você compilar o programa enquanto não incluir um trecho de código que instrua o programa a lidar com ela ou declarar explicitamente que o método provoca tal exceção. O controle de exceções torna o código mais robusto, permitindo lidar com ocorrências anormais em

poucos pontos do programa, em vez de ter que se preocupar com o que vai acontecer em cada linha de execução. A sintaxe é semelhante a C++.

E.2 A linguagem Java

A maioria dos programas em Java contém um ou mais objetos ou unidades de compilação. Cada unidade contém declarações de pacote e importação, além de declarações de classes e interface. Embora possam existir várias classes e interfaces por unidade, cada unidade pode ter no máximo uma interface e uma classe públicas, porque é dessa maneira que a unidade será acessada por outros objetos do sistema. O restante deve ser privado, e pode ser declarado explicitamente como privado na declaração de classe, ou aceito como *default*.

Ao utilizar o compilador Java, *javac*, estará compilando-se código fonte Java em *bytecodes*. Esses *bytecodes* contêm instruções independentes de máquina. Em seguida, o interpretador do Java, também chamado de *java*, e parte do ambiente de execução do Java, interpreta esses *bytecodes*. Ao criar uma unidade de compilação do Java, cada um deverá ter um sufixo *.java*.

B.2.1 Construção léxica da Java

Este item descreve as gramáticas livres de contexto usadas nesta especificação para definir a estrutura léxica e sintática de um programa Java.

Uma gramática livre de contexto consiste em várias produções. Cada produção tem um símbolo abstrato chamado um não terminal ao seu lado esquerdo, e uma sucessão de um ou mais não terminal e símbolos terminais ao seu lado direito. Para cada gramática, os símbolos terminais são tirados de um alfabeto especificado. A partir de uma regra de produção que consiste em um único não terminal distinto, denominado símbolo inicial, uma determinada gramática livre de contexto especifica uma linguagem, isto é, um conjunto infinito de possíveis sucessões de símbolos terminais que podem

ser o resultado da substituição sucessiva de qualquer não terminal de um elemento ao lado direito em uma produção por um não terminal que está ao lado esquerdo.

A gramática léxica para Java tem como seus símbolos terminais os caracteres do conjunto *Unicode*. É definido um conjunto de produções, a partir do símbolo inicial *CompilationUnit*. Os caracteres de entrada e fins de linha resultam em fim de processamento e reconhecimento de linha de entrada, esses elementos são reduzidos a uma sucessão de elementos entrada. Esses elementos que não são nenhum espaço branco ou comentários são os *tokens*. Os *tokens* são os símbolos terminais da gramática sintática Java. Este processo é especificado pelas produções seguintes:

Input:
InputElements _{opt} Sub _{opt}
InputElements:
InputElement
InputElements InputElement
InputElement:
WhiteSpace
Comment
Token
Token:
Identifier
Keyword
Literal
Separator
Operator
Sub:
the ASCII SUB character, also known as "control-Z"

A entrada de um programa Java é descrito como sucessões de caracteres *Unicode*. São escritos programas Java usando o conjunto de caracteres *Unicode*, versão 2.0¹. Com exceção de comentários, identificadores, e dos conteúdos para caracteres e

¹ Informações sobre esta codificação podem ser encontradas em <http://www.unicode.org> e <ftp://unicode.org>

strings de literais, todos os elementos de entrada em um programa Java é formado por caracteres ASCII (ou *Unicode* que resultam em caráter ASCII ANSI X3.4) que é o Código Padrão americano para intercâmbio de Informação. Os primeiros 128 caracteres do *Unicode* são os caracteres ASCII. As implementações de Java primeiro reconhecem os caracteres *Unicode* na entrada, traduzem os caracteres ASCII acrescentando quatro dígitos de hexadecimal para obter o caracter *Unicode* e mantém todos os outros caracteres inalterados. Este passo de tradução resulta em uma sucessão de caracteres de entrada *Unicode* :

UnicodeInputCharacter:
UnicodeEscape
RawInputCharacter
UnicodeEscape:
UnicodeMarker HexDigit HexDigit HexDigit HexDigit
UnicodeMarker:
u
UnicodeMarker u
RawInputCharacter:
any Unicode character
HexDigit: um de
0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F
The \, u, e dígitos hexadecimal que são todos caracteres ASCII.

Os elementos de entrada, como o espaço em branco, e os comentários são eliminados. O espaço em branco (*WhiteSpace*) é definido como espaço em ASCII, tab horizontal, e caracter *form feed*, e também fins de linha:

WhiteSpace:
o ASCII SP character, também conhecido como "space"
o ASCII HT character, também conhecido como "horizontal tab"
o ASCII FF character, também conhecido como "form feed"
Terminadordelinha

Em relação aos comentários (*Comment*), a Java define três tipos:

`/* texto */` - É um comentário tradicional: todo o texto desde o caracter ASCII `/*` até o caracter ASCII `*/`, tudo é ignorado (como em C e C++).

`// texto` - É um comentário de única-linha: todo o texto do caracter ASCII `//` até o fim da linha são ignorados (como em C++).

`/** documentation */` - É um comentário de documentação: o texto incluído pelos caracteres ASCII `/**` e `*/` pode ser processado por uma ferramenta separada para preparar documentação automaticamente gerada a partir de classe, interface, construtor, ou declaração (método ou campo).

Estes comentários são especificados formalmente pelas produções seguintes:

```
Comment:
    TraditionalComment
    EndOfLineComment
    DocumentationComment
TraditionalComment:
    /* NotStar CommentTail
EndOfLineComment:
    // CharactersInLineopt LineTerminator
DocumentationComment:
    /** CommentTailStar
CommentTail:
    * CommentTailStar
    NotStar CommentTail
CommentTailStar:
    /
    * CommentTailStar
    NotStarNotSlash CommentTail
NotStar:
    InputCharacter but not *
    LineTerminator
NotStarNotSlash:
    InputCharacter but not * or /
```


LineTerminator
CharactersInLine:
InputCharacter
CharactersInLine InputCharacter

Formam-se então os símbolos terminais para a gramática sintática Java e são chamados símbolos Java. Estes símbolos são identificadores, palavras chaves, literais, separadores, e os operadores da linguagem Java.

Um identificador é uma sucessão de comprimento ilimitado de letras e dígitos aceitos pela Java , sendo que o primeiro caracter deve ser uma letra. Um identificador não pode ter a mesma sequência de uma palavra reservada, literal *Boolean* , ou o elemento nulo:

Identifier:
IdentifierChars but not a Keyword or BooleanLiteral or NullLiteral
IdentifierChars:
JavaLetter
IdentifierChars JavaLetterOrDigit
JavaLetter:
any Unicode character that is a Java letter (see below)
JavaLetterOrDigit:
any Unicode character that is a Java letter-or-digit (see below)

As palavras reservadas são as sucessões dos caracteres listados a seguir, formadas de letras ASCII, reservadas para uso como palavras chaves e que não podem ser usadas como identificadores:

abstract	default	if	private	throw
boolean	do	implements	protected	throws
break	double	import	public	transient
byte	else	instanceof	return	try
case	extends	int	short	void
catch	final	interface	static	volatile
char	finally	long	super	while
class	float	native	switch	

const	for	new	synchronized
continue	goto	package	this

Um literal é a representação de um valor de um tipo primitivo, um tipo *string*, ou um tipo nulo:

Literal:
IntegerLiteral
FloatingPointLiteral
BooleanLiteral
CharacterLiteral
StringLiteral
NullLiteral

Um inteiro literal pode ser expresso em base decimal (base 10), hexadecimal (base 16), ou octal (base 8):

IntegerLiteral:
DecimalIntegerLiteral
HexIntegerLiteral
OctalIntegerLiteral
DecimalIntegerLiteral:
DecimalNumeral IntegerTypeSuffixopt
HexIntegerLiteral:
HexNumeral IntegerTypeSuffixopt
OctalIntegerLiteral:
OctalNumeral IntegerTypeSuffixopt
IntegerTypeSuffix: one
1 L

O ponto flutuante literal tem as seguintes partes: uma parte é um número inteiro, um ponto decimal (representado por um caracter vírgula ASCII), uma parte fracionária, um expoente e um tipo sufixo. O expoente, se presente, é indicado pela letra ASCII ‘e’ ou ‘E’, seguido por um inteiro com sinal (opcional).

FloatingPointLiteral:
Digits . Digitsopt ExponentPartopt FloatTypeSuffixopt
. Digits ExponentPartopt FloatTypeSuffixopt

Digits ExponentPart FloatTypeSuffixopt
Digits ExponentPartopt FloatTypeSuffix
ExponentPart:
ExponentIndicator SignedInteger
ExponentIndicator: one of
e E
SignedInteger:
Signopt Digits
Sign: one of
+ -
FloatTypeSuffix: one of
f F d D

O tipo *boolean* pode ter dois valores, representado pelos literais verdade (*true*) e falso (*false*), formado de letras ASCII:

CharacterLiteral:
' SingleCharacter '
' EscapeSequence '
SingleCharacter:
InputCharacter but not ' or \

Um *string* literal consiste em zero ou mais caracteres incluídos entre aspas duplas. Cada caracter pode ser representado por uma sequência de *escapes*:

StringLiteral:
" StringCharactersopt "
StringCharacters:
StringCharacter
StringCharacters StringCharacter
StringCharacter:
InputCharacter but not " or \
EscapeSequence

Os caracteres e *string escape* permitem a representação de alguns caracteres não gráficos como também a aspas simples, aspas duplas e *backslash*, letras em caracteres e

strings literais.

EscapeSequence:		
\ b	/* \u0008: backspace BS	*/
\ t	/* \u0009: horizontal tab HT	*/
\ n	/* \u000a: linefeed LF	*/
\ f	/* \u000c: form feed FF	*/
\ r	/* \u000d: carriage return CR	*/
\ "	/* \u0022: double quote "	*/
\ '	/* \u0027: single quote '	*/
\\	/* \u005c: backslash \	*/
OctalEscape	/* \u0000 to \u00ff: from octal value */	
OctalEscape:		
\ OctalDigit		
\ OctalDigit OctalDigit		
\ ZeroToThree OctalDigit OctalDigit		
OctalDigit: one of		
0 1 2 3 4 5 6 7		
ZeroToThree: one of		
0 1 2 3		

O tipo nulo (*null*) tem um valor, a referência nula, representado pelo literal nulo que é formado de caracteres ASCII. Um nulo literal sempre é do tipo nulo.

NullLiteral:
null

Os nove caracteres ASCII que seguem são os separadores Java (*punctuators*):

() { } [] : . ,

Os 37 tokens seguintes são os operadores Java, formados de caracteres ASCII:

=	>	<	!	~	?	:										
==	<=	>=	!=	&&		++	--									
+	-	*	/	&		^	%	<<	>>	>>>						
+=	-=	*=	/=	&=	=	^=	%=	<<=	>>=	>>>=						

Esta gramática tem símbolos para Java definidos pela gramática léxica como seus símbolos terminais. É definido um jogo de produções, a partir do símbolo inicial *CompilationUnit*, isso descreve como sucessões de símbolos podem formar programas em Java corretos sintaticamente:

CompilationUnit : *PackageDeclaration_{opt} ImportDeclarations_{opt} TypeDeclarations_{opt}*

E.2.2 Construções Sintáticas de Java

Uma versão LALR(1)² da gramática sintática é apresentada de forma resumida a seguir. A gramática no corpo desta especificação é bem parecida com a gramática LALR(1) mas mais legível.

A Java é uma linguagem fortemente tipada, o que significa que toda variável e toda expressão tem um tipo que é conhecido no tempo de compilação. Os tipos limitam os valores para uma variável, podem limitar o que uma expressão pode produzir, podem limitar as operações apoiadas nesses valores, e podem determinar o significado das operações. A tipagem forte auxilia a descoberta de erros em tempo de compilação. A seguir são especificados de forma resumida os elementos sintáticos de Java.

Tipos Os tipos da linguagem Java são divididos em duas categorias: tipos primitivos e tipos de referência. Os tipos primitivos são os tipos *boolean* e o tipo numérico. Os tipos numéricos são *byte*, *short*, *int*, *long*, *char*, *floating-point*, *float*, e *double*. Os tipos de referência são os tipos de classe (*class type*), tipos de interface (*interface types*), e tipos de *array* (*array types*). Também há um tipo nulo especial (*null*

² LALR, ou " LookAhead LR ", é uma variante um pouco menos poderosa do analisador LR, e produz tabelas de análises gramaticais menores. Um *parser* LR(k) é um tipo determinístico de *parser* que usa k símbolos de antecipação no máximo para determinar sua próxima ação em qualquer ponto dentro da análise gramatical. Se (k) é omitido, então é assumido que k é 1. Um *parser* de LALR tem vantagens significativas em tamanho em relação a LR. É possível construir gramáticas que podem ser analisadas gramaticalmente por um *parser* de LR mas não podem ser analisadas gramaticalmente por um *parser* de LALR. Quer dizer, o *parser* de LALR analisará essas gramáticas de modo a caracterizá-las como ambíguas, enquanto o *parser* de LR não o fará. Na prática, porém, tais gramáticas parecem ser raras e as ambigüidades normalmente podem ser eliminadas através de revisões secundárias.

type). Um objeto em Java é dinamicamente criado como instância de um tipo de classe ou um *array* dinamicamente criado. Os valores de um tipo de referência são referências a objetos. Todos os objetos, inclusive *arrays*, suportam os métodos de Objeto de classe. Os *string* Literais são representados através de objetos de *string*. A seguir relacionam-se suas regras de produção, juntamente com as regras de produção de valores e variáveis.

Type:
PrimitiveType
ReferenceType
PrimitiveType:
NumericType
boolean
NumericType:
IntegralType
FloatingPointType
IntegralType: one of
byte short int long char
FloatingPointType: one of
float double
ReferenceType:
ClassOrInterfaceType
ArrayType
ClassOrInterfaceType:
Name
ClassType:
ClassOrInterfaceType
InterfaceType:
ClassOrInterfaceType
ArrayType:
PrimitiveType []
Name []
ArrayType []

Os tipos são os mesmos se eles têm os mesmos nomes completos de qualificadores e são carregados pelo mesmo carregador de classe (*classloader*). Nomes de tipos são usados em declarações, em *casts*, em instâncias de classe criando expressões, em expressões de criação de ordem, e em expressões de operador *instanceof*.

Variáveis Uma variável é um local de armazenamento. Uma variável de um tipo primitivo sempre retém um valor de tipo certo. Uma variável de um tipo de classe T pode ter uma referência nula ou uma referência para uma instância de classe T ou para qualquer classe que seja uma subdivisão de classe de T. Uma variável do tipo interface pode ter uma referência nula ou uma referência a qualquer instância de qualquer classe que implementa a interface. Se T for um tipo primitivo, então uma variável de tipo *array* de T pode ter uma referência nula ou uma referência a qualquer *array* de tipo *array* de T. Se T for um tipo de referência, então uma variável do tipo *array* de T pode ter uma referência nula ou uma referência a qualquer *array* do tipo *array* de S , tal que o tipo S é associável ao tipo T. Uma variável de tipo Objeto pode ter uma referência nula ou uma referência a qualquer objeto, por exemplo de classe ou *array*.

Nomes (*names*) São usados nomes para declarar entidades em um programa Java. Uma entidade declarada é um pacote (*package*), tipo de classe (*class type*), tipo de interface (*interface type*), ou membro (*member*) campo ou método (*field* ou *method*) de um tipo de referência, parâmetro (para um *método*, construtor, ou manipulador de exceção), ou variável local. Os nomes em programas Java ou são simples, consistindo em um único identificador, ou qualificados, consistindo em uma sucessão de identificadores separados por *tokens*. Todo nome introduzido por uma declaração tem uma extensão que é parte do programa Java, dentro do qual a entidade declarada pode se referir para por um simples nome. A seguir são listadas suas regras de produção:

Name:
SimpleName
QualifiedName
SimpleName:
Identifier

QualifiedName:

Name . Identifier

Pacotes Os programas Java são organizados como conjuntos de pacotes (*packages*). Cada pacote tem seu próprio conjunto de nomes para tipos, o que ajuda a prevenir conflitos entre nomes. Um tipo é acessível fora do pacote só pela declaração do tipo declarado público. A construção da estrutura para pacotes é hierárquica. Os membros de um pacote são classe e tipos de interface, e são declarados em unidades de compilação do pacote, e sub-pacotes do qual pode conter unidades de compilação e sub-pacotes deles próprios. A seguir relacionam-se suas regras de produção:

compilationUnit:

PackageDeclarationopt ImportDeclarationsopt TypeDeclarationsopt

ImportDeclarations:

ImportDeclaration

ImportDeclarations ImportDeclaration

TypeDeclarations:

TypeDeclaration

TypeDeclarations TypeDeclaration

PackageDeclaration:

package Name ;

ImportDeclaration:

SingleTypeImportDeclaration

TypeImportOnDemandDeclaration

SingleTypeImportDeclaration:

import Name ;

TypeImportOnDemandDeclaration:

import Name . * ;

TypeDeclaration:

ClassDeclaration

InterfaceDeclaration

:

Classes As declarações de classe definem novos tipos de referência e descrevem como eles são implementados. O nome de uma classe tem como sua extensão todas as

declarações de tipo no pacote no qual a classe é declarada. Uma classe pode ser declarada abstrata (*abstract*) e deve ser declarada abstrata se for implementada de forma incompleta; tal classe não pode ser instanciada, mas pode ser estendida através de subdivisões de classe. Uma classe pode ser declarada final no caso de não poder ter subdivisões nessa classe. Se uma classe é declarada pública, então pode ser referida por outros pacotes. Cada classe, menos o Objeto é uma extensão de uma única classe existente, ou seja, uma subclasse de uma classe, e pode implementar interfaces. A seguir são listadas suas regras de produção:

ClassDeclaration:
Modifiersopt class Identifier Superopt Interfacesopt ClassBody
Super:
extends ClassType
Interfaces:
implements InterfaceTypeList
InterfaceTypeList:
InterfaceType
InterfaceTypeList . InterfaceType
ClassBody:
{ ClassBodyDeclarationsopt }
ClassBodyDeclarations:
ClassBodyDeclaration
ClassBodyDeclarations ClassBodyDeclaration
ClassBodyDeclaration:
ClassMemberDeclaration
StaticInitializer
ConstructorDeclaration
ClassMemberDeclaration:
FieldDeclaration
MethodDeclaration
FieldDeclaration:
Modifiersopt Type VariableDeclarators :
VariableDeclarators:

VariableDeclarator
VariableDeclarators . VariableDeclarator
VariableDeclarator:
VariableDeclaratorId
VariableDeclaratorId = VariableInitializer
VariableDeclaratorId:
Identifier
VariableDeclaratorId []
VariableInitializer:
Expression
ArrayInitializer
MethodDeclaration:
MethodHeader MethodBody
MethodHeader:
Modifiersopt Type MethodDeclarator Throwsopt
Modifiersopt void MethodDeclarator Throwsopt
MethodDeclarator:
Identifier (FormalParameterListopt)
MethodDeclarator []
FormalParameterList:
FormalParameter
FormalParameterList . FormalParameter
FormalParameter:
Type VariableDeclaratorId
Throws:
throws ClassTypeList
ClassTypeList:
ClassType
ClassTypeList . ClassType
MethodBody:
Block
:
StaticInitializer:
static Block
ConstructorDeclaration:

Modifierso _{pt} ConstructorDeclarator Throws _{opt} ConstructorBody
ConstructorDeclarator:
SimpleName (FormalParameterList _{opt})
ConstructorBody:
{ ExplicitConstructorInvocation _{opt} BlockStatement _{opt} }
ExplicitConstructorInvocation:
this (ArgumentList _{opt}) ;
super (ArgumentList _{opt}) ;

Interface Uma declaração de interface introduz um novo tipo referência cujos membros são constantes e os métodos são abstratos. Este tipo não tem nenhuma implementação, mas nesse caso, classes sem conexões podem ser implementadas fornecendo implementação para métodos abstratos. Um programa Java pode usar interfaces para implementar classes relacionadas compartilhadas com uma super-classe abstrata comum, ou somar métodos para Objetos. Uma interface pode ser declarada para ser uma extensão direta de uma ou mais interfaces, significando que especifica implicitamente todos os métodos abstratos e constantes das interfaces que estende, com exceção de qualquer constante que deva ser escondida. A seguir são relacionadas suas regras de produção:

InterfaceDeclaration:
Modifierso _{pt} interface Identifier ExtendsInterfaces _{opt} InterfaceBody
ExtendsInterfaces:
extends InterfaceType
ExtendsInterfaces . InterfaceType
InterfaceBody:
{ InterfaceMemberDeclarations _{opt} }
InterfaceMemberDeclarations:
InterfaceMemberDeclaration
InterfaceMemberDeclarations InterfaceMemberDeclaration
InterfaceMemberDeclaration:
ConstantDeclaration
AbstractMethodDeclaration
ConstantDeclaration:

FieldDeclaration AbstractMethodDeclaration: MethodHeader ;
--

Uma classe pode ser declarada para implementar uma ou mais interfaces diretamente, significando que qualquer exemplo da classe implementa todos os métodos abstratos especificados pela interface ou pelas interfaces. Uma classe necessariamente implementa todas as interfaces que suas super-classes direta e super-interfaces diretas criam. Esta (múltipla) herança de interface permite que objetos suportem (múltiplos) comportamentos comuns sem compartilhar qualquer implementação. Uma variável cuja declaração de tipo é um tipo de interface, pode ter como valor uma referência para qualquer objeto, que é uma instância de uma classe declarada para implementar uma interface especificada. Não é suficiente que a classe exista para implementar todos os métodos abstratos da interface; a classe ou uma de suas super-classes devem ser declaradas para de fato implementar a interface, ou então não se considera que a classe implemente a interface.

Array Os *arrays* Java são objetos, dinamicamente criados, e podem ser associados a variáveis de tipo Objeto. Todos os métodos de Objeto de classe são invocados em uma *array*. Um *array* objeto contém várias variáveis. O número de variáveis pode ser zero no caso em que o *array* está vazio. As variáveis contidas dentro de um *array* não tem nenhum nome; ao invés disso elas são referenciadas por expressões de acesso a *array* que usam valores de índice internos não negativos. Estas variáveis são chamadas componentes de *array*. Se um *array* tiver n componentes, dizemos que n é o comprimento do *array*. Os componentes do *array* são referências que usam índices inteiros de 0 a $n-1$. Todos os componentes de um *array* têm o mesmo tipo, denominado de componente do *array*. Se o tipo de componente de um *array* é T , então o tipo do próprio *array* é escrito $T[]$. O tipo de componente de um *array* pode ser um tipo de *array*. Os componentes de tal *array* podem conter referências a *sub-arrays*. A partir de qualquer tipo de *array*, considerando-se seu tipo de componente, e sucessivamente o tipo de componente daquele tipo, em determinado ponto deve ser encontrado um tipo de

componente que não seja um *array*. Este é chamado de tipo de elemento do *array* original, e os componentes a este nível da estrutura de dados são chamados elementos do *array* original. Há uma situação na qual um elemento de um *array* pode ser um *array*: Se o tipo de elemento é Objeto, então alguns ou todos os elementos podem ser *arrays*, porque qualquer objeto de *array* pode ser associado a qualquer variável de tipo Objeto. A seguir são relacionadas as suas regras de produção:

ArrayInitializer:
{ VariableInitializers _{opt} ,opt }
VariableInitializers:
VariableInitializer
VariableInitializers . VariableInitializer

Blocos e declarações A sequência de execução de um programa Java é controlada por declarações que são executadas para si mesmas e não possuem valores. Algumas declarações contêm outras declarações como parte de sua estrutura; tais declarações são sub-declarações da declaração. Diz-se que a declaração S contém uma declaração U se não houver nenhuma declaração T diferente de S e U tal que S contém T e T contém U. Da mesma maneira, algumas declarações contêm expressões como parte de sua estrutura. Algumas declarações são familiares aos programadores de linguagens C e C++, como *empty*, *labeled*, *expression*, *if*, *switch*, *while*, *do*, *for*, *break*, *continue*, e *return*. A linguagem Java, de forma diferente de C e C++, não tem nenhuma declaração *goto*. Porém, declarações *break* e *continue* são usadas em Java para permitir utilizar declarações de rótulos. As declarações Java que não existem na linguagem C são as declarações *throw*, *synchronized*, e *try*. A seguir são listadas suas regras de produção:

lock:
{ BlockStatements _{opt} }
BlockStatements:
BlockStatement
BlockStatements BlockStatement
BlockStatement:
LocalVariableDeclarationStatement

Statement

LocalVariableDeclarationStatement:

LocalVariableDeclaration ;

LocalVariableDeclaration:

Type VariableDeclarators

Statement:

StatementWithoutTrailingSubstatement

LabeledStatement

IfThenStatement

IfThenElseStatement

WhileStatement

ForStatement

StatementNoShortIf:

StatementWithoutTrailingSubstatement

LabeledStatementNoShortIf

IfThenElseStatementNoShortIf

WhileStatementNoShortIf

ForStatementNoShortIf

StatementWithoutTrailingSubstatement:

Block

EmptyStatement

ExpressionStatement

SwitchStatement

DoStatement

BreakStatement

ContinueStatement

ReturnStatement

SynchronizedStatement

ThrowStatement

Try Statement

Empty Statement:

:

LabeledStatement:

Identifier : Statement

LabeledStatementNoShortIf:

Identifier : StatementNoShortIf

ExpressionStatement:

 StatementExpression ;

StatementExpression:

 Assignment

 PreIncrementExpression

 PreDecrementExpression

 PostIncrementExpression

 PostDecrementExpression

 MethodInvocation

 ClassInstanceCreationExpression

IfThenStatement:

 if (Expression) Statement

IfThenElseStatement:

 if (Expression) StatementNoShortIf else Statement

IfThenElseStatementNoShortIf:

 if (Expression) StatementNoShortIf else StatementNoShortIf

SwitchStatement:

 switch (Expression) SwitchBlock

SwitchBlock:

 { SwitchBlockStatementGroupsopt SwitchLabelsopt }

SwitchBlockStatementGroups:

 SwitchBlockStatementGroup

 SwitchBlockStatementGroups SwitchBlockStatementGroup

SwitchBlockStatementGroup:

 SwitchLabels BlockStatements

SwitchLabels:

 SwitchLabel

 SwitchLabels SwitchLabel

SwitchLabel:

 case ConstantExpression :

 default :

WhileStatement:

 while (Expression) Statement

WhileStatementNoShortIf:

while (Expression) StatementNoShortIf

DoStatement:

do Statement while (Expression) ;

ForStatement:

for (ForInitopt ; Expressionopt ; ForUpdateopt)

Statement

ForStatementNoShortIf:

for (ForInitopt ; Expressionopt ; ForUpdateopt)

StatementNoShortIf

ForInit:

StatementExpressionList

LocalVariableDeclaration

ForUpdate:

StatementExpressionList

StatementExpressionList:

StatementExpression

StatementExpressionList , StatementExpression

BreakStatement:

break Identifieropt ;

ContinueStatement:

continue Identifieropt ;

ReturnStatement:

return Expressionopt ;

ThrowStatement:

throw Expression ;

SynchronizedStatement:

synchronized (Expression) Block

TryStatement:

try Block Catches

try Block Catchesopt Finally

Catches:

CatchClause

Catches CatchClause

CatchClause:

catch (FormalParameter) Block
Finally:
finally Block

Expressões Muito do trabalho em um programa Java é gasto avaliando-se expressões, ou seus efeitos secundários, tais como tarefas de associação de variáveis, ou os valores que podem ser usados como argumentos ou operandos em expressões grandes ou afetar a sucessão da execução em declarações, ou ambos. A seguir relacionam-se suas regras de produção:

Primary:
PrimaryNoNewArray
ArrayCreationExpression
PrimaryNoNewArray:
Literal
this
(Expression)
ClassInstanceCreationExpression
FieldAccess
MethodInvocation
Array Access
ClassInstanceCreationExpression:
new ClassType (ArgumentListopt)
ArgumentList:
Expression
ArgumentList . Expression
ArrayCreationExpression:
new PrimitiveType DimExprs Dimsopt
new ClassOrInterfaceType DimExprs Dimsopt
DimExprs:
DimExpr
DimExprs DimExpr
DimExpr:
[Expression]
Dims:

```
[ ]
Dims [ ]
FieldAccess:
    Primary . Identifier
    super . Identifier
MethodInvocation:
    Name ( ArgumentListopt )
    Primary . Identifier ( ArgumentListopt )
    super . Identifier ( ArgumentListopt )
ArrayAccess:
    Name [ Expression ]
    PrimaryNoNewArray [ Expression ]
PostfixExpression:
    Primary
    Name
    PostIncrementExpression
    PostDecrementExpression
PostIncrementExpression:
    PostfixExpression ++
PostDecrementExpression:
    PostfixExpression --
UnaryExpression:
    PreIncrementExpression
    PreDecrementExpression
    + UnaryExpression
    - UnaryExpression
    UnaryExpressionNotPlusMinus
PreIncrementExpression:
    ++ UnaryExpression
PreDecrementExpression:
    -- UnaryExpression
UnaryExpressionNotPlusMinus:
    PostfixExpression
    ~ UnaryExpression
    ! UnaryExpression
```

CastExpression

CastExpression:

(PrimitiveType Dimsopt) UnaryExpression

(Expression) UnaryExpressionNotPlusMinus

(Name Dims) UnaryExpressionNotPlusMinus

MultiplicativeExpression:

UnaryExpression

MultiplicativeExpression * UnaryExpression

MultiplicativeExpression / UnaryExpression

MultiplicativeExpression % UnaryExpression

AdditiveExpression:

MultiplicativeExpression

AdditiveExpression + MultiplicativeExpression

AdditiveExpression - MultiplicativeExpression

ShiftExpression:

AdditiveExpression

ShiftExpression << AdditiveExpression

ShiftExpression >> AdditiveExpression

ShiftExpression >>> AdditiveExpression

RelationalExpression:

ShiftExpression

RelationalExpression < ShiftExpression

RelationalExpression > ShiftExpression

RelationalExpression <= ShiftExpression

RelationalExpression >= ShiftExpression

RelationalExpression instanceof ReferenceType

EqualityExpression:

RelationalExpression

EqualityExpression == RelationalExpression

EqualityExpression != RelationalExpression

AndExpression:

EqualityExpression

AndExpression & EqualityExpression

ExclusiveOrExpression:

```
AndExpression
ExclusiveOrExpression ^ AndExpression
InclusiveOrExpression:
    ExclusiveOrExpression
    InclusiveOrExpression | ExclusiveOrExpression
ConditionalAndExpression:
    InclusiveOrExpression
    ConditionalAndExpression && InclusiveOrExpression
ConditionalOrExpression:
    ConditionalAndExpression
    ConditionalOrExpression || ConditionalAndExpression
ConditionalExpression:
    ConditionalOrExpression
    ConditionalOrExpression ? Expression : ConditionalExpression
AssignmentExpression:
    ConditionalExpression
    Assignment
Assignment:
    LeftHandSide AssignmentOperator AssignmentExpression
LeftHandSide:
    Name
    FieldAccess
    ArrayAccess
AssignmentOperator: one of
    = *= /= %= += -= <<= >>= >>>= &= ^= |=
Expression:
    AssignmentExpression
ConstantExpression:
    Expression
```

Quando um programa Java viola os limites semânticos da linguagem Java, a Máquina Virtual Java sinaliza este erro ao programa como uma exceção (*exception*). Um exemplo de tal uma violação é uma tentativa de indexar fora dos limites de um *array*. Algumas linguagens de programação e suas implementações reagem a tais erros de forma rigorosa e encerram o programa; outras linguagens de programação permitem que

uma implementação reaja de um modo arbitrário ou imprevisível. Nenhuma destas abordagens são compatíveis com as metas do projeto Java: garantir portabilidade e robustez. Assim, a Java especifica a ocorrência de uma exceção quando limites semânticos são violados e provoca uma transferência para um controle não local, do ponto onde a exceção ocorreu para um ponto que pode ser especificado pelo programador. Uma exceção denomina-se *thrown* no ponto onde aconteceu e *caught* no ponto para o qual controle é transferido. Em um programa Java podem ser produzidas exceções explicitamente, usando declarações para especificar essas exceções. Isto permite uma alternativa ao estilo antigo de controlar condições de erro devolvendo valores errados, como um valor inteiro negativo onde um valor negativo normalmente não seria esperado. Experiências demonstram que muito freqüentemente tais valores errados são ignorados ou não conferidos pelos usuários, conduzindo a programas que não são robustos, a comportamentos indesejáveis, ou ambos.

Toda exceção é representada por uma instância da classe *Throwable* ou uma de suas sub-classes; para tal um objeto pode ser usado para levar informação do ponto em que uma exceção ocorre ao elemento que irá manipular essa ocorrência. Os elementos que irão fazer a manipulação são estabelecidos por cláusulas de captura de declarações de ensaio. Durante o processo de lançamento de uma exceção, a Máquina Virtual Java completa abruptamente, uma por uma, cada expressão, declaração, métodos e chamadas de construtores, inicializadores estáticos, e campo de expressões de inicialização que começaram mas não completaram a execução na linha atual. Este processo continua até que é encontrado um manipulador que indica que controla aquela exceção particular, dando nome à classe da exceção ou a uma super-classe da classe da exceção. Se nenhum manipulador é achado, então o método *uncaughtException* é chamado para o *ThreadGroup*, que é o pai da corrente, e é feito todo esforço para evitar que uma exceção não seja manipulada. O mecanismo de exceção Java é integrado com o modelo de sincronização Java, de forma que as travas são definidas como declarações sincronizadas e as chamadas de métodos sincronizados são completadas abruptamente.

ImportDeclarations:

ImportDeclaration

ImportDeclarations ImportDeclaration

TypeDeclarations:

TypeDeclaration

TypeDeclarations TypeDeclaration

A sintaxe completa da gramática Java pode ser vista no apêndice A, na forma BNF, assim como os comandos da linguagem Java. Também seus diagramas sintáticos podem ser vistos no mesmo apêndice.