Um Modelo de Execução para Java no Processador Cell BE

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Francisco Hoyos e aprovada pela Banca Examinadora.

Campinas, 25 de janeiro de 2010.

Rodolfo Jardim de Azevedo (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Fabiana Bezerra Müller – CRB8 / 6162

Hoyos, Francisco Rafael Lorenzo
H853m Um modelo de execução para Java no processador Cell BE/Francisco Rafael Lorenzo Hoyos-- Campinas, [S.P. : s.n.], 2009.

Orientador : Rodolfo Jardim de Azevedo.
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1.Cell Broadband Engine. 2.Java (Linguagem de programação de computador). 3.Framework (Programa de computador). 4.Multi core. 5.Arquitetura de computador. I. Azevedo, Rodolfo Jardim de. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: An execution model for Java on the Cell BE processor

Palavras-chave em inglês (Keywords): 1.Cell Broadband Engine.. 2.Java (Computer program languages. 3.Framework (Computer program). 4.Multi core. 5. Computer architecture.

Área de concentração: Linguagens de Programação

Titulação: Mestre em Ciência da Computação

Banca examinadora: Prof. Dr. Rodolfo Jardim de Azevedo (IC-UNICAMP)
Prof. Dr. Sandro Rigo (IC - UNICAMP)
Prof. Dr. Sérgio Takeo Kofuji (LSI - USP)

Data da defesa: 21/12/2009

Programa de Pós-Graduação: Mestrado em Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 21 de dezembro de 2009, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Sérgio Takeo Kofuji
LSI / USP

Prof. Dr. Sandro Rigo
IC / UNICAMP

Prof. Dr. Rodolfo Jardim de Azevedo
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Um Modelo de Execução para Java no Processador Cell BE

Francisco Hoyos

Janeiro de 2010

Banca Examinadora:

- Rodolfo Jardim de Azevedo (Orientador)
- Sandro Rigo
  IC - UNICAMP
- Sérgio Takeo Kofuji
  LSI - USP
- Paulo Cesar Centoducatte (Suplente)
  IC - UNICAMP
- Ivan Luiz Marques Ricarte (Suplente)
  FEEC - UNICAMP

Resumo

O Cell Broadand Engine (Cell BE) é um processador com arquitetura de múltiplos núcleos heterogêneos, voltado para o uso em aplicações de alto desempenho. Talvez mais conhecido como o processador do PlaysStation 3 da Sony, ele também está presente aos milhares no supercomputador Roadrunner da IBM. Entretanto, o SDK do Cell BE não suporta o desenvolvimento de aplicações em Java. Como é sabido, Java é uma das linguagens mais utilizadas hoje em dia, nas mais variadas plataformas de hardware e para quase todos os tipos de aplicações. Este trabalho introduz um novo modelo para a execução de programas Java no Cell BE. Esse modelo permite ao programador Java executar tarefas (partes do código Java do programa principal) nos Synergistic Processing Elements (SPE), que são núcleos especializados do Cell BE, maiores responsáveis pelo grande poder de processamento desse chip. Enquanto outras soluções tentam esconder completamente a arquitetura de múltiplos núcleos heterogêneos do Cell BE, a nova proposta expõe um modelo de memória explicitamente distribuída, habilitando o programador Java a definir exatamente qual código deve executar nos SPEs. A viabilidade do modelo é então demonstrada através da melhoria de desempenho obtida consistentemente com vários programas executados em uma máquina virtual Java modificada para suportar a plataforma Cell BE. Com seis SPEs, esses programas executam, em média, aproximadamente duas vezes mais rápido do que os mesmos programas na máquina virtual Java original.

Abstract

The Cell Broadband Engine (Cell BE) is a processor with a heterogeneous multicore architecture, targeted at high performance applications. Perhaps best known as the processor of Sony's PlayStation 3, it is also used (thousands of them) in the IBM Roadrunner supercomputer. However, the Cell BE SDK does not support Java application development. It is well known that Java is currently one of the most widely used languages, being present on many different hardware platforms and in almost all types of applications. This work introduces a new model for the execution of Java programs on the Cell BE. Such model allows the Java programmer to execute tasks (pieces of the main program's Java code) on the Synergistic Processing Elements (SPE), which are highly specialized cores in the Cell BE and are the main source of the chip's huge processing power. While other solutions try to completely hide the Cell BE's heterogeneous multicore architecture, this new proposal exposes an explicit distributed memory model, empowering the Java programmer to define exactly what code runs on the SPEs. The feasibility of the model is demonstrated by means of consistent performance improvements achieved with several programs executed on a Java virtual machine, which has been modified to support the Cell BE platform. With six SPEs those programs run, on average, around twice as fast as the same programs on the original Java virtual machine.

Agradecimentos

Eu gostaria de agradecer ao meu orientador, Rodolfo Azevedo, pelo direcionamento sempre sensato e preciso, pelo acompanhamento cuidadoso durante todo o desenvolvimento deste trabalho e pela total disposição em aceitar orientar um aluno sem dedicação exclusiva.

Ao Bruno Teles, pelo desenvolvimento de vários programas utilizados na análise de viabilidade e desempenho do modelo.

Ao Instituto de Computação da UNICAMP, pelo excelente curso de mestrado e pela infra-estrutura disponibilizada para o desenvolvimento deste trabalho.

À Motorola, por conceder-me tempo de dedicação ao mestrado.

À IBM, por ceder o console PlayStation 3 utilizado na implementação do projeto.

Dedicatória

Para Elaine, minha esposa, cuja paciência e apoio durante todo o mestrado foram essenciais para que eu o concluísse.

Para Mariana e Rafael, meus filhos, que souberam entender quando eu não podia dar-lhes toda minha atenção.

Para meus pais, que me proporcionaram ótima educação e incentivo permanente ao estudo.

Acrônimos

API: Application Programming Interface

Cell BE: Cell Broadband Engine

DLL: Dynamically Linked Library

DMA: Direct Memory Access

DSM: Distributed Shared Memory

DSP: Digital Signal Processor

EIB: Element Interconnect Bus

GC: Garbage Collector

GPU: Graphics Processing Unit

HPC: High Performance Computing

JIT: Just In Time

JNI: Java Native Interface

JVM: Java Virtual Machine

LS: Local Store

MFC: Memory Flow Controller

MPI: Message Passing Interface

POSIX: Portable Operating System Interface (for UniX)

PPE: Power Processor Element

RISC: Reduced Instruction Set Computer

RMI: Remote Method Invocation

RPC: Remote Procedure Call

SDK: Software Development Kit

SIMD: Single Instruction, Multiple Data

SPE: Synergistic Processing Element

VMX: Vector Multimedia eXtensions

XDR: eXtreme Data Rate

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

O processador Cell Broadband Engine (Cell BE) [17], é um processador com múltiplos núcleos heterogêneos, de alto desempenho, desenvolvido por Sony, Toshiba e IBM, empresas que formaram a aliança então chamada de STI. Ele está disponível comercialmente desde 2006 e é comumente lembrado por equipar o console de video game PlayStation 3 da Sony e o supercomputador Roadrunner da IBM. A existência de uma distribuição de Linux adaptada para o Cell BE completa o cenário que viabiliza um sistema extremamente poderoso para o desenvolvimento de aplicações, mas com a disponibilidade e o preço de um console de video game.

Java é certamente uma das linguagens mais populares em uso atualmente. Ela está presente em praticamente todas as plataformas e é utilizada nos mais variados tipos de aplicações. Mais recentemente, graças ao desenvolvimento de diversas tecnologias na área de máquinas virtuais, Java vem sendo usada inclusive no campo de alto desempenho. É uma linguagem normalmente elogiada por proporcionar alta produtividade e por liberar o programador de tarefas que facilmente induzem ao erro, como manipulação de apontadores e gerência explícita de memória.

Nada mais natural, então, que se busquem meios de executar programas Java em sistemas baseados no Cell BE, uma vez que o SDK da IBM para o Cell BE suporta o desenvolvimento de aplicações apenas em C, C++ e Fortran. Infelizmente, isso não é tão simples. Desde a concepção do Cell BE, seus projetistas adotaram uma abordagem agressivamente voltada para a obtenção de máximo desempenho. Um aspecto que foi bastante sacrificado em prol de um maior desempenho foi a facilidade de programação. Portanto, para se aproveitar ao máximo o poder de processamento do Cell BE, deve-se entender os detalhes da arquitetura do hardware e utilizar técnicas de programação de

baixo nível. Estas habilidades são normalmente estranhas aos programadores Java. Além disso, a própria linguagem não favorece o uso de técnicas de programação de baixo nível.

O presente trabalho propõe um modelo de execução para programas Java no Cell BE. Ao contrário de outas soluções publicadas anteriormente que procuram oferecer uma visão totalmente transparente do sistema, este trabalho expõe um modelo de memória explicitamente distribuída, dando maior liberdade e controle ao programador Java, o que lhe permite definir exatamente quais partes da aplicação executam em quais núcleos do Cell BE.

O restante desta dissertação é organizado conforme descrito a seguir. O capítulo 2 apresenta os principais conceitos e trabalhos relacionados com esta área de pesquisa, listando e classificando as alternativas existentes para execução de programas Java no Cell BE. No capítulo 3 são descritos em detalhe o novo modelo proposto e a implementação de uma máquina virtual Java que suporta esse modelo, especializada para o Cell BE. Esse capítulo se encerra com a discussão das limitações conhecidas do modelo e de sua implementação. O capítulo 4 demonstra a viabilidade do modelo, assim como apresenta os números de desempenho obtidos com vários programas usados nessa análise. Finalmente, o capítulo 5 conclui o texto, além de indicar alguns possíveis trabalhos futuros.

Capítulo 2

Conceitos Básicos e Trabalhos Relacionados

A principal motivação deste trabalho é proporcionar ao programador Java uma nova forma de explorar o poder de processamento dos múltiplos núcleos disponíveis no Cell BE. Este capítulo introduz vários conceitos e trabalhos relacionados aos tópicos mais relevantes da área, que são:

- O processador Cell BE (seção 2.1);
- A linguagem Java (seção 2.2);
- Sistemas distribuídos em Java (seção 2.3);
- Alternativas para execução de programas Java no Cell BE (seção 2.4).

2.1 O processador Cell BE

O processador Cell BE [17], desenvolvido pelo grupo de empresas STI (Sony, Toshiba e IBM), é um processador *multicore* de alto desempenho. Em sua primeira versão, ele conta com um núcleo principal chamado Power Processor Element (PPE) e oito co-processadores, os Synergistic Processing Elements (SPE). Portanto, ao contrário das implementações *multicore* tradicionais, os núcleos do Cell BE são heterogêneos.

A Figura 2.1 apresenta o diagrama de blocos do processador Cell BE.

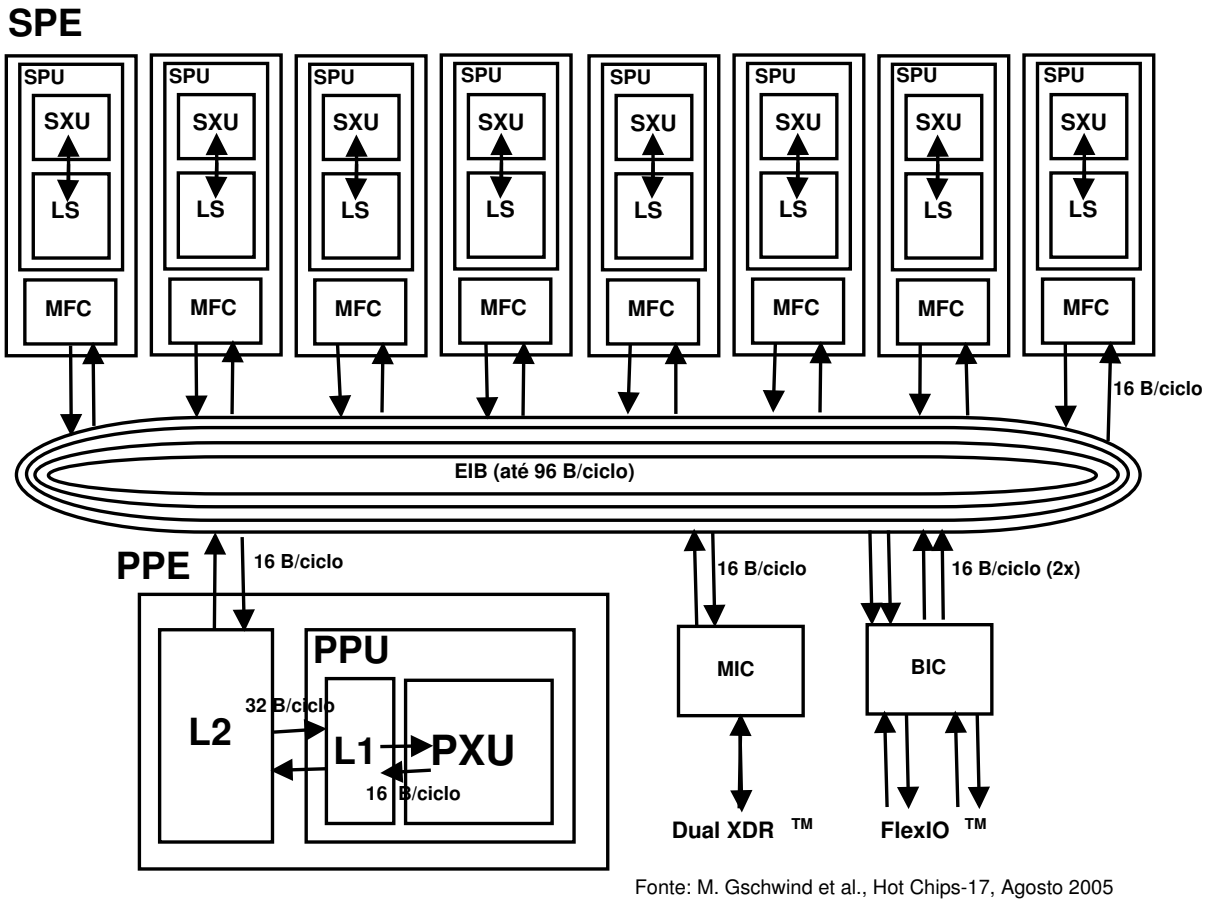

Figura 2.1: Diagrama de blocos do processador Cell BE

O PPE é uma versão simplificada de um processador PowerPC, com arquitetura compatível com a da família Power 970. Ele é um processador de 64 bits de propósito geral, destinado a executar o sistema operacional e a atuar como controlador dos SPEs.

Sua arquitetura é similar a dos primeiros processadores RISC, não dispondo, por exemplo, de suporte para execução fora de ordem. Simplificações desse tipo permitem uma diminuição significativa tanto na área utilizada pelo PPE quanto no consumo de energia. Por outro lado, o PPE conta com os seguintes recursos para aumentar seu desempenho: execução simultânea de duas instruções (*dual issue*), execução simultânea de dois *threads* e suporte para instruções vetoriais (VMX).

Cada núcleo SPE é um processador vetorial especializado. Seu conjunto de instruções é definido por uma nova arquitetura Single-Instruction Multiple-Data (SIMD). Por essa razão, ele é destinado, primordialmente, a computações matemáticas intensas sobre uma *stream* de dados, como tipicamente ocorre com DSPs e GPUs. No entanto, sua arquitetura é mais genérica que a destes últimos, permitindo-lhe executar os mais variados tipos de tarefas. Para a execução das instruções vetoriais, cada SPE dispõe de 128 registradores de 128 bits.

Talvez o aspecto mais notável na arquitetura do SPE seja a ausência de memória cache. No lugar da cache, o SPE conta com uma memória local própria, um tipo de *scratchpad memory*, chamada de Local Store (LS). Com 256 KB, a LS é independente da memória principal, não exigindo controle para manutenção de coerência, o que torna sua implementação extremamente simples quando comparada a das memórias cache convencionais. As transferências entre a LS e a memória principal são feitas através de DMA, sempre coordenadas pelo Memory Flow Controller (MFC), que faz parte do SPE. O MFC também oferece um serviço de *mailbox*, que permite ao SPE se comunicar com o PPE e outros SPEs através da troca de mensagens curtas (32 bytes).

O SPE, assim como o PPE, também é um processador *dual issue* e executa as instruções em ordem. Além do mais, ele não dispõe de *branch prediction*.

O PPE, os SPEs, a memória principal e os controladores de entrada/saída são interconectados pelo Element Interconnect Bus (EIB). O EIB é constituído por quatro anéis, sendo que dois deles transferem informação em um sentido e os outros dois no sentido contrário. Cada anel permite até três transferências simultâneas, o que proporciona uma taxa de transferência teórica máxima de 96 bytes por ciclo.

Dois pontos de grande ênfase na concepção e implementação do Cell BE foram alto desempenho e baixo consumo de energia. Outra forma de se resumir esses dois objetivos

é a busca por um aumento de eficiência por área utilizada de silício. Para esse fim foram fundamentais duas características que permeiam todo o projeto do processador:

- Simplificação do hardware, muitas vezes com soluções inovadoras, indo contra os paradigmas estabelecidos (por exemplo, a eliminação de memória cache nos SPEs);
- Uso de barramentos largos, com altas taxas de transferência, bem como das tecnologias mais rápidas disponíveis para memória principal (Rambus XDR) e sistema de entrada/saída (Rambus FlexIO).

A primeira versão do Cell BE é usada no PlayStation 3 da Sony. Isso significa que, por menos de US$300 nos EUA, qualquer pessoa pode adquirir uma máquina com tecnologia que, até pouco tempo atrás, era usada apenas em supercomputadores (processador vetorial com múltiplos núcleos heterogêneos).

O Software Development Kit para o Cell BE [16], ou simplesmente Cell BE SDK, suporta oficialmente apenas as linguagens C, C++ e Fortran. A ausência de Java nessa lista já indica uma maior preocupação, por parte dos arquitetos do Cell BE, com o desempenho do sistema do que com a produtividade do programador.

Para se tirar proveito do poder de processamento dos SPEs, o programador tipicamente deve:

- Carregar o programa no SPE;
- Iniciar o programa no SPE;
- Transferir dados da memória principal para a LS do SPE (e vice-versa) via DMA;
- Sincronizar a execução do programa no SPE com a do programa no PPE através da troca de mensagens via *mailbox*.

Além disso, deve-se ter a preocupação de alinhar com múltiplos de 128 bytes os endereços de áreas de memória usadas em operações de DMA.

Apesar de existirem recursos que facilitam alguns dos aspectos listados acima, em geral, a programação para o Cell BE demanda um bom entendimento da arquitetura de hardware e o emprego de técnicas de baixo nível.

Uma das motivações para se usar Java no Cell BE é justamente livrar o programador dessa complexidade. Java é uma linguagem de alto nível, que se caracteriza por oferecer classes que abstraem os mais variados recursos do sistema alvo. A máquina virtual Java

(JVM) é responsável por esconder as especificidades da plataforma de hardware, promovendo a portabilidade dos programas Java. Dessa forma, nada mais adequado do que embutir na JVM a programação de baixo nível exigida para a utilização dos SPEs no Cell BE.

2.2 A linguagem Java

Surgida em meados da década de 1990, a linguagem Java [14], originalmente destinada a pequenas aplicações para navegadores de internet, foi gradativamente abrindo espaço em praticamente todas as áreas onde há desenvolvimento de software. De aplicações cliente com ricas interfaces gráficas a programas comerciais e científicos executando em poderosos servidores, Java tornou-se uma das linguagens mais populares da atualidade [32].

Dentre as principais características que contribuíram para o sucesso de Java pode-se destacar:

- Portabilidade (o código fonte é compilado para bytecode, que é um formato binário independente da plataforma de hardware);
- Extensa coleção de bibliotecas e suas respectivas APIs, disponibilizadas junto com o SDK da linguagem;
- Realização automática de *garbage collection*, o que minimiza problemas relacionados a alocação e desalocação de memória;
- Suporte a programação *multithreaded*.

Uma das propriedades mais interessantes e úteis de Java é o suporte à programação *multithreaded*. Ao contrário de outras linguagens populares como C e C++, que dependem de bibliotecas separadas para a criação e sincronização de *threads*, em Java essas funcionalidades existem na própria linguagem, sendo expostas via palavras reservadas e classes da biblioteca padrão. Os dois elementos de Java fundamentais à programação *multithreaded* são a classe `Thread` e a palavra reservada `synchronized`.

Para se criar um *thread* em Java deve-se primeiro criar uma instância da classe `Thread` ou de uma classe derivada desta, da mesma forma como se cria um outro objeto qualquer. Para que o *thread* inicie sua execução, deve-se invocar o método `start` desse objeto, definido na classe `Thread`. O método `start`, por sua vez, invoca o método `run` de um

objeto que implementa a interface `Runnable`. Exatamente qual objeto é esse depende da estratégia de implementação usada, conforme explicado a seguir. O corpo de `run` é o código do *thread* propriamente dito.

A interface `Runnable` é implementada pela própria classe `Thread`. Dessa forma, a maneira mais simples de se obter um *thread* é derivar uma nova classe de `Thread` e implementar o método `run` nessa nova classe. Nesse caso, `start` executa o método `run` definido no próprio objeto *thread*. Uma maneira mais flexível, entretanto, é implementar a interface `Runnable` em uma classe qualquer. Essa estratégia permite que a nova classe derive de uma outra classe arbitrária e não necessariamente de `Thread`. Isso é importante porque Java não permite herança múltipla. Finalmente, nessa maneira alternativa deve-se criar uma instância de `Thread`, passando um objeto da classe que implementa `Runnable` como parâmetro do construtor. Quando `start` é invocado no objeto *thread*, ele chama o método `run` do objeto recebido como parâmetro no construtor.

A exclusão mútua na execução de regiões críticas por *threads* concorrentes em Java se dá através do uso da palavra reservada `synchronized`. A JVM garante que todo código declarado como `synchronized` é executado por apenas um *thread* de cada vez. Esta palavra reservada pode ser usada em dois contextos: tanto um método inteiro quanto um bloco arbitrário de código podem ser declarados `synchronized`.

Todo objeto possui um *lock* associado, que pode ser adquirido e liberado apenas indiretamente. Por exemplo, uma maneira de se fazer isso é através do uso de código `synchronized` (o *lock* apropriado é automaticamente adquirido quando o *thread* entra no trecho `synchronized` e liberado quando o *thread* sai do mesmo). A diferença principal entre as duas formas de `synchronized` é o objeto alvo cujo *lock* deve ser obtido. Quando um método `synchronized` é invocado, o *lock* obtido é o do próprio objeto cujo método vai ser executado. No caso de um bloco de código `synchronized`, um objeto deve ser explicitamente indicado (passado como parâmetro) e é o *lock* deste objeto que deve ser obtido. A segunda forma é a mais geral. A declaração de métodos `synchronized` é simplesmente uma conveniência de programação. O mesmo efeito pode ser obtido envolvendo-se o corpo do método em questão em um bloco `synchronized` e passando-se `this` como argumento.

Completando as facilidades básicas para sincronizar *threads*, a classe `Object` implementa alguns métodos que permitem a comunicação entre *threads*. Por ser definidos em `Object`, eles são herdados por todos os objetos. Para suspender sua execução, um *thread* deve invocar `wait`. Esta chamada tem como efeito colateral a liberação do *lock* do objeto associado, uma vez que `wait` deve ser chamado por código `synchronized`. Em

contrapartida, a invocação de `notify` faz com que um *thread* suspenso por `wait` desperte e readquira o *lock* do objeto associado automaticamente. Para despertar todos os *threads* suspensos por `wait` utiliza-se `notifyAll`.

2.2.1 A máquina virtual Java

A máquina virtual Java, ou simplesmente JVM, como é comumente chamada pela sua sigla em inglês, é responsável pela execução dos programas Java. Ela constitui a peça chave para a independência de plataforma de hardware, característica fundamental de Java.

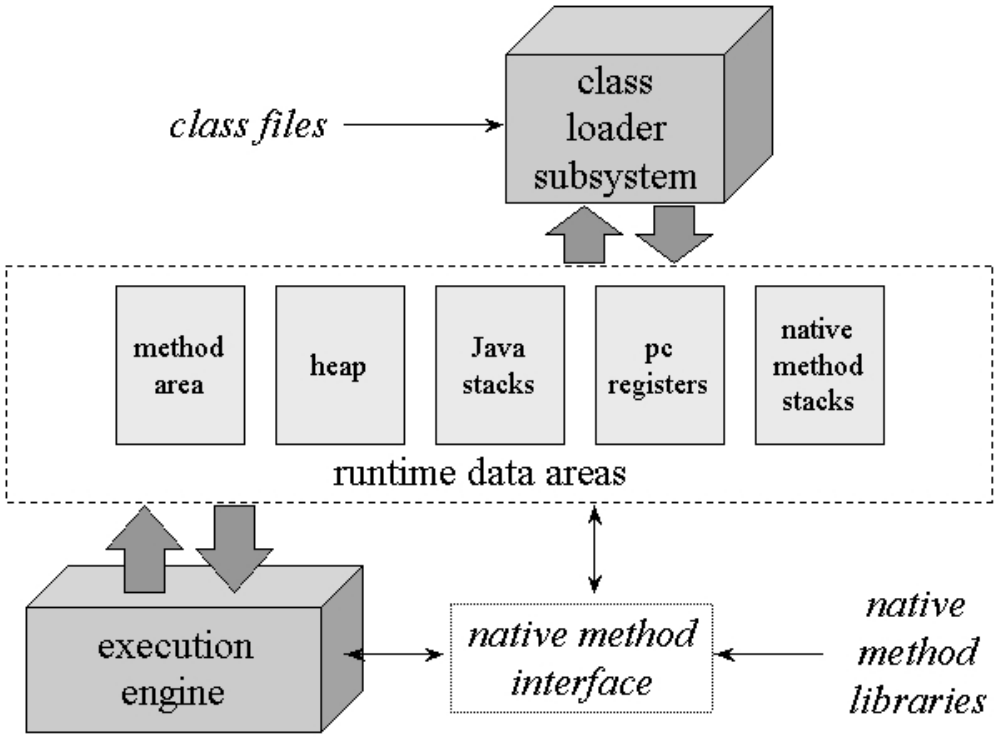

Figura 2.2: Arquitetura da JVM

Um esquema da arquitetura da JVM [34] é mostrado na Figura 2.2.

O subsistema de carregamento de classes (*class loader subsystem*) é responsável por localizar e carregar as classes necessárias à execução do programa. As classes são arma-

zenadas em arquivos com extensão ".class", que contêm a representação binária, isto é, o bytecode, dessas classes. Além de ler o bytecode a partir do sistema de arquivos, esse subsistema também faz o link e a inicialização das classes. A fase de link, por sua vez, constitui-se da verificação do bytecode e alocação de memória para as variáveis estáticas. Já a inicialização consiste em invocar o código Java que inicializa essas variáveis estáticas.

Na parte central da Figura 2.2 aparecem as áreas de dados usadas pela JVM em tempo de execução (*runtime data areas*). A área de métodos (*method area*) e o heap são compartilhados por todos os *threads* na JVM. Por outro lado, cada *thread* possui uma pilha Java (*Java stack*) e um registrador de contador de programa (*PC register*). Se um *thread* invocar um método nativo, ele também possuirá uma pilha de método nativo (*native method stack*).

A área de métodos contém a representação interna de todas as classes carregadas até o momento. Para cada classe (ou tipo), essa área armazena informações como:

- Nome completo do tipo (por exemplo, `java.lang.Object`);
- Nome completo da superclasse direta;
- Se o tipo é classe ou interface;
- Os modificadores do tipo (por exemplo, `public`, `abstract` e `final`);
- Uma lista ordenada dos nomes completos das superinterfaces diretas;
- O *constant pool*, que contém todas as constantes usadas pela classe, sejam literais ou referências simbólicas para tipos, campos e métodos;
- Lista de campos da classe, incluindo nome, tipo e modificadores;
- Lista de métodos da classe, incluindo nome, tipo de retorno, número e tipo de cada parâmetro, modificadores, o código (bytecode) do método, tamanho da pilha de operandos, tamanho da área ocupada pelas variáveis locais, tabela de exceções;
- As variáveis de classe (ou estáticas).

Enquanto a área de métodos contém os dados que descrevem as classes, o heap é a área onde as instâncias das classes, os objetos, são criados. Os seguintes elementos fazem parte, tipicamente, da representação de cada objeto no heap:

- Um apontador para o início da região, dentro da área de métodos, que descreve a classe do objeto;
- As variáveis (campos) da instância;
- Uma tabela de métodos, que viabiliza o acesso rápido a todos os métodos que podem ser invocados nesse objeto, inclusive os herdados das superclasses;
- Uma área de *lock*, usada para garantir acesso exclusivo ao objeto na presença de múltiplos *threads*.

Cada *thread* tem sua própria pilha Java, que é a área usada para a execução propriamente dita dos métodos invocados pelo *thread*. Sempre que um método vai ser invocado, um novo *stack frame* é colocado na pilha Java. O *stack frame* contém a pilha de operandos e a área de variáveis locais do método associado.

A JVM é uma máquina baseada em pilha. Por esse motivo, a pilha de operandos é a principal área manipulada pelas instruções do bytecode Java. Por exemplo, para somar duas variáveis locais, seus valores devem ser primeiro colocados na pilha para que sejam somados.

Cada argumento passado a um método em sua invocação é associado com um elemento da área de variáveis locais. Além disso, como o próprio nome diz, as variáveis locais usadas no código Java fonte do método são alocadas na área de variáveis locais do *stack frame*.

O único registrador existente na JVM é usado para armazenar o contador de programa, que contém o endereço da próxima instrução do bytecode a ser executada. Se o *thread* estiver executando um método nativo, o conteúdo desse registrador é indefinido.

A pilha de métodos nativos tem função similar à pilha Java, mas destina-se à execução de métodos nativos.

Finalmente, na parte inferior da Figura 2.2 estão o *execution engine* e a interface com métodos nativos. O *execution engine* é responsável pela execução do bytecode, e pode ser um interpretador puro ou utilizar alguma forma de compilação JIT. A interface com métodos nativos permite a invocação destes e também o acesso a várias funcionalidades da JVM a partir dos métodos nativos.

2.3 Sistemas distribuídos em Java

Como os SPEs têm acesso à memória principal, pode-se argumentar que os núcleos do Cell BE constituem um sistema de memória compartilhada. Entretanto, a verdadeira memória dos SPEs é a LS. Quando um SPE executa instruções de *load* e *store*, é a LS que está sendo lida e escrita. O acesso à memória principal é realizado apenas através de transferências DMA, que devem ser explicitamente programadas pelo usuário. Portanto, é mais adequado classificar o Cell BE como um sistema de memória distribuída. Isso é consistente com o uso típico feito desse processador, onde cada SPE recebe um lote de dados, trabalha sobre esse lote localmente em sua LS e depois devolve o resultado do processamento ao PPE ou passa o lote processado para um outro SPE. Essa forma de computação é muito semelhante à que ocorre em um sistema distribuído tradicional em rede, sendo que o PPE e os SPEs desempenham o papel dos nós da rede.

2.3.1 RMI

A literatura técnica é rica em trabalhos que abordam o tema de sistemas distribuídos em Java. Muitos desses trabalhos propõe melhorias sobre o sistema padrão oferecido por Java para o uso de objetos distribuídos, o Remote Method Invocation (RMI). RMI [36] foi introduzido com o JDK 1.1 e sua proposta foi levar a funcionalidade do RPC para o mundo de Java e seus objetos. Mais do que simplesmente proporcionar uma maneira de invocar os métodos de um objeto remoto, seus autores tiveram a preocupação de definir um modelo completo para objetos distribuídos em Java.

Para expor métodos remotamente usando RMI, um objeto deve implementar uma interface (como definido na linguagem Java) remota. Uma interface é dita remota se estender a interface padrão `Remote`. Para usar métodos remotos, um cliente RMI deve primeiro obter uma referência de um objeto remoto e então invocar quaisquer dos métodos publicados em uma de suas interfaces remotas. Além disso, como todo método remoto pode lançar exceções do tipo `RemoteException`, que sinalizam algum problema de comunicação entre cliente e servidor, o chamador é obrigado a lidar com esse tipo de erro.

Quando um cliente usa uma referência para um objeto remoto, na verdade essa referência aponta para um objeto local chamado *stub* ou *proxy*. Esse objeto *stub* implementa todos os métodos das interfaces remotas suportadas pelo objeto remoto alvo. Quando um desses métodos do objeto *stub* é invocado, a implementação local simplesmente dispara, através do protocolo de comunicação entre cliente e servidor, a execução do método cor-

respondente no objeto remoto.

RMI preserva muito bem a sintaxe e a semântica de uso de objetos locais, ao aplicá-las aos objetos remotos. A declaração explícita de interfaces remotas e a necessidade de se tratar os erros de comunicação via exceções são diferenças bem vindas, pois marcam de forma inequívoca a diferença entre invocações de métodos locais e remotos.

Outras características positivas do RMI são:

- O bytecode das classes dos objetos *stub*, dos objetos passados como parâmetro e dos objetos devolvidos como resultado de invocações remotas é carregado dinamicamente. A origem do bytecode pode ser inclusive uma máquina não envolvida na invocação remota, como, por exemplo, um servidor web.

- O carregamento de bytecode oriundo de fontes externas à JVM do cliente só é permitido se um objeto `SecurityManager` tiver sido instalado previamente, o que força o emprego de uma política de segurança.

- O *garbage collector* funciona de modo distribuído, de forma a garantir que um objeto seja removido somente se não houver nenhuma referência, local ou remota, apontando para ele.

Uma diferença semântica importante de RMI com relação à invocação de métodos locais é que, quando um objeto não remoto é passado como parâmetro de um método remoto, esse objeto é copiado. Isso significa que quaisquer alterações que o método remoto faça no objeto recebido não terão efeito no objeto original passado como parâmetro. Em chamadas locais, ao contrário, objetos nunca são copiados, mas sim as referências usadas para acessá-los. Dessa forma, mudanças feitas em um objeto recebido como parâmetro são imediatamente vistas pelo chamador, pois todo código Java sempre trabalha com referências para objetos e não com os objetos em si. Em RMI, um objeto devolvido como resultado da invocação de um método remoto também é copiado, desta vez na direção inversa.

A cópia de um objeto de uma JVM para outra consiste na serialização, transmissão e desserialização desse objeto. Para que um objeto seja automaticamente serializado e desserializado em RMI, sua classe deve necessariamente implementar a interface `Serializable`. Para isso, basta incluí-la na lista de interfaces implementadas pela classe, não sendo necessário codificar realmente nenhum método, já que `Serializable` é uma interface do tipo "marker" (não define métodos). O algoritmo de serialização utiliza reflexão para

serializar objetos, uma vez que estes podem ter uma estrutura arbitrariamente complexa, consistindo, no caso mais geral, de um grafo de objetos.

O grande problema de RMI, especialmente para aplicações da área de High Performance Computing (HPC), é seu fraco desempenho. [23] e [25] descrevem uma série de otimizações implementadas na Universidade de Karlsruhe para melhorar a eficiência do RMI. A versão otimizada economizou 45% (mediana) do tempo de execução do RMI original, em uma configuração com PCs conectados via Ethernet. Algumas das melhorias implementadas foram:

- Representação mais compacta do tipo de um objeto serializado;
- Manutenção em memória da informação sobre tipos transmitidos (o RMI original elimina e gera novamente essa informação a cada invocação de um método remoto);
- Melhor utilização de buffers na serialização;
- Redução drástica do número de objetos criados durante uma invocação remota;
- Redução do uso de métodos nativos e de reflexão;
- Uso de chamadas locais quando objetos alvo residem na JVM do cliente (o RMI original não detecta quando objetos locais são usados como alvo de invocações de métodos definidos em interfaces remotas e, portanto, tudo se passa como no caso geral: uso de objeto *stub*, serialização, transmissão via socket).

Manta [20] é um sistema que usa compilação de Java para código nativo com o intuito de oferecer um RMI eficiente. Esse sistema destina-se principalmente à programação paralela, o que explica o foco na comunicação rápida entre os programas Java distribuídos pelas máquinas de um *cluster*. O *overhead* da serialização é quase todo transferido para o tempo de compilação, quando são gerados serializadores especializados para cada classe cujos objetos podem ser transferidos entre máquinas distintas. A interface do RMI é mantida, mas o protocolo de comunicação é substituído por uma versão otimizada para desempenho. Além disso, o RMI original completo (incluindo o protocolo) também é suportado de modo a permitir a interoperabilidade com outras JVMs.

Apesar do projeto do RMI original da Sun indicar uma preocupação com generalidade e flexibilidade em muitos aspectos, paradoxalmente isso não ocorre na definição da camada de transporte que, necessariamente, precisa ser baseada em sockets. Pior, como apontado

em [23], o uso de sockets é exposto à camada de aplicação através da API. Uma outra contribuição de [23] e [25] foi justamente implementar um sistema RMI cuja camada de transporte é independente da tecnologia de comunicação entre os nós da rede.

Além do fraco desempenho, outras limitações do RMI frequentemente citadas são:

- Não é possível acessar diretamente as variáveis de objetos remotos, apenas os métodos declarados em sua interface remota. Para permitir acesso a variáveis é necessário explicitamente adicionar métodos com essa finalidade à interface remota.
- Não há suporte a sincronização distribuída. Por exemplo, métodos como `wait()` e `notify()` não têm efeito sobre os objetos remotos, mas sim sobre os *stubs* locais.

[31] destaca-se dentre os vários trabalhos que implementam sincronização distribuída por propor uma solução que adiciona baixo *overhead* e que não exige alterações no *middleware* (RMI). Segundo os autores desse artigo, o RMI de Java apresenta dois problemas nessa área: não propaga operações de sincronização aos objetos remotos e não mantém a identidade do *thread* que executa operações remotas, de uma máquina para outra. O segundo problema acontece porque um novo *thread* é lançado na máquina onde a invocação remota é recebida e este não é ciente do *thread* que iniciou a chamada na máquina de origem. Na prática dois *threads* são envolvidos na execução do método remoto (um na máquina que origina a chamada e outro na máquina que recebe e executa o método), apesar de, logicamente, existir apenas um. Para uma correta sincronização é essencial que esses dois *threads* sejam tratados como o mesmo. A solução apresentada em [31] consiste em alterar o bytecode de operações relacionadas a sincronização. Quando tais operações são invocadas em objetos *stub* de RMI, elas são propagadas via RMI para o objeto remoto. O bytecode das classes *stub* também é alterado de forma a passar um parâmetro a mais em todas as invocações remotas. Tal parâmetro contém um identificador que é único para todos os *threads* que devem ser tratados como o mesmo.

2.3.2 DSM

Distributed Shared Memory (DSM) é um tema recorrente nos trabalhos relacionados a sistemas distribuídos em Java. O objetivo de um sistema DSM é oferecer ao programador um ambiente de memória compartilhada em uma plataforma de memória distribuída. Com um sistema DSM para Java, o programador pode usar o bem conhecido paradigma de programação *multithreaded* de Java, que supõe memória compartilhada, em um ambiente

de memória distribuída, com pouca ou nenhuma alteração. [13] apresenta uma análise detalhada de sistemas DSM para Java, onde a plataforma de memória distribuída é um *cluster* de máquinas ligadas em rede.

A análise de vários trabalhos na área de sistemas distribuídos em Java indica uma preferência por soluções do tipo DSM, uma vez que a possibilidade de se executar em ambientes distribuídos programas Java *multithreaded* pré-existentes é muito atraente. Cada trabalho, em geral, tem seu foco voltado para algum quesito específico, não existindo uma solução que seja a melhor ou a mais adequada para todo tipo de problema. Os sistemas propostos normalmente oscilam entre duas características difíceis de se conciliar: portabilidade e desempenho.

As soluções mais portáveis são aquelas que não modificam a JVM e, portanto, podem ser aplicadas em JVMs padrão. Os trabalhos nessa linha normalmente introduzem bibliotecas Java adicionais, o que exige um modelo de programação diferente do *multithreaded* tradicional[1], e são baseados em RMI, o que tende a prejudicar o desempenho. JDSM [27] e ProActive [6] são exemplos desse tipo de solução.

Um sistema distribuído para Java que não modifica a JVM e é frequentemente citado é JavaParty [26]. Nesse sistema a única alteração necessária em programas *multithreaded* é o uso do modificador `remote` nas classes cujos objetos devem ser acessados remotamente. JavaParty pode ser descrito como uma camada adicional sobre o RMI. O código Java original passa por um pré-processador que gera uma série de classes adicionais para cada classe marcada com `remote`. Essas classes encapsulam o uso de RMI, o que é bastante conveniente para o programador. A localização dos objetos remotos é definida e ajustada automaticamente pelo sistema de forma a obter uma distribuição balanceada. Quando um objeto de uma classe `remote` está na JVM local, isso é detectado e o objeto é usado localmente. JavaParty usa migração de objetos para aumentar explicitamente a "localidade" dos mesmos. Para melhorar o desempenho, a versão mais recente do sistema usa uma implementação muito mais eficiente de RMI, conservando a API original.

Um dos trabalhos mais interessantes sobre Java distribuído, com proposta de total transparência e portabilidade, é JavaSplit [12]. Qualquer JVM convencional pode ser usada (portabilidade) e os programas Java *multithreaded* não precisam nenhuma alteração (transparência), nem mesmo o uso de novas bibliotecas. Além disso, JavaSplit não emprega RMI como meio de comunicação entre as JVMs. A solução é baseada em instrumentação de bytecode. O bytecode de operações relevantes para a distribuição de objetos

[1]Portabilidade aqui refere-se à solução (JVM) e não aos programas Java do usuário.

é alterado/aumentado para fazer uso de um DSM escrito em Java puro. Isso significa que o sistema DSM é embutido no programa original, através da modificação do bytecode. Como o resultado final também é bytecode, qualquer JVM pode executá-lo.

Já as soluções que procuram favorecer mais o desempenho em detrimento da portabilidade, podem ser classificadas em dois grupos. O primeiro implementa uma JVM distribuída, o que tipicamente envolve o emprego de uma JVM especializada em cada nó participante do *cluster*. [5] e [4] apresentam uma solução desse tipo. O segundo grupo utiliza compilação para código nativo, ao mesmo tempo em que agrega funcionalidade DSM. Exemplos de sistemas que adotam essa abordagem são Jackal [33] e Hyperion [3].

Os artigos voltados especificamente para o uso de Java em aplicações do tipo HPC destacam-se por terem o padrão Message Passing Interface (MPI) como critério de comparação e como inspiração para sistemas distribuídos em Java. Isso se justifica pela liderança de MPI no segmento de aplicações HPC, tipicamente escritas em Fortran, C ou C++. Em [9] os autores descartam totalmente a possibilidade de se usar RMI para conectar múltiplas JVMs e recomendam a utilização de alguma implementação de MPI para Java. Já em [2], existe uma ênfase em produtividade e programação orientada a objetos, seguindo uma tendência mais recente em HPC. Por essa razão, a solução analisada utiliza ProActive, que, como visto anteriormente, não é baseada em MPI, mas sim em bibliotecas Java adicionais. Para melhorar o desempenho de RMI, do qual ProActive depende, é sugerida a substituição do protocolo de transporte original por um mais eficiente, que utiliza Java Fast Sockets [30].

2.4 Alternativas para execução de programas Java no Cell BE

Em processadores *multicore* tradicionais (núcleos homogêneos, memória principal diretamente acessível por todos os núcleos, cache local controlado por hardware), o sistema operacional é responsável por distribuir automaticamente os *threads* do programa entre os processadores. Com isso, basta que o programa Java seja *multithreaded* e a JVM use *threads* nativos (abordagem segundo a qual os *threads* Java são mapeados diretamente em *threads* do sistema operacional) para usar os múltiplos núcleos existentes. Não é necessário usar um sistema DSM, uma vez que a memória já é compartilhada nestas arquiteturas.

No caso de processadores *multicore* com memória distribuída, como o Cell BE, faz

até mais sentido usar um modelo DSM transparente do que em um *cluster* de máquinas ligadas em rede, já que a comunicação entre os núcleos é muito mais rápida do que entre os nós da rede. Por outro lado, como os núcleos são diferentes entre si e especializados para determinados tipos de aplicações, a alocação de *threads* aos núcleos deve levar isso em consideração para obter um melhor desempenho. Ainda assim, o apelo da abordagem DSM permanece forte, tanto que todos os trabalhos sobre Java no Cell BE publicados até o presente e discutidos a seguir, adotam tal estratégia.

Como o PPE é um processador de propósito geral da família PowerPC, basta instalar qualquer JVM compatível com PowerPC em uma máquina Cell BE para poder executar aplicações Java em tal máquina. Entretanto, essa solução constituiria um tremendo desperdício de recursos, uma vez que os SPEs não seriam utilizados. Mesmo que o código executado nos SPEs não seja o mais adequado para sua arquitetura, o simples fato de se utilizar os SPEs e o PPE simultaneamente tem o potencial de gerar ganhos significativos de desempenho, graças à execução paralela. Além disso, a solução ingênua de se utilizar uma JVM comum para PowerPC no Cell BE apresentaria um desempenho pior do que aquele obtido em uma máquina PowerPC convencional de especificações similares ao PPE. Isso se deve ao fato do PPE ser um processador PowerPC simplificado, como visto na seção 2.1.

O desafio é, então, buscar alternativas que permitam executar programas Java no Cell BE, mas que aproveitem o poder de processamento dos SPEs. Uma forma de classificar as soluções para esse desafio baseia-se no número de JVMs utilizadas. As duas categorias segundo esse critério são: JVM única e múltiplas JVMs.

2.4.1 JVM única

Neste tipo de solução há apenas uma JVM que engloba inteiramente o Cell BE. Os principais componentes da máquina virtual executam no PPE e despacham parte de seu trabalho para os SPEs, conforme a estratégia específica adotada. A escolha de qual código transferir aos SPEs para execução é o que define uma solução particular. Existem dois tipos de tarefas que podem ser executadas pelos SPEs: tarefas internas da JVM e código do usuário.

Garbage collection é um exemplo de uma tarefa interna da JVM que pode executar separadamente em um SPE. [11] apresenta um *garbage collector* do tipo *mark-and-sweep* para o Cell BE, no qual a fase de marcação é feita pelo SPE. Neste caso o ganho de

desempenho vem do fato de que o PPE é aliviado da maior parte do trabalho de *garbage collection*, o que tende a diminuir o tempo total de execução da aplicação.

Teoricamente, compilação JIT também pode ser transferida para o SPE. Esta estratégia consiste em compilar de antemão nos SPEs a maior quantidade possível do código do programa Java e armazenar o código compilado na memória principal. O objetivo é ter disponível o código compilado de cada método Java no momento de sua execução. Dessa forma, elimina-se a espera introduzida pela compilação JIT do tempo total de execução.

Uma abordagem diferente é executar partes do programa Java do usuário nos SPEs, como descrito em [28]. O protótipo desenvolvido como parte desse trabalho é capaz de gerar código nativo para o SPE, através de compilação JIT. Tal código é então executado em um SPE. Apesar de algumas opções serem sugeridas, [28] não é conclusivo com relação à estratégia para selecionar os métodos que são executados nos SPEs. Além disso, esse trabalho não inclui resultados relativos a tempos de execução ou à potencial melhora de desempenho.

Uma outra solução que se caracteriza por executar código do usuário nos SPEs é Hera-JVM [22]. Como definido por seus autores, é um sistema que abstrai a heterogeneidade da arquitetura do Cell BE apresentando a ilusão de uma máquina virtual *multithreaded* homogênea. Baseada na JikesRVM [1], ela oferece migração automática de *threads* do PPE para o SPE. Todos os métodos são compilados via JIT antes de ser executados. Para resolver o problema de pouca disponibilidade de memória local no SPE, ambos dados e código são acessados através de um cache na LS implementado em software. Métodos nativos são sempre executados no PPE. Se a intenção é executar no Cell BE aplicações Java *multithreaded* sem alterá-las, Hera-JVM parece bem posicionada para atingir tal objetivo, como indicado pelos resultados preliminares de desempenho apresentados em [22]. Entretanto, atualmente esse sistema não permite ao usuário controlar precisamente qual código é executado nos SPEs.

2.4.2 Múltiplas JVMs

Nesta abordagem, a JVM principal executa no PPE, enquanto JVMs auxiliares executam separadamente em cada SPE. Dadas as limitações do ambiente de execução do SPE (pouca memória, inexistência de suporte a *threads*), a JVM deste deve ser o mais simples e compacta quanto possível. A estratégia deste tipo de solução é executar partes do programa do usuário computacionalmente intensivas nas JVMs dos SPEs. O modelo de

execução proposto nesta dissertação adota esta alternativa.

[24] apresenta a CellVM, onde cada SPE hospeda uma CoreVM e o PPE a ShellVM. Enquanto a CoreVM é basicamente um interpretador, a ShellVM auxilia a CoreVM com tarefas como a execução de instruções complexas do bytecode não suportadas pela CoreVM, *lock/unlock* de objetos e invocação de métodos nativos. A CellVM atribui automaticamente um SPE a cada *thread* no programa Java do usuário. Portanto, este possui controle limitado sobre o que é executado nos SPEs. A decisão de reduzir a JVM que executa nos SPEs a um mero interpretador de bytecode tem a vantagem de deixar mais memória disponível para os dados, uma vez que o tamanho da JVM é reduzido. Contudo, o custo disso é uma maior complexidade e um menor desempenho. Como a ShellVM controla estruturas de dados intensamente usadas como o Java heap, o *constant pool* e a área de métodos, a CoreVM é frequentemente forçada a comunicar-se com a ShellVM para solicitar assistência, o que produz um impacto significativo no desempenho. Por outro lado, com o intuito de minimizar estas transferências de dados e controle entre a CoreVM e a ShellVM, uma série de modificações foram feitas no código da JVM original, como a introdução de caches controlados por software tanto para dados quanto para instruções, preparação de código e re-escrita de opcodes. Estas alterações implicaram no aumento tanto do tamanho como da complexidade do código. Além disso, a versão da CellVM descrita em [24] sofria de incoerência de memória nos caches implementados nos SPEs.

Mais recentemente, [35] apresenta uma análise de várias otimizações aplicadas ao interpretador da CoreVM nos SPEs. As melhorias implementadas incluem tanto técnicas comumente usadas em interpretadores como novas otimizações que exploram as características de hardware específicas do SPE. Algumas delas são apresentadas a seguir:

- *Pipelining* do interpretador, que consiste em realizar o *fetch* do endereço da próxima instrução Java a ser executada durante a execução da instrução Java atual. Com isso, a latência do *load* do endereço da instrução Java é diluída durante a execução.

- *Caching* do topo da pilha Java, que consiste em armazenar um ou mais elementos do topo da pilha em registradores, evitando acessos à memória.

- Uso de super-instruções Java, que consiste em criar novas instruções Java a partir da concatenação de duas instruções Java tipicamente executadas consecutivamente. Super-instruções têm código maior que as instruções Java normais, o que aumenta as chances de uso da instrução de máquina *hint* do SPE, que é o único mecanismo existente no SPE para *branch prediction*. Para evitar um *stall* do pipeline, o *hint*

Solução	JVMs	SPE executa	JIT	Modelo
Garbage collection	Única	Tarefa interna	N/A	N/A
Compilação JIT	Única	Tarefa interna	N/A	N/A
Georg Sorst	Única	Código do usuário	Sim	DSM
Hera-JVM	Única	Código do usuário	Sim	DSM
CellVM	Múltiplas	Código do usuário	Não	DSM

Tabela 2.1: Soluções para Java no Cell BE.

deve ocorrer aproximadamente 15 ciclos antes do *branch*. Como ao final de cada instrução Java executada pelo interpretador da CoreVM ocorre um desvio para o endereço da próxima instrução Java, é necessário que o código da instrução Java seja suficientemente longo para que o *hint* possa ser emitido com a antecedência devida.

- *Pipelining* de operandos, que usa a mesma ideia do *fetch* antecipado, só que desta vez para os operandos da instrução Java, diluindo a latência dos *loads* desses operandos ao longo da execução da instrução anterior.

- Uso de *branch predictors* nas instruções Java de desvio que computam o endereço alvo a partir de seus operandos. Isto é feito, mais uma vez, através da instrução *hint*.

As otimizações que apresentaram os resultados mais consistentes e significativos são as de *pipelining* do interpretador e de operandos. Contudo, o artigo não apresenta a média de melhora de desempenho obtida e nem uma discussão detalhada do efeito de se utilizar diversas combinações das múltiplas otimizações analisadas. Fica claro que algumas dessas técnicas são conflitantes e, portanto, não compensa empregá-las simultaneamente.

A Tabela 2.1 apresenta um resumo das soluções para execução de Java no Cell BE descritas na seção 2.4. Como se pode notar, as características de maior predominância são o emprego de uma JVM única, a execução de código do usuário nos SPEs e a adoção do modelo de memória DSM.

Este capítulo apresentou os principais conceitos e trabalhos relacionados aos temas centrais desta dissertação: o processador Cell BE, a linguagem Java e a JVM, sistemas distribuídos em Java (com ênfase em RMI e DSM) e as alternativas existentes para execução de programas Java no Cell BE. O próximo capítulo detalha o modelo de execução de uma nova solução para Java no Cell BE. Sua implementação é estruturalmente simi-

lar à da CellVM, mas a abordagem é muito distinta, uma vez que não se baseia em um modelo DSM.

Capítulo 3

Um Novo Modelo de Execução para Java no Cell BE

Como exposto no capítulo anterior, todas as alternativas para se executar Java no Cell BE de que se tem conhecimento adotam uma solução do tipo DSM. Isso significa que elas procuram esconder do programador Java a existência dos SPEs. Se por um lado essa abordagem tem a vantagem de permitir a execução de programas *multithreaded* pré-existentes sem alterações, ela dificulta ou mesmo impossibilita uma utilização ótima dos SPEs. Essa limitação se deve ao fato de que em um sistema DSM a alocação de código aos SPEs deve ser feita automaticamente, sem a participação do programador.

O modelo de execução apresentado nesta dissertação busca proporcionar maior controle ao programador, permitindo-lhe definir exatamente qual código será executado nos SPEs. Essa característica tem o potencial de promover um melhor uso dos SPEs do que em sistemas DSM. Pelo próprio fato de proporcionar acesso direto aos SPEs, a abordagem adotada por este modelo é a de memória explicitamente distribuída. Dessa forma, não se procura criar uma ilusão de memória compartilhada.

Conforme mencionado na seção 2.4.2, esta dissertação propõe um ambiente de execução Java para o Cell BE que tira proveito do poder de processamento dos SPEs iniciando uma JVM em cada um deles. As JVMs dos SPEs executam tarefas Java criadas e iniciadas pelo programa Java do usuário que, por sua vez, executa na JVM do PPE. O programador é responsável por definir o conteúdo dessas tarefas, o que determina o código que vai ser executado nos SPEs e os dados sobre os quais esse código vai agir. A definição das tarefas é feita em Java puro, não exigindo a introdução de nenhum elemento novo na linguagem.

3.1 O Modelo de Execução

A JVM do PPE é uma JVM tradicional e completa. O programa do usuário é submetido a esta JVM que, como esperado, começa executando esse programa pelo método `main` da classe passada como argumento na linha de comando. Por outro lado, a JVM do SPE é muito especial, uma vez que ela executa apenas as tarefas Java solicitadas pela JVM do PPE.

Para executar código Java em um SPE, o programador deve completar os seguintes passos:

1. Definir uma classe *task* (tarefa) que deve implementar a interface `SPETask`;
2. Criar um objeto da classe definida no passo anterior e configurá-lo com todos os dados de entrada exigidos pela classe *task* (esses são os dados necessários para a execução da tarefa);
3. Iniciar a execução da tarefa em um SPE através da invocação de `SPETaskDispatcher.executeTask(taskObject)`, onde `taskObject` é o objeto criado no passo anterior.

A interface `SPETask` é a seguinte:

```
public interface SPETask<T> {
    T execute();
}
```

Como pode ser visto nesta interface extremamente simples, o único requisito imposto pelo modelo a uma tarefa para o SPE é que esta deve implementar o método `execute`. O corpo deste método é o código que vai ser executado em um SPE. Ele pode invocar quaisquer outros métodos Java, se for preciso. Apesar do modelo não proibir que a tarefa executada no SPE invoque qualquer código Java arbitrariamente complexo, o ambiente de execução do SPE, que é bastante limitado em termos de recursos, impõe restrições significativas, como será detalhado mais adiante.

`SPETask` é uma interface parametrizada. Seu parâmetro define o tipo de saída do método `execute` da classe tarefa ou, em outras palavras, o tipo do objeto produzido como resultado da execução da tarefa.

`SPETaskDispatcher` é uma classe Java disponibilizada junto com a JVM do PPE para facilitar a ativação de tarefas no SPE. Essa classe é composta por um único método, `executeTask`, como mostrado a seguir:

```
public class SPETaskDispatcher {
    public static <T> T executeTask(SPETask<T> task);
}
```

O método `executeTask` envia o objeto tarefa, recebido como argumento, para execução em um SPE. Ele é um método estático, o que facilita sua invocação, já que não é necessário criar um objeto `SPETaskDispatcher` para tal. Além disso, esse método também é parametrizado com o mesmo parâmetro T da interface `SPETask`, uma vez que ele tem como saída o objeto resultado da tarefa.

Como a implementação de `executeTask` é síncrona, ele bloqueia até que a execução da tarefa termine e o resultado da mesma seja enviado de volta pelo SPE. Por essa razão, tipicamente o programador invocará `executeTask` a partir de um *thread* Java adicional.

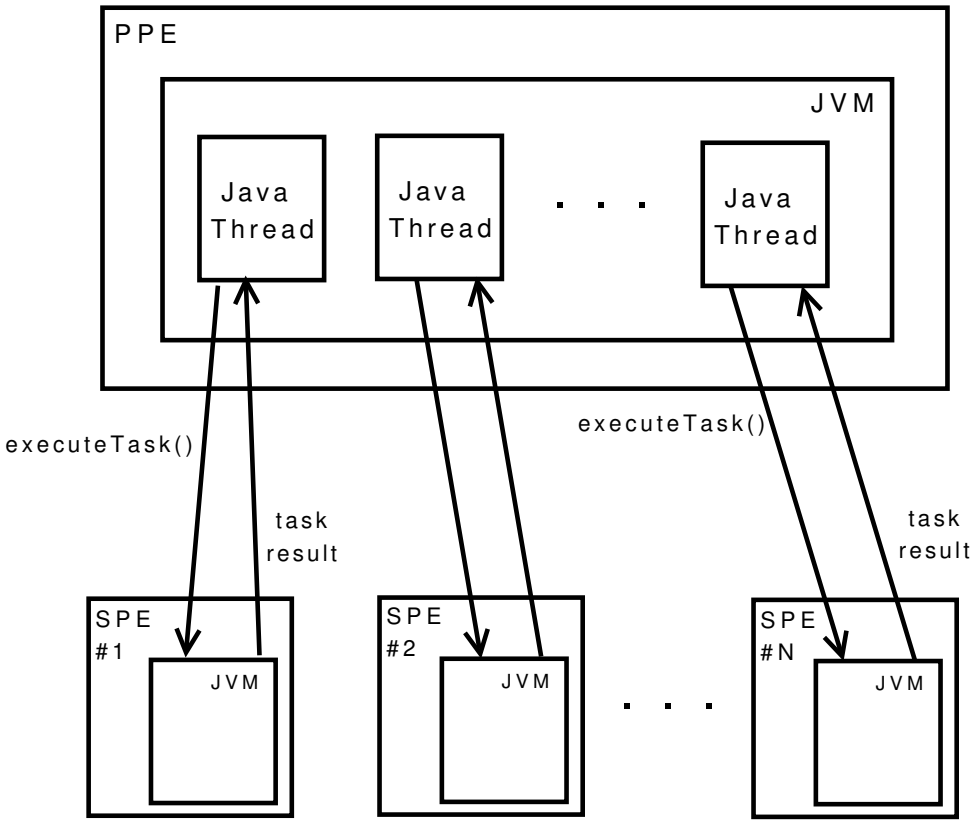

Figura 3.1: Modelo de Execução

A Figura 3.1 ilustra uma representação esquemática do modelo de execução aqui proposto. Como se pode observar nessa figura, vários *threads* no programa do usuário tra-

balham concorrentemente, cada um utilizando um SPE para a execução de tarefas.

Para a definição da interface `SPETask` e da classe `SPETaskDispatcher` inspirou-se no tutorial de RMI da Sun [29], onde um servidor RMI é usado para executar tarefas genéricas remotamente.

3.1.1 Exemplo de uso

O código de um exemplo muito simples, mas completo, é apresentado na Figura 3.2. Esse exemplo inclui a definição de uma tarefa e a invocação da mesma a partir do programa principal.

```
1  public class MyInt {
       public MyInt(int value) {
3          this.value = value;
       }
5      public int value;
   }
7
   public class MyTask implements SPETask<MyInt> {
9      public MyTask(int i1, int i2) {
           this.i1 = i1;
11         this.i2 = i2;
       }
13     public MyInt execute() {
           return new MyInt(i1 + i2);
15     }
       private int i1;
17     private int i2;
   }
19
   public class MyProg {
21     public static void main(String[] args) {
           MyTask task = new MyTask(1,2);
23         MyInt result = SPETaskDispatcher.executeTask(task);
           System.out.println("Task result: " + result.value);
25     }
   }
```

Figura 3.2: Exemplo de definição e uso de uma tarefa

A primeira coisa a ser observada neste exemplo é a definição da classe `MyInt` (linha 1). Ela simplesmente armazena um inteiro. Esta classe é usada como o argumento de tipo da

interface `SPETask` (linha 8) e que, como visto anteriormente, define o tipo do resultado da tarefa. Dado que um tipo parametrizado em Java aceita apenas tipos referência como argumento, é necessário usar uma classe, não um tipo primitivo, como o argumento de `SPETask` (usar `SPETask<int>` seria ilegal). A razão pela qual `MyInt` é utilizada ao invés da classe padrão `Integer` é economizar memória no momento em que ela é carregada no SPE, pois o bytecode de `MyInt` é muito menor do que o de `Integer`.

`MyTask` é uma tarefa que simplesmente soma dois números inteiros. Uma vez que o método `execute` não tem parâmetros, toda a informação necessária a uma tarefa para realizar seu trabalho tem que ser passada em tempo de construção do objeto tarefa ou imediatamente após construção, via métodos adicionais definidos para esse propósito. O importante é que no momento em que `execute` for invocado, o objeto tarefa já contenha todos os dados necessários para sua execução. No exemplo da Figura 3.2, os dois inteiros a ser somados são passados ao construtor da classe tarefa (linha 22).

A última classe deste exemplo é `MyProg`, que é a classe por onde a execução do programa começa. Ela cria um objeto do tipo `MyTask` (uma instância da classe tarefa) e o inicializa com os valores 1 e 2 (linha 22). Depois ela inicia a execução da tarefa em um SPE (linha 23) e, finalmente, imprime o resultado da tarefa (linha 24). Este é atribuído a uma variável do tipo `MyInt` (linha 23), como especificado pela definição de `MyTask`.

Um programa real irá normalmente usar todos os SPEs simultaneamente. Para isso, devem ser criados múltiplos *threads* Java que, concorrentemente, irão cada um lançar uma ou mais tarefas nos SPEs.

3.1.2 Considerações sobre o modelo

Obviamente o modelo de execução aqui apresentado requer que programas *multithreaded* pré-existentes sejam modificados para que estes possam fazer uso dos SPEs. O programa principal, que executa no PPE, é responsável por coordenar a execução das tarefas nos SPEs de forma que o resultado desejado seja alcançado. Além disso, para maximizar o rendimento, o programa principal deve balancear a carga de trabalho nos SPEs, de modo que todos eles fiquem ocupados pelo maior tempo possível.

Este modelo não é destinado, *a priori*, a nenhuma categoria específica de aplicações. Contudo, os tipos de programas que presentemente podem tirar proveito desta solução dependem, em grande medida, das limitações da implementação atual da JVM, listadas na seção 3.3.

3.2 Implementação da solução

Para suportar o novo modelo, partiu-se de uma JVM tradicional pré-existente e dela foram derivadas duas novas JVMs, uma para o PPE e outra para os SPEs. A JVM do PPE, somada a uma ou mais JVMs de SPE, constituem o novo sistema de execução de programas Java para o Cell BE.

A seguir são apresentados a JVM original, usada como ponto de partida para as novas JVMs, e vários detalhes de implementação dessas JVMs especializadas.

3.2.1 JamVM

A JVM escolhida como base para implementação do projeto é a JamVM [19]. Esta JVM apresenta as seguintes características importantes, que determinaram a sua escolha:

- Tem tamanho reduzido (sua implementação é extremamente compacta);
- Seu código é aberto, sendo disponibilizado gratuitamente;
- Suporta a arquitetura PowerPC (o PPE tem arquitetura PowerPC e a do SPE é baseada em PowerPC);
- Satisfaz plenamente a especificação oficial da JVM, em sua versão 2 [18].

Como um dos principais objetivos da JamVM é manter seu tamanho compacto, ela não inclui um compilador JIT, o que significa que todo o código Java é interpretado. Interpretação pura, como se sabe, normalmente acarreta um desempenho consideravelmente pior do que aquele obtido via compilação JIT e posterior execução de código nativo. Entretanto, se a JamVM contivesse um compilador JIT, muito provavelmente ela não caberia no SPE (ou o espaço que sobraria para dados seria tão reduzido que não se tornaria uma solução viável). Além disso, um compilador JIT pré-existente não seria capaz de gerar código para os SPEs, então um esforço significativo teria que ser investido para fazer essa geração.

Para aumentar a velocidade da interpretação, a JamVM implementa várias melhorias nesse sentido, tais como:

- *Threaded interpreter*, o que diminui o custo do *dispatch* de cada instrução;
- *Prefetch* da próxima instrução;

- *Cache* do topo da pilha de operandos;
- *Method inlining*, para diminuir o número de invocações de métodos.

A maioria dessas melhorias pode ser ligada ou desligada seletivamente. De acordo com o autor da JamVM, o emprego dessas otimizações de desempenho permite a essa JVM obter tempos de execução comparáveis aos de uma com um compilador JIT simples.

3.2.2 Detalhes de implementação

Ao ser iniciada, a JVM do PPE recebe como argumento, na linha de comando, o número de SPEs que devem ser usados. Como parte do seu processo de inicialização, a JVM do PPE então lança uma JVM por SPE até que o número de SPEs especificado pelo usuário seja atingido.

Lançar uma JVM no SPE requer a criação de dois *threads* no PPE. O primeiro é responsável por disparar a execução do programa do SPE que, neste caso, é a versão da JamVM para o SPE. Isto tem que ser feito em um *thread* separado porque a função que inicia um programa no SPE bloqueia até o término desse programa. Antes de entrar nessa chamada bloqueante, o primeiro *thread* lança o segundo, que é responsável por processar as mensagens de solicitação e notificação daquele SPE. Portanto, para cada SPE existe um *thread* dedicado no PPE para atender seus pedidos e receber suas notificações. Isto foi feito para aumentar a responsividade do PPE quando múltiplas JVMs de SPE estão executando simultaneamente.

Os dois *threads* mencionados no parágrafo anterior são internos à JVM e, portanto, são *threads* POSIX criados pela implementação em C da JVM. Como notado anteriormente, o programa do usuário irá tipicamente criar mais um *thread* Java para cada SPE em uso. A regra geral é que a execução de uma tarefa deve ser invocada a partir de um *thread* Java separado do restante do programa.

Antes de começar a executar o programa do usuário, a JVM do PPE espera até que todas as JVMs dos SPEs completem sua inicialização. Para que isso ocorra, cada SPE envia uma mensagem para o PPE assim que sua inicialização termina. Essa mensagem de notificação é recebida pelo *thread* dedicado correspondente no PPE.

Reduzindo o tamanho da JVM do SPE

A versão da JamVM para o SPE foi significativamente modificada para reduzir seu tamanho o máximo possível, já que memória é um recurso escasso no SPE. Até mesmo algumas das otimizações de desempenho opcionais da JamVM tiveram que ser desligadas para preservar ainda mais memória no SPE. As principais alterações implementadas na JamVM do SPE foram:

- Já que o SPE sempre executa um programa com um único *thread*, todo o código da JamVM relacionado com suporte a *threads* foi removido. Este código não era necessário de qualquer forma, pois suporte a *threads* não faz parte das bibliotecas de sistema do SPE que acompanham o SDK do Cell BE. A remoção do suporte a *threads* implica na supressão dos *threads* internos da JVM, bem como na eliminação da capacidade da JVM de criar e gerenciar *threads* Java do programa do usuário. No primeiro caso foram desativados três *threads* relacionados à realização assíncrona de *garbage collection*. Já no segundo caso, como apenas um *thread* Java executa na JamVM do SPE, todo o código relacionado à sincronização de *threads* foi removido ou desativado. Dessa forma, foram suprimidas da JVM as funções `lock` e `unlock`, usadas na implementação de código `synchronized`, e as funções `wait`, `notify` e `notifyAll`, que implementam os métodos de mesmo nome da classe `Object`. Além disso, foi removido o código de suporte a monitores, usado na implementação de todas as operações de sincronização mencionadas acima. Finalmente, a garantia de que sempre apenas um *thread* está executando permitiu também a eliminação do controle de acesso exclusivo às estruturas internas da JVM que normalmente precisam ser protegidas de acessos concorrentes.

- O SPE não realiza operações de I/O diretamente (estas são sempre delegadas ao PPE), então todo o código relacionado com acesso ao sistema de arquivos foi removido, inclusive o responsável pelo carregamento dinâmico de bibliotecas (DLL).

- Suporte a *class loaders* foi removido. Isso significa que não pode ser usado nenhum *class loader* que não seja o da própria JVM, nem mesmo o chamado *system class loader*. Isto não constitui uma limitação, pois *class loaders* tradicionais não funcionariam no SPE, dado que este não realiza operações de I/O. Conforme explicado mais adiante, o PPE carrega as classes em prol do SPE.

- Suporte a *parse* de *classpaths* e busca dentro de arquivos zip foram removidos, pois o carregamento de classes é sempre feito pelo PPE.

- Suporte a JNI foi removido. Este código é responsável pela interação entre a JVM e métodos nativos, nos dois sentidos. A maior parte do código envolvido nesse suporte permite o acesso à JVM a partir de métodos nativos.

- Suporte a reflexão foi removido. Programação reflexiva é recomendada apenas em casos muito específicos (por exemplo, manipulação em tempo de execução de código genérico), que não se aplicam aos programas tipicamente executados no Cell BE.

- Suporte a propriedades Java foi removido. Nenhum dos programas escritos para o novo modelo fazem uso de propriedades Java. Como visto a seguir, este suporte pode ser habilitado, se necessário.

- Várias classes Java das hierarquias `Exception` e `Error` que eram originalmente carregadas durante a inicialização, agora são carregadas de forma *lazy* (sua carga é postergada até o momento em que cada classe é realmente necessária).

Exceto pelo código removido nos quatro primeiros itens da lista acima, que realmente não tem utilidade no SPE, as alterações descritas nos itens restantes podem ser revertidas, se assim for exigido por alguma aplicação específica desta solução. Contudo, aumentar o tamanho das áreas de código e/ou dados da JVM implica em uma correspondente redução da memória disponível para os dados do programa Java sendo executado, fator que deve ser considerado cuidadosamente.

Como mencionado no segundo item da lista acima, o SPE não acessa o sistema de arquivos diretamente. A consequência disso é que sempre que uma nova classe precisa ser carregada, o SPE solicita seu bytecode ao PPE. Este é outro pedido atendido pelos *threads* dedicados no PPE. Para melhorar o desempenho do sistema como um todo, a JamVM do PPE foi alterada para armazenar em memória o bytecode de todas as classes carregadas. Assim, sempre que um pedido de carregamento de classe é recebido, o PPE primeiro checa se a classe já foi previamente carregada. Nesse caso, o bytecode dela é imediatamente lido da memória, evitando o acesso ao disco. Um tipo de programa comumente executado no Cell BE divide os dados da entrada em partes de tamanhos iguais e distribui essas partes menores entre os SPEs. Cada SPE executa o mesmo código sobre sua parte dos dados e devolve o resultado do processamento ao PPE. Portanto,

Item	Valor original	Novo valor
Tamanho mínimo do heap	2 MB	4 KB
Tamanho máximo do heap	128 MB	32 KB
Tamanho da pilha	64 KB	4 KB

Tabela 3.1: Tamanhos do heap e da pilha Java na JamVM original e na versão para o SPE.

todos os SPEs necessitam as mesmas classes e, consequentemente, o PPE vai receber várias vezes o pedido de carregamento de cada uma dessas classes. Nesse cenário, a estratégia de manter em memória o bytecode das classes carregadas proporciona um ganho de desempenho significativo.

Uma outra medida tomada para economizar memória no SPE foi descartar o bytecode referente ao corpo dos métodos de cada classe carregada pela JVM do SPE. Apenas o índice de cada método dentro do bytecode da classe é guardado. Quando um método está prestes a ser executado pela JVM do SPE, se o seu código ainda não está disponível localmente, o SPE solicita o mesmo ao PPE, enviando o índice desse método como parâmetro do pedido. O PPE atende a um pedido de carregamento de método muito rapidamente, pois a classe associada já foi necessariamente carregada antes, o que significa que seu bytecode está presente em memória.

Também foram alterados os valores padrão dos tamanhos do heap e da pilha utilizados pela JamVM no SPE, de forma a adequá-los ao espaço limitado da LS, conforme a Tabela 3.1.

Execução de uma tarefa no SPE

A Figura 3.3 ilustra, passo a passo, o que ocorre quando `SPETaskDispatcher.executeTask` é invocado por um *thread* no programa do usuário. A descrição desses passos é dada a seguir:

1. `executeTask` invoca um método nativo da JVM do PPE, chamado `pushToSPEAndExecute`;
2. `pushToSPEAndExecute` identifica um SPE ocioso;
3. `pushToSPEAndExecute` serializa o objeto tarefa recebido, gerando um array de bytes;
4. `pushToSPEAndExecute` envia um pedido de execução de tarefa, por meio de uma mensagem de *mailbox*, ao SPE identificado no segundo passo;

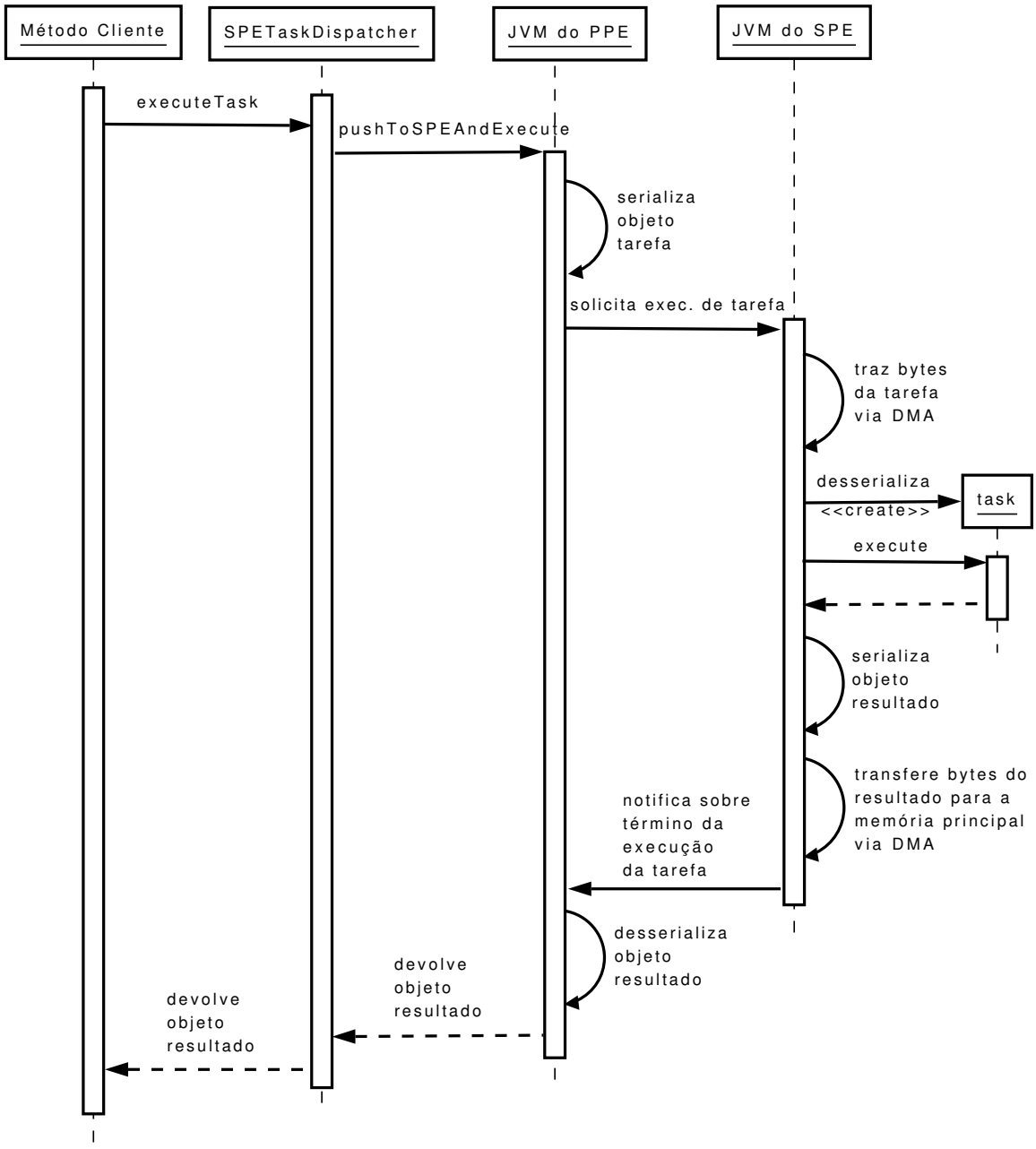

Figura 3.3: Execução passo a passo de uma tarefa

5. O SPE transfere para sua LS, via DMA, o array de bytes do passo 3 e reconstitui (desserializa) o objeto tarefa;
6. A JVM do SPE executa a tarefa simplesmente invocando `execute` no objeto tarefa;
7. O SPE serializa o objeto resultado da tarefa, gerando um array de bytes;
8. O SPE transfere para a memória principal, via DMA, os bytes do objeto resultado;
9. O SPE notifica o PPE, através de uma mensagem de *mailbox*, sobre o término da execução da tarefa;
10. `pushToSPEAndExecute` desserializa os bytes do resultado, reconstituindo o objeto resultado;
11. `pushToSPEAndExecute` devolve o objeto resultado para `executeTask` que, por sua vez, devolve-o ao método cliente no programa do usuário.

Ao obter um SPE ocioso no passo 2 acima, `pushToSPEAndExecute` usa uma estrutura interna para marcar esse SPE como ocupado. Se não há SPEs ociosos nesse momento, `pushToSPEAndExecute` simplesmente espera até que um se torne disponível. Essa é uma espera não ocupada, de forma a não consumir ciclos desnecessariamente no PPE. Inicialmente todos os SPEs são marcados como ociosos. Após tornar-se ocupado, um SPE volta a ser marcado como ocioso quando a execução da tarefa que ele está executando termina e o PPE recebe o resultado correspondente.

Um resumo das interações entre PPE e SPE é mostrado na Figura 3.4. Com exceção da primeira interação, todas as outras são mensagens enviadas através do mecanismo de *mailbox*. As interações são apresentadas de forma cronológica, desde o momento em que o PPE inicia a JVM do SPE até o término da execução de uma tarefa. As mensagens de pedido de carregamento de classe e método, assim como suas respectivas respostas, podem ocorrer várias vezes tanto entre o início e fim da inicialização da JVM quanto entre o início e fim da execução da tarefa. Como já discutido antes, do lado do PPE há um *thread* dedicado ao processamento dessas mensagens de *mailbox* para cada SPE.

Serialização

Como mencionado anteriormente, a transferência de um objeto entre PPE e SPE (e vice-versa), exige sempre a serialização desse objeto no lado do enviador e sua correspondente desserialização no lado do receptor. Inicialmente, cogitou-se a utilização das classes

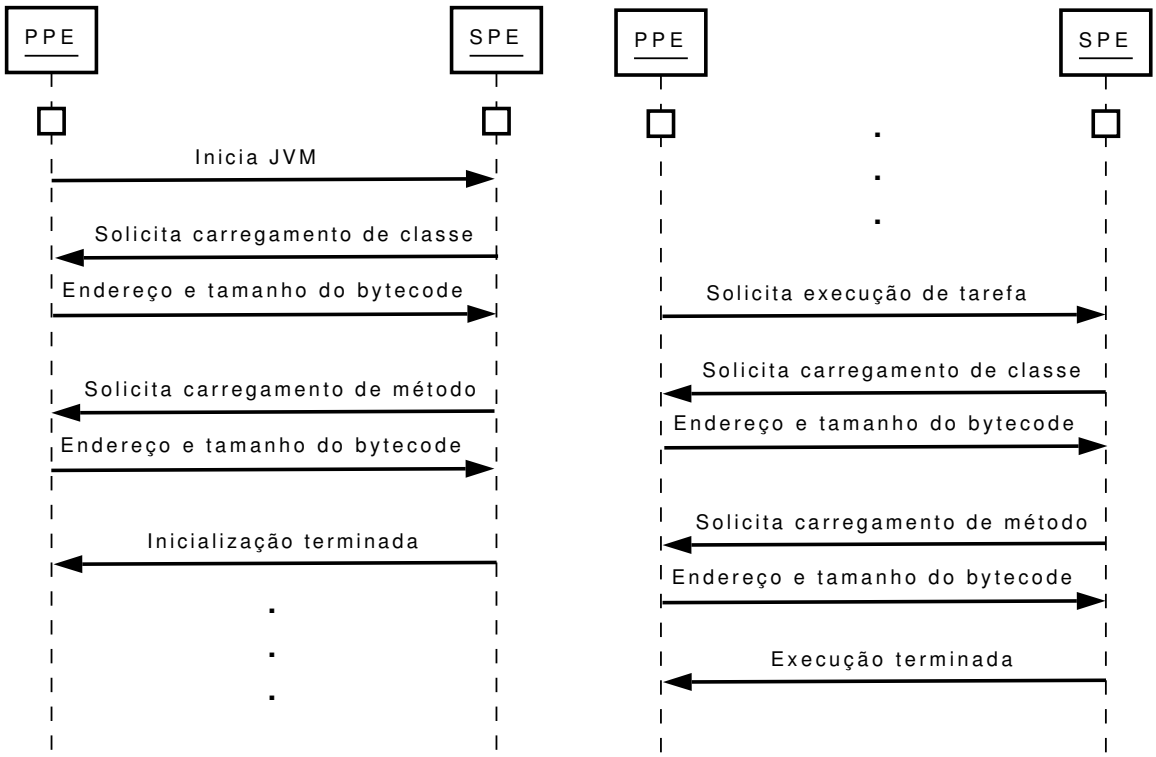

Figura 3.4: Interações entre PPE e SPE

padrão de Java para serialização, que são usadas pelo RMI. Infelizmente, essas classes são extremamente numerosas e grandes, pois foram desenvolvidas da forma mais geral e abrangente possível. Esse fato, por si só, já inviabiliza o uso dessa solução no Cell BE devido, mais uma vez, à pouca memória do SPE. Além disso, conforme já discutido na seção 2.3.1, o desempenho da serialização padrão de Java deixa muito a desejar.

Optou-se, então, por implementar algoritmos *ad hoc* de serialização e desserialização, aproveitando-se do conhecimento sobre a representação interna de objetos na JamVM. A serialização transforma, recursivamente, um objeto arbitrariamente complexo (que pode conter variáveis que são outros objetos e assim sucessivamente) em uma sequência de bytes. Arrays em Java também são classes, mas como têm uma estrutura peculiar, precisam ser tratados separadamente. A recursão também se aplica a eles, já que um array pode conter objetos como seus elementos, inclusive outros arrays. A cada novo nível de recursão, um objeto (regular ou array) é serializado e, juntamente com seu conteúdo, o nome da classe do objeto também é inserido na sequência de bytes sendo gerada. A desserialização, como esperado, faz o processo inverso da serialização, recursivamente transformando os bytes de entrada em uma cópia fiel do objeto originalmente serializado. O resultado da desserialização é um novo objeto na JVM onde esse procedimento foi invocado. Para isso, a função original da JamVM para criação de objetos é utilizada.

Garbage Collection

Uma alteração radical que chegou a ser implementada para reduzir ainda mais o tamanho da JamVM do SPE foi a eliminação completa do *garbage collector* (GC). Como a execução assíncrona de *garbage collection* teve que ser necessariamente desligada (dada a ausência de suporte a *threads* no SPE), o único momento em que ainda se invocava o GC era quando não havia memória suficiente durante o processamento de um pedido de alocação de memória. Entretanto, as chances de que o GC obtivesse a memória necessária em uma situação dessas não eram muito boas, o que justificaria então a sua completa remoção (como o GC acabaria falhando de qualquer forma, então não seria muito diferente falhar imediatamente quando faltasse memória em um pedido de alocação). Por outro lado, quando o SPE termina de executar uma tarefa, é importante liberar o máximo possível de memória em preparação ao recebimento da próxima tarefa. Nada mais adequado do que invocar o GC explicitamente nesse momento, como acabou sendo feito. Desse modo, apesar de ter sido realizado todo o trabalho de permitir a compilação da JVM do SPE sem o GC, na versão final do projeto essa opção não foi usada e o GC continua fazendo parte

da JVM. Ele só não é invocado assincronamente, mas entra em cena quando falta memória para atender um pedido de alocação e ao final da execução de cada tarefa, conforme se acabou de descrever.

```
  void jamSPUMain()
2 {
      InitArgs args;
4     unsigned int requestAddr;
      task_execution_request task_req __attribute__ ((aligned (128)));
6
      setDefaultInitArgs(&args);
8     initVM(&args);

10    /* Notifica PPE que a JVM deste SPE está pronta (chamada
         bloqueante). */
12    spu_write_out_intr_mbox(SPE_VM_INIT_DONE_NOTIF_ID);

14    /* Entra em espera por pedidos de execução de tarefa do PPE. */
      while (1)
16    {
          /* Espera enquanto mailbox está vazia. */
18        requestAddr = spu_read_in_mbox();
          /* Transfere via DMA pedido de execução de tarefa. */
20        mfc_get(&task_req, requestAddr,
                  sizeof(task_req), spe_tag_id, 0, 0);
22        mfc_read_tag_status_all();
          /* Executa tarefa. */
24        processTaskRequest(&task_req);
          /* Invoca garbage collector explicitamente. */
26        gc1();
      }
28 }
```

Figura 3.5: Loop principal da JVM do SPE

A Figura 3.5 contém uma versão simplificada do laço principal da JVM do SPE. Como se pode observar, essa JVM fica em um laço infinito (linha 15) aguardando por um pedido de execução de tarefa (linha 18). Quando isso ocorre, a tarefa é transferida para a LS do SPE (linha 20), executada (linha 24) e, finalmente, o GC é invocado (linha 26) para liberar a memória ocupada pelos objetos alocados durante a execução da tarefa.

Dada a maneira como a desserialização foi implementada, o objeto sendo reconstituído passa a existir do ponto de vista da JVM antes mesmo da desserialização terminar com-

pletamente. Isso originou um problema durante a execução da *garbage collection*, em que o GC tentava processar um objeto sendo reconstituído como um objeto qualquer, o que causava resultados imprevisíveis, pois esse objeto ainda possuía campos com valores indefinidos. Para solucionar esse problema, foram introduzidas novas verificações para evitar que objetos sendo reconstituídos sejam processados pelo GC.

3.3 Limitações

Uma vez que a estratégia do projeto era colocar uma JVM em cada SPE, o desenvolvimento da versão da JamVM para o SPE priorizou a obtenção de um tamanho reduzido e não um bom desempenho. Como consequência, a interpretação de bytecode no SPE tem um desempenho que fica aquém do que a JamVM pode teoricamente conseguir, quando todas suas otimizações são ativadas. Contudo, o desempenho geral de um programa Java que faz uso dos SPEs, como proporcionado por este projeto, é significativamente melhor do que o do mesmo programa executando exclusivamente no PPE, conforme mostrado no próximo capítulo.

Devido à quantidade limitada de memória disponível para código Java e para dados do usuário depois que a JVM do SPE é carregada e inicializada, apenas tarefas pequenas podem ser executadas nos SPEs. Isto significa que o tamanho total das classes necessárias à execução de uma tarefa deve ser modesto, incluindo as classes da tarefa propriamente dita, do resultado da tarefa e quaisquer outras classes referenciadas por estas duas.

A implementação original da JamVM depende do pacote GNU Classpath, que implementa as principais bibliotecas de classes Java. Várias dessas bibliotecas, por sua vez, recorrem a bibliotecas nativas, que também fazem parte da distribuição do GNU Classpath. Entretanto, já foi mencionado anteriormente que a JVM do SPE não é capaz de carregar bibliotecas nativas dinamicamente. Por causa disso, todo método nativo que precisar ser invocado, seja do GNU Classpath ou definido pelo usuário, deve ser estaticamente ligado ao executável da JVM.

Finalmente, o modelo apresentado aqui não é adequado para problemas que seriam tipicamente resolvidos com uma abordagem baseada em memória compartilhada, dado que ele adota o paradigma de memória distribuída. Em particular, na versão atual uma tarefa em um SPE não pode lançar diretamente outra tarefa em outro SPE, então o usuário deve decompor o problema original em tarefas que não interagem diretamente entre si.

Este capítulo apresentou um novo modelo de execução para programas Java no Cell BE. O diferencial mais importante desse modelo é o total controle oferecido ao programador sobre qual código executar nos SPEs. Isso se dá através da definição de classes tarefa, que implementam o código a ser executado nos SPEs. Foi mostrado como uma tarefa é executada e uma série de detalhes de implementação das versões especializadas da JamVM para o PPE e o SPE. Finalmente, listou-se as limitações conhecidas do modelo e de sua implementação atual. No próximo capítulo são mostrados alguns números de desempenho que comprovam a viabilidade da nova solução.

Capítulo 4

Análise de viabilidade e desempenho

O capítulo 3 detalhou um novo modelo que permite ao programador Java utilizar todos os núcleos do processador Cell BE. Para comprovar a viabilidade dessa alternativa, o presente capítulo irá apresentar uma série de programas que empregam esse modelo, bem como seus correspondentes tempos de execução. Os resultados obtidos demonstram um ganho significativo em desempenho quando os SPEs são usados na execução dos programas.

4.1 Ambiente e metodologia

O sistema usado para a análise de viabilidade e desempenho foi um PlayStation 3, com uma distribuição Linux Fedora 7 e kernel 2.6.24. Dos 8 SPEs presentes no chip do Cell BE, apenas 6 estão disponíveis para aplicações de usuário no PlayStation 3. Um é inutilizado pelo processo de manufatura da Sony e outro é reservado para o sistema operacional do aparelho.

Para cada programa Java incluído nesta análise, partiu-se de sua implementação sequencial, que foi executada na versão original da JamVM. Obviamente a JamVM padrão não faz uso dos SPEs, portanto a versão sequencial de cada programa executa exclusivamente no PPE. O tempo de execução desta configuração foi usado como base de comparação para a versão paralela do mesmo programa.

A versão paralela utiliza a solução proposta nesta dissertação, que introduz uma "versão Cell BE" da JamVM, batizada de JAM_PPU (o executável da JAM_PPU contém tanto a JVM do PPE quanto a JVM do SPE). O programa paralelo foi executado em seis configurações, que se distinguem pelo número de SPEs usados, de 1 a 6. A JAM_PPU foi

testada também com uma cofiguração de zero SPEs, executando a versão sequencial do programa.

Todos os tempos foram obtidos através de instrumentação dos próprios programas Java sendo observados. Para isso, utilizou-se `System.nanoTime()`, que tem precisão de nanossegundos, porém os tempos finais foram convertidos para milissegundos, o que é suficiente para os propósitos desta análise e mais conveniente para a apresentação dos valores.

Uma das categorias de programas paralelos que podem ser usadas mais facilmente com o novo modelo proposto é a que resolve problemas através da decomposição geométrica [21] dos dados de entrada. Quando se utiliza essa abordagem, a estrutura básica do programa pode ser decomposta nos seguintes passos:

1. Os dados de entrada são divididos em S partes iguais, onde S é o número de SPEs configurados para uso.

2. Criam-se S *threads* Java, um por SPE. Cada *thread* fica encarregado pelo processamento de 1/S dos dados.

3. Cada *thread* divide seu lote de dados em N partes iguais, onde N depende do volume máximo de dados que o SPE consegue acomodar em sua LS para a execução de uma tarefa. Dessa forma, N é o resultado da divisão do tamanho do lote de dados passado ao *thread* pelo tamanho máximo da área de dados disponível no SPE.

4. Cada *thread* executa um laço de N iterações. A cada iteração, o *thread* envia ao SPE uma tarefa contendo uma das N partes dos dados e vai consolidando os resultados obtidos.

Uma outra forma simples de se tirar proveito da habilidade de executar programas Java nos SPEs é ter cada SPE processando dados não relacionados entre si. Esta abordagem é útil nos casos em que for difícil ou impossível criar uma versão paralela de um algoritmo. Nesses casos, pode-se aumentar a produtividade simplesmente pondo cada SPE para executar o mesmo algoritmo sobre uma entrada diferente. Por exemplo, o algoritmo MD5, que gera uma sequência *hash* para uma string arbitrarimente longa, pode ser colocado em uma tarefa e processado por um SPE. Dessa forma é possível calcular o *hash* de vários arquivos ao mesmo tempo (cada arquivo sendo processado por um SPE). Isso foi implementado, utilizando-se como entrada 6 arquivos de 4,4 MB. Quando processados sequencialmente pela JamVM original, o tempo de execução foi de 20,25 segundos.

Quando cada arquivo foi processado por um SPE, concorrentemente aos outros, o tempo de execução foi de 8,04 segundos. Portanto, utilizando-se essa abordagem com 6 SPEs obteve-se um ganho de 2,52 para o algoritmo de MD5, com relação à versão original executando na JamVM padrão.

4.2 Descrição detalhada de um programa usado na avaliação do modelo

Esta seção apresenta uma descrição detalhada de um programa Java que utiliza a decomposição geométrica para enviar os dados de entrada aos SPEs para processamento. O programa original foi extraído de [8, 7], que fazem parte de uma série de artigos sobre processamento de imagens em Java.

A versão original do programa contém uma interface gráfica que permite ao usuário alterar alguns parâmetros utilizados no processamento e observar a imagem antes e depois da aplicação dos efeitos. Todo o código relativo à interface gráfica foi removido, uma vez que não era relevante para os testes realizados.

Com o objetivo de reduzir o consumo de memória no SPE, optou-se por manter a representação interna dos pixels da imagem como um array simples de inteiros, onde cada elemento corresponde a um pixel. No programa original [8] essa representação, que é usada ao se ler os pixels da imagem, é transformada em um array tri-dimensional de inteiros, onde as dimensões representam linha, coluna e valor do componente ARGB (a terceira dimensão tem 4 posições, uma para cada componente do ARGB). Essa representação alternativa é mais conveniente para o processamento da imagem, mas requer muito mais espaço. Para que a comparação entre as versões sequencial e paralela do algoritmo fosse justa, a mesma representação interna dos pixels (array simples de inteiros) foi usada em ambas.

Dentre as várias rotinas de processamento contidas no programa, foi selecionada para execução nos SPEs a que calcula a média quadrática dos valores dos componentes RGB de todos os pixels da imagem. A versão sequencial desse algoritmo é implementada pelo método `getRms` da Figura 4.1.

Na versão paralela do cálculo da média quadrática, implementada pelo método `parallelGetRms` (Figura 4.2), a imagem é dividida em fatias (grupos de linhas consecutivas), uma para cada SPE (linha 2). O tamanho da fatia é arredondado para conter

```
int getRms(int[] data, int imgRows, int imgCols) {
  int pixelCntr = 0;
  long accum = 0;
  for (int row = 0;row < imgRows;row++) {
    for (int col = 0;col < imgCols;col++) {
      for (int i = 1;i <= 3;i++) {
        int n = (3 - i) * 8;
        int val = data[row*imgCols + col];
        if (n > 0)
          val = val >> n;
        val = val & 0xFF;
        accum += val * val;
      }
      pixelCntr += 3;
    }
  }
  int meanSquare = (int)(accum/pixelCntr);
  int rms = (int)(Math.sqrt(meanSquare));
  return rms;
}
```

Figura 4.1: Média quadrática dos pixels da imagem - versão sequencial

um número exato de partes (linha 4), sendo que cada uma dessas partes corresponde aos dados de uma única tarefa enviada ao SPE. Como discutido anteriormente, para se calcular o tamanho de uma dessas partes, leva-se em conta a capacidade da área de dados do SPE. No código da Figura 4.2, tanto o número de SPEs (`numOfSPEs`) quanto o número máximo de inteiros processados em cada tarefa enviada a um SPE (`maxIntsPerSPE`) são valores definidos por parâmetros na linha de comando da invocação do programa.

O processamento de cada fatia fica a cargo de um novo *thread*. Portanto, criam-se `numOfSPEs` *threads*, que são instâncias da classe `ImageProcThread`, apresentada na Figura 4.3.

Após lançar os *threads*, `parallelGetRms` processa as linhas da imagem que eventualmente tenham sobrado da divisão em fatias (linhas 13 a 15). Para isso é usado o método `addSquares`, que faz o mesmo que `getRms` (Figura 4.1), mas apenas para as linhas a partir da especificada pelo parâmetro `row`. Finalmente, os resutados parciais de cada *thread* e da eventual sobra são consolidados (linhas 17 a 25) e a média quadrática é calculada tirando-se a raiz quadrada da média aritmética dos quadrados de todos os componentes RGB da imagem (linhas 26 a 28).

A classe `ImageProcThread` (Figura 4.3) implementa um *thread* que processa uma fatia da imagem. Para isso, executa um laço que envia consecutivamente uma nova tarefa ao SPE (linha 18) até que os dados se esgotem. Os resultados de cada tarefa vão sendo somados em `squaresSum` (linha 19) até que, ao final da execução do laço, essa variável contenha a soma dos quadrados dos componentes de todos os pixels da fatia.

O cálculo das somas dos quadrados é efetivamente realizado pela classe tarefa, `ImageSquaresSumTask`, mostrada na Figura 4.4. Como o objeto tarefa inteiro é serializado para envio ao SPE, é importante que ele contenha somente os dados realmente necessários. É por essa razão que um novo array `pixels` é criado para cada nova tarefa (Figura 4.3, linha 14), contendo apenas os pixels de uma iteração. O resultado da execução da tarefa é do tipo `MyLong` (linha 18), que é semelhante à classe `MyInt` da Figura 3.2, porém ao invés de encapsular um inteiro de 32 bits, `MyLong` contém uma variável do tipo `long` (inteiro de 64 bits).

A classe tarefa `ImageSquaresSumTask` (Figura 4.4) é muito simples. Como já mencionado, o seu método `execute` apenas calcula a soma dos quadrados dos componentes dos pixels no array recebido no construtor, de forma idêntica ao código da versão sequencial (Figura 4.1).

```
int parallelGetRms(int[] data, int imgRows, int imgCols) {
  int slice = imgRows / numOfSPEs;
  int rowsPerSPE = maxIntsPerSPE / imgCols;
  slice = ((int)(slice / rowsPerSPE)) * rowsPerSPE;
  ImageProcThread[] t = new ImageProcThread[numOfSPEs];
  int row = 0;
  for (int i = 0; i < numOfSPEs; i++, row += slice) {
    t[i] = new ImageProcThread(data, slice, imgCols, row, rowsPerSPE);
    t[i].start();
  }

  // processamento das eventuais linhas que sobraram
  long squaresSum = 0;
  if (row < imgRows)
    squaresSum = addSquares(data, imgRows, imgCols, row);

  for (int i = 0; i < numOfSPEs; i++) {
    try {
      t[i].join();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
    squaresSum += t[i].getSquaresSum();
  }
  int pixelCount = imgRows * imgCols * 3;
  int meanSquare = (int)(squaresSum/pixelCount);
  int rms = (int)(Math.sqrt(meanSquare));
  return rms;
}
```

Figura 4.2: Média quadrática dos pixels da imagem - versão paralela

```
public class ImageProcThread extends Thread {
  public ImageProcThread(int[] data, int sliceRows, int imgCols,
                         int startRow, int rowsPerSPE) {
    this.data = data;  this.sliceRows = sliceRows;
    this.imgCols = imgCols;  this.startRow = startRow;
    this.rowsPerSPE = rowsPerSPE;
  }

  public void run() {
    int upperBound = startRow + sliceRows;
    squaresSum = 0;
    for (int row = startRow; row < upperBound; row += rowsPerSPE) {
      int size = rowsPerSPE * imgCols;
      int[] pixels = new int[size];
      System.arraycopy(data, row*imgCols, pixels, 0, size);
      ImageSquaresSumTask task = new
        ImageSquaresSumTask(pixels, rowsPerSPE, imgCols);
      MyLong result = SPETaskDispatcher.executeTask(task);
      squaresSum += result.value;
    }
  }

  public long getSquaresSum() {
    return squaresSum;
  }

  private int[] data;
  private int sliceRows, imgCols, startRow, rowsPerSPE;
  private long squaresSum;
}
```

Figura 4.3: Média quadrática dos pixels da imagem - classe thread

```
public class ImageSquaresSumTask implements SPETask<MyLong> {
  public ImageSquaresSumTask(int[] pixels, int rowsPerSPE,
                             int imgCols) {
    this.pixels = pixels;
    this.rowsPerSPE = rowsPerSPE;
    this.imgCols = imgCols;
  }

  public MyLong execute() {
    long accum = 0;
    for (int row = 0; row < rowsPerSPE; row++) {
      for (int col = 0; col < imgCols; col++) {
        for (int i = 1; i <= 3; i++) {
          int n = (3 - i) * 8;
          int val = pixels[row*imgCols + col];
          if (n > 0)
            val = val >> n;
          val = val & 0xFF;
          accum += (val * val);
        }
      }
    }
    return new MyLong(accum);
  }

  private int[] pixels;
  private int rowsPerSPE;
  private int imgCols;
}
```

Figura 4.4: Média quadrática dos pixels da imagem - classe tarefa

Opção	Descrição
-Xspecount:	Número de SPEs usados (0 a 6).
-Xspeminheap:	Tamanho mínimo do heap Java no SPE em KB.
-Xspemaxheap:	Tamanho máximo do heap Java no SPE em KB.
-Xspestack:	Tamanho da pilha Java no SPE em KB.

Tabela 4.1: Opções de linha de comando exclusivas da JAM_PPU.

Parâmetro	Descrição
ImgMod23	Classe que determina a transformação aplicada à imagem.
Riverside_Nature_Center3.jpg	Nome do arquivo da imagem.
8100	Número máximo de inteiros processados por uma tarefa no SPE.
1	Número de SPEs utilizados por ImgMod23 para distribuir as tarefas.

Tabela 4.2: Parâmetros passados à versão paralela do programa de processamento de imagem.

4.2.1 Execução e resultados obtidos

A imagem usada como entrada para o programa de cálculo da média quadrática é do tipo JPEG, com 2048 x 1536 pixels. A cada tarefa enviada a um SPE, três linhas (24576 bytes) são processadas.

O comando para executar a versão paralela deste programa na JAM_PPU com 1 SPE é o seguinte:

```
JAM_PPU -Xmx1024M -Xspecount:1 -Xspeminheap:36 -Xspemaxheap:36
ImgMod02a ImgMod23 Riverside_Nature_Center3.jpg 8100 1
```

A opção `-Xmx1024M` especifica um heap com tamanho máximo de 1024 MB (ou 1 GB) para a JVM do PPE. `-Xmx` é uma opção padrão nas JVMs.

Já as três opções subsequentes no comando acima foram introduzidas na implementção da JAM_PPU, juntamente com uma outra. Todas elas são detalhadas na Tabela 4.1.

Os parâmetros passados ao programa Java (cuja classe principal é `ImgMod02a`), em sua versão paralela, são descritos na Tabela 4.2.

O comando para executar a versão sequencial do mesmo programa na JAM_PPU é:

```
JAM_PPU -Xmx1024M -Xspecount:0 ImgMod02a ImgMod23
Riverside_Nature_Center3.jpg
```

Sempre que a versão sequencial do programa é usada, a JAM_PPU é configurada com

zero SPEs, o que faz com que a JVM do PPE não inicie nenhuma JVM de SPE. Como nesta linha de comando a classe `ImgMod23` contém a versão sequencial do algoritmo, todas as opções e parâmetros específicos da execução paralela não são necessários.

Finalmente, o comando para executar a versão sequencial do programa na JamVM original é:

```
jamvm -Xmx1024M ImgMod02a ImgMod23 Riverside_Nature_Center3.jpg
```

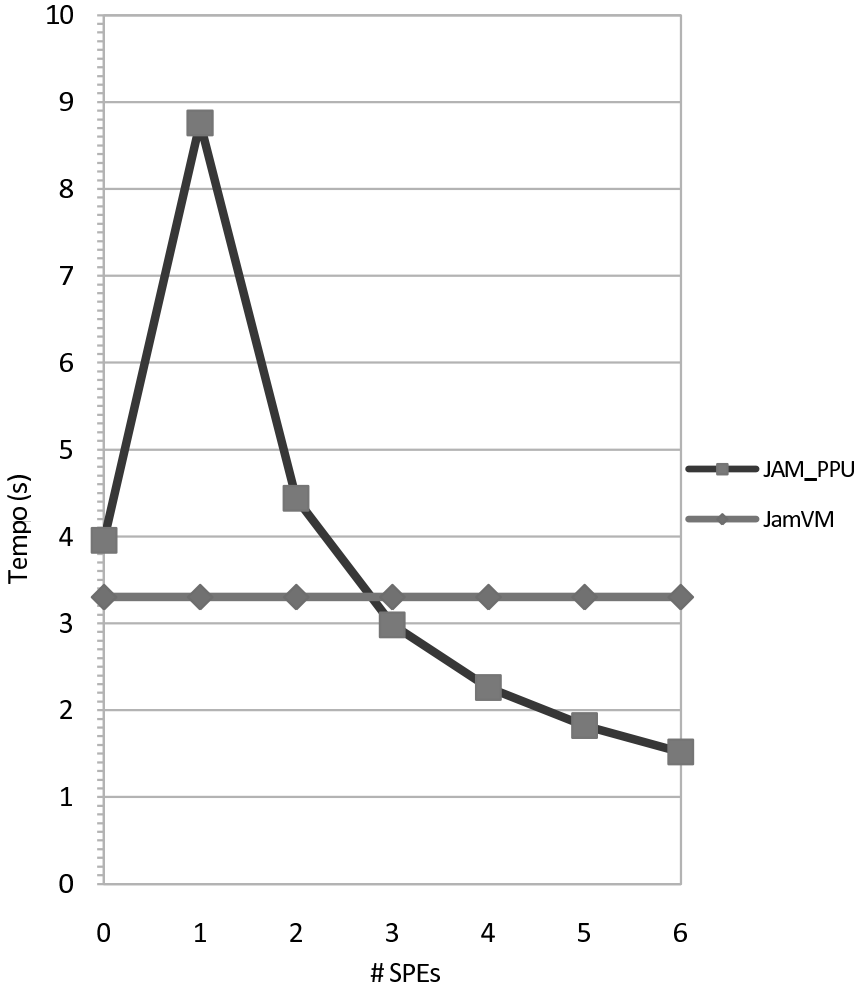

Figura 4.5: Tempo de execução versus número de SPEs - processamento de imagem

A Figura 4.5 contém o gráfico do tempo de execução para cada uma das sete configurações (0 a 6 SPEs) do programa executado pela JAM_PPU. A título de comparação, também foi plotado no mesmo gráfico o tempo da versão sequencial executada na JamVM original, identificado na figura como "JamVM".

Quando a JAM_PPU foi executada sem o auxílio de SPEs, seu desempenho foi apro-

ximadamente 20% pior do que o da JamVM original. Após uma análise inicial, não foi identificada nenhuma razão que justificasse essa perda de desempenho. Uma possibilidade é a diferença na forma como as duas JVMs são compiladas: enquanto a JamVM foi gerada diretamente no PlayStation 3 com seus scripts próprios de instalação, a JAM_PPU foi criada com o Cell SDK, em uma máquina x86, através de *cross compiling*.

A configuração da JAM_PPU com apenas um SPE foi 2,65 vezes mais lenta do que a JamVM. Isto se deve principalmente ao fato do interpretador da JVM do SPE não conter muitas das otimizações possíveis da JamVM (estas otimizações estão ligadas na JamVM original usada como base de comparação), o que deixa a interpretação significativamente mais lenta. Além disso, como apenas um SPE é utilizado, a execução nesse caso é essencialmente sequencial também, pois praticamente todo o processamento da imagem é feito no único SPE disponível.

À medida que mais SPEs foram incrementalmente adicionados, o tempo de execução caiu rapidamente, já que com dois ou mais SPEs o processamento passa a ser verdadeiramente concorrente. Com três SPEs, a JAM_PPU já foi mais rápida que a JamVM. Com seis SPEs o programa executou em 46% do tempo original, o que representa uma melhora de desempenho de 2,17 vezes.

É importante ressaltar que apesar de, à primeira vista, ser óbvio obter-se um ganho de desempenho devido à utilização de múltiplos núcleos, o que se está demonstrando aqui é a viabilidade do modelo e sua presente implementação. Poderia ter ocorrido que a solução proposta introduzisse tanto *overhead* que, mesmo com 6 SPEs, o tempo de execução do programa paralelo ainda fosse mais alto que o da versão sequencial na JamVM original.

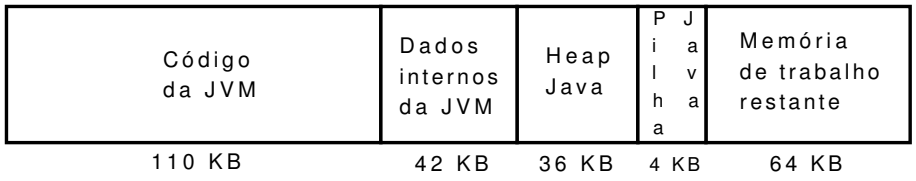

Figura 4.6: Uso da memória no SPE

A Figura 4.6 detalha a utilização da memória do SPE durante a execução deste programa. A versão para o SPE da JamVM tem aproximadamente 110 KB. No momento em que a JVM termina sua inicialização, outros 82 KB já foram tomados, dos quais 36 KB são reservados para o heap Java, 4 KB para a pilha Java do único *thread* existente e 42 KB são usados por dados internos da JVM, principalmente classes Java pré-carregadas. Com isso, sobram 64 KB de memória de trabalho para o recebimento e execução de tarefas.

Os tamanhos do heap e da pilha são configuráveis via linha de comando, enquanto que os tamanhos do código da JVM e de seus dados internos não variam, seja lá qual tarefa estiver sendo executada. A memória de trabalho restante deve ser grande o suficiente para conter a forma serializada do objeto tarefa e do objeto resultado (não simultaneamente), mais as estruturas de dados da JVM criadas ao se carregar as classes Java necessárias para a execução da tarefa.

4.3 Outros programas

O segundo programa usado para validação do modelo é um simples conversor de caracteres. Ele lê um arquivo texto de entrada, converte todos seus caracteres para letras maiúsculas e grava o resultado em um arquivo de saída. Na versão paralela, os SPEs são usados para converter simultanemanete os caracteres de diferentes partes do arquivo de entrada.

Na busca de maximizar a quantidade de caracteres convertidos a cada tarefa executada por um SPE, chegou-se ao valor de 22528 bytes. Como tanto o objeto tarefa quanto o objeto resultado contêm um vetor com esse mesmo tamanho em bytes e ao final da execução da tarefa os dois objetos existem simultaneamente no heap, o tamanho deste último deve ser no mínimo o dobro de 22528. O valor que satisfez as necessidades de uma tarefa e que, portanto, foi utilizado na execução do conversor de caracteres, é 47 KB.

Como entrada para o conversor de caracteres foi passado um arquivo de aproximadamente 17 MB.

O terceiro programa ordena um vetor de inteiros utilizando uma abordagem mista de merge sort e quick sort para melhorar o desempenho do merge sort puro. O merge sort é chamado recursivamente nas duas metades do vetor de entrada até que o tamanho do vetor a ser ordenado fique abaixo de um valor de corte. Quando esse tamanho é atingido, o vetor é ordenado via quick sort.

O estabelecimento de um tamanho de corte apropriado permite tirar vantagem do uso dos SPEs para executar os quick sorts. Assim, fixou-se o tamanho de corte do vetor em 10240 elementos, o que possibilita a ordenação de um desses vetores inteiramente em um SPE. O tamanho do heap da JVM do SPE usado nesta configuração foi de 42 KB.

O merge sort é invocado recursivamente apenas na versão sequencial do algoritmo. Na versão paralela, como nos programas anteriores, o vetor inicial é dividido pelo número disponível de SPEs e cada uma destas partes fica a cargo de um *thread* Java. Cada *thread* subdivide sua parte em pedaços que podem ser processados individualmente por um SPE,

onde estes pedaços são ordenados via quicksort. Quando os SPEs terminam os quicksorts, o programa principal no PPE utiliza o merge sort iterativamente para combinar todos os sub-vetores em um vetor final inteiramente ordenado.

A entrada para o terceiro programa consistiu em um vetor com um milhão de inteiros entre 0 e 10000, gerados aleatoriamente.

O quarto programa obtém o valor mínimo de um vetor de inteiros. Cada SPE encontra o mínimo de uma parte do vetor inicial e o mínimo geral é então obtido no PPE. Como entrada foi usado um vetor com 30 milhões de inteiros. O heap da JVM do SPE foi configurado com 47 KB e cada tarefa processou 11264 inteiros de uma vez no SPE.

O quinto programa realiza a soma de duas matrizes. Cada SPE fica responsável pela obtenção de uma coluna da matriz final, ou seja, faz a soma de duas colunas, uma de cada matriz de entrada, nas mesmas posições relativas dentro de sua matriz de origem. Nesta execução foram somadas duas matrizes iguais de 2048 colunas com 11264 inteiros cada uma. O heap da JVM do SPE foi configurado com 47 KB.

O sexto programa modifica o quinto da seguinte forma: além da soma de dois elementos da coluna atual, uma série de operações aritméticas são efetuadas para simular um processamento mais intenso por elemento da matriz final. A cada elemento da matriz final obtido, são realizadas também 32 iterações, sendo que cada uma destas consiste de 2 multiplicações, 2 divisões e 2 somas, todas com números inteiros. As matrizes de entrada tinham 32 x 11264 elementos e o tamanho do heap foi o mesmo que no quinto programa.

Os gráficos dos tempos de execução do segundo ao sexto programas, nas sete configurações da JAM_PPU e na configuração base da JamVM são mostrados nas Figuras 4.7 a 4.11. Pode-se observar nitidamente que esses gráficos têm todos a mesma forma do obtido com o programa de processamento de imagem (Figura 4.5). Esse resultado já era esperado, uma vez que todos esses algoritmos utilizam a decomposição geométrica em suas versões paralelas.

A Tabela 4.3 lista os tempos de execução de todas as configurações para os programas apresentados anteriormente.

Consolidando-se os resultados obtidos com todos os programas, obtém-se o gráfico da Figura 4.12. Ele representa o ganho médio proporcionado pelo uso da JAM_PPU, conforme se aumenta o número de SPEs de zero até seis, com relação ao tempo base obtido com a JamVM original. Os valores individuais dos ganhos médios estão listados na Tabela 4.4. Confirmando o que já havia sido observado no primeiro programa, com três SPEs, em média, o desempenho já é melhor do que o da JamVM original (ganho médio de

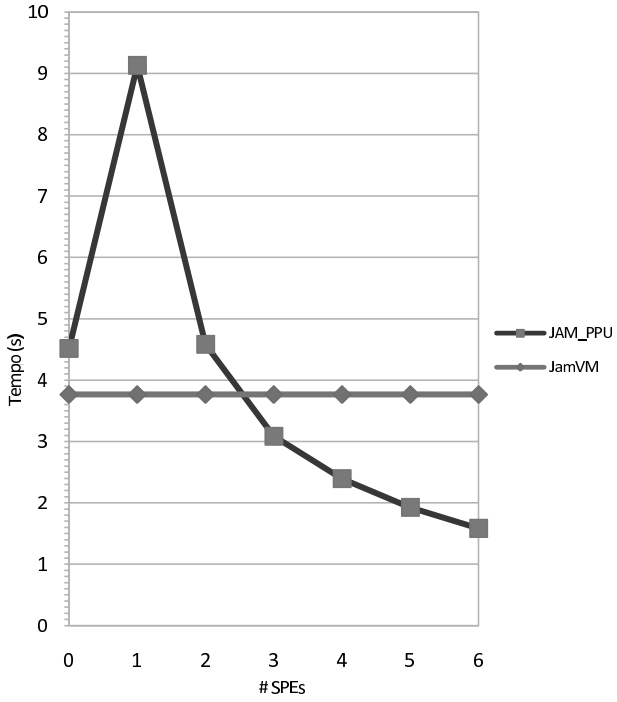

Figura 4.7: Tempo de execução versus número de SPEs - Conversão de caracteres

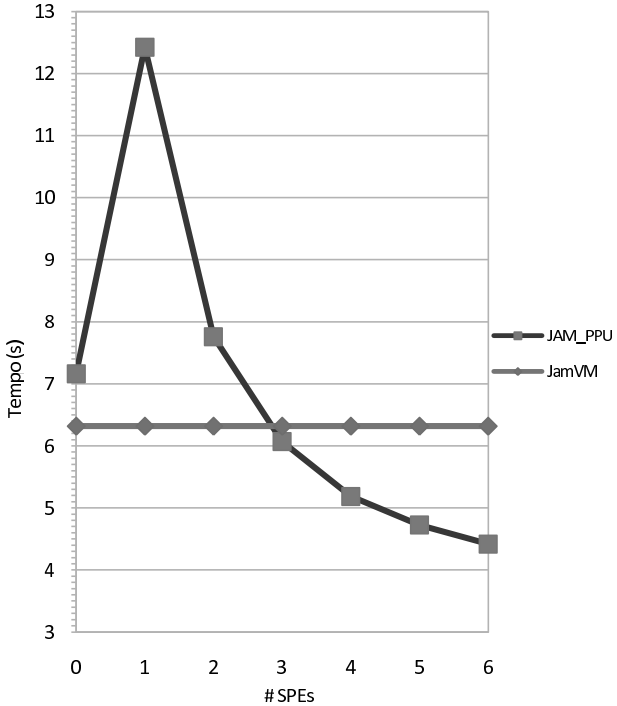

Figura 4.8: Tempo de execução versus número de SPEs - Ordenação de inteiros

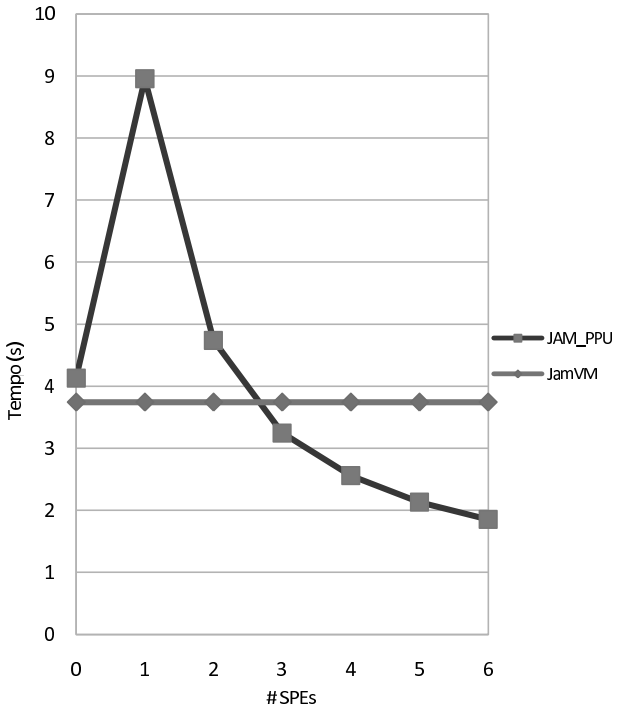

Figura 4.9: Tempo de execução versus número de SPEs - Obtenção do mínimo

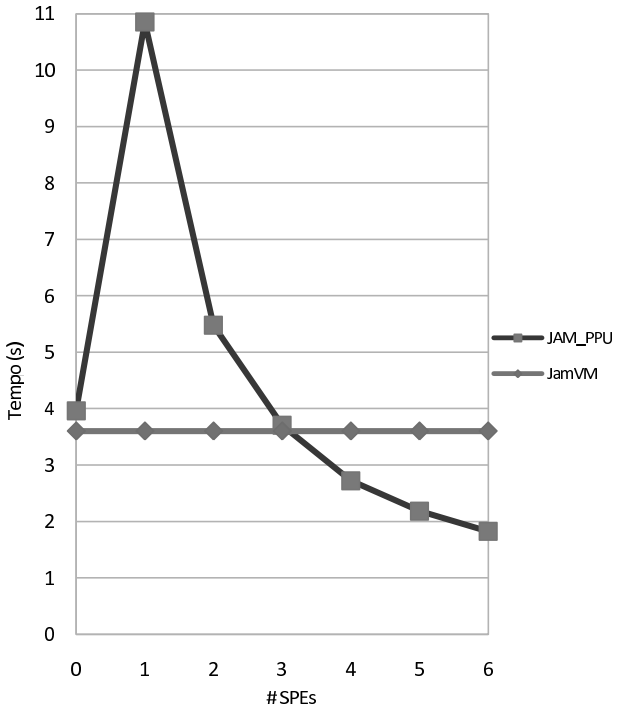

Figura 4.10: Tempo de execução versus número de SPEs - Soma de matrizes

Configuração	Proc. de Imagem (ms)	Conversão de caracteres (ms)	Ordenação de inteiros (ms)	Obtenção do mínimo (ms)	Soma de matrizes (ms)	Proc. intenso c/ matrizes (ms)
JamVM	3303	3765	6318	3744	3601	3743
JAM_PPU + 0 SPEs	3955	4516	7163	4131	3961	4301
JAM_PPU + 1 SPE	8761	9129	12424	8955	10851	8737
JAM_PPU + 2 SPEs	4439	4582	7760	4740	5476	4372
JAM_PPU + 3 SPEs	2982	3085	6068	3247	3703	3009
JAM_PPU + 4 SPEs	2260	2394	5184	2559	2721	2187
JAM_PPU + 5 SPEs	1822	1925	4724	2131	2182	1914
JAM_PPU + 6 SPEs	1519	1584	4417	1853	1823	1641

Tabela 4.3: Tempos de execução

Configuração	Ganho médio
JAM_PPU + 0 SPEs	0,87
JAM_PPU + 1 SPE	0,41
JAM_PPU + 2 SPEs	0,78
JAM_PPU + 3 SPEs	1,12
JAM_PPU + 4 SPEs	1,46
JAM_PPU + 5 SPEs	1,74
JAM_PPU + 6 SPEs	2,04

Tabela 4.4: Ganho médio

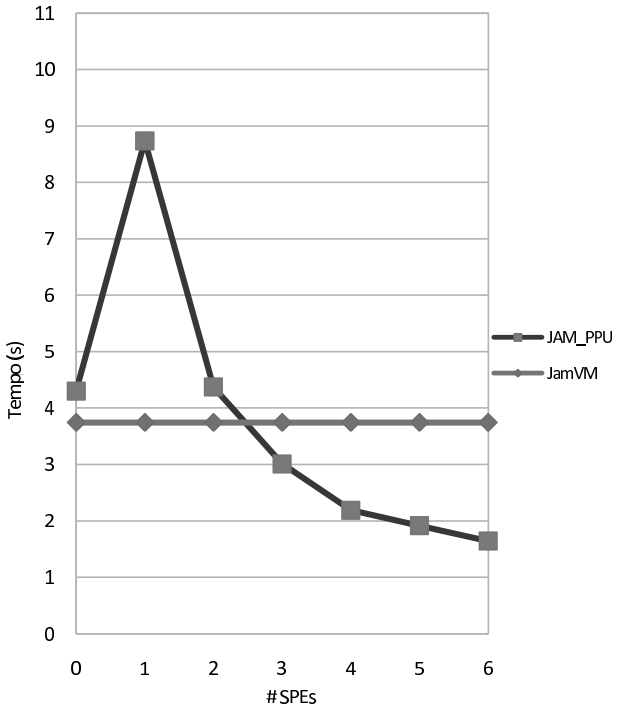

Figura 4.11: Tempo de execução versus número de SPEs - Processamento intenso com matrizes

1,12), enquanto com seis SPEs, em média, o desempenho dobra aproximadamente (ganho médio de 2,04).

4.4 Melhorando o desempenho do interpretador no SPE

Conforme mencionado na seção 3.2.2, as otimizações opcionais da JamVM foram desligadas na JVM do SPE para economizar memória. Contudo, é possível ligar algumas delas, se desejado. O importante é analisar o efeito final no desempenho do programa, pois as otimizações aumentam a velocidade de interpretação mas, como em geral o código da JVM também aumenta, o espaço disponível para todas as outras áreas da JVM é reduzido na mesma proporção. Com isso, tipicamente o que ocorre é uma diminuição da quantidade de dados que uma tarefa processa de cada vez e um consequente aumento no número de tarefas executadas.

Como exemplo do tipo de ganho de desempenho que pode ser obtido com a utilização

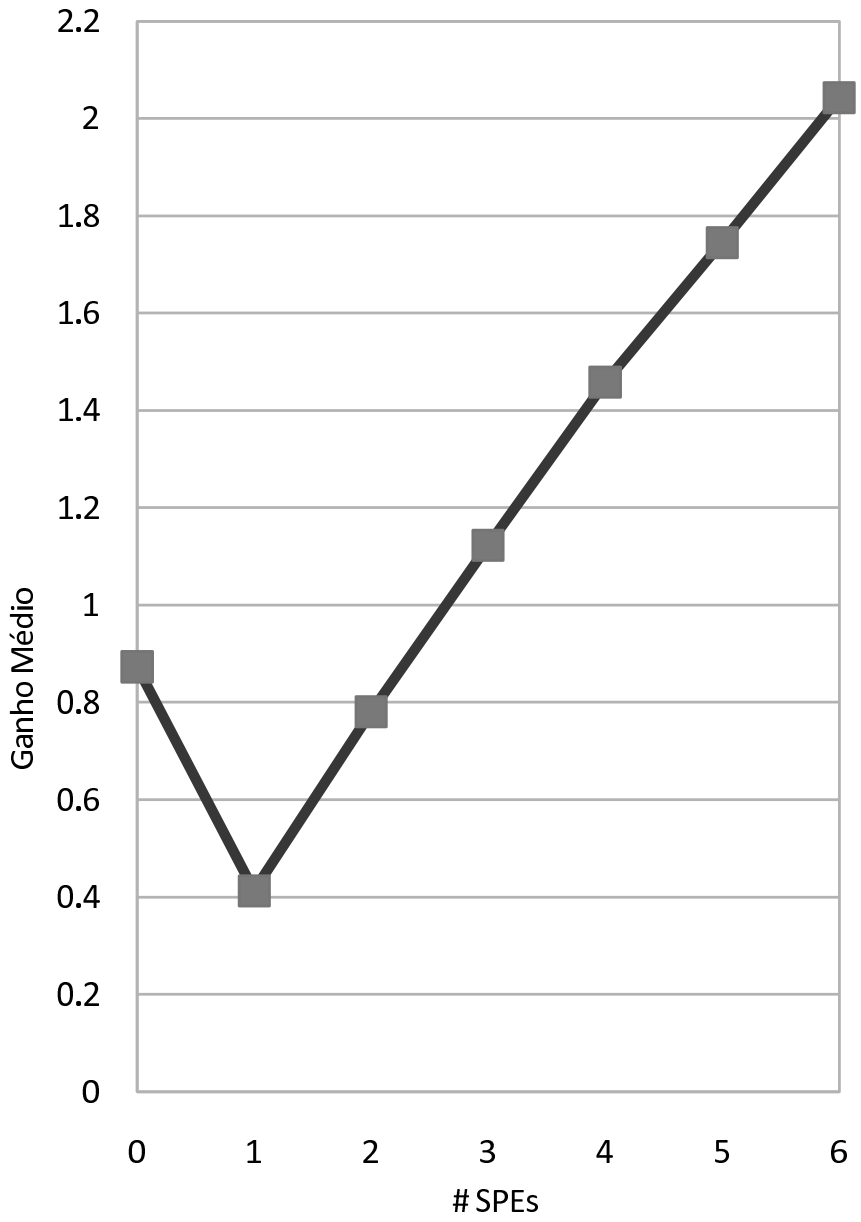

Figura 4.12: Ganho médio versus número de SPEs

de algumas dessas otimizações, o programa de processamento de imagem foi executado novamente nas configurações com um ou mais SPEs. Desta vez as otimizações *threaded interpreter* e *cache* do topo da pilha de operandos (ver seção 3.2.1) foram ligadas na JamVM do SPE. Todos os parâmetros de invocação do programa original foram mantidos. No caso do programa de processamento de imagem não foi necessário reduzir a quantidade de dados em cada tarefa pois já havia uma certa folga nesse sentido.

Configuração	Sem Otimizações (ms)	Com otimizações (ms)
JAM_PPU + 1 SPE	8761	6959
JAM_PPU + 2 SPEs	4439	3546
JAM_PPU + 3 SPEs	2982	2380
JAM_PPU + 4 SPEs	2260	1827
JAM_PPU + 5 SPEs	1822	1464
JAM_PPU + 6 SPEs	1519	1265

Tabela 4.5: Tempos de execução do programa de processamento de imagem com e sem otimizações na JamVM do SPE

Os tempos de execução obtidos no processamento de imagem com as otimizações da JamVM mencionadas anteriormente estão listados na Tabela 4.5, ao lado dos valores correspondentes da primeira execução, sem otimizações. A melhora de desempenho observada, em média, foi de 19,4%. Esse resultado, bastante expressivo, indica ser vantajoso o uso das otimizações no interpretador da JamVM, mesmo com a redução da memória disponível para outras partes da JVM que não seu próprio código. O fato de neste teste não ter sido necessário reduzir a carga de dados por tarefa para acomodar o aumento no código da JVM não invalida os resultados pois, conforme indicado na próxima seção, variações expressivas no tamanho dos dados das tarefas não alteram significativamente o tempo de execução das mesmas.

4.5 Custo da interpretação

O tempo total de execução de uma tarefa, medido pela duração da invocação de `SPETaskDispatcher.executeTask()`, é dominado pela interpretação do bytecode no SPE (ver a Figura 3.3 para uma sequência de todos os passos envolvidos na invocação de `executeTask`). Isto foi verificado através de um novo teste no qual se aumentou o total de dados contidos em cada tarefa, de 3 para 4 linhas, no programa de processamento de imagem. Essa alteração representa um aumento de 33,33% na carga de dados de cada

tarefa e uma redução correspondente no número de iterações (tarefas executadas). Não foi observada melhora significativa, já que o tempo total de interpretação foi praticamente o mesmo, enquanto o tempo consumido pelas transferências via DMA e pela serialização e desserialização manteve-se desprezível nos dois casos (com 3 e 4 linhas de imagem por tarefa).

Um último teste feito para avaliar o desempenho da solução proposta foi compará-lo ao de uma JVM tradicional com compilação JIT. Com esse objetivo foi usada a CACAO [10] para executar a versão sequencial do programa de processamento de imagem. O tempo de execução na CACAO foi 1105 ms, melhor inclusive que o resultado da JAM_PPU configurada com 6 SPEs e compilada com algumas otimizações. Vale notar que esse tempo é aproximadamente um terço do obtido com a JamVM original, o que demonstra claramente a superioridade da execução de código nativo (mesmo com o tempo adicional gasto em compilação JIT) sobre a execução puramente interpretada. No entanto, este resultado não invalida este trabalho. Ele apenas indica um potencial ainda maior de ganho de desempenho quando o modelo de execução proposto aqui for adaptado a uma implementação baseada em JIT. Esse ganho potencial, por sinal, já era de se esperar.

Neste capítulo foi demonstrada a viabilidade do novo modelo de execução para Java no Cell BE através dos resultados obtidos com alguns programas implementados de acordo com esse modelo. A implementação e a execução de um programa de processamento de imagem foi descrita em detalhe. Também se indicou o ganho de desempenho obtido com o emprego de algumas otimizações no interpretador da JamVM. Finalmente, identificou-se o custo da interpretação do bytecode como um grande impedimento à obtenção de resultados mais expressivos em termos de desempenho.

Capítulo 5

Conclusões

A contribuição principal deste trabalho foi propor um novo modelo de execução para programas Java no processador Cell BE. Assim como as outras soluções existentes, a nova proposta almeja aproximar o mundo Java (linguagem muito popular, grande número de programadores) do mundo Cell BE (alto poder de processamento, facilmente acessível através do PlayStation 3). Ela abstrai a programação de baixo nível, utilizada tipicamente no desenvolvimento de aplicações para o Cell BE, expondo os SPEs através de Java puro.

A nova alternativa permite que se explore o poder de processamento dos SPEs lançando uma JVM separada em cada um deles. O uso de múltiplas JVMs faz com que essa abordagem se assemelhe a sistemas Java distribuídos. Ao contrário do que ocorre comumente com estes, porém, não se usa RMI para a comunicação entre os nós do sistema e nem as bibliotecas padrão de Java para serialização. As transferências de dados entre as JVMs do PPE e dos SPEs se dão via DMA, enquanto que pequenas mensagens de controle são carregadas pelo serviço de *mailbox* do Cell BE. Já a serialização e desserialização de objetos trocados entre os núcleos foi uma implementação totalmente nova, especializada para uso na JamVM, a máquina virtual escolhida como base para o projeto.

O modelo proposto tem como característica central a definição de classes tarefa, que contêm o código a ser executado em um SPE. Quando um objeto tarefa é criado, ele deve ser inicializado com todos os dados necessários para a execução dessa tarefa. Uma tarefa é então enviada para execução em um SPE simplesmente invocando-se o método `SPETaskDispatcher.executeTask`. O uso de tarefas permite ao programador definir exatamente qual código irá executar nos SPEs, característica única da solução aqui proposta. Esse controle fino tem o potencial de proporcionar uma melhor utilização dos SPEs, uma vez que alocações totalmente automáticas dificilmente atenderão satisfatoriamente as ne-

Solução	JVMs	SPE executa	JIT	Modelo
Garbage collection	Única	Tarefa interna	N/A	N/A
Compilação JIT	Única	Tarefa interna	N/A	N/A
Georg Sorst	Única	Código do usuário	Sim	DSM
Hera-JVM	Única	Código do usuário	Sim	DSM
CellVM	Múltiplas	Código do usuário	Não	DSM
JAM_PPU	Múltiplas	Código do usuário	Não	Memória Distribuída

Tabela 5.1: Soluções para Java no Cell BE e a nova alternativa.

cessidades de todos os programas.

Como o modelo expõe explicitamente os SPEs ao programador, ele adota uma abordagem de memória distribuída. Este é outro ponto divergente com relação às outras soluções existentes, que adotam modelos do tipo DSM. A Tabela 5.1 contém as mesmas soluções apresentadas na seção 2.4, mas acrescenta a nova alternativa proposta, chamada aqui de JAM_PPU, para fins de comparação.

Foi demonstrada também a viabilidade do novo modelo, apresentando-se os tempos de execução de vários programas Java adaptados ao mesmo. Esses programas consistentemente apresentaram um menor tempo de execução do que a versão original sequencial, já a partir da configuração com três SPEs (em média). Com seis SPEs, o novo modelo proporcionou um melhora de desempenho média de aproximadamente duas vezes. Ficou claro que o custo da interpretação do bytecode é bastante elevado e que a mesma solução baseada em uma implementação com compilação JIT deve apresentar um desempenho final significativamente melhor.

É importante ressaltar que o modelo proposto aqui é genérico e, portanto, pode ser empregado em sistemas com outros processadores de múltiplos núcleos heterogêneos.

Finalmente, alguns dos possíveis tópicos para trabalhos futuros são apresentados a seguir:

- Aplicação do novo modelo a um sistema baseado em JIT para Java no Cell BE;
- Extensão do modelo para permitir que um SPE invoque uma nova tarefa em outro SPE diretamente - isto pode ser interessante para algoritmos do tipo *pipeline*, em que o mesmo lote de dados sofre diferentes transformações consecutivamente, uma em cada SPE;
- Uso de cache implementado em software, tanto para a área de código quanto a área

de dados nos SPEs, o que minimizaria o problema da escassez de memória nesses núcleos;

- Invocação de métodos nativos localizados em bibliotecas carregadas dinamicamente pelo PPE.

O modelo de execução para Java no Cell BE descrito nesta dissertação foi tema de um artigo [15] apresentado, em agosto de 2009, no XIII Simpósio Brasileiro de Linguagens de Programação.

Referências Bibliográficas

[1] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal*, 44(2):399–417, 2005.

[2] Brian Amedro, Vladimir Bodnartchouk, Denis Caromel, Christian Delbe, Fabrice Huet, and Guillermo Taboada. Current State of Java for HPC. Technical report, INRIA, 2008.

[3] Gabriel Antoniu, Luc Bougé, Philip Hatcher, Mark MacBeth, Keith McGuigan, and Raymond Namyst. The Hyperion system: Compiling multithreaded Java bytecode for distributed execution. *Parallel Computing*, 27(10):1279–1297, 2001.

[4] Y. Aridor, M. Factor, A. Teperman, T. Eilam, and A. Schuster. A high performance cluster JVM presenting a pure single system image. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 168–177, New York, NY, USA, 2000. ACM.

[5] Yariv Aridor, Michael Factor, and Avi Teperman. cJVM: A Single System Image of a JVM on a Cluster. In *ICPP '99: Proceedings of the 1999 International Conference on Parallel Processing*, page 4, Washington, DC, USA, 1999. IEEE Computer Society.

[6] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[7] Richard G. Baldwin. Processing Image Pixels Using Java: Controlling Contrast and Brightness. *developer.com*, 2004.

[8] Richard G. Baldwin. Processing Image Pixels Using Java: Getting Started. *developer.com*, 2004.

[9] Mark Bull and Scott Telford. Programming Models for Parallel Java Applications. Technical report, Edinburgh Parallel Computing Centre, 2000.

[10] CACAOVM. http://www.cacaovm.org/, 2008.

[11] Chen-Yong Cher and Michael Gschwind. Cell GC: Using the Cell Synergistic Processor as a Garbage Collection Coprocessor. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 141–150, New York, NY, USA, 2008. ACM.

[12] Michael Factor, Assaf Schuster, and Konstantin Shagin. A platform-independent distributed runtime for standard multithreaded Java. *International Journal of Parallel Programming*, 34(2):113–142, 2006.

[13] Kevin Fenwick. A Performance Analysis of Java Distributed Shared Memory Implementations. Master's thesis, Adelaide University, Department of Computer Science, 2001.

[14] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[15] Francisco Hoyos, Bruno Teles, and Rodolfo Azevedo. An Execution Model for Java on Cell BE Processor. In *XIII Simpósio Brasileiro de Linguagens de Programação*, pages 75–88, 2009.

[16] IBM. Software Development Kit for Multicore Acceleration Version 3.1 - Programmer's Guide, http://www.ibm.com/developerworks/power/cell/, 2008.

[17] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, 2005.

[18] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[19] Robert Lougher. JamVM, A Compact Virtual Machine, http://jamvm.sourceforge.net/, 2003.

[20] Jason Maassen, Rob Van Nieuwpoort, Ronald Veldema, Henri Bal, Thilo Kielmann, Ceriel Jacobs, and Rutger Hofman. Efficient Java RMI for parallel programming. *ACM Transactions on Programming Languages and Systems*, 23(6):747–775, 2001.

[21] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, 2004.

[22] Ross McIlroy and Joe Sventek. Hera-JVM: Abstracting Processor Heterogeneity Behind a Virtual Machine. In *HotOS XII: Proceedings of the 12th Workshop on Hot Topics in Operating Systems*, 2009.

[23] Christian Nester, Michael Philippsen, and Bernhard Haumacher. A more efficient RMI for Java. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 152–159, New York, NY, USA, 1999. ACM.

[24] Albert Noll, Andreas Gal, and Michael Franz. CellVM: A Homogeneous Virtual Machine Runtime System for a Heterogeneous Single-Chip Multiprocessor. Technical report, University of California, Irvine, 2006.

[25] Michael Philippsen, Bernhard Haumacher, and Christian Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, 2000.

[26] Michael Philippsen and Matthias Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.

[27] Yukihiko Sohda, Hidemoto Nakada, and Satoshi Matsuoka. Implementation of a portable software DSM in Java. In *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 163–172, New York, NY, USA, 2001. ACM.

[28] Georg Sorst. Java on Cell BE. Master's thesis, Aachen University, Department of Electrical Engineering and Information Technology, 2007.

[29] Sun Microsystems, Inc. Designing a Remote Interface, The Java Tutorials, RMI Trail, http://java.sun.com/docs/books/tutorial/rmi/designing.html, 2008.

[30] Guillermo L. Taboada, Juan Touriño, and Ramón Doallo. Efficient Java Communication Protocols on High-speed Cluster Interconnects. In *In Proc. 31st IEEE Conf. on Local Computer Networks (LCN 06)*, pages 264–271, 2006.

[31] Eli Tilevich and Yannis Smaragdakis. Portable and efficient distributed threads for Java. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 478–492, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[32] Tiobe Software. TIOBE Programming Community Index, http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, 2009.

[33] R. Veldema, R.A.F. Bhoedjang, and H.E. Bal. Distributed Shared Memory Management for Java. In *Proceedings of the Sixth Annual Conference of the Advanced School for Computing and Imaging (ASCI 2000)*, pages 256–264, 2000.

[34] Bill Venners. *Inside the Java 2 Virtual Machine*. McGraw-Hill, second edition, 1999.

[35] Kevin Williams, Albert Noll, Andreas Gal, and David Gregg. Optimization Strategies for a Java Virtual Machine Interpreter on the Cell Broadband Engine. In *CF '08: Proceedings of the 2008 conference on Computing frontiers*, pages 189–198, New York, NY, USA, 2008. ACM.

[36] Ann Wollrath, Roger Riggs, and Jim Waldo. A Distributed Object Model for the Java System. In *COOTS'96: Proceedings of the 2nd conference on USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 17–17, Berkeley, CA, USA, 1996. USENIX Association.