3
O Modelo Numina

Nossa hipótese é de que o uso de um ambiente de maior abstração, baseado em camadas construídas com uma linguagem de programação interpretada, permite oferecer mais facilidades para o desenvolvedor de aplicações paralelas. O uso de uma linguagem interpretada apresenta um impacto no desempenho, mas acreditamos que isso não deva comprometer o desempenho necessário para a solução do problema do usuário, além de poder ser tratado empregando-se técnicas de otimização às camadas do modelo e no ambiente de interpretação da linguagem.

Com base em nossa hipótese vamos discutir e apresentar, neste capítulo, as questões referentes ao uso de um modelo em camadas para abstrair os detalhes do paralelismo fornecendo serviços de forma gradativa de acordo com as diretrizes apresentadas em (Skillicorn1998).

Dentre as alternativas relacionadas à abstração, o uso de máquinas virtuais se apresenta como um fator importante para criar uma camada que isole o programador dos detalhes das arquiteturas que serão usadas.

3.1
Camadas de Abstração de Numina

Nosso modelo é baseado em um padrão Master/Worker e suporta tanto os modos de trabalho SPMD quanto MPMD, sendo que nossa implementação do protótipo não suporta a comunicação entre os trabalhadores. Esse tipo de abordagem abrange grande quantidade de problemas de paralelismo e portanto se mostra propícia aos nossos estudos.

A visão de camadas do nosso modelo propõe abstrair os detalhes envolvidos no desenvolvimento de aplicações paralelas. Identificamos quatro camadas descritas a seguir:

1. Camada de arquitetura de processadores – com o objetivo de diminuir o impacto da diversidade de processadores, nossa primeira camada é responsável por tratar a arquitetura, apresentando para o desenvolvedor de aplicações a abstração *Core* que representa um núcleo de execução. O desenvolvimento de novas plataformas de *hardware* busca prover

máquinas cada vez mais poderosas, no entanto isso implica em explorar alternativas de máquina que possam melhorar o desempenho. Em alguns casos, como o do Cell/BE e das GPUs, a arquitetura é totalmente diferente das máquinas tradicionalmente utilizadas o que leva a uma grande curva de aprendizado para o desenvolvedor de aplicações que precisa lidar com todos os novos detalhes. Nesta camada o desenvolvedor de sistema pode aplicar todas as otimizações que forem cabíveis a fim de obter um ambiente que utilize o máximo possível de facilidades oferecidas pela máquina.

2. Camada de controle e configuração – essa camada oferece a abstração *CoreGroup* que representa um grupo de núcleos de execução formando um processador paralelo multi-núcleo virtual. São fornecidos para o desenvolvedor de aplicações mecanismos para controlar a execução de suas aplicações indicando que trechos de código e dados precisam ser carregados no CoreGroup. Esses detalhes passam a ser implementados na API do modelo de maneira transparente. Surge nessa camada a primeira referência a configuração de políticas de passagem de parâmetros para as funções do usuário.

3. Camada de comunicação – essa camada define os mecanismos de comunicação tanto entre núcleos quanto entre máquinas distintas através da abstração *Cluster*. O desenvolvedor de aplicações usa a mesma API que utiliza para CoreGroup no desenvolvimento de aplicações para a rede. Os detalhes relativos a endereços de máquinas são isolados em uma configuração de ambiente de forma que não afetem o desenvolvimento, além disso tanto o número de CoreGroups conectados quanto o número de Cores em cada CoreGroup podem ser recuperados dinamicamente o que permite preparar as aplicações para possíveis variações no ambiente de execução.

4. Camada de programação – na parte de mais alto nível do modelo Numina o desenvolvedor de aplicações utiliza uma API simples para configurar e controlar o desenvolvimento de aplicações paralelas. O desenvolvedor pode acessar as duas abstrações principais do modelo que são CoreGroup e Cluster sem que precise se preocupar com os detalhes específicos da arquitetura.

Para atingir o objetivo de criar uma camada de separação entre o programador e a arquitetura da máquina, propomos basear o modelo Numina em uma linguagem de programação interpretada, de modo que uma máquina

virtual possa ser carregada em cada núcleo disponível. A linguagem escolhida foi Lua (Ierusalimschy1996) por apresentar uma máquina virtual pequena e ter fácil integração com a linguagem C.

Com o uso de uma linguagem interpretada o ciclo de programação/teste torna-se mais ágil, pois cada alteração pode ser rapidamente executada. Uma vez que as máquinas virtuais são carregadas em núcleos diferentes, o usuário pode carregar programas Lua que serão executados diretamente em cada um destes núcleos. Não há necessidade de compilação especial, nem de compiladores específicos. A plataforma Numina esconde as tarefas tediosas e fornece ao programador uma interface simples, consistente e livre de compilação, porém esse benefícios tem um custo de desempenho.

Inicialmente o modelo Numina surgiu como um estudo para simplificar o uso de uma arquitetura específica multi-núcleo, o processador Cell/BE, e posteriormente evoluiu para se tornar um modelo mais abstrato para o desenvolvimento de aplicações paralelas. Atualmente contamos com duas implementações, uma para Cell/BE e outra para arquitetura Intel/AMD multi-núcleo. A parte mais próxima da máquina precisa ser modificada e compilada de forma específica, funcionando como um *driver*. Já as camadas superiores são escritas em Lua, permitindo seu reuso mesmo entre diferentes arquiteturas.

3.2 A API Numina - Um Exemplo

A API Numina conta com um pequeno conjunto de comandos que têm como objetivo permitir ao desenvolvedor de aplicações controlar o ambiente de execução de seu código paralelo. Como propomos tratar o paralelismo tanto em máquinas paralelas quanto em ambientes *cluster* buscamos uma forma de oferecer uma API o mais consistente possível em relação a visão do programador.

Para simplificar o entendimento listamos abaixo um trecho de código Numina.

Listagem 3.1: Exemplo de código Numina

```
1 require  "coregroup"
2 group = CoreGroup:new(1)
3 ffunc=[[function func(id, x)
4            return id * x
5         end]]
6 function main()
7         group:load(ffunc, "func", "scatter", "broadcast")
8         totalCount = group:getCount()
```

```
 9         for i=1,totalCount do
10             parmsid[i] = id
11             id = id + 1
12         end
13         group:func(parmsid,10)
14         result = group:reduce("SUM")
15         print("result="..result)
16 end
17 main()
18 group:close()
19 group:exit()
```

Na linha 1, importamos a biblioteca que representa um processador multi-núcleo. Em seguida, criamos um grupo na linha 2 para representar o processador. Através desse processo são abertas as *threads* que vão executar as máquinas virtuais Lua, sendo uma para cada núcleo do processador.

A próxima etapa é definir o código que precisa ser carregado em cada núcleo, isso é feito com programação Lua tradicional e está representado da linha 3 a 5.

Como forma de simplificar a legibilidade, criamos uma nova função na linha 6 que concentrará todo o código de coordenação da aplicação. A primeira tarefa deste código é carregar a função do usuário em cada um dos núcleos, isso é realizado através da chamada `load`, que recebe como parâmetros o texto da função, o nome que deve ser usado em cada núcleo, e uma lista com as políticas de distribuição de argumentos.

Atualmente contamos com duas políticas: *scatter* e *broadcast*. Além disso o usuário pode prover novas políticas conforme sua necessidade. Esse mecanismo é utilizado para definir como distribuir os dados que são passados como argumentos para a chamada de função e serão abordados na Seção 3.9.

Das linhas 8 a 12, criamos uma tabela contendo identificadores dos núcleos através de um laço que consulta a quantidade de núcleos dinamicamente. Isso permite o desenvolvimento de aplicações mais portáteis porque o algoritmo pode ser parametrizado para uma quantidade indefinida de núcleos.

Na linha 13, a função é chamada sobre o grupo, isso é possível porque a API Numina incorpora a função carregada como se fizesse parte do próprio CoreGroup, o que facilita a leitura do desenvolvedor. Isso é possível graças aos recursos da linguagem Lua que permitem incorporar uma função a uma estrutura de dados. Como argumentos para a chamada, são passados a tabela criada anteriormente e um literal numérico. Como a política do primeiro parâmetro foi definida como *scatter*, a tabela é percorrida e cada um de seus

elementos é enviado para um núcleo, espera-se que o tamanho da tabela seja compatível com a quantidade de núcleos. Já o segundo parâmetro foi definido como *broadcast*, assim o valor literal é copiado para todos os núcleos.

Uma vez executada a função precisamos recuperar os valores e a API oferece um conjunto de facilidades para isso. Entre as opções usamos, na linha 14 do código, a função *reduce* passando o parâmetro *SUM*. Dessa forma o próprio ambiente se encarrega de recuperar o retorno de cada núcleo e realizar a soma dos resultados parciais. Outras opções de retorno são: *gather*, *reduce("PROD")*, *reduce("MAX")* e *reduce("MIN")*.

Por fim usamos as funções *close* e *exit* para encerrar as máquinas Lua de cada núcleo e encerrar as *threads* do ambiente.

3.3 Arquitetura Numina

O modelo que propomos é ter uma máquina virtual Lua por núcleo de processamento de modo que seja possível carregar funções diferentes em cada núcleo. Desta forma, pode-se especializar um ou mais núcleos em um subconjunto de tarefas específicas, se necessário, ou carregar as mesmas funções em todos os núcleos e implementar o padrão SPMD(Mattson2004). A Figura 3.1 mostra uma visão geral da arquitetura Numina.

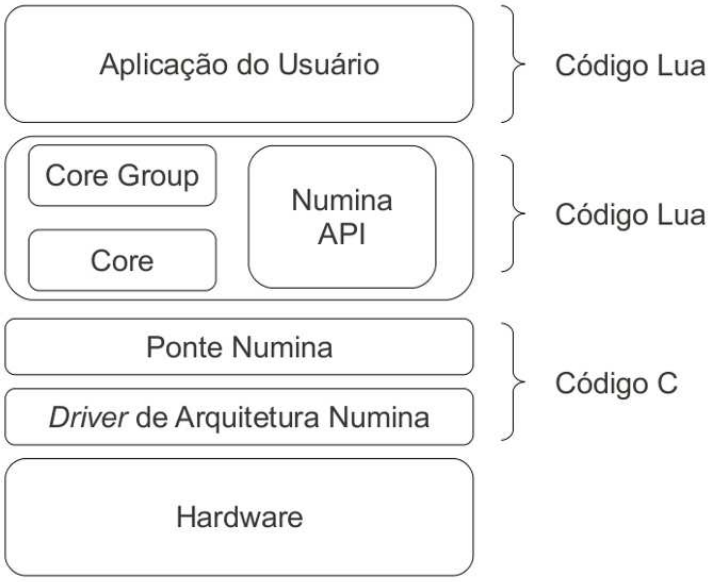

Figura 3.1: Arquitetura do modelo Numina

Seguindo as camadas apresentadas na Figura 3.1 de baixo para cima temos:

- *Driver* de Arquitetura Numina – essa camada mais próxima da máquina é responsável por implementar os mecanismos necessários ao modelo de acordo com as características disponibilizadas pela máquina. Nesta parte do código podemos aplicar otimizações específicas a fim de obter o melhor desempenho possível.
- Ponte Numina – essa camada é responsável por prover os serviços básicos do modelo para uma dada arquitetura, mas sem contemplar otimizações específicas. De uma maneira geral essa parte do código precisa ser compilada para cada arquitetura suportada, mas sem modificações no código.
- API Numina – primeira camada do modelo implementada na linguagem Lua, e é responsável por fazer a ligação entre o ambiente Lua e a biblioteca de baixo nível escrita em C. Juntamente com as duas camadas anteriores, forma o ambiente de execução Numina, já sendo possível ao usuário escrever aplicações, porém seria necessário controlar toda a execução sem grandes facilidades.
- Core – essa camada abstrai o conceito de um núcleo de execução. Um núcleo é capaz de carregar e executar funções do usuário, verificar o estado da execução e retornar o resultado de uma execução. Essa abordagem é diferente da encontrada em (Augonnet2011) e (Tsoi2011) em que o usuário trabalha com o conceito de tarefa.
- CoreGroup – essa camada representa um grupo de núcleos, e disponibiliza o primeiro nível de políticas de distribuição de dados. O uso de políticas de distribuição de dados permite ao usuário usar uma chamada única a uma função passando um conjunto de dados, que, de acordo com a política, é distribuído para os núcleos executores. Com CoreGroup o usuário tem acesso aos mesmos serviços da camada anterior, porém com uma visão de grupo, assim é necessário pensar em como disponibilizar os dados de processamento considerando uma política de distribuição. Essa camada oferece algumas funcionalidades para simplificar a coleta de resultados contando com opções de *gather* e *reduce* similares as oferecidas em MPI.
- Aplicação do Usuário – essa camada é a aplicação do usuário efetivamente. É necessário carregar o código da camada anterior e usar sua abstração, juntamente com as funções que resolvem o problema da aplicação.

Um ponto chave dessa arquitetura é a junção do *Driver* de Arquitetura, que concentra os detalhes específicos da máquina, com a ponte Numina que oferece um primeiro nível de abstração sobre a máquina. Utilizamos esse grau

de isolamento para que seja possível portar a implementação do modelo para diferentes arquiteturas de forma mais conveniente para o desenvolvedor de sistema. Propomos assim ouso de uma API de paralelismo unificada para diferentes arquiteturas, desde que estas ofereçam a capacidade de mapeamento necessárias. Caso a arquitetura destino ofereça a capacidade computacional necessária, o desenvolvedor de sistema só precisa escrever o *driver* e recompilar a ponte Numina para oferecer o suporte à nova máquina. Em especial as placas de GPU não oferecem a possibilidade desse mapeamento porque seus núcleos não são capazes de executar uma máquina virtual Lua, sendo direcionados para a execução de operações matemáticas em um formato semelhante ao de processadores de array.

3.4 Ambiente de Execução Numina - Modelo de Cópia de Memória

Para atingir o objetivo, Numina tem que carregar o programa do usuário em cada máquina Lua dos núcleos de processamento. Para isso, o código Lua que tem que ser carregado e passado como um parâmetro para uma função da API Numina chamada `loaddump`. Em seguida, copiamos o código Lua da memória principal para cada núcleo.

Lua se destina a ser usada como uma linguagem de extensão para C. Assim, é possível manipular seu estado de execução a partir de um programa escrito em C, e, especificamente, carregar uma string como um programa em um estado[1] de máquina Lua. Esta facilidade é explorada aqui como o meio para carregamento do programa do usuário, que é decomposto em funções. O ambiente de execução Numina conta com uma máquina Lua principal que interpreta a aplicação do usuário e controla as máquinas executoras através de comandos Numina. Uma vez que o segmento em execução no núcleo recebe a string de código, ele verifica se o comando associado é `loaddump`. Se for, a string de código é carregada como código Lua na máquina virtual correspondente.

Uma vez que as funções estejam carregadas em núcleos diferentes, o programador pode chamá-las diretamente do programa Lua com a função Numina `execute`. Isso é feito usando a interface C para passar os argumentos para o estado da máquina Lua e depois recuperar o resultado, se for o caso. Lua usa uma pilha de comunicação com C, assim, quando uma execução de função retorna valores, estes são disponibilizados na pilha e o programador deve recuperá-los. Para o programador Numina, isso é realizado através de uma chamada à `getresults`.

[1]Uma máquina virtual Lua mantém um estado de execução na forma de uma pilha que pode ser manipulada em um código C, isso facilita a integração entre as duas linguagens.

Quanto ao desempenho, vamos analisar os resultados no Capítulo 4, mas existe um impacto tal como o experimentado entre C e Lua[2]. O programador pode fazer chamadas a bibliotecas em C que tenham sido incorporadas à máquina Lua. No caso da arquitetura Intel, essas bibliotecas podem ser incorporadas também dinamicamente.

3.5 Camadas Numina de Abstração

Utilizamos um ambiente mínimo e nativo da máquina para disponibilizar os recursos para as camadas superiores de abstração. Essas camadas são implementadas em Lua para que possamos usufruir da portabilidade da própria linguagem e também para que possamos disponibilizar certos recursos mais dinâmicos tais como a incorporação de funções do usuário durante o carregamento.

Quando o usuário carrega uma função no ambiente Numina, tanto o Core quanto o CoreGroup criam uma entrada em suas estruturas internas com o mesmo nome da função de forma que o usuário possa fazer a chamada a função diretamente no CoreGroup. Essa facilidade diminuiu muito a complexidade do uso do ambiente paralelo porque o usuário passa a trabalhar com uma separação entre código de aplicação e código de configuração.

Na Figura 3.2 abaixo temos uma visão mais clara do uso do modelo Numina. Quando a aplicação do usuário é executada deve criar uma instância de CoreGroup, que por sua vez vai disponibilizar um conjunto de Cores ligados a um *thread* do sistema cada um. A quantidade de *threads* depende do sistema, no caso da implementação para Cell/BE temos 6, já para a plataforma Intel/AMD podemos ter variações da quantidade de núcleos e por isso essa quantidade pode ser configurada através de um arquivo do ambiente Numina.

Em alguns casos podemos, de acordo com a necessidade do usuário, trocar um pouco da facilidade de uso para atingir um melhor desempenho. Para isso oferecemos a biblioteca Native Arrays, escrita em C, que conta com um conjunto de operações de manipulação de arrays compiladas em linguagem de máquina de forma a utilizas as otimizações que estejam disponíveis na arquitetura utilizada. Em especial o operador dotproduct oferece um produto entre vetores de forma otimizada tanto em Intel/AMD quanto em Cell/BE, utilizando instruções SIMD em ambos os casos.

[2]Consideramos aqui o uso da linguagem Lua totalmente interpretada, ou seja, não compilado com JIT

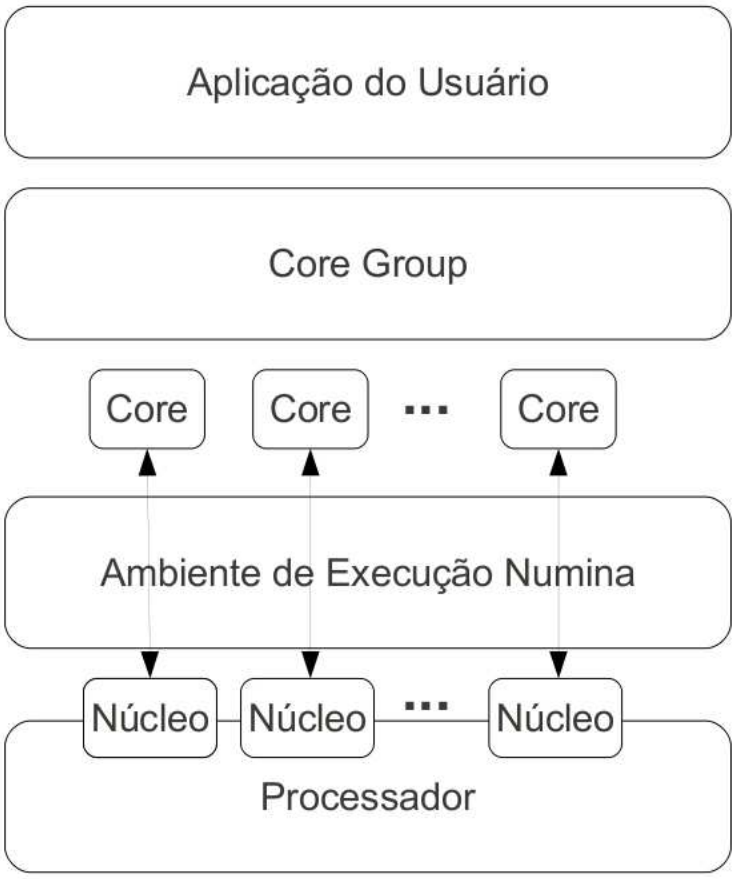

Figura 3.2: A execução de uma aplicação em Numina

Caso o programador precise atingir um melhor desempenho pode utilizar essa mesma abordagem e fornecer partes de seu algoritmo escritas em C e compiladas para a arquitetura destino, utilizando o ambiente Numina para coordenar a execução de sua aplicação com mais facilidade.

3.6 Cell Broadband Engine - Cell/BE

O Cell Broadband Engine (Cell/BE) (Kahle2005) é um processador multi-núcleo baseado na tecnologia PowerPC utilizado no Playstation 3 da Sony (Sony2005) e em servidores IBM BladeCenter (IBM2005). Ele foi desenvolvido para oferecer mais poder de processamento para aplicações multimídia e jogos. No entanto, devido à sua arquitetura multi-núcleo e da possibilidade de instalar um sistema operacional Linux, o Playstation 3 começou a ser usado como uma alternativa de baixo custo e alto desempenho em *clusters* (Rabbah2007) e (Kurzak2008). Essa alternativa está mais restrita agora que o processador foi descontinuado pela IBM, porém ainda existem muitas instalações que fazem uso dessa arquitetura, em especial do uso de consoles

Playstation 3. Em 2010 a Sony lançou uma atualização que removia a capacidade de rodar outro sistema operacional no Playstation 3 o que dificultou ainda mais o uso dessa arquitetura.

O Cell/BE é um chip multi-núcleo heterogêneo composto por um núcleo principal – o *Power Processor Element* (PPE) – e oito co-processadores, os *Synergistic Processing Units* (SPU). Os múltiplos núcleos não estão diretamente disponíveis para o sistema operacional, sendo assim, o programador deve estruturar o seu aplicativo de forma a usar explicitamente os núcleos SPU. Esta arquitetura oferece um grande poder de processamento e controle total sobre a máquina, mas ao mesmo tempo uma complexidade maior no desenvolvimento de aplicações.

O desenvolvimento de aplicações para o processador Cell/BE, embora seja muito interessante por conta de seu alto poder de processamento, é muito complexo e só atinge desempenho máximo se o desenvolvedor tiver um profundo conhecimento do sistema (Mueller2007, Kurzak2008, Rabbah2007, Kahle2005). Muitas atividades de controle são feitas diretamente pelo programador através de comandos de baixo nível, o que torna o desenvolvimento de aplicações muito caro e demorado.

Além de ser muita informação, é também improdutivo para o programador lidar com muitos detalhes que não fazem parte da aplicação. A compilação é realizada em um processo de três etapas que requer que o código seja compilado, em seguida convertido em uma biblioteca embutida, e finalmente ligado com o programa principal que deve ser compilado para o PPE. Para exemplos simples isto pode ser gerenciado manualmente, mas para qualquer aplicação maior, este processo se torna muito complexo. Para lidar com isso, recomenda-se que o programador use os *makefiles* fornecidos juntamente com o SDK da IBM. O que leva a copiar e adaptar esses *makefiles* e a estrutura de projeto para projeto, em um ciclo propenso a erros e improdutivo.

Após todas essas etapas, o programador pode executar o seu programa, ou no simulador do sistema ou em um processador real. Se algo der errado todo o processo deve ser executado novamente.

Com o objetivo inicial de simplificar este ambiente de desenvolvimento, foi feito um estudo sobre como criar uma API que encapsulasse essas tarefas para o programador, mas que não automatizasse as decisões sobre o desenvolvimento da aplicação em questão. Um outro objetivo do modelo era permitir sua evolução de forma transparente para o programador. O uso de uma linguagem interpretada se mostrou conveniente, já que permite aplicar modificações tanto na máquina virtual da linguagem quanto no ambiente de suporte a execução de forma independente das aplicações.

O processador Cell/BE deu origem ao modelo Numina, tendo sido a primeira arquitetura destino. Uma característica que foi determinante no uso da linguagem Lua foi que cada SPU tem apenas 256KB de memória local. Foi necessário modificar a máquina virtual Lua para deixar o máximo de memória livre possível. Também foi necessário compilar a máquina Lua para a arquitetura SPU usando diretivas para a otimização de espaço ao invés de desempenho. A máquina Lua é gerada como uma biblioteca que é ligada ao programa controlador do núcleo SPU. Já a máquina Lua que é carregada no PPE é uma máquina Lua tradicional compilada para a arquitetura PowerPC.

A infraestrutura da plataforma Numina é programada na linguagem C, que é a interface fornecida pelo processador Cell/BE. Esta infraestrutura tem os meios básicos para controlar as máquinas Lua em cada SPU. Cada SPU em seguida, executa o programa que contém a máquina Lua, e este programa é controlado através de comandos que são enviados juntamente com o código do aplicativo. Graças as facilidades oferecidas pela integração de Lua com C, que permite chamar código C em Lua e vice-versa, o uso de Lua como linguagem embutida se mostrou uma escolha natural.

O primeiro obstáculo a ser vencido foi a barreira de memória. Isto porque este processador utiliza um esquema de memória hierárquica, onde cada núcleo de processamento tem sua própria memória local de alta velocidade, mas de apenas 256KB.

O segundo obstáculo foi a questão de visibilidade dos núcleos. Neste processador existe uma diferenciação entre os núcleos, sendo 8 núcleos especiais para processamento intenso, que não ficam expostos para o sistema operacional. Dessa forma o desenvolvimento de aplicações se dá em várias etapas, começando com o desenvolvimento de funções que serão carregadas internamente, como bibliotecas estáticas, nesses núcleos. Para lidar com essa característica foi criado um pequeno módulo de controle que executa em cada um desses núcleos e que inicia a máquina virtual Lua modificada. A partir daí é iniciado um ciclo de comunicação com o processador principal que roda a aplicação a fim de receber as funções Lua e os dados para processamento.

O Cell/BE está equipado com um conjunto especial de registradores chamados de *mailboxes* que permitem a comunicação entre o PPE e SPUs. Cada SPU tem seu próprio conjunto de *mailboxes* sendo dois de entrada e um de saída e o trabalho sobre esses registradores funciona de forma bloqueante. Assim, usamos esses registradores para fins de sincronização, garantindo os comandos adequados e a troca de dados. Um *comando* é uma estrutura que contém os seguintes campos: um código de operação, uma string que pode ser usada para fins de nomes de estruturas, um ponteiro para uma área de

memória, juntamente com o tamanho desta área, o que ajuda a troca de dados entre os SPUs e o PPE. Esta característica permite um acesso à memória de forma assíncrona realizado por *hardware* especializado liberando o SPU para executar outras operações o que permite sobrepor o tempo de transferência de dados com processamento de dados já presentes na memória do SPU. Cada SPU tem a sua própria área de comandos na memória principal, o que permite iniciar uma cópia de memória para cada um separadamente.

Implementamos uma série de otimizações a partir desse modelo de execução inicial. A primeira otimização introduzida foi a criação de um novo tipo de dado em Lua, baseada na integração Lua/C. Fornecemos vetores nativos, que são criados no ambiente de C e referenciados na máquina virtual usando *lightuserdata*[3] Lua. Introduzimos também operações para permitir ao usuário manipular os dados nestas estruturas, como `dotproduct`, que é um código C para realizar o produto entre dois vetores nativos passados como argumentos. O principal benefício disso é usar otimizações do compilador para o código SIMD já que a criação do vetor nativo cumpre os requisitos de alinhamento de dados do Cell/BE. Vetores nativos são melhor tratados com o código C, uma vez que residem fora da máquina Lua, tornando-se a melhor maneira de troca de grandes quantidades de dados entre o PPE e SPU. Ao carregar um vetor nativo em um SPU, o ambiente de trabalho usa funções fornecidas pelo SDK IBM para que os benefícios de SIMD possam ser usados. Da mesma maneira, a remoção de um vetor nativo usa uma função fornecida pelo SDK. Isso é necessário porque os vetores nativos não são vistos pelo coletor de lixo Lua. Atualmente, a área de memória de um SPU pode conter até 5 vetores nativos de 4096 integers/floats ou 2048 doubles. Esses tamanhos são derivados de limitações na capacidade de transferência de dados máxima do Cell/BE.

A segunda otimização foi a utilização de registradores *mailbox* para retornar valores resultantes do código Lua executado no SPU. Este é um mecanismo comumente usado por programadores Cell/BE, utilizar os registradores para passar dados que se encaixem em 32 bits. Como a execução principal dos SPUs é voltada para a aritmética de precisão simples, isso é uma otimização relevante considerando que o programador Lua não tem que se preocupar com os detalhes.

A terceira otimização implementada, seguindo a ideia de explorar os registradores de *mailbox* para comunicação de dados, foi enviar as operações e parâmetros de execução. Especificamente, para a execução de funções, pre-

[3]Em Lua o usuário pode manipular estruturas que tenham sido criadas em C carregando o endereço de memória dentro da máquina virtual através de *lightuserdata*

cisávamos criar um protocolo simples para lidar com o número de parâmetros e o tamanho do nome da função. Esse protocolo envia uma sequência de mensagens. A primeira contém a operação indicando que é uma chamada de função, o segundo passo é enviar o número de caracteres do nome da função. Estes são enviados um a um, como um número inteiro de 32 bits sem sinal, e depois reordenados em string dentro do SPU. Em seguida enviamos a quantidade de parâmetros e o SPU se prepara para receber essa quantidade valores double.

3.7 Intel/AMD Multi-núcleo

Nessa seção descrevemos a implementação do modelo Numina na arquitetura Intel/AMD.

A primeira escolha foi a de infraestrutura de execução paralela de nosso ambiente. Como podemos ver pelos resultados apresentados em (Zhong2007), o uso de Pthreads permite um controle de mais baixo nível sobre as *threads* do sistema, um dos efeitos disso é que podemos reaproveitar as *threads* em execução. Isso é um fator fundamental para o ambiente Numina já que cada *thread* tem sua própria máquina virtual Lua e por isso deve permanecer em execução. Já em (Stamatakis2008) os autores concluem que OpenMP e Pthreads não apresentam desempenho significantemente diferente quando comparados em várias plataformas, porém Pthreads oferecem maior controle principalmente em relação a alocação de memória.

Como as comparações entre o uso de Pthreads e OpenMP não mostraram diferenças consideráveis no desempenho, e considerando que a implementação para Cell/BE já faz uso de Pthreads para abstrair a execução dos SPUs, optamos por continuar usando essa biblioteca. Foram utilizados mecanismos baseados em variáveis *mutex* e de condição, disponíveis em Pthreads, para gerenciar a comunicação e sincronização entre o controlador Numina e os núcleos de processamento que executam o código do aplicativo.

O uso da biblioteca Pthreads juntamente com Linux, fornece uma *kernel-thread*[4] por núcleo, sendo que a biblioteca inclui *mutex* e variáveis de condição para permitir uma execução *thread-safe* do código do usuário. Realizamos algumas experiências utilizando parâmetros para permitir a execução em modo privilegiado e trocando a estratégia de escalonamento das *threads* o que aumentou o desempenho. No entanto essa melhora tem um custo de dedicação da máquina exclusivamente ao processamento daquela aplicação em particular.

[4] A implementação da biblioteca Pthreads em ambientes Linux com kernel 2.6 ou superior provê um comportamento 1:1, ou seja, fornece uma *kernel-thread* para cada *thread* aberta na aplicação

Como essa condição pode ser muito específica, optamos por deixar esse tipo de configuração a cargo do administrador do ambiente.

Outra possibilidade foi a de experimentar com o uso de uma máquina virtual Lua com recursos de JIT. O projeto LuaJIT (Pall2011) conta com uma versão para Intel/AMD, mas não com uma versão para Cell/BE, dessa forma não temos como avaliar o seu uso misto, porém os resultados, quando consideramos apenas a arquitetura Intel/AMD, foram muito interessantes e serão apresentados no Capítulo 4.

3.8 Cluster Híbrido

Para o suporte a *clusters* optamos por ampliar a visão do *CoreGroup*, assim podemos manter a API usada para programar uma máquina multi-núcleo para controlar um conjunto de máquinas. Consideramos também que as máquinas que compõe o *cluster* são também multi-núcleo por questão de simplicidade. Como base do nosso desenvolvimento optamos por usar a biblioteca LuaSocket(Nehab2007) que provê os mecanismos para comunicação via rede para a linguagem Lua.

A forma de transferência de dados se dá através do envio de strings, por conta disso optamos por enviar a comunicação codificada em Base64 (Josefsson2006) de forma a não acontecerem perdas devido ao uso de caracteres especiais. O conjunto de máquinas que faz parte do processamento deve executar o *daemon* Numina que fica esperando a conexão do nó principal no qual a aplicação é iniciada. O nó principal deve ter um arquivo de configuração com o ip e a porta de cada nó participante do processamento.

As funções devem ser enviadas no formato string. Isso porque as diferentes versões de Lua apresentam detalhes diferentes nas suas implementações em relação ao formato binário de funções, assim, o código do usuário é declarado e enviado para cada um dos nós da rede. Em cada nó, o *daemon* Numina inicia um CoreGroup com a quantidade de Cores correspondente a quantidade de núcleos da máquina a fim de carregar as funções que serão executadas. A quantidade de CoreGroups que faz parte do Cluster, bem como a quantidade de Cores por CoreGroup, podem ser recuperadas dinamicamente durante a execução, o que permite ao desenvolvedor preparar a sua aplicação para lidar com mudanças no ambiente de execução.

A Figura 3.3 abaixo apresenta uma configuração de *cluster* Numina. Neste ambiente as diversas máquinas executoras aguardam receber os comandos de controle para executar localmente as funções do usuário, a aplicação do usuário por sua vez deve instanciar um Cluster ao invés de um CoreGroup,

porém toda a interação seguirá a mesma API.

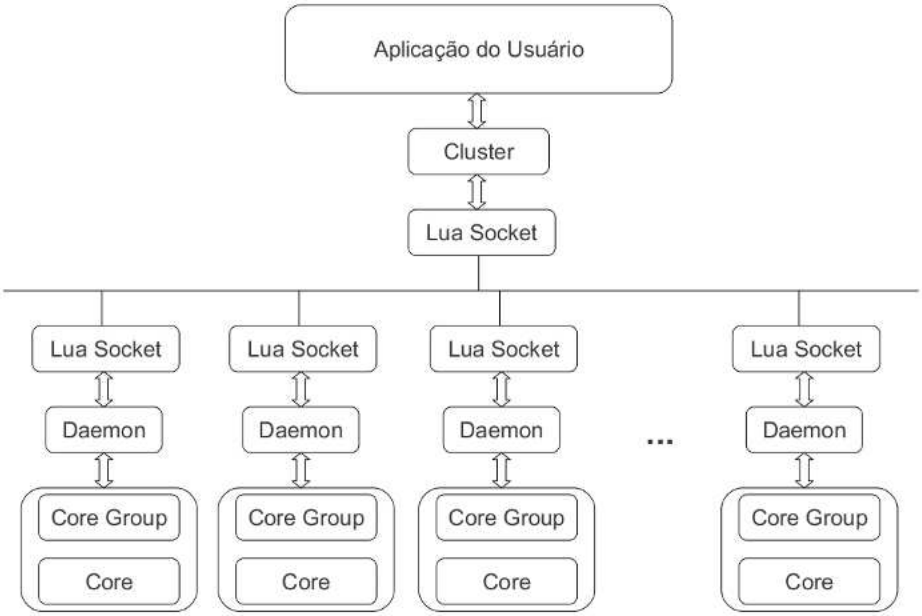

Figura 3.3: Visão de um Cluster Numina

Apresentamos a seguir o mesmo exemplo de código Numina agora voltado para a execução em *cluster*.

Listagem 3.2: Exemplo de código Numina voltado para *cluster*

```
1 require "cluster"
2 cluster = Cluster:new(1)
3 ffunc=[[function func(id, x)
4            return id * x
5         end]]
6 function main()
7         cluster:load(ffunc, "func", "BS", "BB")
8         nodeCount = cluster:getNodeCount()
9         corePerNode = {}
10        for i=1,nodeCount do
11           corePerNode[i] = cluster:getNodeCoreCount(i)
12        end
13        parmsid = {}
14        id = 1;
15        for i=1,nodeCount do
16           parmsid[i] = {}
17           for j=1,corePerNode[i] do
18              parmsid[i][j] = id
```

```
19                id = id + 1
20             end
21          end
22          cluster:func(parmsid,10)
23          result = cluster:reduce("SUM")
24          print("result="..result)
25 end
26 main()
27 cluster:close()
```

A principal diferença que temos desse código para o anterior está no particionamento de dados que precisa acontecer em duas dimensões. As políticas também mudam porque agora precisam ser definidas quanto a como distribuir os dados em relação aos nós da rede e como cada nó tratará os dados internamente.

O desenvolvedor de aplicação pode contar com funções especiais para recuperar o número de nós do *cluster* e o número de núcleos por nó como podemos ver nas linhas 8 e 11 respectivamente. Além disso é possível recuperar a quantidade total de núcleos do *cluster*.

3.9 Revendo as Políticas de Distribuição de Dados

Agora que apresentamos duas versões de código Numina podemos retomar a questão de como tratar o particionamento de dados em cada um dos casos. A primeira questão a ser levantada é que para nossa implementação consideramos que os dados podem ser particionados de forma a corresponderem ao número de núcleos que temos para execução. Dessa forma se utilizarmos uma máquina com quatro núcleos esperamos receber como entrada quatro elementos ou tabelas com quatro elementos que possam ser distribuídos para os núcleos.

Quando trabalhamos com o paralelismo apenas em uma máquina utilizamos a abstração CoreGroup e o particionamento acontece em relação aos núcleos, assim, o usuário precisa decidir se os dados passados em uma chamada devem ser copiados para cada um dos núcleos (*broadcast*), ou se devem ser separados para serem enviados para cada núcleo (*scatter*). Esta segunda opção permite passar uma tabela para a chamada e cada um de seus elementos é enviado para um núcleo diferente.

Como podemos ver na Figura 3.4 quando passamos uma tabela utilizando a política *broadcast* a tabela é totalmente copiada para cada um dos núcleos,

já quando usamos a política *scatter* a tabela tem seus elementos separados para envio individual para cada núcleo.

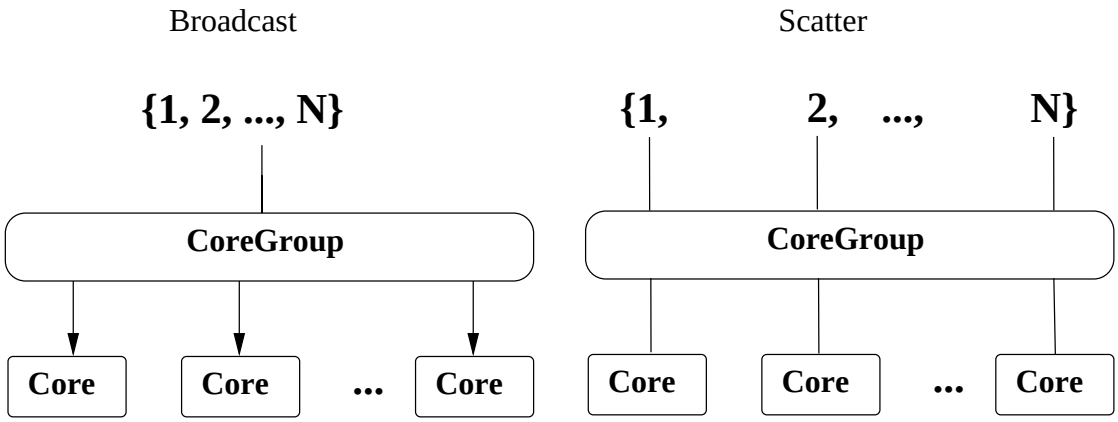

Figura 3.4: Políticas de distribuição de dados com CoreGroup

Quando trabalhamos com o paralelismo baseado em um *cluster* utilizamos a mesma ideia de particionamento, porém agora temos que decidir como os dados serão enviados para cada nó da rede e como tratar os dados localmente. Isso nos dá quatro tipos de políticas definidas a seguir:

1. BB – quando utilizamos a política broadcast-broadcast os dados serão replicados para todos os nós da rede e localmente serão replicados para todos os núcleos de cada nó. Nesta opção podemos passar um literal como argumento.

2. BS – quando utilizamos a política broadcast-scatter os dados serão replicados para todos os nós da rede, porém localmente serão separados e enviados um para cada núcleo de cada nó da rede. Nesta opção passamos uma tabela de uma dimensão.

3. SB – quando utilizamos a política scatter-broadcast os dados serão separados para envio para cada um os nós da rede e localmente serão replicados entre os núcleos de cada máquina. Nesta opção também passamos uma tabela de uma dimensão.

4. SS – quando utilizamos a política scatter-scatter os dados serão separados para o envio para cada nó e localmente serão novamente separados para o envio para cada núcleo de máquina. Nesta opção passamos uma tabela de duas dimensões para que possa ser quebrada nos dois níveis.

Uma outra questão importante é que como podemos ter máquinas com diferentes quantidades de núcleos precisamos estar atentos com o tipo de

particionamento de dados. No exemplo de código apresentado na seção anterior a tabela de identificadores considerava que cada posição poderia conter uma tabela de tamanho diferente, respeitando a contagem de núcleos de cada máquina.

Como podemos ver na Figura 3.5 as políticas se sobrepõem de forma que a tabela possa ser enviada para os núcleos.

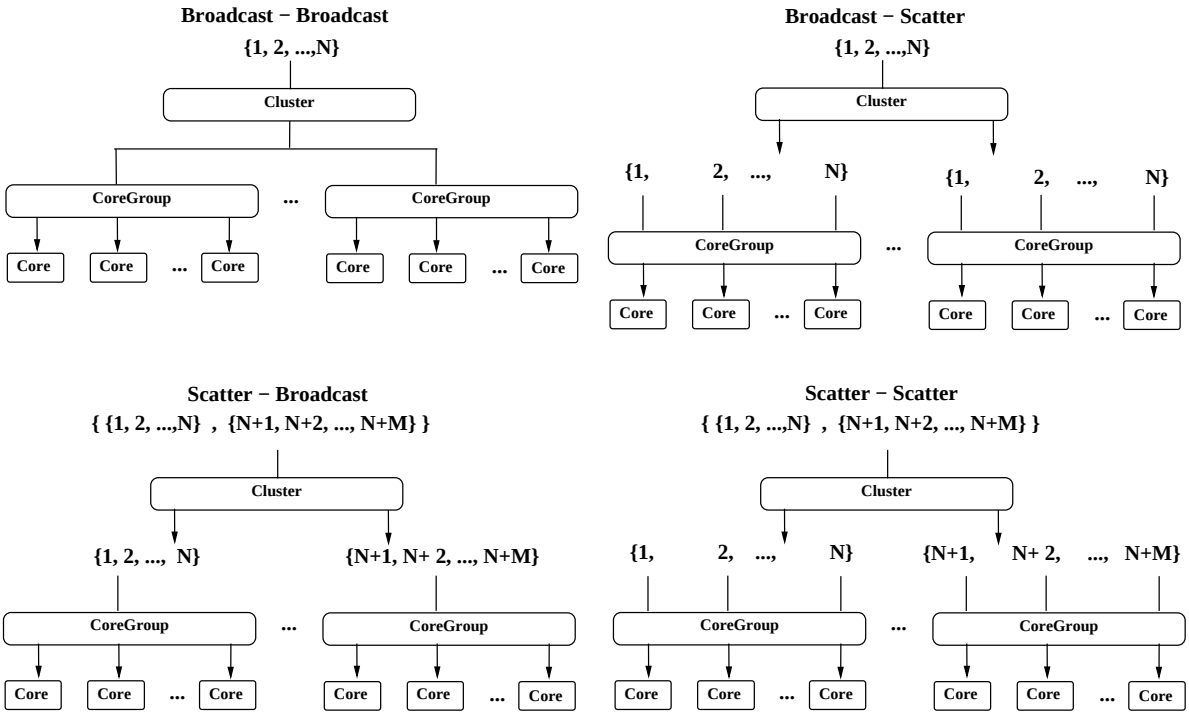

Figura 3.5: Políticas de distribuição de dados com Cluster

3.10 Conclusão

A implementação das camadas de abstração propostas para o modelo Numina se deram através de um conjunto de códigos que aproveitam a integração entre Lua e C isolando o código mais específico da arquitetura em um *driver* e contando com código Lua para as camadas comuns. Dessa forma aproveitamos as facilidades da linguagem Lua para prover alguns dos mecanismos mais dinâmicos do modelo sem que seja necessário criar uma versão para cada arquitetura suportada.

O suporte a *clusters* de máquinas é garantido através de uma abstração, também em Lua, que mantém, para o desenvolvedor de aplicações, uma API consistente com a utilizada para programar uma máquina multi-núcleo que não esteja em ambiente de rede, dessa forma distribuir uma aplicação Numina requer pouco esforço.

O uso de políticas de distribuição de dados permite que o desenvolver de aplicações controle a aplicação com uma granularidade maior, sem precisar tratar o controle individual de máquinas ou de núcleos de processamento. Embora o particionamento de dados requeira atenção especial do desenvolvedor, este mecanismo é parte inerente ao problema de desenvolvimento de aplicações paralelas.