4
Resultados e Avaliação

Com base em nossa hipótese, apresentada no Capítulo 3, torna-se fundamental avaliar o ambiente nos aspectos de facilidade de uso e impacto no desempenho, utilizando algumas aplicações paralelas para mostrar os diversos aspectos envolvidos nessa tarefa.

É mais fácil avaliar o ambiente na questão quantitativa, pois essa se baseia na comparação direta entre os tempos de execução de aplicações. Já do ponto de vista qualitativo, a questão é um pouco mais difícil de avaliar por envolver a subjetividade da percepção do usuário quanto a facilidade de uso, mas podemos utilizar os parâmetros apresentados na Seção 2.5 além do próprio tamanho do código criado.

Para analisar o custo de desempenho também realizamos medidas para identificar o impacto do uso de camadas de abstração e a melhoria de desempenho atingida através do uso de técnicas de JIT em arquitetura Intel/AMD.

4.1
Estratégia de Testes

Tendo atingido uma plataforma executável que indicava ser capaz de responder adequadamente às necessidades de desempenho foi então criado um plano de teste com a divisão em duas categorias:

1. Quantitativa

   (a) Tempo de Execução - cada versão da aplicação é executada 10 vezes e os tempos são utilizados em uma média aritmética. Foram utilizadas as linguagens C e Java como parâmetros de comparação com a implementação em Numina.

2. Qualitativa

   (a) Linhas de Código - cada versão é medida em termos de quantidade de linhas de código.

(b) Aderência ao Paradigma de Programação da Linguagem - são analisadas as questões referentes à similaridades na forma de desenvolver aplicações proposta pela linguagem e pelo ambiente que provê recursos de paralelismo para a mesma.

As aplicações teste que foram usadas são:

– Teste de primalidade de um conjunto de números – exemplo que demanda grande volume de operações, mas se baseia em apenas um ponto de comunicação para transmitir para cada processador os valores inicial e final de seu conjunto de testes. Por conta disso é interessante para medir fatores de custo.

– Multiplicação de matrizes – exemplo tradicional de programação paralela que apresenta grande carga de processamento, porém de fácil paralelização. Explora mecanismos de comunicação e compartilhamento de dados que podem ser feitos em diferentes padrões.

– Aproximação de pi utilizando método de Monte-Carlo – explora a utilização do método de Monte-Carlo o que tem um impacto direto na aplicação pela necessidade de ter um gerador aleatório único.

– Cálculo da integral da função e^x através do método de trapézios – exemplo tradicional na literatura de paralelismo de fácil entendimento para introduzir os conceitos de comunicação e coleta de resultados.

– Simulação N-Body – exemplo tradicional de um problema real resolvido através do processamento paralelo que demanda mecanismos mais refinados de comunicação.

Cada uma dessas aplicações foi executada com um conjunto distinto de parâmetros de entrada a fim de identificar o efeito do tamanho da entrada no desempenho atingido por cada uma das versões da aplicação.

As máquinas utilizadas nos testes foram:

1. AMD Turion X2 2.1 GHz e 2GB RAM- dois núcleos
2. AMD Phenom II X4 3.4 GHz e 6GB RAM - quatro núcleos
3. Cell/BE 3.2 GHz e 256MB RAM - seis SPEs por ser baseado em Playstation 3

4.2 Análise Quantitativa

Foram consideradas para esta análise duas aplicações: teste de primalidade de um conjunto de números, multiplicação de Matrizes. Isto porque essas aplicações apresentam uma carga computacional maior sendo assim mais adequadas a medidas de tempo de execução. As duas aplicações tiveram implementações sequenciais e paralelas executadas na máquina Phenom II X4 utilizando os 4 núcleos de processamento disponíveis.

Para medir o tempo de execução utilizamos a ferramenta *time* disponível no Linux. Foram medidas 10 execuções, e o tempo considerado foi dado pelo valor da coluna *real-time*. Utilizamos a média dos tempos para chegar ao tempo de execução. Para as versões paralelas de C e Java utilizamos 4 *threads* de execução, já que o modelo Numina prevê usar uma *thread* de execução por núcleo disponível.

Vamos apresentar a seguir os resultados de cada uma dessas aplicações de forma isolada comparando as diferentes versões para depois comparar os resultados entre as diferentes aplicações. Utilizamos primeiro o tempo de execução das versões sequenciais das aplicações como referência do ganho atingido pelo uso de paralelismo em cada um dos problemas. Em seguida apresentamos o desempenho da versão paralela de cada implementação correlacionando os valores com as outras tecnologias.

4.2.1 Teste de Primalidade de um Conjunto de Números

Esta aplicação serve como um bom experimento por demandar um grande poder computacional. O algoritmo que usamos apenas testa o número dado p em relação ao conjunto $\{k : (\exists n \in \mathbb{N})(k = 2n + 1 \wedge 3 \leq k \leq \lfloor\sqrt{p}\rfloor)\}$ para buscar divisores, e se não encontrar considera o número como primo.

Para que essa aplicação realmente tenha um tempo de execução considerável, trabalhamos com o intervalo de números [99999999999901,99999999999999], variando o número inicial para conseguirmos conjuntos de 100, 1000 e 10000 números.

Para podermos comparar a melhoria de tempo de execução precisamos conhecer o tempo de execução das versões sequenciais da aplicação. A Tabela 4.1 apresenta a média dos tempos, em segundos, para completar a tarefa com versões sequenciais da aplicação na máquina Phenom II X4.

Intervalo	C	Java	Numina
100	0,6	0,7	4,9
1000	3,5	3,5	29,4
10000	29,0	29,0	245,2

Tabela 4.1: Média dos tempos de execução - teste de primalidade sequencial

As medidas do tempo de execução paralelo vão nos permitir identificar o fator de aceleração obtido em relação às versões sequenciais de cada versão da aplicação. A Tabela 4.2 apresenta a média dos tempos, em segundos, para completar a tarefa com versões paralelas da aplicação, utilizando uma *thread* por núcleo na máquina Phenom II X4.

Intervalo	C	Java	Numina
100	0,5	1,1	2,5
1000	1,7	3,9	9,6
10000	11,9	26,5	65,6

Tabela 4.2: Média dos tempos de execução - teste de primalidade paralelo

Podemos estabelecer a relação entre esses números indicando que houve uma melhora no tempo de execução pelo uso de uma versão paralela, exceto para a versão Java. Consideramos como fator de aceleração a equação 4-1

$$\text{Aceleração} = \frac{T_{\text{sequencial}}}{T_{\text{paralelo}}} \tag{4-1}$$

A Tabela 4.3 mostra o fator de aceleração pelo uso de uma versão paralela para cada uma das tecnologias. Podemos perceber que o fator de aceleração melhora conforme o conjunto de dados aumenta. O custo do uso de *threads* em Java fica evidente pela piora de tempo nos conjuntos de entrada menores.

Intervalo	C	Java	Numina
100	1,2	-0,6	1,9
1000	2,1	-0,9	3,1
10000	2,4	1,1	3,7

Tabela 4.3: Fator de aceleração pelo uso de paralelismo - teste de primalidade

4.2.2 Multiplicação de Matrizes

A multiplicação de matrizes, além de aparecer em diferentes problemas de computação científica, é um exemplo tradicional de programação paralela devido à grande quantidade de operações de somas e multiplicações necessárias.

Para que essa aplicação realmente tenha um tempo de execução considerável, trabalhamos com ordens de matrizes 500, 1000, 2000, 4000 e 8000. As matrizes foram preenchidas com números aleatórios e não são esparsas.

A Tabela 4.4 apresenta a média dos tempos, em segundos, para completar a tarefa com versões sequenciais da aplicação na máquina Phenom II X4. A versão Java apresentou um erro de execução para a matriz de ordem 8000 impossibilitando sua medição.

Ordem	C	Java	Numina
500	0,2	0,2	2,2
1000	1,3	1,3	2,2
2000	11,1	11,4	15,3
4000	89,9	88,7	105,3
8000	719,8	-	774,5

Tabela 4.4: Média dos tempos de execução - multiplicação de matrizes sequencial

A Tabela 4.5 apresenta a média dos tempos, em segundos, para completar a tarefa com versões paralelas da aplicação, utilizando uma *thread* por núcleo na máquina Phenom II X4.

Ordem	C	Java	Numina
500	0,1	0,1	0,3
1000	0,4	0,4	1,3
2000	3,3	3,4	6,7
4000	25,7	25,5	39,1
8000	198,6	196,9	250,9

Tabela 4.5: Média dos tempos de execução - multiplicação de matrizes paralela

Podemos estabelecer a relação entre esses números indicando que houve uma melhora no tempo de execução pelo uso de uma versão paralela. Novamente utilizamos a equação 4-1 para considerar o fator de aceleração. A Tabela 4.6 mostra o fator de aceleração pelo uso de uma versão paralela.

Ordem	C	Java	Numina
500	3,5	2,0	8,0
1000	3,5	3,1	1,7
2000	3,4	3,4	2,3
4000	3,5	3,5	2,7
8000	3,6	-	3,1

Tabela 4.6: Fator de aceleração pelo uso de paralelismo - multiplicação de matrizes

Esses testes também foram executados no ambiente Cell conforme apresentado na Tabela 4.7 abaixo.

Ordem	C Sequencial	C	Numina
500	70	4	29
1000	757	21	116
2000	6587	144	467
4000	-	738	1473

Tabela 4.7: Média dos tempos de execução em ambiente Cell - multiplicação de matrizes paralela

Na primeira coluna da tabela apresentamos os tempos de uma versão de sequencial do código feita em C para referência. O código para a implementação da multiplicação de matrizes em Cell difere principalmente por conta de não haver memória compartilhada, o que aumenta significantemente o tempo de processamento. No entanto essa característica permanece tanto para a versão em C quanto para versão Numina o que nos leva a um fator semelhante ao obtido em plataforma AMD/Intel.

4.2.3 Análise de Resultados Quantitativos

Um aspecto interessante que é observado em ambos os exemplos é que o tempo de execução melhora à medida que o tamanho da entrada cresce. Isso fica mais evidente com o uso de uma linguagem interpretada, pois o custo para tratar um conjunto de entrada menor se torna mais alto.

Observamos que houve uma melhora no tempo de execução das aplicações das versões sequenciais para as versões paralelas para cada uma das tecnologias como já era esperado. Além disso, identificamos os fatores de perda no tempo de execução nos dois exemplos quando comparamos as médias de tempos em C com as médias de tempo em Numina. Na aplicação de teste de primalidade de números, a versão Numina fica 5,4 vezes mais lenta do que a versão em C. Já na aplicação de multiplicação de matrizes essa diferença cai para 2,2 vezes. Esse resultado melhor na multiplicação de matrizes se deve ao uso de uma biblioteca própria para trabalhar com vetores e fornecer operações escritas em C. Esta é uma primeira abordagem para melhoria de tempo de execução transparente para o usuário.

4.3 Análise Qualitativa

Nesta seção avaliaremos as questões de tamanho de código fonte, mistura entre código de comunicação/funcional, e quantidade de detalhes específicos de arquitetura utilizados nas aplicações.

Foram consideradas para esta análise duas aplicações: aproximação de pi utilizando método de Monte-Carlo e cálculo da integral da função e^x através do método de trapézios. Isto porque essas aplicações não apresentam uma carga computacional pesada, porém são exemplos tradicionais na literatura de paralelismo sendo assim adequadas a análise qualitativa.

Devido a natureza dessa análise são necessárias as diferentes versões de código fonte de cada aplicação. As listagens para o exemplo de aproximação de pi utilizando método Monte-Carlo podem ser encontradas no Apêndice B, as listagens dos demais códigos podem ser encontradas on-line em numina.com.br.

4.3.1 Aproximação de Pi Utilizando Método de Monte-Carlo

Os métodos de Monte-Carlo são baseados no uso de um gerador de números aleatórios. O problema é que para garantir o correto funcionamento do método esse gerador deve ser único no sistema, o que apresenta uma dificuldade em sistemas paralelos, pois somente um núcleo pode gerar os números. Em sistemas paralelos e distribuídos podemos dedicar um núcleo a essa geração de números, no entanto contamos com menos um processador no cálculo da solução.

Essa aplicação é utilizada como exemplo na literatura por ser de simples compreensão e também para discutir o problema de unicidade do gerador. Por conta desse requisito, precisamos usar mecanismos que permitam sincronizar a execução. Em C utilizamos variáveis *mutex* e de condição juntamente com os mecanismos de *lock/unlock*, *wait/broadcast* fornecidos pela biblioteca Pthreads. Em Java utilizamos blocos *synchronized* e os métodos *wait/notifyAll* fornecidos pela classe Object. Em Numina não temos um mecanismo explícito de controle de sincronização, porém a função *gather*, que permite recuperar os resultados de cada Core, bloqueia internamente, garantindo a barreira de sincronia de que precisamos.

Uma diferença importante entre as implementações é que em C e Java programamos com *threads* diretamente. Quando abrimos uma *thread* de execução o código fica em execução contínua, dessa forma precisamos de um mecanismo que coordene o acesso a área de dados. Isso é feito via mecanismos de exclusão mútua. Em Numina a aplicação não tem acesso direto a *threads* de execução. A aplicação deve ser projetada em termos de chamadas de função que trabalham sobre um conjunto de dados fornecido. A cada passo do algoritmo fornecemos um novo conjunto e realizamos uma nova chamada à função. A cada chamada utilizamos a função *gather* para recuperar os resultados parciais, garantindo assim a sincronização.

A Tabela 4.8 apresenta a quantidade de linhas de código para cada versão, além dos mecanismos usados para sincronização.

	C	Java	Numina
Linhas de Código	107	116	57
Sincronismo	*lock/unlock* *wait/broadcast*	*synchronized* *wait/notifyAll*	*gather* nova chamada

Tabela 4.8: Quantidade de Linhas de Código e Mecanismos de Sincronização

Podemos perceber que, em relação ao tamanho do código, a versão Numina tem praticamente metade do tamanho. Já em relação aos mecanismos de sincronismo, o controle em relação ao acesso é feito pelo ambiente, sendo que o usuário só precisa atentar para a ordem de transferências de dados e chamadas às suas funções. Nas versões em C e Java, grande parte do código é dedicada ao controle dos mecanismos de sincronismo que acabam por interferir na legibilidade das aplicações.

4.3.2 Cálculo de Integral da Função e^x Através do Método de Trapézios

Esta aplicação é um exemplo que discute o particionamento de dados para distribuição e posteriormente coleta e redução dos dados (Andrews2000). Neste caso não temos controle de sincronização entre as *threads* porque o cálculo é realizado uma única vez.

A Tabela 4.9 a seguir apresenta a quantidade de linhas de código para cada versão da aplicação.

	C	Java	Numina
Linhas de Código	56	58	33

Tabela 4.9: Quantidade de Linhas de Código por versão

Novamente o código Numina apresenta praticamente metade do tamanho das outras versões, mas a grande diferença neste exemplo está no particionamento de dados. Em Numina, o programador não trabalha com uma visão de threads, e sim com chamadas de função sobre um CoreGroup, que acontecem em todos os núcleos em paralelo. Como só é feita uma chamada de função, os dados para cada núcleo precisam ser particionados em uma tabela de forma a serem distribuídos de acordo com uma política escolhida pelo usuário conforme apresentado na Seção 3.9. Em C e Java isso acontece de forma implícita porque o usuário é responsável por criar cada *thread* e passar os dados correspondentes.

Em Numina evitamos utilizar muitos laços *for* para controlar *threads* e suas passagens de parâmetros, o que deixa o entendimento mais claro porque o usuário só tem um ponto de chamada de função. Por outro lado o usuário é levado a pensar em seu algoritmo através de uma abordagem de mais alto nível para projetar a forma como particiona os dados de sua chamada.

4.3.3 Análise de Resultados Qualitativos

Para a análise qualitativa, escolhemos trabalhar com duas aplicações mais simples para permitir evidenciar as características do modelo. Em relação à quantidade de código o resultado foi que podemos escrever as aplicações com metade do código necessário nas outras versões.

Já em relação aos mecanismos de sincronismo pode ser observada no código uma melhoria considerável, pois todo o controle é feito pelo próprio ambiente, enquanto em C e Java todo o controle fica a cargo do programador o que causa um acoplamento do código da aplicação com o código de controle.

Quanto ao particionamento de dados, observamos que o modelo promove uma melhoria por conta de levar o programador a trabalhar em um nível mais alto através da escolha de políticas de particionamento a serem aplicadas aos seus dados.

4.4 Simulação N-Body

Esta aplicação é muito utilizada para medir desempenho de arquiteturas novas por necessitar de muito processamento e necessitar de um esquema de comunicação de dados eficiente. Neste caso utilizamos a implementação baseada em MPI do curso de paralelismo da *Syracuse University*[1]. Para comparação criamos uma implementação Numina do mesmo código utilizando a abstração de *cluster* fornecida por nosso modelo.

Nesse exemplo temos algumas limitações no processamento devido as características da própria aplicação. O primeiro ponto que optamos foi usar uma aplicação base de comparação em C disponível na internet, para que pudéssemos contar com um código totalmente imparcial. O segundo ponto é que não havia uma implementação de MPI para o ambiente do Cell/BE disponível, por isso essa arquitetura não foi utilizada na medição de tempos desse exemplo apenas as máquinas Phenom II X4 e Turion X2. Essa configuração oferece, em termos do modelo Numina, um Cluster composto de dois CoreGroups, um com 4 e outro com 2 Cores interligados por uma rede de 100Mbps, totalizando 6 núcleos de processamento. Utilizamos 100 passos de execução da simulação para 3000, 6000 e 12000 corpos na simulação.

A implementação MPI utilizada foi a openMPI versão 1.2 e disponível nas máquina Linux utilizadas no teste. A versão Numina utilizada foi puramente interpretada, sem recursos JIT nem a biblioteca de processamento de vetores.

Por ser uma aplicação mais completa e real optamos por analisar os resultados sob as duas óticas apresentadas anteriormente. Em relação a visão quantitativa a Tabela 4.10, a seguir, apresenta os resultados das medições de tempo de execução Numina e MPI. Usamos o mesmo racional apresentado na Seção 4.2 de executar 10 vezes as aplicações, coletar os tempos de execução e realizar a média.

Duas características de nossa implementação do modelo Numina têm um peso no tempo de execução atingido nesse exemplo. A primeira delas é que a biblioteca LuaSocket trabalha exclusivamente com a transmissão de dados no formato *string*. Dessa forma, a cada passo temos que converter os dados

[1] A versão do código pode ser acessada em http://www.new-npac.org/projects/cdroms/cewes-1999-06-vol2/cps615course/examples96/

Corpos	openMPI	Numina Cluster
3000	12,0	310
6000	34,7	743,4
12000	125,15	2415,6

Tabela 4.10: Média dos tempos de execução - simulação N-Body

de formato numérico para *string* a fim de transmitir as informações para os outros processadores. A segunda é que não contamos ainda com um mecanismo de comunicação inter-núcleo. Usamos como base para o modelo Numina uma visão Master/Worker onde um nó central distribui tanto tarefas quanto dados para os outros. Sendo assim, a cada passo as informações são coletadas de cada processador para o código central do usuário, para em seguida serem redistribuídas. Isso não só afeta o desempenho pela transmissão de dados, mas também tem um impacto no aproveitamento dos processadores que acabam ficando limitados pelos núcleos mais lentos.

Em relação a visão qualitativa, a Tabela 4.11, a seguir, apresenta a quantidade de linhas de código e a forma de comunicação de dados utilizada.

	openMPI	Numina Cluster
Linhas de Código	283	204
Comunicação	–*Pipe* –*AllReduce*	–*gather* –funções de conversão strings/tabelas –nova chamada à função de cálculo

Tabela 4.11: Quantidade de Linhas de Código e Mecanismos de Comunicação

Podemos observar que o código na versão Numina tem 79 linhas a menos que a versão MPI. A visão de execução é a mesma discutida na Seção 4.3.1, onde a cada passo o usuário deve chamar sua função de processamento novamente.

Graças as facilidades apresentadas pela linguagem Lua foi possível escrever as funções de conversão de tabelas em strings como funções do usuário, já que a abstração Cluster não apresenta uma forma própria de transferir tabelas Lua. Dessa forma vemos que o usuário pode expandir o ambiente para atender suas necessidades seguindo o mesmo modelo de desenvolvimento que usa para codificar o algoritmo de sua solução.

Se a implementação já contasse com os mecanismos de transmissão de tabelas isso teria um impacto na quantidade de linhas de código gerando mais uma simplificação na aplicação do usuário.

4.5 Melhoria de Desempenho Através de JIT

Um argumento que defendemos em relação ao uso de linguagens interpretadas é que otimizações podem ser empregadas no ambiente de execução da linguagem de forma transparente para o usuário final. Podemos experimentar e avaliar essa questão na implementação de nosso modelo na arquitetura Intel/AMD trocando a máquina virtual de Lua tradicional pela LuaJIT(Pall2011)[2].

A primeira observação importante é que o código da aplicação do usuário permanece inalterado. Isto porque a mudança é realizada diretamente no ambiente de execução que passa a utilizar a máquina virtual Lua modificada para usar JIT. Com isso o usuário passa a ter um tempo de execução melhor mantendo as facilidades de uso originais.

A seguir apresentamos na Tabela 4.12 os resultados das execuções da aplicação de Teste de Primalidade em um ambiente Numina *versus* Numina com JIT. A primeira coluna apresenta os dados correspondentes a Tabela 4.2, a segunda coluna apresenta a execução da mesma aplicação utilizando um ambiente Numina modificado para trabalhar com a máquia LuaJIT, a terceira coluna apresenta o fator de aceleração.

Intervalo	Numina	Numina com JIT	Fator Numina/JIT
100	2,5	0,2	11,2
1000	9,6	1,4	6,6
10000	65,6	9,8	6,7

Tabela 4.12: Comparação dos tempos de execução com o uso de JIT

Podemos perceber uma melhoria muito grande no tempo de execução o que valida nossa visão de que otimizações no ambiente são uma boa alternativa para o usuário final uma vez que o programa executado não precisou ser modificado. Como o primeiro intervalo de números é consideravelmente menor que os outros seu fator de melhora não pode ser tomado como base, mas podemos perceber nos outros dois intervalos que o fator de melhoria fica em torno de 6,6 vezes.

Observando a Tabela 4.2 podemos perceber que o uso de uma máquina JIT obteve um tempo melhor que o de nossa versão em C. Um dos principais motivos para esse comportamento é que a máquina LuaJIT oferece suporte a menos arquiteturas do que o compilador gcc 4.5.0 utilizado nos testes. Dessa forma o ambiente LuaJIT pode aplicar otimizações mais direcionadas.

[2]Não foi possível experimentar essa técnica com a arquitetura Cell/BE por não existir uma implementação do projeto LuaJIT que suporte esse processador

4.6 Impacto do Uso de Camadas de Abstração

Outra questão que propomos em nossa hipótese é que o uso de camadas permite fornecer mais facilidades para o desenvolvedor de aplicações, mas que não deveria implicar em um custo computacional muito grande.

Para avaliar esse impacto utilizamos como base a aplicação de Teste de Primalidade por não ter uso de comunicação intensa permitindo avaliar o impacto de cada camada sobre o desempenho da aplicação. Como nosso modelo é baseado em camadas podemos criar aplicações em cada uma das camadas do modelo ficando com a responsabilidade de controlar manualmente os núcleos de execução. Assim criamos três versões diferentes da aplicação:

1. Acessar diretamente as funções da API Numina
2. Utilizar a abstração Core, mas controlar os núcleos manualmente
3. Utilizar a abstração CoreGroup

Os testes foram executados nas máquina Phenom II X4 nos intervalos 1000 e 10000. Cada uma das três versões da aplicação foi executada dez vezes com cada intervalo gerando uma média de tempo.

Apresentamos na Tabela 4.13 os resultados dos tempos de execução para cada uma das versões utilizando o ambiente Numina e Numina JIT.

Versão	Numina	Numina com JIT
API	114,43	5,6265
Core	114,65	5,6230
CoreGroup	116,04	5,6275

Tabela 4.13: Fatores de impacto no desempenho pelo uso de Camadas

Podemos perceber que mesmo quando usamos o ambiente totalmente interpretado o impacto pelo uso de camadas não é significativo ficando em torno de 2 segundos, já com o uso do ambiente habilitado para JIT a diferença é ainda menor. Quando consideramos todas as facilidades fornecidas ao desenvolvedor de aplicações pelo uso de camadas, esse impacto parece não comprometer o desempenho necessário para as soluções de aplicações paralelas. O código das três versões da aplicação está listado no Apêndice C.

4.7 Conclusão

Uma caracterização importante é que nossa análise quantitativa foi feita sobre nosso protótipo de implementação do modelo Numina, enquanto a análise qualitativa é feita sobre o próprio modelo. Evoluções do protótipo podem promover melhorias no desempenho das aplicações, conforme a própria proposta do trabalho.

Após analisar o desempenho de nosso modelo podemos identificar uma perda de 5,4 na aplicação de teste de primalidade de números e uma perda de 2,2 na aplicação de multiplicação de matrizes em relação as versões dessas aplicações em C. Os ganhos do ponto de vista qualitativo apontam o uso de camadas e linguagens interpretadas como um caminho promissor para simplificar a tarefa do desenvolvedor de aplicações.

Foram analisados ainda os impactos no desempenho do custo de usar camadas de abstração bem como de aplicar uma otimização ao ambiente de forma transparente para o usuário através do emprego de JIT. O impacto pelo uso de camadas não foi significativo, porém a melhoria de desempenho através da mudança no ambiente apresentou uma diferença muito grande.

Essas análises mostram que em aplicações que possamos pagar a diferença de tempo de execução – entre 2 e 5 vezes mais lento – teremos facilidades se usarmos o modelo proposto. Para os casos em que seja necessário um tempo de execução melhor podemos aplicar otimizações e usar construções em C em pontos específicos.

Nossa proposta de uso de uma linguagem interpretada não está ligada diretamente a uma execução puramente interpretada. Embora acreditemos que o tempo de execução atingido com o ambiente puramente interpretado – para quando não existem opções de uso JIT – seja satisfatório do ponto de vista do desenvolvedor de aplicações, acreditamos que a execução pode e deve empregar todo tipo de técnicas de melhoria de desempenho disponíveis, em especial JIT. Dessa forma podemos atingir o desempenho esperado de uma linguagem compilada, mas com as facilidades proporcionadas por uma linguagem interpretada e um modelo de camadas de abstração.

A análise da aplicação de simulação N-Body mostrou um tempo de execução insatisfatório, embora do ponto de vista qualitativo tenha sido mais fácil de escrever a aplicação. Seria necessário aplicar mecanismos de otimização ao ambiente para melhorar o tempo de execução. Devido as características do próprio problema, uma parte do problema poderia ser desenvolvida em C e, utilizando as facilidades de integração oferecidas por Lua, fornecer um desempenho melhor na parte que necessita de mais processamento. Com esta

abordagem o desenvolvedor de aplicações ainda teria todas as facilidades de coordenação do ambiente, tendo um esforço menor no desenvolvimento e na manutenção de sua aplicação.

Embora todos os exemplos utilizados sejam aplicações que empregam o padrão SPMD, essa não é uma restrição de nosso modelo. Podemos desenvolver aplicações MPMD – Multiple Programs Multiple Data – no entanto para efeitos de comparação inicial consideramos que a contribuição desse tipo de aplicação não simplificaria o entendimento.