5
Trabalhos Relacionados

Tendo identificado que o desenvolvimento de aplicações paralelas é uma tarefa complicada por conta da quantidade de informações que o desenvolvedor precisa dominar, podemos apresentar o contexto deste trabalho.

Uma questão importante é a discussão sobre facilidades no desenvolvimento de aplicações paralelas, e como abstrair os detalhes envolvidos nessa tarefa, principalmente no que diz respeito a dependências de arquitetura e detalhes para atingir o desempenho necessário para a solução do problema.

Em nosso estudo de caso estamos trabalhando com processadores de duas arquiteturas diferentes, Intel/AMD e Cell/BE. Este último foi desenvolvido para proporcionar alto desempenho para aplicações de multimídia. Por conta disso o desenvolvimento de aplicações para Cell/BE torna-se específico e difícil, o que gerou uma grande quantidade de trabalhos e discussões para propor ferramentas que permitissem simplificar essa tarefa. As investigações realizadas para essa plataforma, de uma maneira geral, não se aplicam à outra arquitetura que abordamos, servindo como um indicador da dificuldade enfrentada pelo desenvolvedor de aplicações paralelas.

Em nosso trabalho consideraremos as pesquisas sobre o desenvolvimento de aplicações paralelas sob a ótica de três vertentes que tratam de diferentes interesses e aspectos.

A primeira concentra a visão de que a melhoria de desempenho deve ser uma responsabilidade do sistema e não do desenvolvedor de aplicações. Nesse sentido as pesquisas devem apontar mecanismos que permitam diferentes níveis de melhorias de maneira transparente para o usuário final.

A segunda apresenta modelos de abstração de paralelismo de forma que seja possível isolar as questões referentes a diferentes níveis de paralelismo bem como diferentes visões e estratégias para abordar os problemas que necessitam de aplicações paralelas.

A terceira trata de bibliotecas para suporte a paralelismo. Como existem muitas bibliotecas diferentes para a arquitetura Intel/AMD e seu uso é bastante conhecido, decidimos por explorar as bibliotecas para o processador Cell/BE que, tendo características muito específicas, conta com um conjunto diverso de

abordagens para paralelismo.

5.1 Desempenho como Responsabilidade do Sistema

De acordo com (Andrews2000), um programa paralelo é caracterizado pelo uso de vários processos para resolver um dado problema em menos tempo do que seria necessário utilizando uma versão sequencial, ou para resolver uma instância maior do problema na mesma quantidade de tempo. Em qualquer uma das situações, podemos perceber que um fator importante sobre paralelismo é o desempenho, já que seu principal objetivo é melhorar o tempo de execução quando comparado ao desenvolvimento sequencial. No entanto, como argumentado em (Foster1995), o desempenho sozinho não pode ser considerado como a única métrica para aplicações paralelas. Alguns outros fatores, tais como eficiência do paralelismo, requisitos de memória, latência, taxas de entrada/saída, largura de banda da rede, potencial para reuso, custos de hardware, portabilidade, e escalabilidade, devem ser considerados quando quantificamos uma aplicação paralela.

Devido a essas características acreditamos que o desempenho deve ser fornecido através de mecanismos do sistema, estando assim na esfera de trabalho dos desenvolvedores de sistema. Para o desenvolvedor de aplicações sempre existe a preocupação com o desempenho que será atingido, mas isso passa a ser feito em uma esfera mais confortável, sem que seja necessário dominar os detalhes de cada tipo de arquitetura.

Em (Sankaralingam2010) o autor expõe que muito do software escrito atualmente é baseado em linguagens interpretadas e a eficiência no código é deixada para o sistema graças a diferentes *frameworks* que tentam prover o máximo de funcionalidade de forma que o desenvolvedor de aplicações permaneça concentrado no seu domínio de problema. O autor apresenta a visão de que o paralelismo deva ser deixado para especialistas porque a maior parte das aplicações comerciais não apresenta problemas de real paralelismo e sim de concorrência sem dependência de dados. Acreditamos que essa visão é restrita, pois mesmo para o desenvolvimento de *frameworks* para suporte ao desenvolvimento de aplicações comerciais os desenvolvedores precisam de melhores ferramentas para lidar com o paralelismo, mas concordamos que o uso de linguagens interpretadas permite maiores facilidades para os desenvolvedores de sistema fornecerem melhorias de desempenho.

Em (Vishkin2011) é defendida a ideia de que a mudança para suportar a computação paralela deve acontecer na própria abstração que usamos para representar os algoritmos. A proposta do autor é ter uma visão que permita

ao programador de aplicações mapear seu algoritmo PRAM[1] diretamente sobre uma máquina que, diferente do modelo von Neuman, permite executar múltiplas instruções de maneira concorrente. Esta máquina – chamada XMT – conta com multi-núcleo e multi-thread explícito e é baseada em ter uma *thread* principal que controla a execução e um conjunto de *threads* que executam instruções de forma concorrente, sendo que o número de *threads* pode chegar até a quantidade de núcleos disponíveis. Neste sentido, nosso ambiente se assemelha a essa visão de máquina, porém não utilizamos a execução de uma única instrução por núcleo e sim carregando um conjunto de funções por núcleo que executa sua máquina Lua independente. Especificamente na arquitetura Intel/AMD temos uma alternativa que permite o uso de memória compartilhada. Podemos ainda considerar que o autor apoia a mesma ideia que defendemos de que deve existir uma fronteira bem definida entre desenvolvedores de sistema e de aplicação que permita proteger estes de todos os detalhes envolvidos na evolução e melhoria de desempenho.

A literatura mostra que podemos encontrar formas de melhorar o desempenho através da melhoria do ambiente de suporte como pode ser visto em (Karwande2003), (Ke2004), (Faraj2006) e (Vadhiyar2000), onde são apresentadas variantes de MPI com desempenho otimizado. Nosso modelo não se baseia em MPI, mas nossa camada de rede é suficientemente abstrata para permitir o seu uso o que nos permitiria tirar proveito dos benefícios dessas implementações modificadas sem que isso afetasse as aplicações de nossos usuários.

Já em (Tan2003), (Faraj2005) e (Rosing1999) podemos ver a abordagem baseada em geração de código para ambientes paralelos. Essa abordagem também pode ser empregada na forma de um serviço através da camada de API do modelo que poderia utilizar tanto alternativas JIT como de geração de código para arquiteturas específicas. Essa última alternativa tem sido especialmente utilizada quando são usados dispositivos baseados em GPU.

Esses estudos mostram que diferentes grupos buscam otimizações para liberar o desenvolvedor de aplicações de ter que tratar detalhes de mais baixo nível. Nesta visão de separação de papéis, (Gorlatch2004) afirma que desenvolver código para tratar detalhes de baixo nível é uma tarefa para desenvolvedores de sistema que estão acostumados aos detalhes desse tipo de tarefa.

Em (Miller2002), o autor afirma que uma perda de desempenho de uma ordem de grandeza pode ser aceitável quando consideramos os benefícios que os programadores têm quando usam uma abstração de mais alto nível. Pode-

[1]PRAM ou *Parallel Random Access Machine* é uma abstração que permite descrever algoritmos paralelos em termos de uma máquina de memória compartilhada abstrata de forma que possa ser avaliado sua complexidade

mos considerar que essa ideia é apoiada por (Skillicorn1998), onde os autores afirmam que o desempenho máximo é desnecessário, especialmente se comprometer a manutenibilidade e aumentar o custo de desenvolvimento. Isso encoraja nossa escolha por utilizar uma linguagem interpretada como meio de fornecer os mecanismos de paralelismo, pois embora sejam tipicamente mais lentas em sua execução oferecem mais oportunidades para que os desenvolvedores de sistema implementem otimizações.

Quando coordenadas, essas ideias apontam para um modelo que isole os detalhes de infraestrutura e otimizações do desenvolvedor de aplicações. O uso de camadas de software atribui ao sistema a responsabilidade pela busca de soluções de melhor desempenho, garantindo aos desenvolvedores de sistemas a liberdade necessária para a evolução das plataformas. Vemos isso como uma motivação para a investigação do custo pelo uso de camadas para abstrair os detalhes de infraestrutura e desempenho. Além disso, o uso de camadas permite a separação estrutural entre a aplicação e o sistema. Desde que seja mantida uma interface consistente, a camada de infraestrutura pode evoluir sem impactos.

Uma outra questão, bem delicada, é estabelecer o que deve ser visível e o que deve ser transparente para a programação da aplicação. O compromisso com a transparência deve ser uma constante entre os desenvolvedores de sistema, evitando ao máximo a visibilidade de uma característica específica. Porém, certas características fazem parte da semântica das aplicações e a existência de critérios para fundamentar a tomada de decisão sobre essa visibilidade, durante a fase de definição de suas abstrações, ainda é um problema em aberto. Em nosso trabalho abordamos esse problema através do uso de camadas de abstração extensíveis de forma que o desenvolvedor de aplicação possa, a seu critério, trocar de papel e atuar como um desenvolvedor de sistema criando pontos específicos para um problema ou arquitetura de acordo com suas necessidades. Assim conseguimos fornecer a transparência desejada sem comprometer possíveis demandas que venham a surgir em domínios de aplicação, tais como o uso de uma linguagem de domínio específico embarcada (Catanzaro2011) e (Chakravarty2011).

5.2 Modelos de Abstração de Paralelismo

Devido ao alto grau de complexidade no uso de bibliotecas de paralelismo muitos trabalhos têm sido dedicados a propor modelos que possam simplificar essa tarefa. Tipicamente, essa abordagem está associada a ambientes baseados em componentes, que, através de uma camada de infraestrutura, proveem uma

forma de trabalho mais abstrata. Em nosso modelo buscamos oferecer essa visão de camadas, porém sem que seja necessário um modelo de componentes associado à aplicação.

Nesta seção apresentamos diversos trabalhos que compartilham essa visão de prover serviços de paralelismo através de camadas de abstração para o desenvolvedor de aplicações.

O uso de camadas pode ajudar a diminuir as dificuldades oriundas da integração de bibliotecas e *frameworks* para paralelismo. Em (Alameh2007) é apresentada a dificuldade de explicar o uso de recursos de paralelismo para programadores com pouca experiência ou para alunos em cursos de programação. Geralmente, estes recursos apresentam um maior grau de dificuldade no aprendizado por causa da interferência que causam na abstração oferecida pela linguagem.

Em (Matthey2004) os autores reportam que o uso de uma abstração maior, através de uma infraestrutura orientada a objetos para modelar a dinâmica de moléculas, ajuda usuários a entender aplicações existentes e desenvolver novos algoritmos de forma independente da infraestrutura necessária para execução. Embora o principal objetivo do trabalho não seja o de paralelismo, o ambiente apresentado pelos autores permite uma abordagem baseada no padrão Master/Worker (Mattson2004) simplificada através de interfaces. O paralelismo é oferecido de forma desacoplada do ambiente, o que permitiria sua evolução de forma independente sem afetar estruturalmente as aplicações existentes. Essa ideia é similar à nossa proposta do uso de camadas com abstrações gradativas, porém com uma abordagem para um domínio específico, enquanto em nosso ambiente o objetivo é proporcionar facilidades para o desenvolvedor de aplicações paralelas independente do domínio de aplicação. Além disso nosso trabalho se concentra em proporcionar mecanismos de paralelismo com uma preocupação de fornecer meios para se atingir o desempenho necessário à solução da aplicação.

Uma abordagem para tratar o paralelismo é o uso de diretivas para marcar trechos de código. Um exemplo recente dessa linha pode ser encontrado em (Feng2011), onde é apresentado SpiceC, um ambiente baseado em múltiplas *threads* executoras que contam com sua própria memória local além de compartilhar uma visão para uma memória acessível por todas as *threads*. O desenvolvimento é realizado através de diretivas inseridas no código para permitir a transferência de dados. Neste ambiente o desenvolvedor de aplicação começa a ter um isolamento da camada de infraestrutura realizando seu desenvolvimento através de diretivas. A principal diferença em relação a nosso ambiente está no enfoque em máquinas com memória compartilhada. Em Numina fornecemos

uma API para memória distribuída o mais próxima e coerente possível com a API que usamos para memória compartilhada. Em nosso modelo cada máquina virtual Lua tem seu próprio ambiente de memória reservada além de também poder acessar uma memória compartilhada com os outros núcleos de execução. No entanto isso não fica tão claro quando falamos de uma execução em *cluster* já que não fornecemos controle de coerência para a memória distribuída.

StarPU (Augonnet2011) é um ambiente para agendamento de tarefas que trata os aceleradores através de bibliotecas de processamento específico, voltadas para alguma tarefa matemática, juntamente com um mecanismo de agendamento de tarefas para execução. Uma camada abstrai a arquitetura do acelerador e permite agendar uma *task* através de uma fila de execução associada a um *worker*. O usuário pode decidir dinamicamente sobre a política de agendamento que será usada no tratamento dos *workers*. Tal como em nosso ambiente, StarPU prove uma abstração para o tratamento de hierarquia de memórias de forma transparente para o usuário, porém em nosso modelo usamos uma abstração de mais alto nível quando expomos para o usuário o conceito de núcleo no qual podem ser carregadas funções para execução. Nossa visão sobre políticas de agendamento se concentra em permitir que as *threads* de nosso ambiente executem em modo de superusuário sendo que o administrador do sistema pode escolher entre as diferentes políticas de agendamento de *threads* disponíveis no sistema operacional[2].

Em (Chafi2011) é apresentada uma discussão de que atualmente é necessário mapear as aplicações para diferentes tipos de arquiteturas paralelas e que esta é uma tarefa que depende de recursos de baixo nível o que dificulta seu uso por programadores comuns. Podemos dizer que os autores apoiam a mesma ideia que defendemos de que o tratamento da complexidade envolvida no desenvolvimento de aplicações paralelas não é razoável. Sendo assim torna-se essencial disponibilizar abstrações que permitam o uso de paralelismo sem que seja necessário tornar-se um especialista nessa área. Além disso reforçam que a aplicação deve atingir bom desempenho sem que isso sacrifique a produtividade, assim consideram a métrica desempenho/homem-hora. O ambiente que propõem em seu trabalho é baseado em camadas que abstraem gradativamente os detalhes do suporte ao paralelismo, porém diferem de nossa proposta por especializarem seu ambiente no uso de linguagens de domínio específico. Fornecem uma ferramenta para auxiliar na criação deste tipo de linguagem e um ambiente para execução que mapeia diferentes tipos de arquitetura, em especial GPUs com suporte CUDA. No uso de GPUs são oferecidas duas formas de tratamento, o desenvolvedor pode fornecer seu

[2]Em ambientes Posix temos pelo menos as políticas FIFO e Round Robin

próprio código para GPU ou então marcar trechos de código para o qual deve ser gerado um código CUDA. Como o ambiente é baseado em Scala e conta com uma etapa de compilação então esse processamento para geração de código pode ser executado de forma transparente em relação ao usuário. Uma outra diferença em relação ao nosso modelo é que o desenvolvimento é direcionado para uma máquina multi-núcleo sem suporte a um modelo de execução em *cluster*.

Em (Tsoi2011) os autores concordam que um *framework* de programação genérico para unificar o desenvolvimento de aplicações paralelas é uma necessidade real já que atualmente o desenvolvedor de aplicações conta com um conjunto de arquiteturas fundamentalmente diferentes. Eles apresentam um *framework* direcionado para o tratamento de *clusters* heterogêneos basados em aceleradores tais como GPUs e FPGAs além de uma API para controle de execução e transferência de dados. O modelo apresentado conta com uma visão de *worker* e *group of works* bastante similar ao que usamos em Numina, porém em nosso modelo a abstração é centrada em um núcleo e não em uma tarefa. O *framework* lida com diferentes tipos de aceleradores empregando diferentes módulos compilados para cada arquitetura e agregando esses diferentes módulos em uma mesma aplicação. Esses módulos precisam ser controlados através de um arquivo XML que descreve a estrutura da aplicação em si. Em Numina essa tarefa é simplificada por dois aspectos: o primeiro é o fato de Lua ser uma linguagem que permite a descrição dos módulos da aplicação como parte da própria aplicação e o segundo é que por ser interpretada cada acelerador só precisa ter a máquina portada para se tornar visível no ambiente Numina, já que uma parte de nossa API é construída na própria linguagem Lua.

Uma abordagem inversa é apresentada em (Arandi2011) onde é criado um ambiente orientado a dados de forma que o problema possa ser decomposto em certas unidades de execução. O ambiente de execução é semelhante ao nosso no sentido de contar com um gerenciador de execuções e *threads* executoras, porém o ambiente é orientado para tratar a concorrência explicitada pelo programador através de macros ou de um compilador de fonte para fonte para expressar coleções concorrentes. Acreditamos que esse tipo de abordagem dificulta a adoção do modelo por necessitar uma adaptação maior das aplicações e algoritmos do usuário.

Uma modificação no ambiente Octave voltado para o processamento de matrizes é apresentada em (Khoury2011). Como é uma implementação específica direcionada para a resolução de cálculos de matrizes torna-se mais fácil prover um tipo de paralelismo automático. O ambiente conta com novo

tipo de dado chamado p_matrix além de contar com uma sobrecarga nos operadores padrão para tratar esse tipo de dado de forma paralela. Essa solução difere da nossa por ser voltada para uma linguagem de propósito específico enquanto nosso objetivo é prover facilidades de paralelismo para uma linguagem de propósito geral.

Em (Catanzaro2011) é apresentada uma argumentação de que a programação paralela ainda é vista como uma arte obscura dominada por alguns poucos especialistas e o uso de linguagens orientadas para um melhor desempenho tais como C e C++ geralmente apresentam um maior custo no ciclo de desenvolvimento do que linguagens orientadas para produtividade tais como Python e Ruby. O trabalho propõe uma linguagem embutida em Python, especificamente para tratar trechos de código que devem ser compilados para arquiteturas aceleradoras, mais especificamente para GPUs com suporte a CUDA. Eles se baseiam em fazer a transformação de código Python para C e posteriormente de C para CUDA. Essa abordagem é interessante porque abre um caminho para que nosso modelo possa suportar GPUs de maneira semelhante, isso porque as GPUs não suportam a execução de uma máquina Lua em seus núcleos de execução voltados especificamente para cálculos e com memória local mais restrita do que o processador Cell/BE.

Seguindo essa linha de embutir uma linguagem de propósito específico, é apresentada em (Chakravarty2011) uma linguagem para expressar operações coletivas sobre *arrays* juntamente com um gerador de código para CUDA de forma que as expressões são embutidas em Haskell mas executam em uma GPU. Em nosso ambiente criamos NativeArrays para permitir que fossem aplicadas otimizações de arquitetura diretamente pelo compilador, essa abordagem parece se tornar uma das escolhas mais comuns quando se trata de usar aceleradores devido às suas características.

Em (Baduel2005), os autores apresentam a experiência de aplicar o modelo de programação SPMD[3] a um ambiente orientado a objetos, discutindo o mecanismo de *typed group communication* (Baduel2002) onde um grupo de objetos expõe um tipo de dados que pode ser usado por um cliente. As facilidades do ambiente permitem ao usuário concentrar-se na aplicação em vez de se preocupar com os detalhes da comunicação e infraestrutura necessárias ao programa. As relações e interações entre os objetos podem ser isoladas e a parte de computação do algoritmo pode ser programada quase como se fosse uma versão sequencial do sistema. Em nosso modelo usamos a ideia de chamada em grupo através de um componente do ambiente no qual o

[3]SPMD – *Single Program Multiple Data* em que um único programa é executado em vários nós e cada nó recebe um subconjunto distinto de dados para processar.

usuário carrega as funções para execução paralela, a diferença é que a função carregada não precisa ser desenvolvida segundo uma visão de componentes, apenas considerando que será executada em paralelo levando em conta o particionamento de dados.

Em (Baude2007), os autores apresentam o uso de *Collective Interfaces*, que são definições de tipos para descrever quais estratégias de paralelismo serão usadas. Ao invés de escrever o código que trata a interação entre objetos, o usuário indica o que é necessário para atingir o objetivo e o ambiente usa a informação para aplicar o código de comunicação entre os objetos que fazem parte do grupo de processamento. A programação paralela é elevada a um nível mais alto de abstração em que o usuário não precisa escrever padrões de código recorrentes e pode se concentrar exclusivamente no problema que está sendo resolvido. Usamos uma ideia semelhante a essa definindo políticas de comunicação dos parâmetros para chamadas das funções definidas pelo usuário. Dessa forma, quando o desenvolvedor carrega uma função no componente do sistema deve indicar que políticas usará para cada um dos parâmetros de cada função carregada, isso permite que posteriormente a chamada seja feita uma única vez e os parâmetros sejam devidamente distribuídos.

O ambiente JOPI (Mohamed2002) se propõe a permitir que o programador migre de um paradigma de passagem de mensagem para passagem de objetos. Tal como as anteriores, essa ferramenta é baseada em Java, o trabalho com ambientes heterogêneos é herdado através do uso da máquina virtual Java. No estudo são apresentados dados que mostram a semelhança do desempenho entre a ferramenta e o uso de C com MPI, o que aponta para a possibilidade de usar melhores abstrações sem que isso implique em grandes perdas de desempenho. Em nosso ambiente usamos uma abordagem similar tornando a serialização de dados para chamadas remotas transparente para o usuário que realiza uma chamada que segue o mesmo modelo de uma local.

Em (Bigot2007), os autores apresentam uma outra abordagem para obter aplicações paralelas baseadas em componentes usando CORBA e MPI para modelar as diferentes estratégias de operações do grupo. Esta abordagem difere da (Baude2007) no sentido de que fornece uma abstração da infraestrutura subjacente, mas não fornece a infraestrutura de comunicação. Em (Baude2007), a paralelização é fornecida pelo *middleware* com recursos próprios. Ambas as abordagens indicam que o uso de camadas de infraestrutura pode ajudar a gerir a complexidade do software, assim dedicamos uma camada de nosso modelo ao controle de comunicação utilizando LuaSocket (Nehab2007).

Podemos ainda considerar que, isolando o MPI em uma camada de infraestrutura, é possível substituir a implementação usada inicialmente por outra

mais otimizada tal como as que podem ser encontradas em (Karwande2003) e (Ke2004).

Uma abordagem completamente diferente, vista por exemplo em (Charles2005), é a concepção de uma linguagem paralela totalmente nova, embora esta abordagem não seja uma opção nova por si só. Os autores apresentam X10, visando a computação em *cluster* não-uniforme. A ideia da linguagem é ter um novo conjunto de ferramentas que são projetadas a partir do zero para lidar com as especificidades que encontramos no desenvolvimento de aplicações paralelas. Uma característica importante é a escolha de usar uma máquina virtual, reforçando o conceito de abstrair os detalhes de máquina.

5.3 Bibliotecas para Suporte a Paralelismo no Cell/BE

Existe um número muito grande de bibliotecas para suporte a paralelismo em ambientes Intel/AMD, por isso consideramos apenas as bibliotecas para suporte ao processador Cell/BE pois, sendo um caso mais restrito, permite exemplificar a diversidade de abordagens que existe quando buscamos ferramentas auxiliares para o desenvolvimento de aplicações paralelas. Além disso, trata-se de uma arquitetura reconhecidamente mais complexa que a dos processadores tradicionais e devido às características específicas de sua arquitetura e das dificuldades envolvidas em programar corretamente para essa plataforma, torna-se fundamental o uso de ferramentas para auxiliar essa tarefa.

Seguindo o enfoque tradicional, os ambientes de suporte buscam prover o máximo de desempenho possível sem um comprometimento maior com o grau de facilidade para o programador. Isso gera uma dificuldade para o desenvolvedor de aplicações, pois cada arquitetura acaba tendo um conjunto diferente de ferramentas que, por terem um enfoque mais voltado para o desempenho, dificultam a visão da solução de problemas que precisam de paralelismo.

Dentre as opções de bibliotecas para suporte a paralelismo podemos considerar alguns dos exemplos destinados a arquitetura do Cell/BE.

Em (McIlroy2010) é apresentada uma máquina virtual Java modificada para processar o código do usuário de forma que *threads* do usuário possam ser migradas e executadas nos SPUs do Cell/BE. Da mesma forma que em nosso modelo, uma grande parte do ambiente é escrita na própria linguagem da plataforma para permitir maior reuso, porém em nosso modelo optamos por ter uma máquina virtual independente por núcleo. Já em (Williams2008) é apresentada uma máquina virtual Java para o processador Cell/BE composta

de duas implementações, ShellVM – mais completa – para rodar no PPE e a CellVM simplificada para rodar nos SPUs. Diferente de nosso modelo, essa máquina virtual modificada não apresenta para o desenvolvedor de aplicações paralelas nenhum tipo de abstração para permitir o controle da paralelização, ao invés disso o trabalho se concentra em permitir a execução de aplicações Java também nos SPUs.

Em (Ferrer2008) vemos uma abordagem totalmente diferente do nosso modelo baseada no uso de anotações no código para indicar o que deve ser considerada uma tarefa a ser executada em um acelerador. Um compilador modificado é fornecido de forma que o desenvolvedor possa programar em C (ou C++) e expressar os trechos que devem ser paralelizados. No entanto esse modelo é mais direcionado para o uso específico em aceleradores sem considerar a execução em *clusters*.

Uma alternativa utilizada em CellSs (Perez2007) e Sequoia (Fatahalian2006) é usar um tradutor de código para permitir que o programador marque em seu código partes que devem ser analisadas e paralelizadas pelo ambiente de execução. Em ambos os casos são usadas *threads* executoras que são controladas pelo ambiente principal. Embora nosso modelo utilize a ideia de ter um conjunto de executores, a aplicação não precisa ter marcações para indicar trechos paralelos, o desenvolvedor de aplicações escreve funções sem explicitar o paralelismo. Especificamente em Sequoia, é exposta ao usuário uma visão de memórias hierárquicas, sendo tratadas de forma recursiva através de tarefas que se subdividem. Embora o usuário não precise se preocupar com a distribuição de dados, precisa projetar sua solução adaptando seu problema a essa visão.

Uma outra forma de trabalhar é criar um ambiente para um domínio específico de problema como é o caso de PythonPS3 (Doswell2008). Este ambiente não é exatamente um modelo para processadores multi-núcleo, mas uma implementação independente para desenvolvimento de jogos para Playstation 3 baseada em Python. O interessante desse projeto é que provê uma abstração para que o programador não precise tratar todos os detalhes da programação do Cell/BE, porém limitada ao desenvolvimento de jogos simples. Como o desenvolvedor de aplicações só tem acesso a um conjunto de funções C predefinidas que executam nos SPEs, o paralelismo é muito limitado. Embora estejamos interessados em prover suporte a paralelismo para uma linguagem de propósito geral, o uso de algum tipo de linguagem de domínio específico de forma embutida pode ser interessante por mesclar as duas visões e permitir o desenvolvimento de código otimizado para uma arquitetura.

Uma abordagem que provê um aproveitamento melhor dos recursos da

máquina é CorePy (Mueller2007), que tem o propósito de esconder os detalhes do gerenciamento de *hardware* do programador fornecendo uma API para a máquina alvo, por exemplo o Cell/BE. O programador interage com objetos Python que representam o *hardware* e o código executável é gerado de forma dinâmica diretamente na memória do processador, neste caso na memória local dos SPU. Uma desvantagem desse ambiente é que para cada máquina suportada toda sua arquitetura deve ser implementada em objetos Python, o que torna o processo de suporte a novos *hardwares* mais demorado. Embora o programador seja poupado do processo de compilação de seus programas, o que no caso do Cell/BE é um grande diferencial, as aplicações não são escritas em Python, essa linguagem é utilizada apenas como ambiente de suporte para o desenvolvimento de aplicações que acontece em código *assembly* da máquina destino. Cada instrução da máquina deve estar disponível na biblioteca Python para ser usada pelo programador, mas o desenvolvimento de aplicações se torna mais complicado. Dessa forma consideramos como alternativa o uso de técnicas JIT na máquina virtual como uma alternativa mais interessante.

Um trabalho que motivou nossa visão de fornecer um ambiente voltado para simplificar o desenvolvimento de aplicações é BlockLib (Alind2008). Através de uma biblioteca que fornece modelos de padrões de paralelismo os autores permitem que o usuário se concentre mais em sua aplicação, porém o código deve ser inserido através de um ponteiro de função para ser compilado com as bibliotecas do ambiente. Embora o código que o programador escreva não seja específico para Cell/BE, é necessário conhecer os detalhes da arquitetura para poder usar corretamente as facilidades fornecidas pela ferramenta. Tentamos minimizar essa necessidade em nosso ambiente embutindo algumas das especificidades da arquitetura tal como controle de alinhamento de dados em memória, além disso acreditamos que o uso de uma linguagem interpretada pode trazer mais benefícios para a evolução e otimização do modelo.

Em todos esses trabalhos, a principal preocupação é atingir o melhor desempenho possível com algum grau de simplificação em relação as ferramentas de programação do Cell/BE originais. O problema com essa abordagem é limitar as possibilidades por conta de um compromisso prematuro de manter o desempenho. Nosso trabalho inverte essa abordagem com o objetivo de fornecer melhores facilidades ao desenvolvedor embutindo as questões de paralelismo no ambiente para que possam ser evoluídas de forma independente.

Nossa visão é semelhante a que pode ser encontrada em Charm++ (Kunzman2009), onde os autores abordam o problema da dificuldade de programação. Utilizando um novo tipo de método de entrada, capaz de

ser agendado para o processamento em aceleradores[4], os autores abstraem as diferenças de hardware do desenvolvedor de aplicações e tomam para o ambiente a decisão de como executar aquele código da melhor maneira. O Charm++ é baseado em um conjunto de bibliotecas que são ligadas entre si com o código e, embora possa ser usado em várias implementações, ele deve ser compilado para o host específico. O trabalho é diferente do nosso, no sentido em que diferencia o uso de aceleradores de núcleos normais explicitamente.

5.4 Conclusão

O que podemos perceber é que existem duas visões disjuntas sobre o problema de desenvolvimento paralelo: de um lado a busca pelo melhor desempenho possível e do outro a busca por melhores abstrações que permitam expressar os conceitos relativos ao desenvolvimento de aplicações paralelas. Embora a busca por melhor desempenho seja uma constante as pesquisas começam a apontar caminhos para simplificar a tarefa dos desenvolvedores de aplicações paralelas através do uso de algumas abstrações baseadas no uso de algum tipo de camada.

O reconhecimento da importância de dividir os papéis de desenvolvedores entre aplicação e sistema e, mais ainda, de que a responsabilidade pela melhoria de desempenho pode ser deixada para o ambiente de infraestrutura permite embasar a busca pelo uso de um ambiente baseado em camadas que abstraiam gradativamente os detalhes de aplicações paralelas.

No entanto ainda não existe um consenso sobre qual seria a melhor forma de prover facilidades para o desenvolvedor de aplicações. Uma possibilidade para esta situação é de ainda não terem sido claramente definidos os conceitos e abstrações relativos a essa área como apresentado em (Vishkin2011), o que acaba por dificultar implementações que permitam expressar operações mais claramente. Este tipo de definição permitiria o desenvolvimento de otimizações internamente, em componentes, o que refletiria no desempenho do código de aplicações que utilizassem esses componentes.

Finalmente, o uso de linguagens de propósito específico embutidas em outras linguagens de programação de propósito geral, em especial linguagens interpretadas, parece despontar como uma solução para o uso de dispositivos aceleradores como as placas GPU.

[4]Acelerador pode ser qualquer tipo de núcleo especializado, como as unidades Cell/BE SPE