Uma Infraestrutura de Alto Desempenho para a Execução de Aplicações Paralela

Juliano Henrique Foleiss, Rodrigo Hübner, Anderson Faustino da Silva
Laboratório de Linguagens, Compiladores e Programação Paralela
Departamento de Informática
Universidade Estadual de Maringá
Maringá, Brasil
Email: julianofoleiss@gmail.com, rhubner@gmail.com, anderson@din.uem.br

***Resumo*—Este artigo apresenta a `TVM`, uma máquina virtual baseada em processos, que gerencia um ambiente de execução multitarefa com suporte à memória transacional. Uma máquina virtual com estas características é alcançada por meio de um conjunto de instruções bem projetado que incluem primitivas de controle de fluxo paralelo e acessos à memória com semântica transacional. `TVM` também é responsável por criar e escalonar tarefas utilizando políticas de prioridade. O objetivo principal é prover um ambiente portável, leve e de alto desempenho para projeto e desenvolvimento de linguagens de programação paralela.**

***Palavras Chave*-Paralelismo; Máquina Virtual; Desempenho; Memória Transacional;**

I. Introdução

Até meados da década passada o avanço na tecnologia de microprocessadores era baseado em aumentar a frequência do *clock* [1]. Essa metodologia, no entanto, esbarra em uma barreira física: o consumo de energia aumenta drasticamente, juntamente com o calor que é dissipado. Com o objetivo de ultrapassar esta barreira, novas pesquisas seguiram em um caminho diferente. Ao invés de aumentar a frequência do *clock* foram incluídos vários núcleos processadores em um único chip [1], [2].

Nos últimos anos, a adoção dessa tecnologia vem se tornando cada vez mais difundida, permitindo a execução de programas paralelos em um número cada vez maior de processadores comerciais.

No entanto, programar para um ambiente paralelo não é algo trivial. Whu *et al.* [2] discutem que a criação de programas paralelos com modelos explícitos pode se tornar algo altamente propensa a erros, além de exigir um certo grau de *expertise* para o projeto e implementação de aplicações simples. Portanto, a premissa de Whu *et al.* é que existe a necessidade de mecanismos que simplifiquem o desenvolvimento de aplicações paralelas.

Com a complexidade crescente das aplicações, é necessário que todos possam ser capazes de projetar e implementar programas que usufruam das capacidades que o *hardware* moderno oferece.

Os modelos implícitos de programação paralela normalmente deixam o trabalho de paralelizar a aplicação a cargo do *toolchain* de compilação, e em certos casos, até de um *hardware* específico. O nível de complexidade desses compiladores pode chegar a níveis avançados, utilizando técnicas avançadas de análise e transformação de código [2], [3], [4], e principalmente, de uma infraestrutura capaz de gerenciar o ambiente de execução paralela [5], [6].

Seguindo uma abordagem diferente, este trabalho propõe uma infraestrutura capaz de gerenciar eficientemente um ambiente de execução paralela. O objetivo principal é facilitar o desenvolvimento de compiladores para linguagens paralelas, provendo um ambiente de execução leve e versátil. A infraestrutura proposta é composta por um conjunto de ferramentas denominado `TVMTK` (*TVM ToolKit*) que possui uma máquina virtual, um compilador e um analisador. A máquina virtual, denominada `TVM` (*Task-based Virtual Machine*), que é o principal componente desta infraestrutura, tem como principais características proporcinar acessos transacionais à memória disponibilizando um mecanismo nativo de sincronização e proporcionar um sistema de gerenciamento e escalonamento de tarefas eficiente.

São duas as principais contribuições deste trabalho. Primeiro, disponibilizar uma infraestrutura de execução paralela de alto desempenho, capaz de utilizar uma arquitetura de *hardware* híbrida. E segundo, prover uma estratégia que proporcione às aplicações portabilidade com escalabilidade.

Este artigo está organizado como segue. A Seção II discute os trabalhos relacionados que motivaram este trabalho. A Seção III descreve as características da `TVM`. E finalmente, a Seção IV apresenta as conclusões e perspectivas futuras.

II. Trabalhos Relacionados

A programação paralela *multicore* tem sido foco de diversas pesquisas nos últimos anos [7]. Com o surgimento e apoio amplamente difundido das *threads* nos sistemas operacionais modernos, diversas bibliotecas como `pthreads` [8], `GNU Threads` [9] e `OpenMP` [10] foram propostas para facilitar e simplificar o desenvolvimento de programas paralelos em linguagens como `C` e `Fortran`.

Contudo, o modelo de memória compartilhada, inerente de programação com *threads*, necessita de mecanismos de sincronização confiáveis que permitam a proteção a acessos concorrentes aos dados em memória principal [7]. Na programação com *threads* é necessário que os pontos de

sincronização sejam determinados pelo usuário, o que pode levar a erros de programação difíceis de serem encontrados. Com o objetivo de simplificar a programação paralela utilizando *threads*, algumas linguagens de programação contendo construções paralelas foram propostas para facilitar o desenvolvimento de aplicações paralelas, como `Concurrent C` [11] e `UPC` [12], entre outras.

Embora essas linguagens facilitem o desenvolvimento de aplicações paralelas, ainda existem dois problemas importantes a serem solucionados, a saber: (a) em alguns casos a contenção gerada pelos mecanismos tradicionais de sincronização chega a níveis inaceitáveis [7]; e (b) toda a gerência da sincronização de dados continua sendo responsabilidade dos desenvolvedores. Herlihy e Moss [13] propõem um modelo de sincronização denominado memória transacional, que dispensa a utilização de *locks* e *barreiras* como uma uma solução plausível tanto para (a) quanto para (b).

Possivelmente o maior desafio na programação paralela é o particionamento do problema. Analisar um algoritmo candidato para solucionar um problema e dividí-lo para explorar o máximo do paralelismo subjacente não é algo trivial [7]. Além disto, a complexidade do ambiente de execução aumenta drasticamente em relação aos ambientes convencionais. Isso se deve ao fato de que várias tarefas são executadas por um tempo não pré-estabelecido e de maneira não-determinística, e ainda por arquiteturas que podem conter máquinas com perfis diferentes de *hardware* e *software*.

A linguagem de programação `Java` [5] possui um ambiente de execução especificado pela Sun Microsystems denominado *Java Virtual Machine* (JVM). Além de um ambiente de execução robusto e bem projetado, a JVM possui algumas facilidades para programação paralela como a implementação de monitores e *threads*. `Java` é independente de plataforma e atualmente seu desempenho é comparável à linguagens tradicionais como `C` e `Fortran` [14].

O objetivo da `TVM` é ser um ambiente de execução independente de plataforma e de linguagem fonte, permitindo o desenvolvimento de compiladores para linguagens paralelas por meio de um mecanismo de gerência de tarefas e memória transacional.

III. A Máquina Virtual

A `TVM` é uma máquina virtual baseada em tarefas que gerencia um ambiente de execução paralelo e possibilita a intercomunicação entre suas tarefas por meio de um espaço de endereçamento comum que possui semântica transacional. A máquina oferece tais facilidades por meio de um conjunto de instruções de baixo nível, que além de incluir instruções lógicas, aritméticas e de controle de fluxo, também inclui primitivas de gerência paralela e acesso à memória transacional.

A. A Arquitetura da TVM

A organização da `TVM` está disposta na Figura 1. Basicamente, a `TVM` é organizada em duas áreas, a saber:

1) Área de Dados Compartilhados (ADC); e
2) Área de Processadores (AP).

A ADC consiste na área de memória compartilhada entre todos os processadores em execução e também é dividida em duas seções:

1) Seção de Dados Gerenciais (SDG); e
2) Seção de Dados Transacionais (SDT).

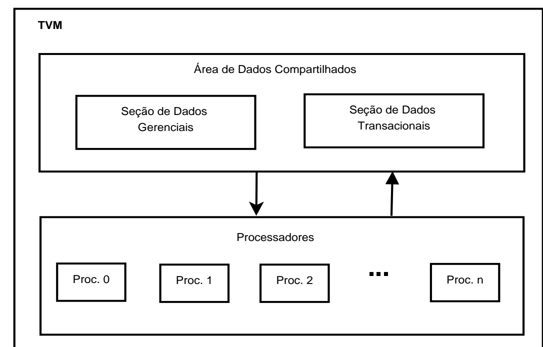

Figura 1. Organização da TVM

A SDG constitui-se de dados importantes para a gerência do ambiente de execução, a saber: *descritores de processadores*, *descritores de tarefas*, *repositório de código nativo* gerado por um compilador *Just-In-Time* e o *repositório de tarefas nativas*. Essa memória é manipulada somente pelos mecanismos internos da `TVM` e não é visível ao usuário.

A SDT é a área da memória destinada para o processamento útil da `TVM`, ou seja, é a área de memória compartilhada entre todos os processadores e suas tarefas. O acesso a esta seção é feita apenas por meio de instruções *load/store* quando a máquina está no modo de acesso transacional.

A AP é a área da `TVM` que abriga os *processadores virtuais*. Cada processador é visto como uma abstração de um computador contendo as informações necessárias para a execução de suas tarefas. Os processadores são responsáveis por executar e escalonar as tarefas que são alocadas a eles. Tais tarefas são definidas pelo usuário e são invocadas por meio de instruções específicas de máquina.

Não há nenhuma restrição sobre como o ambiente deve ser implementado pragmaticamente. Cada processador pode ser uma instância de um processo, uma *thread* ou até processos em computadores distintos. A organização da memória segue o mesmo princípio. A memória não está necessariamente localizada em uma única máquina hospedeira, podendo ser distribuída entre os computadores de um *cluster* (por meio de um *software* DSM [15]), por exemplo. A escolha de uma organização específica é feita durante a configuração do ambiente.

B. O Processador

O processador da TVM é mostrado na Figura 2. O *motor de execução* é responsável pela decodificação e execução das instruções de máquina da TVM. O *escalonador de tarefas* é o mecanismo que permite a execução de diversas tarefas por um processador. Este módulo trabalha gerenciando uma *fila* de tarefas contendo os descritores das tarefas locais, de modo a privilegiar as tarefas com a maior quantidade de acessos à memória compartilhada, garantindo assim, um tempo maior de processamento. O modelo de execução utilizado pela TVM é bem peculiar, sendo apresentado na Seção III-E.

Cada tarefa possui seu *contexto*, que é salvo automaticamente (no módulo de contextos) quando a tarefa sai de execução e restaurado quando esta volta a ocupar o processador. Com efeito, as tarefas da TVM assemelham-se as tarefas de um sistema operacional convencional.

Um contexto de uma determinada tarefa é constituído por um subsistema de memória local e pelos dados que descrevem o estado da execução de cada tarefa: *os registradores de propósito geral*, *os registradores de status do sistema*, *os registradores de pilha* e *o contador de programa*.

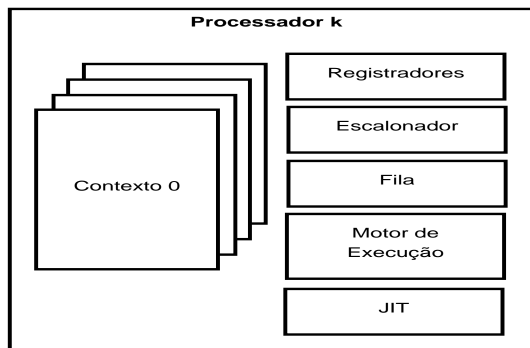

Figura 2. O Processador da TVM

C. O Conjunto de Instruções

O conjunto de instruções da TVM, denoninado TVMIS, é inspirado no conjunto da LLVM [16], o LLVA [17], contando com marcações de alto nível para melhorar o desempenho das otimizações. O conjunto de instruções é constituído de instruções lógica e aritmética, desvios condicionais e incondicionais e entrada e saída. Além destas instruções, o TVMIS possui instruções para o controle de fluxo paralelo, como instruções para iniciar uma determinada tarefa, trocar o modo de acesso à memória (transacional ou local) e alocação de memória na pilha e/ou na *heap*.

D. O Subsistema de Memória

O subsistema de memória é composto por uma memória local e por uma memória global. A memória local de um processo é o seu espaço de endereçamento privado. Por outro lado, a memória global, que contém os dados compartilhados entre diversas tarefas, é um espaço de endereçamento comum.

Para fazer a distinção entre os diferentes espaços de endereçamento, a TVM fornece dois modos de acesso, a saber: *modo de acesso local* e *modo de acesso transacional*. No modo de acesso local, as instruções *load/store* acessam o espaço de endereçamento privado. Já no modo de acesso transacional, estas instruções acessam a SDT. O modo pode ser alternado durante a execução da aplicação, por meio da utilização de instruções específicas para este propósito.

E. O Modelo de Execução

O carregador do sistema é responsável pela carga do código-objeto do programa, que pode conter tarefas e/ou procedimentos. Inicialmente, são criados descritores de todas as tarefas que são armazenados no repositório de descritores de tarefas. Estes descritores contém informações úteis para a gerência da execução, como o *custo da tarefa* e os *apontadores para todas as versões executáveis da tarefa* (versões binárias TVM e JIT). Para que não haja necessidade de sincronização, as tarefas que são carregadas na inicialização da TVM são somente-leitura. Tarefas adicionais podem ser carregadas em tempo de execução com a utilização de uma primitiva associada, tornando-se somente leitura após sua carga.

Os descritores dos procedimentos também são armazenados no repositório de descritores de tarefas e possuem as mesmas propriedades que as tarefas. Desta forma, a execução de procedimentos e tarefas podem ser feitos de maneira simples e compatível. É importante ressaltar que somente procedimentos globais possuem descritores no repositório. Procedimentos locais não necessitam de descritores compartilhados pois são apenas invocados a partir da tarefa que o inclui.

A indexação das tarefas é feita com base no nome da tarefa e a política de indexação pode ser escolhida pelo usuário no momento de configuração do ambiente de execução. Este pode optar por uma política de utilização de tabelas *hash* ou de árvores de busca rearranjáveis.

Após a carga de todas as tarefas, a TVM cria o primeiro processador e atribui como sua primeira tarefa, a tarefa principal. Conforme novas tarefas são instanciadas a partir de outras tarefas, novos processadores são criados, até atingir um limite pré-estabelecido.

Para cada processador criado, um descritor é colocado na área de gerência de processadores. Cada descritor contém informações como: *tarefas em execução*, *custo total da execução corrente*, *tarefas candidatas a relocação* e outras informações importantes para a sinergia entre os processadores.

Quanto o limite de processadores estiver atingido, as próximas tarefas criadas são inseridas nas filas dos próprios processadores, que são responsáveis por escaloná-las. O

escalonador é preemptivo com o objetivo de realizar a troca do contexto das tarefas de maneira transparente e eficiente. A opção por um escalonador preemptivo provém de duas premissas básicas, a saber: (a) simplificação do código do usuário, que não precisa se preocupar com o mecanismo de execução; e (b) permite à `TVM` que de maneira inteligente, encontre uma boa sequência de execução das tarefas para que o programa possa ser executado da maneira mais eficiente possível.

Em particular, o escalonador é responsável por três operações básicas, a saber: (1) escolher o tempo e a ordem de execução de cada tarefa, seguindo uma política de prioridade; (2) selecionar a tarefa que gostaria que fosse relocada a outro processador; e (3) quando ocioso ou com uma carga abaixo de um limiar, executar uma tarefa de outro processador.

A política de prioridade utilizada é denominada *custo da tarefa*. Esse custo é baseado no custo raso de suas operações (contando apenas com o custo das operações de suas chamadas locais em um único nível) e na quantidade de variáveis compartilhadas que acessa. Este custo, que está presente no código-objeto, é calculado por um módulo do montador da `TVM`. Além desta métrica, o ambiente de execução pode utilizar a prioridade estabelecida pelo próprio usuário. Esta prioridade auxilia a `TVM` no cálculo da prioridade real quando o custo calculado para uma tarefa não representa de forma significativa o custo real da tarefa.

IV. Conclusões e Perspectivas Futuras

Neste trabalho foi apresentado um *overview* de uma infraestrutura para a execução de aplicação paralelas, cujo principal componente é a `TVM`, uma máquina virtual baseada em tarefas que visa a execução de programas paralelos independentes de arquitetura. O principal objetivo desta infraestrutura é facilitar o desenvolvimento de novos paradigmas de linguagens de programação paralela, facilitando assim a criação e a implementação dos ambientes de execução e mecanismos de sincronização para modelos implícitos e explícitos.

O conjunto de ferramentas `TVMTK` está no momento em desenvolvimento. Após a finalização deste *toolkit*, o próximo objetivo deste projeto é desenvolver uma versão paralela de Prolog.

Agradecimentos

Agradecemos ao CNPq pelo auxílio financeiro que fomenta este projeto.

Referências

[1] W. Stallings, *Computer Organization and Architecture: Design and Performance*, 8th ed. USA: Prentice Hall, 2009.

[2] W. mei Hwu *et al.*, "Implicitly Parallel Programming Models for Thousand-core Microprocessors," in *Proceedings of the Design Automation Conference*, 2007, pp. 754–759.

[3] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006, pp. 151–162.

[4] J. Mak, K.-F. Faxen, S. Janson, and A. Mycroft, "Estimating and Exploiting Potential Parallelism by Sourcelevel Dependence Profiling," in *Proceedings of the International EuroPar*, 2010, pp. 26–37.

[5] K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Language*, 2nd ed. USA: Addison Wesley, 2000.

[6] A. F. da Silva, M. Lobosco, and C. L. Amorim, "An Evaluation of cJava System Architecture," in *Proceedings of the Symposium on Computer Architecture and High Performance Computing*, 2003.

[7] C. Lin and L. Snyder, *Principle of Parallel Programming*, 1st ed. California, USA: Addison Wesley, 2009.

[8] D. R. Butenhof, *Programming with POSIX threads*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[9] Free Software Foundation, "GNU Portable Threads," 2010, http://www.gnu.org/software/pth/; acesso em 20 de Setembro de 2010.

[10] B. Chapman, G. Jost, and R. v. d. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007.

[11] N. H. Gehani and W. D. Roome, "Concurrent C," *Softw., Pract. Exper.*, vol. 16, no. 9, pp. 821–844, 1986.

[12] T. El-Ghazawi and L. Smith, "UPC: unified parallel C," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, 2006.

[13] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-free Data Structures," in *Proceedings of the International Symposium on Computer Architecture*, 1993, pp. 289–300.

[14] J. H. Foleiss and A. F. da Silva, "Reavaliando a Lacuna do Desempenho entre as Linguagens Java, C e C++," in *Proceedings of the XXXVI Conferência Latino Americana de Informática*, 2010.

[15] J. Protic, M. Tomasevic, and V. Milutinovic, "Distributed Shared Memory: Concepts and Systems," *Parallel Distributed Technology: Systems Applications, IEEE*, vol. 4, no. 2, pp. 63 –71, 1996.

[16] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[17] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke, "LLVA: A Low-level Virtual Instruction Set Architecture," in *Proceedings of the 36th annual ACM/IEEE international symposium on Microarchitecture (MICRO-36)*, San Diego, California, Dec 2003.