PFUN PARA AMBIENTES DE MEMÓRIA COMPARTILHADA

Douglas E. Rosa[1], Lucas D. Fonseca[1,*], Cícero Camargo[1], André R. Du Bois[3], Maurício L. Pilla[2], Gerson G. H. Cavalheiro[2]

[1]Curso de Bacharelado em Ciência da Computação – Universidade Federal de Pelotas

[2]Departmento de Informática – Universidade Federal de Pelotas

[3]Centro Politécnico – Universidade Católica de Pelotas

{douglas.erosa, lucas.fonseca, cicero.camargo}@anahy.org,
{mauricio.pilla, gerson.cavalheiro}@ufpel.edu.br, dubois@ucpel.tche.br

1. Introdução

As arquiteturas *multicore* já fazem parte da realidade da maioria dos usuários de computadores. Este fato implica em um grande aumento da importância de dominar os conceitos de programação paralela. É raro encontrar programadores que conseguem ter um entendimento pleno dessa área, principalmente devido a alta complexidade na expressão do paralelismo das aplicações.

Linguagens funcionais apresentam uma propriedade interessante nesse contexto. Um programa funcional puro não necessita que suas subexpressões sejam avaliadas seqüencialmente e em uma ordem definida. O controle das comunicações de resultados se dá pelo controle do fluxo de dados [Hammond 1999].

Tais propriedades favorecem o uso de algoritmos de lista [Graham 1969]. No projeto Anahy [Cavalheiro 2006] foi proposto um ambiente de execução eficiente baseado em algoritmo de lista. A implementação deste em Cordeiro (2005) confirmou que programas paralelos cujo controle acontece pelo fluxo de dados podem ser executados de forma eficiente em arquiteturas multiprocessadas.

Du Bois (2007) propôs uma extensão a linguagem funcional Haskell, com o objetivo de facilitar a notação do paralelismo presente nas aplicações. Esta nova linguagem usou como ambiente de execução uma máquina virtual (RTS – *RunTime System*) a ser executada em arquiteturas distribuídas. Este trabalho propõe a modificação desta máquina virtual de modo que a execução passe a acontecer em arquiteturas *multicore*, tendo como suporte um núcleo de escalonamento baseado em algoritmo de lista.

2. A Linguagem pFun

A linguagem pFun [Du Bois 2007] é uma linguagem funcional pura que herda a sintaxe de Haskell, porém acrescida de duas primitivas para a descrição da concorrência: *par* e *sync*. A primitiva *par* indica que a expressão pode ser avaliada em paralelo e a primitiva *sync* retorna o resultado de uma expressão *par*.

A implementação da linguagem está organizada em três camadas, sendo a de mais alto nível correspondente à interface de programação.

A camada intermediária é responsável pela portabilidade de código. A execução em ambientes heterogêneos é garantida através da compilação para *bytecode*, o qual é executado na máquina virtual pFun (RTS). O interpretador de *bytecodes* é baseado na

*Bolsista BIC FAPERGS

SECD machine [Landin 1964], utilizando-se apenas uma pilha. As estruturas internas são alocadas em tempo de execução em uma área de memória controlada denominada *heap*, as quais são automaticamente liberadas por um coletor de lixo quando não são mais necessárias.

A camada de mais baixo nível é responsável pela portabilidade e desempenho. A distribuição da carga computacional é feita por um escalonador, o qual garante a ordem de execução das expressões segundo as anotações no código das primitivas *par* e *sync*.

Há duas estruturas relevantes no funcionamento da linguagem: a fila de tarefas e a fila de *threads* executáveis. A fila de tarefas, também chamada de *taskpool* (ou *ready queue*) é uma lista de tarefas que podem potencialmente ser executadas em paralelo. Tarefas são objetos *Par* que são criados ao ser encontrada a primitiva *par.* A fila de *threads* executáveis, ou *threadpool*, tem a função de guardar *threads* que bloquearam-se, esperando avaliação de outra *threads*, mas foram desbloqueadas e estão aptas a seguirem sua execução. O conceito *thread* aqui representa um objeto *Par* em execução.

3. Implementação Distribuída

A linguagem pFun foi desenvolvida originalmente para arquiteturas com memória distribuída, como *clusters*. A estratégia de escalonamento é próxima ao modelo cliente-servidor. A troca de mensagens implementada pela RTS ocorre de maneira distribuída entre um *work server* (servidor) e seus *slaves* (clientes). Um programa pFun é executado inicialmente por um *work server*, o qual gera tarefas para serem distribuídas entre seus *slaves* ou até mesmo outros *work severs*.

Quando um *slave* pede por trabalho, o *work server* retira uma tarefa da sua *taskpool*, e envia para ser avaliada pelo *slave*. O *slave* não possui *taskpool*, portanto, se durante a avaliação do programa o *slave* encontra uma chamada *par*, uma *thread* é criada para avaliar a expressão localmente, enquanto isso a *thread* que estava em execução fica bloqueada aguardando. Em virtude disto, surge o principal problema desta implementação: quando o programa gera tarefas de diferentes granulosidades pode ocorrer um grave desequilíbrio na distribuição de carga no sistema, onde alguns *slaves* possuem tarefas grandes para serem avaliadas ficando assim sobrecarregados, enquanto outros *slaves* avaliam pequenas tarefas, podendo ficar grande parte do tempo ociosos. Nesta implementação não há comunicação entre *slaves*.

Quando um *work server* recebe um pedido de trabalho, a representação da tarefa no *heap* (o *Par)*, é serializada e enviado para o destinatário. Caso o *work server* não possua tarefas em sua *taskpool*, ele então envia uma mensagem comunicando que não possui trabalho.

Em suma, a tarefa de escalonamento distribuída é executada no seguinte laço:

1 – Se há tarefas na *threadpool*, executa-se uma delas.

2 – Se a *threadpool* não possui tarefas busca-se trabalho em um *work-server*. No caso do w*ork server,* cria-se uma nova tarefa para avaliar um objeto *Par* da *taskpool*.

3 – No w*ork server,* se a execução não foi finalizada e a *taskpool* está vazia, busca-se trabalho em outro *work-server*.

4. Implementação para Memória Compartilhada

Nesta versão, atualmente em desenvolvimento [Camargo 2008], é feito o uso de apenas uma máquina virtual pFun. A organização deixa de ser no modelo cliente-servidor em que o escalonamento de tarefas é realizado pelo *work server* e passa a ser utilizada a idéia de processadores virtuais, onde o escalonamento é controlado por mecanismos de sincronização. O uso de processadores virtuais eliminam a hierarquia existente no modelo distribuído entre *work server* e *slaves*. Os processadores virtuais são processos leves que compartilham o mesmo espaço de memória.

A Figura 1 ilustra as principais diferenças entre o modelo distribuído e o para memória compartilhada. Cada processador virtual manterá um *heap* e uma *threadpool* locais, de forma semelhante à versão distribuída. O motivo dessa decisão é evitar que o uso de um *heap* global possa serializar a execução, além de evitar possíveis condições de corrida. Os processadores virtuais possuem suas próprias *taskpool*, porém há uma *taskpool* global contendo ponteiros para a *taskpool* de cada processador virtual, criando a abstração de uma *taskpool* globalmente compartilhada. Esta abstração foi adotada para permitir a estratégia de escalonamento baseada em *Work Stealing* [Blumofe 1994]. Esta estratégia baseia-se no balanceamento de carga entre os processadores, de modo que um processador ocioso roube uma tarefa de outro processador sobrecarregado. O roubo deve ser de tarefas com granulosidade grossa, i.e., que apresentam maior carga computacional, pois evita a necessidade de muitos roubos sucessivos e diminui a sincronização.

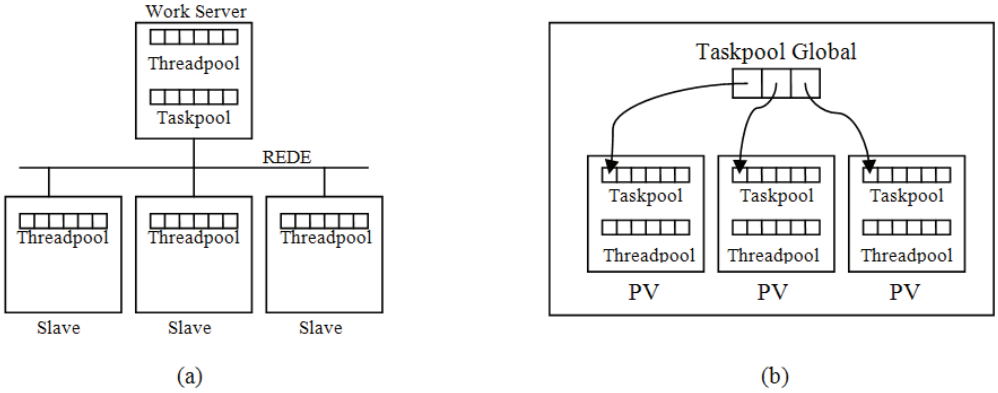

Figura 1 – Comparação entre (a) Implementação Distribuída e (b) Implementação com Memória Compartilhada

O modelo distribuído não apresentava bom desempenho em aplicações com diferentes granulosidades. A versão para memória compartilhada adota o uso de *work stealing* entre os processadores virtuais, o que faz com que a carga computacional seja melhor distribuída, eliminando este problema. Em aplicações do tipo *bag of tasks*, onde todas as tarefas possuem granulosidade homogênea, o desempenho foi maior devido a não haver os custos de comunicação da rede.

5. Conclusões

Neste artigo foi apresentada a implementação adaptativa da linguagem pFun para sistemas multiprocessados de memória compartilhada. Foram mostradas soluções para problemas encontrados na versão distribuída.

Em trabalhos futuros pretende-se terminar a validação e realizar testes de desempenho no pFun com *work stealing*. Além disso, pretende-se implementar a estratégia de *work stealing* na versão distribuída e após, uma versão híbrida para possibilitar a exploração de paralelismo inter e intra nó.

Referências

Blumofe, R.; Leiserson, C. **Scheduling multithreaded computations by workstealing**. Proceedings of the 35th Annual Symposium on Foundations of ComputerScience, Santa Fe, New Mexico, p. 356–368, November 1994.

Camargo, C.; Rosa, D. E.; Fonseca, L. D.; Cavalheiro, G.; Du Bois, A. R. **Adaptação da Linguagem pFun para Ambientes de Memória Compartilhada**. In: Second Workshop on Languages and Tools for Parallel and Distributed Programming. 2008.

Cavalheiro, G. G. H. ; Gaspary, L. P. ; Cardozo Júnior, M. A. ; Cordeiro, O. C. **Anahy: A programming Environment for cluster computing**. In: High Performance Computing for Computational Science - VECPAR 2006, 2007, Rio de Janeiro. LNCS High Performance Computing for Computational Science - VecPar 2006. Heildelberg : Springer-Velag, 2006. p. 198-211.

Cordeiro, O. C. ; Peranconi, D. S. ; Villa Real, L. C. ; Dall Agnol, E. C. ; Cavalheiro, G. G. H. . **Exploiting Multithreaded Programming on Cluster Architectures**. In: HPCS 2005 - International Symposium on High Performance Computing Systems and Applications, 2005, Guelph. 19th International Symposium on High Performance Computing Sustems and Applications. Los Alamitos : IEEE Computer Society, 2005. v. 1. p. 90-96.

Du Bois, A. R.; Cavalheiro, G.; Yamin, A.; Pilla, M. **pFun: A semi-explicit parallel purely functional language**. In: Electronic Proceedings of the First Workshop On Languages and Tools for Parallel and Distributed Programming. 2007.

Graham, R. L. **Bounds on Multiprocessing Timing Anomalies**. SIAM Journal ofApplied Mathematics, v.17, n.2, p.416-429, 1969.

Hammond, K., Michaelson, G., editors. **Research Directions in Parallel Functional Programming**. Springer-Verlag, UK, 1999.

Landin, P. The mechanical evaluation of expressions. The Computer Journal,

6(4):308–320, 1964.