Construção do Ambiente *Multithread* Dinâmico Anahy-3: Escalonamento com Suporte à *Work-Stealing*[*]

Alan S. de Araujo[†], **Cícero Augusto S. Camargo**[‡], **Guilherme Porto B. Cousin**[§], **Gerson Geraldo H. Cavalheiro**

Universidade Federal de Pelotas – UFPel
Centro de Desenvolvimento Tecnológico – CDTec
Laboratory of Ubiquitous and Parallel Systems – LUPS

{asdaraujo,cadscamargo,gpbcousin,gerson.cavalheiro}@inf.ufpel.edu.br

Abstract. *Ambientes dinâmicos de execução além de facilitar a descrição do programa, possuem um núcleo de execução próprio capaz de executar a aplicação* multithreaded *melhorando seu desempenho. Anahy é um destes ambientes desenvolvido para explorar a execução sobre* clusters*. Entretanto, otimizações de escalonamento de* threads *internas aos nós não são consideradas pela ferramenta. A partir disso, surge a necessidade de construção de um novo ambiente de execução otimizado para arquiteturas multicore.*

1 Introdução

O núcleo de escalonamento dos ambientes *multithread* dinâmicos, tais como Cilk [Frigo et al. 1998], KAAPI [Gautier et al. 2007] e Anahy [Cavalheiro et al. 2007], age como um gestor de decisão que mapeia as tarefas concorrentes descritas no código provido pelo programador, sobre o conjunto de processadores virtuais (VPs) disponíveis na abstração de arquitetura concebida por estes ambientes. Este mapeamento é feito por um algoritmo de escalonamento que segue alguma política de prioridade pré-estabelecida e, então, realiza a alocação das tarefas (organizadas como um grafo de dependências) aos VPs. O ambiente Anahy foi inicialmente concebido para oferecer uma abstração de arquitetura multiprocessada sobre uma arquitetura de *cluster*. Seu núcleo de execução é, portanto, composto por uma coleção de VPs distribuídos nos nós do *cluster*.

Ao longo do tempo surgiu a necessidade de atualizar a implementação do ambiente Anahy para que o núcleo de execução possa considerar as novas tecnologias de hardware multiprocessado. A arquitetura de Anahy foca no uso de *clusters* e não considera a otimização do escalonamento dos *threads* internamente aos nós. Isso gera muita contenção entre os VPs, o que penaliza o desempenho de execução do ambiente. Neste artigo, é apresentado o desenvolvimento de um núcleo de execução otimizado para arquiteturas multicore. Este novo núcleo faz parte do novo ambiente dinâmico de execução Anahy-3 e seu núcleo de escalonamento suporta operações de *work-stealing* [Guo 2010].

[*]Este trabalho é parte do projeto "Green Grid", financiado pelo Programa PRONEX FAPERGS/CNPq.
[†]Bolsista PIBIC/CNPq
[‡]Bolsista CAPES
[§]Bolsista PIBIC/CNPq

2 Núcleo de Escalonamento de Anahy-3

O núcleo de execução de Anahy-3 é composto por VPs, implementados como *threads* sistema e criados no início da execução do programa. Dessa forma o escalonamento é realizado em dois níveis. O primeiro consiste no mapeamento dos VPs aos núcleos físicos da arquitetura. O segundo nível consiste no mapeamento dos *threads* descritos pelo programador sobre os VPs. As dependências de dados entre os *threads* são mantidas pelo Anahy-3 sob a forma de um Grafo Cíclico Dirigido (DCG). Na alocação dos *threads* sobre VPs é considerada a ordem de execução dos *threads*, refletindo assim o controle semântico de toda a execução. A estratégia de escalonamento local, aplicada sobre cada VP, prioriza a execução dos *threads* mais recentemente criados sobre o próprio VP. Desta forma, o grafo que descreve as dependências de execução entre as tarefas dos *threads* é distribuído entre os VPs da arquitetura. Isso tem como objetivo aumentar a independência de execução dos VPs, explorando de maneira mais eficiente as dependências de execução entre *threads* pai e filho.

2.1 *Work-Stealing* em Anahy-3

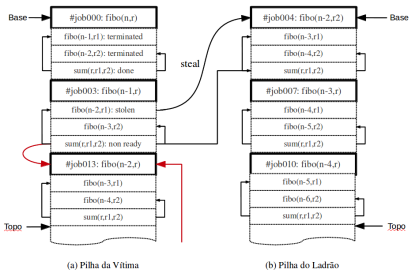

Figura 1. Pilhas de execução de VPs em uma operação de roubo de trabalho.

Operações de *work-stealing* fazem uso de listas de trabalhos aptos à execução, sendo estes trabalhos pertencentes ao escopo de cada VP do ambiente. Um VP insere tarefas aptas à execução na frente da sua lista local e as retira da mesma extremidade, caracterizando desta forma uma pilha e priorizando a execução do grafo em profundidade. Entretanto, quando esta pilha local fica vazia, o VP torna-se um ladrão e, então, escolhe outro VP (vítima) para realizar um roubo de trabalho. A tarefa roubada é então retirada do final da pilha local do VP vítima, potencializando assim a exploração do paralelismo nos extremos onde os roubos acontecem.

A Figura 1 são apresentadas as pilhas de execução de dois VPs durante o roubo de trabalho e ilustra o seguinte cenário. `#job003` é armazenado na pilha do VP_1 que acaba de executar o `#job000`. O VP_2 fica sem trabalho e rouba o *thread* mais antigo da pilha do VP_1, e com isso enche sua pilha de execução novamente. O VP_1 tenta realizar uma operação de sincronização somando o resultado dos *threads* `#job003` e `#job004`. Como o *thread* roubado ainda não está terminado, a operação de *join* falha e o VP_1 abre um novo ramo de execução na árvore, desviando com isso seu fluxo de execução para `#job013`. Ao final desta execução, o *join* é tentado novamente. Neste contexto, não é preciso haver sinalização entre os VPs indicando operações de roubo, uma vez

que operações atômicas são utilizadas para controlar o *status* de cada *thread* e os VPs trabalham diretamente com a referência para o *thread* em execução.

3 Avaliando o Desempenho do Ambiente

Para avaliar o desempenho foram construídos algoritmos de testes. O primeiro realiza o cálculo da combinação de 28 elementos arranjados 14 a 14. O segundo utilizado o algoritmo de Smith-Waterman [Smith and Waterman 1981] para realizar o alinhamento de uma sequência genética de 750 caracteres. Em ambos os algoritmos foram introduzidas cargas sintéticas médias e grandes com vistas a avaliar o desempenho dos ambientes com diferentes cargas computacionais. O testes foram executados em uma máquina Intel®Core™i7, 3.4 GHz, 8 GB de RAM, 34 KB em L1, 256 KB em L2 e 8 MB em L3.

Para o algoritmo da combinação, a carga atribuída realiza diversas operações em ponto flutuante, sendo que a carga grande tem custo computacional 3 vezes maior que carga média. Para o algoritmo Smith-Waterman, a carga é atribuída a partir da divisão da matriz de caracteres em blocos de $40x40$ para simular uma carga média e blocos de $20x20$ para simular uma carga grande.

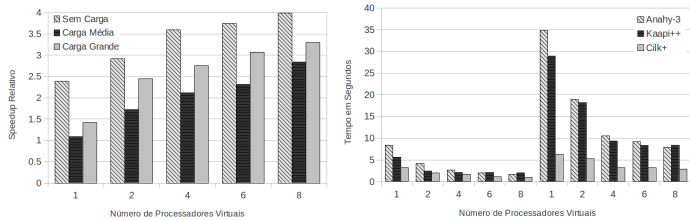

Figura 2. Algoritmo recursivo da combinação de 28, 14 a 14.

A Figura 2 apresenta os resultados de desempenho em relação ao algoritmo da combinação. À esquerda é mostrado o *speedup* relativo entre Anahy e Anahy-3. É notável o grande desempenho de Anahy-3 e relação a Anahy para a execução sem carga sintética. Aplicando a carga média, o *speedup* relativo tem uma degradação. Porém, quando a carga grande aplicada ao algoritmo, o desempenho de Anahy diminui. A média do *speedup* considerando a carga grande é 29.53% maior em relação à carga média e 21.32% menor em relação à execução sem carga sintética. O Gráfico à direita compara os tempos de execução de Anahy-3 aos outros ambientes de execução citados neste trabalho. Na primeira metade do gráfico, quando é aplicada a carga média, o desempenho de Anahy-3 aumenta em proporção ao número de VPs, o que não acontece com KAAPI. Isso reflete o quão bem Anahy-3 consegue explorar a virtualização da arquitetura. Por este motivo o desempenho de KAAPI acaba sendo 8.76% menor que Anahy-3 quando o algoritmo é executado com 8 VPs. O mesmo comportamento se repete na segunda metade do gráfico de tempo, onde a carga grande é aplicada.

A Figura 3 apresenta para gráfico do *speedup* relativo o mesmo comportamento observado para o algoritmo da combinação. O diferencial é que o desempenho de KAAPI diminui já a partir do uso de 6 VPs, chegando a ter um desempenho 14% menor, quando são utilizados 8 VPs, em relação à Anahy-3. Isso acontece tanto para primeira metade do

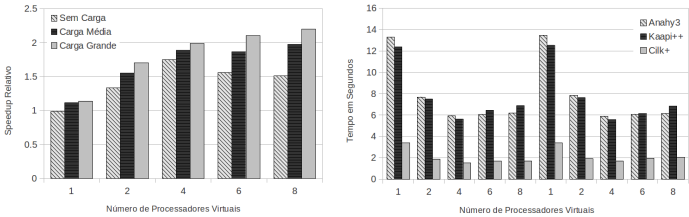

Figura 3. Algoritmo de Smith-Waterman para uma sequência de 750 caracteres.

gráfico quanto para a segunda metade, sendo que a diferença média de desempenho entre estes dois ambientes alcança 17.32%. À esquerda a figura apresenta o *speedup* relativo do Anahy-3 em relação a Anahy. Ambos os ambientes tem uma degradação de desempenho proporcional ao aumento da carga sintética aplicada. Porém, Anahy-3 consegue ter um desempenho melhor. É interessante observar que a diferença de desempenho de Anahy-3 diminui até 11% quando o algoritmo é executado explorando a virtualização da arquitetura e sem a carga sintética aplicada.

4 Conclusão

Em ambos os testes apresentados é notável que o novo ambiente Anahy-3 alcança um maior desempenho em relação à antiga versão de Anahy. Porém, é preciso ressaltar que Anahy foi concebido para explorar o paralelismo em *clusters*, sem se preocupar com otimizações de desempenho internas aos nós. Quando Anahy-3 é comparado a KAAPI, seu desempenho é superior quando é explorada a virtualização da máquina. Como KAAPI não suporta a execução com mais VPs do que o número total de núcleos lógicos existentes, seu desempenho é comprometido. Em ambas as execuções Cilk alcançou, como esperado, índices de desempenho significativos. Entretanto, este desempenho diminui quando o algoritmo não segue o modelo de execução recursivo, o que é o caso do algoritmo Smith-Waterman.

Referências

Cavalheiro, G. G. H., Gaspary, L. P., Cardozo, M. A., and Cordeiro, O. C. (2007). Anahy: A programming environment for cluster computing. In *VII High Performance Computing for Computational Science*, Berlin. Springer-Verlag. (LNCS 4395).

Frigo, M., Leiserson, C. E., and Randall, K. H. (1998). The implementation of the cilk-5 multithreaded language. volume 35, pages 212–223, Montreal, Quebec, Canada. ACM SIGPLAN Notices.

Gautier, T., Besseron, X., and Pigeon, L. (2007). Kaapi: A thread scheduling runtime system for data flow computations on cluster of multiprocessors. pages 15–23, New York, NY, USA. ACM.

Guo, Y. (2010). *A Scalable Locality-Aware Adaptive Work-Stealing Scheduler for Multi-Core Task Parallelism*. PhD thesis, Rice University, Houston, Texas, USA.

Smith, T. and Waterman, M. (1981). Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197.