Estrutura de Dados de Baixa Contenção no Núcleo de Execução de Anahy-3

GUILHERME COUSIN[1]; GERSON GERALDO H. CAVALHEIRO[2]

[1]*Universidade Federal de Pelotas – gpbcousin@inf.ufpel.edu.br*
[2]*Universidade Federal de Pelotas – gerson.cavalheiro@inf.ufpel.edu.br*

1. INTRODUÇÃO

Programas *multithread* se beneficiam do hardware paralelo quando são capazes de explorar os processadores disponíveis em uma arquitetura de forma efetiva durante sua execução. A ociosidade de um processador, durante a execução de um programa, representa perda potencial de desempenho [COUSIN et al. 2013]. Esta ociosidade pode ser reflexo de sincronizações frequentes entre as atividade em execução, resultando em sobrecusto de execução e/ou na criação de dependências desnecessárias entre atividades e eventuais perdas do paralelismo potencial do programa. Contudo, a programação *multithreaded* exige um grande esforço de aprendizagem por parte do programador e o constante aumento do número de cores nestas arquiteturas tornam a migração para o modelo de programação paralela lenta [MA et al. 2011].

Devido a isso, há uma classe de ferramentas especializadas, capazes de fornecer abstrações de programação para desacoplar o hardware da descrição da concorrência de um programa [SKILLICORN; TALIA 1998]. Isso é feito por meio de um mapeamento (escalonamento) das atividades concorrentes do programa sob os recursos de processamento paralelo disponíveis. Este escalonamento visa estabelecer uma ordem de execução para as atividades concorrentes segundo alguma heurística implementada pela ferramenta de execução. Existem inúmeras ferramentas que especializadas , entre elas se encontram OpenMP [CHANDRA 2001], Cilk Plus [ROBISON 2014], TBB [REINDERS 2007] e Anahy [CAVALHEIRO et al. 2006]. Em comum estas ferramentas oferecem recursos de escalonamento para abstrair a exploração do paralelismo. Para que não haja comprometimento do desempenho, estas ferramentas devem ser implementadas de forma eficiente.

Este trabalho visa otimizar e comparar a ferramenta Anahy-3 de forma como em que esta ferramenta tenha um melhor tempo de execução as ferramentas anteriormente citada, para a otimização de Anahy-3 foi utilizado algorítimos não bloqueantes, a seção 2 mostra a forma em que Anahy-3 está organizado e duas opções de estruturas de dados a ser utilizadas.

2. METODOLOGIA

A arquitetura atual de Anahy [CAVALHEIRO et al. 2006], é denominado de Anahy-3 [DE ARAUJO 2013], é composta basicamente por três partes: processadores virtuais (Vps) que compõem o núcleo de execução do ambiente; *Jobs* que são a unidades básicas de escalonamento do ambiente; AnahyVM, responsável gerenciar os componentes do ambiente e fornecer a interface de programação.

As unidades responsáveis por realizar a execução do programa sãao compostas por um conjunto de objetos que definem os VPs do Anahy-3. Estes objetos contêm em sua estrutura uma unidade de execução implementada como um *thread* sistema, formando parte do núcleo de execução da ferramenta. Os VPs

implementam um modo de execução *help first* [GUO et al. 2009], i. e., quando um VP cria um novo thread (Job em Anahy-3), este Job é colocado em uma lista local para que este *thread* possa ser roubado por outro e distribuído entre os VPs da arquitetura. A+ e Skeletons são componentes fornecidos pela AnahyVM e à aplicação *multithread* para que esta possa criar um tipo especial de Job no ambiente.

O escalonamento em arquiteturas multiprocessadas consiste em alocar *n* tarefas sobre *m* unidades de processamento idênticas com o objetivo de reduzir o tempo de execução do programa. Quando o número de tarefas é muito maior que o número de processadores disponíveis, o grande número de alocações destas tarefas não compromete o desempenho de execução. Algoritmos de escalonamento de lista fornecem soluções aproximadas para este problema de decisão, atribuindo prioridade de execução às tarefas. Algoritmos de lista são conhecidos por serem eficientes devido sua simplicidade e eficiência, desde que sejam construídos de forma adequada. Uma das principais limitações do desempenho de um ambiente de execução corresponde á maneira como suas estruturas de dados gerenciam as *threads* do ambiente e a quantidade de tempo gasto nas alocações de novas *threads*. Diante deste problema, fora desenvolvidas duas estruturas de dados para o ambiente, a *SmartHeap* [DE ARAUJO 2013] e uma estrutura de dados sem bloqueio [COUSIN et al. 2014]. Estas estruturas utilizam estratégias de baixa contenção para aumentar a concorrência destas de acesso a lista.

2.1 SMARTHEAP

No início da execução do programa, o ambiente aloca para o *heap* um bloco de memória estático. Consideramos que o tamanho inicial do macro-bloco suporte o armazenamento de 4 *Jobs*. Quando o programa ocupa todos os espaços disponíveis, uma realocação é necessária e, então, o ambiente utiliza uma heurística própria para realizar esta nova alocação. Quando um *Job* ocupa a posição *n* da lista e outro *Job* precisa ser armazenado, a *SmartHeap* faz uma realocação da lista com dobro do tamanho anterior. Esta estratégia é baseada na estratégia descrita por [CORMEN et al. 2001] em *Table Expansion*. Entretanto, a estratégia adotada pela *SmartHeap* utiliza o dobro do tamanho da estrutura para realizar a realocação, isso porque estamos interessados em reduzir o número de futuras alocações na estrutura e não com o tamanho que a estrutura pode alcançar. Isso visa reduzir o número alocações realizadas a cada novo *Job* criado, economizando o tempo de processamento que seria gasto devido as várias chamadas da primitiva de alocação para cada *Job*. Com o uso desta estratégia é visível a reduz o número de realocações sofridas pela estrutura de dados, uma vez que são necessárias apenas 31 realocações para que a estrutura comporte o armazenamento de mais de 1 bilhão de *Jobs*. O limite desta estratégia ocorre quando uma realocação falha, diante disso o gerenciador da lista tenta um realocação com a metade do tamanho pretendido, e isso se repete até que a realocação seja feita.

2.2 ESTRUTURAS DE DADOS SEM BLOQEUIO

O algoritmo proposto como a alternativa a *Smart Heap*, atualmente em uso em Anahy, implementa uma lista duplamente encadeada. A estrutura concebida, possui dois nodos fixos. Entre os nodos fixos, nodos contendo informações relevantes do programa em execução, são inseridos e removidos. Desta forma,

há a possibilidade de inserção e de remoção tanto no início quanto no final da lista sem perda da consistência da estrutura da lista. Esta consistência é mantida com o auxilio de uma *flag* em cada nodo que identifica se o nodo em questão já está sendo utilizado por outros *threads*. A manipulação das *flags* é feita com a primitiva atômica *Compare and Swap*, a qual testa o valor da *flags* e altera seu valor caso a comparação realizada seja verdadeira em uma única instrução.

A inserção é feita do seguinte modo: um *thread* tenta alterar a *flag* do nodo principal da extremidade, caso haja sucesso, o *thread* tenta alterar a *flag* do nodo seguinte. No caso de novo sucesso, o elemento é inserido entre os dois nodos, retornando, então, as *flags* aos seus estados originais. Caso alguma das *flags* já tenha sido alterada por outro *thread*, o algoritmo libera as *flags* alteradas por ela e retorna ao início da inserção, evitando assim *deadlocks* e assumindo que a inserção não foi realizada.

Na remoção, um *thread* tenta alterar a *flag* do nodo principal da extremidade na qual deseja-se fazer a remoção, logo após efetuar a troca deste nodo, o *thread* tenta alterar as *flags* dos outros dois nodos seguintes. Caso haja sucesso, o elemento mais próximo ao nodo principal é removido. Caso alguma das *flags* já tenham sido alterada por outro *thread*, o algoritmo libera as *flags* já alteradas por ele, e volta ao início da remoção assumindo que a remoção não foi realizada e nova tentativa deve ser feita [COUSIN et al. 2014].

3. RESULTADOS E DISCUSSÃO

Foram realizados testes de desempenho para a aplicação do problema de satisfatibilidade booliana 3-Sat, o problema da Satisfatibilidade Booleana (SAT) consiste em verificar se, há uma atribuição de valores verdade para as variáveis de uma fórmula lógica, que tornem esta fórmula verdadeira. Os algoritmos para resolver o problema SAT irão retornar se a fórmula é satisfatível ou não, analisando as cláusulas de cada instância.

Os resultados apresentados foram executados em computador, em uso dedicado, dotado com um processador Intel Core i7-2600 CPU @ 3.40GHz (4 núcleos com SMT ativo), com memória de 8GiB, os resultados estão apresentados em forma de gráfico que mostra a média de 30 execuções, foram feitos testes estatísticos para a validação destes resultados.

Na avaliação de desempenho, foram comparadas Anahy-3 que é a versão utilizando a *SmartHeap*, Anahy-3.1 que utiliza estrutura de dados sem bloqueio, Cilk, OpenMP e TBB.

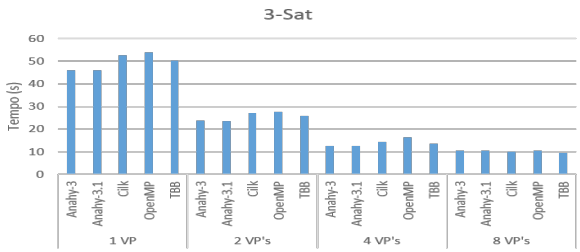

Figura 1: Tempo de execução em segundos do 3-Sat

Considerando a Figura 1, observa-se que a versão de Anahy que utiliza estruturas sem bloqueio, não impactou de forma significativa o desempenho da aplicação. Em relação ao demais ambientes, observa-se que as outras ferramentas introduziram um sobrecusto, mesmo o problema sendo trivialmente

paralelizável, este sobrecusto é gerado pelo mecanismo de escalonamento dos demais ambiente.

4.CONCLUSÕES

A comparação de Anahy aos outros ambientes de execução teve o objetivo de analisar o desempenho do ambiente em relação a outras ferramentas com funcionalidades similares. Em relação ao Anahy, *Cilk* e TBB utilizam os mesmos conceitos de execução e técnicas de escalonamento, mas com diferenças em suas heurísticas. Embora que *Cilk* e TBB tenham suporte do compilador e restrinja muito seu modelo de execução para obter desempenho, Anahy garante seu desempenho devido a simplicidade de sua arquitetura e das estratégias de execução utilizadas pelo ambiente.

5. REFERÊNCIAS BIBLIOGRÁFICAS

MA, K., Li, X.; CHEN, M.; WANG, X. Scalable power control for manycore architectures running multi-threaded applications. **ACM SIGARCH Computer Architecture News**, volume 39, pages 449–460. 2011.

COUSIN, G. P. B.; DE ARAUJO, A. S.; CAVALHEIRO, G. G. H. . Utilização de estruturas de dados não bloqueantes em programação multiprocessada. **WSCAD-WIC**. 2013.

SKILLICORN, D. B.; TALIA, D.. Models and languages for parallel computation. **ACM Computing Surveys,** 30(2):123–169,1998

CHANDRA, R., **Parallel programming in OpenMP**, Morgan Kaufmann. 2001.

REINDERS, J.. **Intel threading building blocks: outfitting C++ for multi-core processor parallelism**. O'Reilly Media, Inc. 2007.

CAVALHEIRO, G. G. H.; GASPARY, L. P.; CARDOZO, M. A.; CORDEIRO, O. C.. Anahy: A programming environment for cluster computing. **VECPAR**. 198–211. 2006.

GUO, Y., BARIK, R., RAMAN, R., AND SARKAR, V. (2009). Work-first and help-first scheduling policies for async-finish task parallelism. **Parallel & Distributed Processing. IEEE International Symposium on**. 1–12. 2009.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., STEIN, C., et al. **Introduction to algorithms**, volume 2. MIT press Cambridge.2001.

COUSIN, G., FAVARETTO, R., ARAUJO, A., and CAVALHEIRO, G. G. H. Programação paralela utilizando de estruturas de dados não bloqueantes. **ERAD/RS.** 2014.

ROBISON, A. D..Composable Parallel Patterns with Intel Cilk Plus.**Computing in Science & Engineering.IEEE Computer Society**.V 15. 66-71. 2013.