Avaliação da Estrutura de Código de Programas *Multithread* em uma Estratégia de Escalonamento Eficiente de *Threads*[*]

Alan S. de Araujo[†], Cícero A. S. Camargo[‡], Matheus G. Nachtigall[§], Rodolfo M. Favaretto, André R. Du Bois, Gerson G. H. Cavalheiro

[1]Universidade Federal de Pelotas – UFPel - Computação – CDTec
Laboratory of Ubiquitous and Parallel Systems – LUPS

{asdaraujo,cadscamargo,mgnachtigall,

rmfavaretto,dubois,gerson}@inf.ufpel.edu.br

1 Introdução

No contexto deste trabalho, a estratégia de escalonamento é incorporada ao ambiente Anahy [Cavalheiro et al. 2007] e tem o objetivo de fornecer subsídios para as tomadas de decisão do núcleo responsável pelo escalonamento, no nível usuário. O ambiente de execução de Anahy oferece uma interface de programação que permite ao programador descrever a concorrência da aplicação sem considerar o *hardware* em que esta será, de fato, executada. O mapeamento dos *threads* descritos pelo programador, sobre as unidades de processamento, é feito em tempo de execução pela estratégia de escalonamento de Anahy. Esse mapeamento, quando feito de maneira eficiente, tem um impacto significativo sobre o consumo de energia. A hipótese é de que este mapeamento possa considerar os custos computacionais das diferentes tarefas geradas pelo programa. Assim, a estratégia tem como intuito, reduzir o consumo de energia causando pouco ou nenhum impacto no desempenho final da aplicação.

Neste artigo, são apresentados estudos sobre a exploração da estrutura de código de programas *multithread* na implementação de uma estratégia de escalonamento que forneça suporte à execução destes programas sobre arquiteturas *multicore*. A estratégia explora eficientemente os recursos de *hardware* disponíveis na arquitetura. Isso permite reduzir o consumo de energia de programas em execução por meio de um melhor uso das *caches* e dos núcleos de processamento. Neste escalonamento são requeridas apenas informações do custo computacional de cada *thread* criado pelo programa. Tais informações, fornecidas pelo programador, podem ser coletadas pelo Anahy, e então estes custos são incorporados aos atributos dos *threads* ou podem ser obtidas por meio de heurísticas de escalonamento de listas [Graham 1976].

2 Escalonamento para Economia de Energia

O núcleo de execução de Anahy é composto por processadores virtuais (PVs), 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

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

dos PVs aos núcleos físicos da arquitetura. O sengundo nível consiste no mapeamento dos *threads* descritos pelo programador sobre os PVs. As dependências de dados entre os *threads* são mantidas pelo ambiente de execução na forma de um Grafo Cíclico Dirigido (DCG) de *threads*. Na alocação dos *threads* sobre PVs 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 PV, prioriza a execução dos *threads* mais recentemente criados sobre o próprio PV. O escalonamento global, por sua vez, é acionado quando algum PV encontra-se sem trabalho. Neste caso, o *thread* criado a mais tempo aguardando para ser executado é lançado sobre este PV. A estratégia ao mesmo tempo em que explora a localidade dos *threads*, realiza uma execução em profundidade no grafo, com a expectativa de privilegiar a execução sobre o caminho crítico. O princípio da proposta de escalonamento é distribuir a carga computacional entre os PVs a partir do custo computacional dos *threads* do programa, e então adequar a frequência de operação dos núcleos de processamento tendo como referência o somatório dos custos associados aos *threads*.

Quando o custo associado a cada fluxo de execução não está presente nos atributos dos *threads*, explora-se a estrutura do DCG, e então aplica-se heurísticas baseadas em escalonamento de lista. Nesta heurística assume-se que um conjunto de *threads* do DCG contém a maior sequência de instruções a serem computadas, esses *threads* por sua vez delimitam o desempenho do programa [Graham 1976]. Com base nisso associamos os *threads* deste caminho à um processador, ou a um conjunto específico de processadores. É possível afimar que, com um número suficientemente grande de *cores*, o custo associado a qualquer processador não irá exceder o custo atribuído ao conjunto de processadores responsáveis pela execução do caminho crítico. Assim frequências menores podem ser atribuídas aos *cores* que não executam *threads* no caminho crítico, de forma a economizar energia sem perda de desempenho.

3 Estudo de Caso

Para avaliar a estratégia de escalonamento proposta foram desenvolvidos dois conjuntos de testes e executados em uma máquina com processador Intel®Core™2 Quad, com frequências de 1.33GHz a 2.66GHz, primeiro nível de *cache* com 32KB para instruções e 32KB de dados, sendo que cada *cache* L2 é compartilhado entre 2 *cores* com tamanho igual à 3MB. O primeiro realiza o cálculo da sequência de Fibonacci e o segundo calcula o número de combinações de M, N a N. Em ambos os algoritmos foi introduzida uma carga computacional extra, com cálculos em ponto flutuante, para que o *overhead* das operações de escalonamento de múltiplos PVs não excedesse o custo computacional dos *threads* do programa. Vale ressaltar que a escolha dos algoritmos dá-se com o intuito de explorarmos o alto grau de paralelismo existentes em suas estruturas, conforme os pseudocódigos apresentados na Figura 1. Nestes algoritmos, há um desbalanceamento

```
fibo(n)
    if n >= 2 return 1
    else return fibo(n-1) + fibo(n-2)
```

```
comb(m,n) {
 if n == 1 return m
 if n == m return 1
 if m > n  return comb(m-1,n-1) + comb(m-1,n)
```

Figura 1. Pseudocódigo do Fibonacci e da Combinação recursivos

causado pela diferença na expansão dos nodos do grafo durante a execução. Conhecendo

a forma como o escalonamento é realizado é possível organizar o programa, ou seja as criações dos *threads*, de maneira que *threads* com maior custo computacional e potencial de geração de paralelismo, sejam criados primeiro na lista de execução. A expectativa desta estratégia é de que *threads* gerados a partir dos *threads* executando nos processadores mais rápidos possam ser migrados para outros PVs. Como estes *threads* possuem uma carga de trabalho menor, dada a natureza recursiva dos algoritmos selecionados, poderão ser finalizados em tempo hábil, em relação ao caminho crítico, mesmo quando executados sobre processadores com frequências reduzidas. As Figuras 2 e 3 descrevem os resultados em termos de eficiência e apresentam a média obtida de 20 execuções para cada aplicação. Para uma melhor visualização do consumo em relação ao desempenho, os gráficos são apresentados em termos de eficiência de desempenho (*speedup / número de processadores*) e de consumo (*1-(consumo observado / consumo máximo)*).

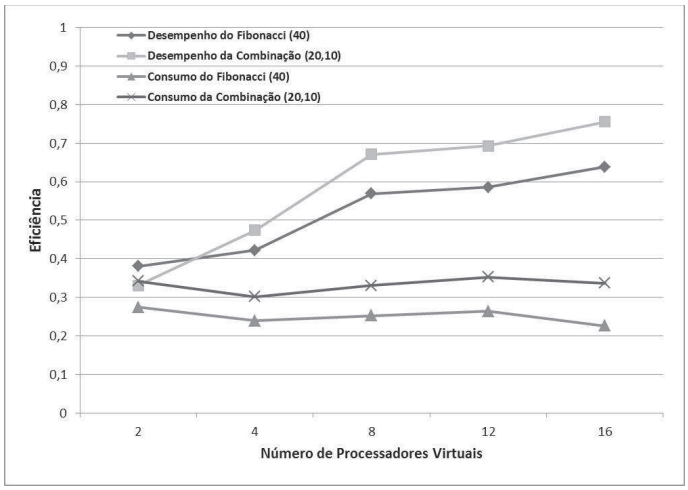

Figura 2. Fibonacci e Combinação com desbalanceamento à direita

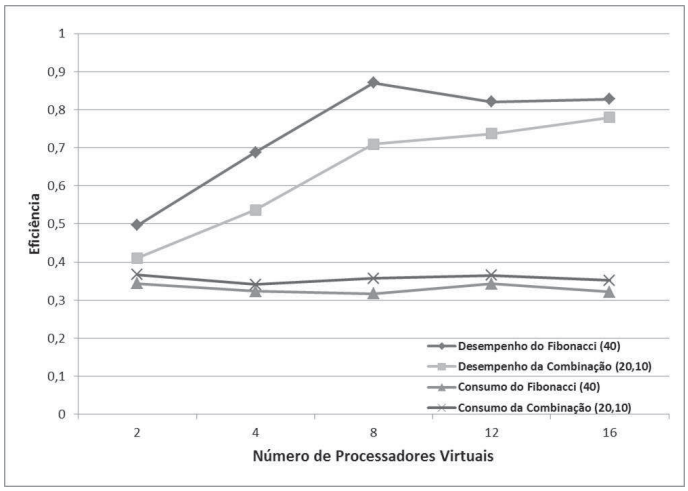

Figura 3. Fibonacci e Combinação com desbalanceamento à esquerda

Os gráficos mostram o impacto que a organização do código tem sobre a execução

do programa, tanto no desempenho quanto no consumo de energia. O *speedup* e o consumo de energia da Figura 2 apresentam a influência deste conceito tem sobre a organização do código. Estes algoritmos foram construídos com o desbalanceamento estrutural à direita do grafo, não explorando desta forma a prioridade de execução em profundidade do escalonador dos *threads*. O contrário disto pode ser visto na Figura 3 com o consumo mais estável em relação ao desempenho. Nesta Figura o desbalanceamento dá-se à esquerda do grafo, priorizando muito bem a execução dos *threads* mais antigos, ou seja, pertencentes ao caminho crítico do grafo.

A Tabela 1 apresenta os dados brutos coletados das duas execuções. Esses dados estão na unidade de Watts e foram medidos utilizando-se dois multímetros, um para medir a tensão e outro para coletar os dados de corrente, na entrada de energia do computador. Para o Fibonacci a economia média de energia vista durante a execução é de 10,72%, o mesmo acontece para a Combinação, com uma redução de 3,50% no consumo de energia.

Tabela 1. Tabela com consumo em Watts do Fibo(40) e da Comb(20,10)

Num. PVs	Fibonacci			Combinação		
	Esq.	Dir.	Economia	Esq.	Dir	Economia
1	61.586	65.631	06.56%	59.729	64.688	08.30%
2	62.274	68.938	10.70%	60.061	62.465	04.00%
4	64.292	72.216	12.32%	62.558	64.570	03.21%
8	64.838	71.015	09.52%	61.037	62.528	02.44%
12	62.838	69.873	11.19%	60.273	60.476	00.33%
16	64.433	73.509	14.08%	61.256	62.956	02.77%

4 Conclusão

Os gráficos mostram a economia de energia dos algoritmos durante a execução. Essa economia aumenta significativamente quando é considerado o tempo total de execução das aplicações, tendo com referência o total consumido pelo programa, podendo alcançar a partir dos resultados apresentados neste trabalho, índices superiores a 36% de economia de energia. Os *speedups* mostraram, para as duas aplicações, a diferença de desempenho quando a organização da estrutura do código é ou não considerada. O que ocorre, conforme verificado no núcleo de escalonamento, é uma maior quantidade de migrações de tarefas entre os PVs devido à tranferência dos *threads* com cargas menores. Por outro lado, as curvas de consumo explicitam um outro problema relacionado a má organização do código, nos experimentos da Figura 2 o consumo aumenta. Isto pode estar relacionado ao maior número de trocas de contexto nos *caches* dos *threads* com maior carga computacional.

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).

Graham, R. L. (1976). *Bounds on the Performance of Scheduling Algorithms*. John Wiley & Sons.