Introdução ao Estudo do Consumo de Energia de Programas Paralelos

Luís Gabriel Lima

Centro de Informática
Universidade Federal de Pernambuco
lgnfl@cin.ufpe.br

Resumo Com a ascensão da computação móvel a proliferação de dispositivos alimentados por bateria a utilização inteligente da energia desses dispositivos é de suma importância. Muitos estudos têm se focado nesse problema, porém a maioria deles envolve as camadas mais baixas da computação como projeto de hardware e sistemas operacionais. Este trabalho tem como objetivo fazer uma uma apresentação dos principais tópicos relacionados ao consumo de energia de programas paralelos em níveis mais próximos da aplicação.

1 Introdução

As arquiteturas multi-core são uma realidade. A maioria dos processadores produzidos industrialmente contém pelo menos 2 núcleos. Devido à esse direcionamento da indústria de hardware, se faz necessário que os programadores explorem o paralelismo durante o desenvolvimento das aplicações para que seja possível tirar proveito desse tipo de arquitetura. Entretanto, escrever programas que envolvam concorrência e paralelismo não é uma tarefa fácil. Um dos motivos para tal é a característica não-determinística desse tipo de aplicação, que leva o programador a se preocupar com problemas como condição de corrida, *deadlocks*, falha de atomicidade, entre outros.

Para se adaptar aos desafios introduzidos por essa nova plataforma de hardware, a comunidade de software tem sido bastante ativa no estudo e desenvolvimento de novas técnicas para tratar uma grande variedade de propriedades de programas concorrentes e paralelos como corretude, desempenho e programabilidade. [8] Porém, um outro grande desafio que até agora tem recebido pouca atenção é o consumo de energia. Melhorar o desempenho de um sistema utilizando-se *threads* e construções concorrentes é bastante palpável hoje, porém, se essa melhora de desempenho acontece atrelada a uma degradação considerável no consumo de energia, para muitos casos esse *trade-off* não é aceitável.

O problema de reduzir o consumo de energia, tradicionalmente, tem sido abordado no nível operacional (por exemplo, desabilitando completamente o sistema ou parte dele) ou no nível de design de hardware (por exemplo, utilizando componentes especializados de baixo consumo de energia). No contexto de sistemas de software, grande parte das pesquisas têm se focado em otimizações de

baixo nível (no nível de código de máquina) [9]. Nos níveis mais altos, especialmente para o nível da aplicação, esse problema ainda foi pouco explorado.

Este texto tem como objetivo apresentar alguns trabalhos relacionados à consumo de energia que estão sendo ativamente pesquisados. O foco aqui é dado principalmente aos níveis mais altos das camadas da computação visando o desenvolvedor de aplicações. Para fazer essa apresentação os trabalhos foram divididos em três categorias: técnicas que focam na análise teórica de algoritmos paralelos (Seção 2), técnicas que utilizam alteração dinâmica de voltagem (Seção 3) e trabalhos que investigam o consumo de energia de construções utilizadas em programação paralela (Seção 4). Por fim, na Seção 5 a conclusão.

2 Análise de algoritmos paralelos

Uma das linhas de estudo em consumo de energia de programas paralelos se baseia em uma análise teórica de alto nível dos algoritmos paralelos com objetivo de determinar estaticamente como minimizar o consumo de energia de uma aplicação. A ideia por trás desse tipo de abordagem é tentar representar de maneira formal o consumo de energia de um algoritmo, através, por exemplo, de uma fórmula matemática e, a partir dessa formalização, tentar chegar a conclusões como: qual o ajuste ótimo dos parâmetros que pode minimizar o consumo de energia, qual o comportamento do consumo de energia desse algoritmo de acordo com a variação do tamanho da entrada, etc.

Dentro desse tipo de estudo, podemos destacar os trabalhos desenvolvidos por Korthikanti e Agha em [3], [5], [4], [6] e [7]. Eles criaram um *framework* de análise algoritmos que a partir de uma modelagem do hardware (quantidade de núcleos do processador, hierarquia de memória, custo de comunicação, etc) é possível derivar da especificação do algoritmo uma fórmula de complexidade de consumo de energia, como um Big-O, sendo possível responder perguntas como: qual o número ótimo de núcleos que devem ser utilizados para obter desempenho máximo dado uma restrição de consumo de energia?

Outro trabalho que deve ser citado é o de Rauber e Rünger [12]. Eles fizeram uma modelagem do consumo de energia de programas que utilizam o modelo de programação *parallel tasks* [11]. Este é um modelo de programação em dois níveis que distingue entre um nível superior *coarse-grained* com tarefas paralelas que cooperam entre si e um nível mais baixo *fine-grained* que captura o paralelismo dentro dessas tarefas. Os autores fazem uma modelagem analítica do consumo de energia desse modelo de programação e mostram que o *overhead* do consumo de energia é menor se tarefas independentes forem executadas concorrentemente por um conjunto disjunto de processadores se comparado à execução consecutiva das mesmas.

Para ficar mais claro como esse tipo de análise é feito, veremos a seguir um exemplo extraído de [3] que analisa o consumo de energia de um algoritmo de soma paralelo.

2.1 Exemplo de análise: Somando em paralelo

Algoritmo Considere um algoritmo paralelo simples que adiciona N números utilizando M núcleos. Inicialmente todos os N números são igualmente distribuídos entre os M núcleos. No primeiro passo do algoritmo, metade dos núcleos enviam a soma que eles computaram para outra metade de forma que nenhum núcleo recebe a soma de mais de um núcleo. Os núcleos que recebem as somas adicionam o número ao seu somatório local que foi computado. O mesmo passo é executado recursivamente até que sobre apenas um núcleo. Ao final da computação um núcleo terá armazenado o somatório de todos os N números.

Metodologia Queremos analisar esse algoritmo de forma que seja possível descobrir qual o número ótimo de núcleos que minimiza o consumo de energia em função do tamanho da entrada. Utilizaremos a mesma análise que os autores utilizam em [3], que se trata da análise da escalabilidade de energia considerando a preservação de desempenho (*iso-performance*). Em [6] e [7] são apresentadas outras métricas de escalabilidade de energia além da preservação de desempenho. Para realizar tal análise, os autores assumem as seguintes simplificações arquiteturais:

1. Todos os núcleos operam em uma mesma frequência e a frequência dos núcleos podem ser variadas.
2. O tempo de computação dos núcleos podem ser dimensionados através da mudança da frequência dos núcleos.
3. O tempo de comunicação entre os núcleos é constante.
4. O tempo de acesso a memória é constante (não existe uma hierarquia de memória).
5. Cada núcleo tem sua própria memória e os núcleos sincronizam através de comunicação entre os núcleos.

Sob essas condições, o tempo de execução T em um dado núcleo é proporcional à quantidade de ciclos μ executados neste núcleo. Sendo X a frequência que o núcleo está operando, então:

$$T = (\textit{quantidade de ciclos}) \times \frac{1}{X} \tag{1}$$

Sabendo que um crescimento linear na voltagem leva à um crescimento linear da frequência do núcleo e que um crescimento linear na voltagem também leva à um crescimento não-linear (tipicamente cúbico) na potência, a energia consumida por um núcleo E pode ser modelada da seguinte forma:

$$E = E_c \times T \times X^3 \tag{2}$$

onde E_c é uma constante do hardware.

Dados essas informações, os autores propõem que sejam seguidos os seguintes passos para analisar um algoritmo paralelo:

1. Achar o caminho crítico do algoritmo paralelo. O caminho crítico é o caminho mais longo do grafo de dependência de tarefas do algoritmo.
2. Particionar o caminho crítico em passos de computação e comunicação.
3. Dimensionar os passos de computação do caminho crítico para que o desempenho da paralelização seja igual ao requisito de desempenho estabelecido. Ou seja, calcular a nova frequência na qual todos os M núcleos alocados para aplicação devem executar. Isso é feito alterando o tempo de computação do caminho crítico (Passo 2) para diferença do desempenho requerido e o tempo de comunicação do caminho crítico.
4. Avaliar a número total de mensagens processadas. Dependendo do algoritmo a quantidade de mensagens pode depender apenas da quantidade de núcleos como também tanto da quantidade de núcleos quanto do tamanho da entrada N.
5. Avaliar o tempo total em que os núcleos estarão ociosos.
6. Obter a expressão de energia na seguinte forma: $E = E_{comp} + E_{comm} + E_{idle}$, onde E_{comp} é a energia consumida para realizar computação, E_{comm} a energia consumida para realizar comunicação e E_{idle} a energia consumida em modo ocioso.
7. Analisar a equação para obter a quantidade de núcleos requeridos para um consumo de energia mínimo em função do tamanho da entrada

Análise Assumindo, sem perda de generalidade, que a quantidade de núcleos disponível é alguma potência de dois. O algoritmo executa em $\log(M)$ passos. Nesse algoritmo é fácil de identificar o caminho crítico: é o caminho de execução do núcleo que ao final têm a soma de todos os número (Passo 1). É possível perceber que existem $\log(M)$ passos de comunicação e $(\frac{N}{M} - 1 + \log(M))$ passos de computação (Passo 2). Agora, podemos obter a frequência reduzida em que cada um dos M núcleos deve executar para completar a tarefa em um tempo T (Passo 3):

$$X' = F \cdot \frac{(\frac{N}{M} - 1 + \log(M)) \cdot \beta}{T \cdot F - \log(M) \cdot K_c} \tag{3}$$

onde F é a frequência máxima de um núcleo, β representa a quantidade de ciclos necessária por adição e K_c a quantidade de ciclos executados em frequência máxima para comunicação de uma mensagem.

A quantidade de troca de mensagens total necessário pelo algoritmo é $(M-1)$ para Mnúcleos (Passo 4). Dessa forma, o tempo ocioso total de todos os núcleos executando com a nova frequência X' (Passo 5) é:

$$T_{iddle} = \frac{B}{X'} \cdot (M(log(M) - 1) + 1) + \frac{1}{F} \cdot K_c \cdot (M(log(M) - 2) + 2) \tag{4}$$

onde o primeiro termo representa o tempo ocioso total gasto pelos núcleos ociosos enquanto outros núcleos estão ocupados computando e o segundo termo

representa o tempo ocioso total basto pelos núcleos ociosos enquanto os outros núcleos estão se comunicando via mensagem.

Dado que $(N-1)$ é a quantidade total de passos de comunicação, as equações de energia podem ser definidas da seguinte forma (Passo 6):

$$E_{comp} = E_c \cdot (N-1) \cdot \beta * X'^2 \tag{5}$$

$$E_{comm} = E_m \cdot (M-1) \tag{6}$$

$$E_{idle} = P_s \cdot T_{idle} \tag{7}$$

onde E_m é a energia consumida para uma mensagem de comunicação entre núcleos e P_s é a potência estática consumida por um núcleo ocioso.

O último passo (Passo 7) é analisar a expressão de energia obtida. Note que a expressão de energia é dependente de muitas variáveis como N, M, B, K_c, E_m e P_s. É possível simplificar alguns desses parâmetros sem perda de generalidade. Por exemplo, na maioria das arquiteturas a quantidade de ciclos necessário para realizar uma adição é um, então os autores assumem $\beta = 1$. Também é possível assumir a energia ociosa consumida por ciclo como $(\frac{P_s}{F}) = 1$.

Outras simplificações que comprometem a generalidade da análise são necessárias para que seja possível criar um gráfico da equação. O autor assume que a razão de energia consumida por ciclo do núcleo operando na frequência máxima F e do núcleo ocioso é 10, ou seja, $E_c \cdot F^2 = 10(\frac{P_s}{F})$ e que $E_m = k \cdot E_c \cdot F^2$.

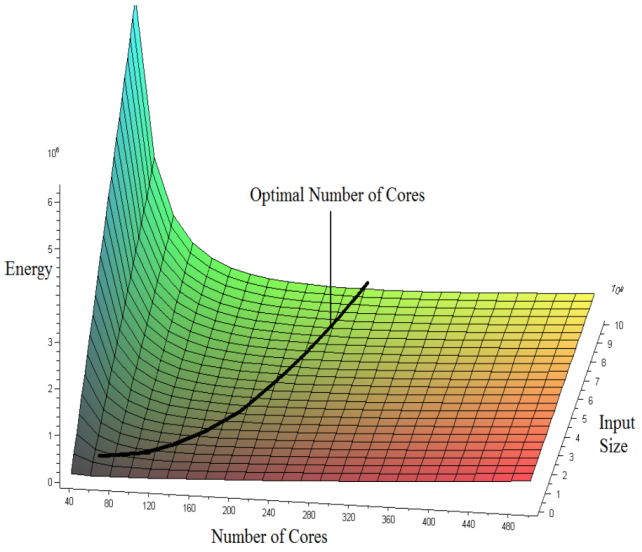

Figura 1. Superfície de energia onde $k = 500$, $\beta = 1$ e $k_c = 5$. A curva preta no plano XY representa a quantidade ótima de núcleos para um consumo mínimo de energia de acordo com a variação do tamanho da entrada.

A Figura 1 consiste no gráfico representa a energia E como uma função do tamanho da entrada e da quantidade de núcleos, considerando as simplificações descritas anteriormente. É possível perceber que para qualquer tamanho de entrada N, inicialmente a energia decresce de acordo com o crescimento de M e em seguida cresce com o crescimento de M. Esse comportamento pode ser explicado pelo fato da energia gasta para computação decresce com o crescimento da quantidade de núcleos executando a uma frequência reduzida, e a energia gasta para comunicação cresce de acordo com o crescimento de M. Dessa forma, é possível perceber que aumentando o tamanho da entrada leva ao crescimento da quantidade ótima de núcleos necessário para um consumo mínimo de energia.

2.2 Discussão

Inicialmente é importante salientar que o exemplo apresentado anteriormente é muito simples e a metodologia utilizada também é bastante limitada se compararmos com o mundo real onde vários outros fatores influenciam no consumo de energia de uma aplicação. O principal objetivo de utilizá-lo neste trabalho foi dar uma ideia de como esse tipo de análise pode ser feito. Em trabalhos posteriores como [6] e [7] os autores consideram fatores mais realistas nas análises como um modelo de memória com hierarquia e um modelo de comunicação mais próximo do real.

Embora o exemplo tenha sido simples, é fácil perceber que esse tipo de análise pode ser muito útil para os desenvolvedores visto que, tendo uma metodologia que aproxime-se bem da realidade, é possível pensar e analisar o consumo de energia de um algoritmo ainda na fase de projeto e concepção de uma solução.

3 Alteração dinâmica de voltagem

Existe uma outra linha de trabalhos que foca na utilização de uma tecnologia chamada DVFS (*Dynamic Voltage and Frequency Scaling*) para construir programas paralelos que gastem menos energia. Essa tecnologia está presente na maioria dos processadores multi-core atuais e consiste em dinamicamente alterar a frequência de *clock* e a voltagem dos núcleos de um processador. Essa técnica vem sendo utilizada porque na maioria das CPUs atuais a potência dissipada (a taxa de consumo de energia) pode ser representada de maneira simplificada por $P = C \cdot V^2 \cdot F$, onde V é a voltagem, F a frequência de *clock* e C a capacitância. Dessa forma, alterar a voltagem e a frequência do processador tem influência direta no consumo de energia. Uma observação importante é que devido a forma com que os circuitos dos processadores são projetados a frequência e a voltagem são frequentemente alteradas conjuntamente.

Um dos trabalhos que pode-se destacar nessa área é o de Liu [8] onde ele observa como seria possível economizar energia através do conhecimento das principais estruturas utilizadas para construir programas paralelos. Como mecanismos de sincronização são peças fundamentais para o desenvolvimento desse tipo de software, o autor ataca esse problema e analisa como é possível diminuir

o consumo de energia de algoritmos e padrões que utilizam sincronização. Ele parte da premissa que o tempo gasto durante um bloqueio (no caso de *mutex*) ou uma espera ocupada (no caso de *spinlocks*) consome energia sem contribuir para o progresso do programa. Isso se dá porque espera ocupada consome energia sem estar executando nenhuma instrução efetiva do programa e bloqueio aumenta a quantidade de trocas de contexto que leva à um aumento considerável de falhas de *cache*, que é bastante custoso em termos de consumo de energia.

Uma das ideias que o autor prega nesse trabalho é que o tempo de bloqueio ou espera ocupada pode ser reduzido se as *threads* participantes executarem "cooperativamente"em velocidade. Por exemplo, admita que as *threads* T_1 e T_2 estão executando em direção a um ponto de sincronização e elas irão demorar 2 e 10 segundos, respectivamente. Nesse cenário T_1 terá que esperar 8 segundos. A ideia é que T_1 diminua sua velocidade para atingir o ponto de sincronização ao mesmo tempo que T_2. Para que isso seja possível, é necessário diminuir a frequência de *clock* do núcleo que está executando T_2. Dessa forma, com a diminuição da velocidade, o consumo de energia também será menor nesse núcleo, contribuindo para o objetivo principal de reduzir o consumo de energia do programa.

Pattern	Figure	Strategy	Example Scenarios
(A) Dependent Join		downscale "parent" thread	futures (MultiLisp, Java, C++0x, C#, *etc*) promise pipelining (Argus, E) fork-join async-finish (X10)
(B) Counted Sync		upscale late comers based on counter	Clock (X10) CountDownLatch (Java 1.5) N Semaphores
(C) Declarative Sync		upscale late comers based on program structure	Chord (C#) Exchange (Java 1.5) synchronized SEND/RECV (MPI)
(D) Critical Path		upscale the critical execution; downscale late comers	monitors synchronized blocks (Java, C#) CTRITICAL directive (OpenMP)
(E) Symmetric Join		scheduling dependent	MapReduce loop parallelization (OpenMP, X10, Fortress, *etc*)

Figura 2. Padrões de sincronização e estratégias utilizando DVFS

A partir dessa ideia de diminuir ou aumentar a velocidade de algumas *threads* para reduzir o consumo de energia do programa, o autor identifica cinco padrões que são recorrentes no desenvolvimento de aplicações paralelas que podem se beneficiar desse tipo de abordagem. Esses padrões bem como as estratégias propostas estão ilustrados na Figura 2. É importante salientar que embora intuitivamente faça bastante sentido as ideias apresentadas em [8], o autor não chegou a realizar nesse trabalho implementações e experimentos que verificassem na prática os resultados esperados. Fatores como a forma com que as *threads* são

escalonadas no sistema operacional ou na máquina virtual podem impactar diretamente nesses resultados.

Outro trabalho que se baseia em DVFS que pode ser citado é o de Bartenstein e Liu [1]. Neste trabalho os autores propõem uma solução para o modelo de programação baseado em *streams* utilizando DVFS para minimizar o gasto de energia. Esse modelo de programação é bem específico e está relacionado à aplicações que processam grande quantidade de dados. Nesse cenário o consumo de energia está fortemente relacionado à quantidade de dados e como esses dados são processados. Um programa nesse modelo é representado por um grafo de processamento onde os nós são unidades de processamento (chamadas de filtros) e as arestas representam o fluxo do dos dados no grafo. A proposta, chamada de *Green Streams*, usando o grafo que representa o programa, identifica alguns padrões semelhantes aos que foram apresentados na Figura 2 e aplica soluções baseadas DVFS para "balancear" a execução de forma que quando há uma dependência no grafo as *threads* de processamento se ajustem para chegar ao mesmo tempo nos pontos de sincronização. Neste trabalho os autores implementaram a solução e os experimentos realizados mostraram uma economia de energia de 28% e um ganho de performance de 7% em relação a uma solução que não utiliza DVFS.

Embora as soluções baseadas em DVFS se mostrem bastante promissoras no que tange a economia de energia em processadores *multi-core*, esse tipo de abordagem é um pouco distante do desenvolvimento de aplicações. De fato, alterar a frequência de operação de alguns núcleos do processador não deve ser preocupação de uma aplicação, e sim do sistema operacional, de um sistema de tempo de execução ou até de uma biblioteca especifica. Porém, é possível que sejam criados abstrações em formas de bibliotecas ou modelos de programação utilizando o conceito de DVFS para prover aos desenvolvedores de aplicação ferramentas voltadas para diminuição do consumo de energia de programas paralelos.

4 Análise experimental de construções concorrentes

Os desenvolvedores de software têm uma gama de opções para modelar concorrência em seus programas, desde primitivas de sincronização como *mutex*, *spinlocks* e blocos sincronizados até construtores de criação e gerenciamento de *threads* como *thread pools*. Porém, não é claro como as escolhas de como organizar e representar concorrência nos programas impacta no consumo de energia. Com o intuito de entender na prática esse impacto, alguns trabalhos têm focado em analisar experimentalmente o comportamento de programas reais para encontrar essa resposta.

Um exemplo é o de Gautham *et al.* [12] que compara o consumo de energia de duas técnicas de sincronização de dados: travas e memória transacional em software (STM). Para realização dos experimentos os autores utilizaram a suíte de benchmark STAMP que consiste uma série de algoritmos de diversos domínios com grande carga de sincronização que utiliza STM. Como o STAMP é um benchmark escencialmente de STM, os autores tiveram que alterar as

implementações para incluir também sincronização com travas para realizar a comparação. Os resultados dos experimentos mostraram que STM pode ser mais eficiente em termos de desempenho e consumo de energia para cenários onde se gasta muito tempo dentro de regiões críticas.

Outro trabalho importante nesse contexto é o de Pinto *et al.* [10] onde o foco é entender o comportamento com relação ao consumo de energia de alguns construtores de gerenciamento de *threads* disponíveis no ambiente da linguagem Java. Nesse trabalho são comparados três construtores: o gerenciamento explícito de *threads*, *thread pools* e *work-stealing*. O primeiro engloba a criação e gerenciamento manual das *threads* onde o desenvolvedor é responsável por lidar manualmente com a divisão do trabalho e ciclo de vida das *threads* e o segundo utiliza *thread pools* de tamanho fixo que automaticamente gerencia as threads e o desenvolvedor é responsável por submeter tarefas que representam unidades lógicas independentes de trabalho para executado pelo *thread pool* quando conveniente. Já o último tem funcionamento similar a um *thread pool* do ponto de vista do desenvolvedor, porém as *threads* são gerenciadas de outra forma, ao invés de manter um *buffer* global de tarefas como é o caso do *thread pool*, o *framework* de *work-stealing* mantém um *buffer* para cada *thread* e permite que as *threads* "roubem" tarefas de *threads* vizinhas quando seu *buffer* estiver vazio.

O objetivo desse trabalho é entender se e como a escolha de construtores, número de *threads* e estratégias de divisão de trabalho impacta no consumo de energia dos programas. Para isso os autores realizaram experimentos que exploraram um amplo espectro de configurações. Foram utilizados nove programas distintos, alguns escritos especificamente para o trabalho outros já conhecidos de algumas suites de *benchmark*. Esses programas foram escolhido de forma que explorassem diferentes características de problema: uso intensivo de CPU, uso intensivo de memória, uso intensivo de entrada e saída, embaraçosamente paralelo, etc. Um resultado interessante desses experimentos é que nenhum dos construtores utilizados se sobressaiu em relação aos outros de uma forma geral, cada um se destacou em situações distintas.

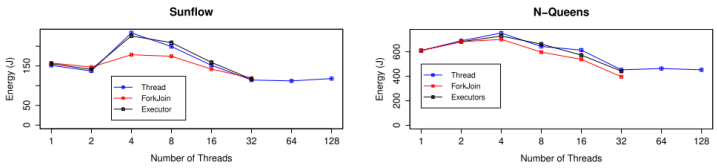

Figura 3. Exemplos da curva Λ

Outro resultado interessante desse trabalho é o comportamento do consumo de energia de acordo com o variação da quantidade de *threads*. Os autores identificaram que em praticamente todos programas testados, o crescimento do número de *threads* acontece em conjunto com um crescimento no consumo de energia

até um determinado número de *threads* onde o consumo de energia começa a decrescer. Os autores denominaram esse comportamento de curva Λ, devido ao formato da curva que representa esse comportamento no gráfico, como podemos ver na Figura 3. Os autores sugerem que esse comportamento aconteça devido às características dos processadores multi-core atuais. A parte crescente da curva pode ser explicada porque os núcleos quando estão inutilizados, executam em uma frequência menor, consumindo menos energia; dessa forma o programa que roda com 8 *threads* consome mais energia do que o programa que roda com 4. Porém, quando passa de um certo número de *threads*, o ganho em desempenho é tão considerável (o programa termina mais rápido) que faz com que o consumo de energia passe a decrescer.

Os experimentos realizados por esse trabalho é bastante extenso, e foge ao escopo deste texto abordá-los como um todo. Porém, o principal resultado apresentado é que os três construtores de gerenciamento de *threads* utilizados impactam o consumo de energia de formas diferente, e o consumo de energia está intimamente ligado com várias escolhas como: que construtor utilizar, a quantidade de *threads*, a granularidade das tarefas e o tamanho da entrada.

5 Conclusão

Como podemos ver, o estudo do consumo de energia de programas paralelos em níveis mais altos de abstração é uma área incipiente. Boa parte dos trabalhos relacionados são recentes e de caráter exploratório. Apesar disso, é fácil perceber que encontrar o equilíbrio entre consumo de energia e desempenho no contexto de programação paralela não é uma tarefa simples. Analisar o consumo de energia de uma perspectiva teórica leva à equações matemáticas complexas e também necessitam de simplificações que podem comprometer a generalidade da análise de várias formas. Trabalhar com alteração dinâmica de voltagem se mostra bastante desafiador pois além de ser uma operação de baixo nível que de certa forma quebra o princípio da abstração, pode interferir no funcionamento de camadas inferiores como o sistema operacional. Por fim, os experimentos comentados na Seção 4 mostram que a relação entre consumo de energia e desempenho levando-se em conta as várias abstrações para programação paralela como mecanismos de sincronização e gerenciamento de *threads* é extremamente sensível à diversas características do problema em si, tornando difícil generalizar conceitos ou técnicas.

É importante salientar que este texto pode (e será) melhorado. Em futuras revisões é possível melhorar desde a organização do texto em si quanto a didática para expor o problema de uma forma mais clara. Seria interessante também adicionar uma análise crítica para alguns dos trabalhos citados deixando mais evidente pontos que podem ser explorados em futuras pesquisas.

Referências

1. Thomas W Bartenstein and Yu David Liu. Green streams for data-intensive software. In *Proceedings of the 2013 International Conference on Software Engineering*,

pages 532–541. IEEE Press, 2013.

2. Ashok Gautham, Kunal Korgaonkar, Patanjali Slpsk, Shankar Balachandran, and Kamakoti Veezhinathan. The implications of shared data synchronization techniques on multi-core energy efficiency. In *Presented as part of the 2012 Workshop on Power-Aware Computing and Systems*. USENIX, 2012.
3. Vijay Anand Korthikanti and Gul Agha. Analysis of parallel algorithms for energy conservation in scalable multicore architectures. In *Parallel Processing, 2009. ICPP'09. International Conference on*, pages 212–219. IEEE, 2009.
4. Vijay Anand Korthikanti and Gul Agha. Avoiding energy wastage in parallel applications. In *Green Computing Conference, 2010 International*, pages 149–163. IEEE, 2010.
5. Vijay Anand Korthikanti and Gul Agha. Towards optimizing energy costs of algorithms for shared memory architectures. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pages 157–165. ACM, 2010.
6. Vijay Anand Korthikanti, Gul Agha, and Mark Greenstreet. On the energy complexity of parallel algorithms. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 562–570. IEEE, 2011.
7. Vijay Anand R Korthikanti. *Towards energy-performance trade-off analysis of parallel applications*. PhD thesis, University of Illinois at Urbana-Champaign, 2012.
8. Yu David Liu. Energy-efficient synchronization through program patterns. In *Green and Sustainable Software (GREENS), 2012 First International Workshop on*, pages 35–40. IEEE, 2012.
9. Jean P Mermet. *Low power design in deep submicron electronics*, volume 337. Springer, 1997.
10. Gustavo Pinto, Fernando Castor, and Yu David Liu. Understanding energy behaviors of thread management constructs. In *Proceedings of the 28th ACM Conference on Object-Oriented programming systems, languages, and applications*, page to appear. ACM, 2014.
11. Thomas Rauber and Gudula Rünger. Tlib—a library to support programming with hierarchical multi-processor tasks. *Journal of Parallel and Distributed Computing*, 65(3):347–360, 2005.
12. Thomas Rauber and Gudula Rünger. Modeling the energy consumption for concurrent executions of parallel tasks. In *Proceedings of the 14th Communications and Networking Symposium*, pages 11–18. Society for Computer Simulation International, 2011.