

# Computação Verde na Camada de Aplicação

MONOGRAFIA APRESENTADA  
AO  
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA  
DA  
UNIVERSIDADE DE SÃO PAULO  
PARA  
OBTENÇÃO DO TÍTULO  
DE  
BACHAREL EM CIÊNCIA DA COMPUTAÇÃO

Thales Areco Bandiera Paiva

Orientador: Prof. Dr. Alfredo Goldman vel Lejbman

São Paulo, dezembro de 2014



# Agradecimentos

Ao Prof. Dr. Alfredo Goldman pela orientação e paciência.

Ao Prof. Dr. Hermes Senger por gentilmente emprestar o sistema de testes.

Ao seu aluno Denis Oliveira por ter resolvido minhas dúvidas sobre o sistema.



# Sumário

<b>Lista de Figuras</b>	<b>vii</b>
<b>Lista de Tabelas</b>	<b>ix</b>
<b>1 Introdução</b>	<b>1</b>
<b>2 Computação Verde</b>	<b>3</b>
2.1 Regulamentações . . . . .	3
2.2 Métricas e <i>benchmarks</i> . . . . .	4
2.3 Uso eficiente de recursos computacionais . . . . .	4
<b>3 Eficiência Energética</b>	<b>5</b>
3.1 Hardware e Firmware . . . . .	5
3.1.1 Desativação Dinâmica de Componentes . . . . .	5
3.1.2 <i>Advanced Power Management</i> . . . . .	5
3.1.3 Regulagem Dinâmica de Frequência e Voltagem . . . . .	6
3.2 Sistema Operacional . . . . .	7
3.2.1 <i>Advanced Configuration and Power Interface</i> . . . . .	7
3.2.2 Gerenciadores para uso do processador . . . . .	8
3.3 Aplicação . . . . .	8
3.3.1 Recomendações de alto nível aos usuários . . . . .	8
3.3.2 Recomendações às gerências de TI . . . . .	8
3.3.3 Guias de desenvolvimento . . . . .	9
3.3.4 Código dependente de contexto . . . . .	9
<b>4 Metodologia de medição</b>	<b>11</b>
4.1 Sobre o sistema utilizado . . . . .	11
4.2 Gravando o consumo de potência . . . . .	12
4.3 Medindo o consumo de aplicações . . . . .	13
<b>5 Experimentos na camada de aplicação</b>	<b>15</b>
5.1 Perfis de Consumo de Potência . . . . .	15
5.1.1 Especificação do experimento . . . . .	16
5.1.2 Resultados obtidos . . . . .	16
5.1.3 Análise dos resultados . . . . .	18
5.2 Escolha de Algoritmos . . . . .	18

5.2.1	Especificação do experimento . . . . .	19
5.2.2	Resultados obtidos . . . . .	19
5.2.3	Análise dos resultados . . . . .	21
5.3	Eliminação de <i>Code Smells</i> . . . . .	22
5.3.1	Especificação do experimento . . . . .	22
5.3.2	Resultados obtidos . . . . .	23
5.3.3	Análise dos resultados . . . . .	24
5.4	Agendamento de Processos . . . . .	25
5.4.1	Especificação do experimento . . . . .	25
5.4.2	Resultados obtidos . . . . .	25
5.4.3	Análise dos resultados . . . . .	28
<b>6</b>	<b>Conclusões</b>	<b>29</b>
	<b>Referências Bibliográficas</b>	<b>31</b>

# Lista de Figuras

4.1	Acesso por Telnet . . . . .	11
4.2	Consumo de potência obtido pela biblioteca para execuções paralelas de uma aplicação	13
4.3	Exemplo de execução do <i>energyanalyser</i> . . . . .	14
5.1	Perfil do consumo de potência para uma tarefa <i>CPU bound</i> . . . . .	16
5.2	Perfil do consumo de potência para uma tarefa <i>Memory bound</i> . . . . .	17
5.3	Perfil do consumo de potência para uma tarefa <i>IO bound</i> . . . . .	17
5.4	Potência consumida pelos algoritmos para diferentes tamanhos de entrada . . . . .	20
5.5	Tempo de execução dos algoritmos para diferentes tamanhos de entrada . . . . .	20
5.6	Energia consumida pelos algoritmos para diferentes tamanhos de entrada . . . . .	21
5.7	Impacto médio da retirada de cada <i>code smell</i> nas variáveis de interesse . . . . .	24
5.8	Potência média consumida pelo sistema para cada $n$ . . . . .	26
5.9	Tempo médio de execução da tarefa para cada $n$ . . . . .	26
5.10	Energia média consumida pela tarefa para cada $n$ . . . . .	27
5.11	Energia média consumida pelo sistema para cada $n$ . . . . .	27



# Lista de Tabelas

3.1	Estados dos dispositivos sob a norma APM v1.2 . . . . .	6
3.2	Especificação dos estados do sistema pela APM v1.2 . . . . .	6
3.3	Gerenciadores de uso do processador no Linux . . . . .	8
5.1	Linha de comando para cada tipo de tarefa . . . . .	16
5.2	<i>Code smells</i> considerados . . . . .	22
5.3	Médias amostrais das variáveis estudadas . . . . .	24
5.4	Tabela de impacto da eliminação de cada <i>code smell</i> . . . . .	25
5.5	Tabela de melhora pela utilização de $n = 7$ para cada $n$ . . . . .	28



# Capítulo 1

## Introdução

O impacto negativo de sistemas computacionais no meio ambiente é evidente. Como Murugesan nota em [Mur08], cada fase do ciclo de vida de um computador implica em diferentes problemas ambientais. A Computação Verde combina política, ciências, e engenharia para tentar resolver esses problemas.

Um dos ramos da computação verde é o uso eficiente de recursos computacionais. Para estudar este ramo, é interessante dividi-lo nas camadas *hardware*, sistema operacional, aplicação, e redes como Ardito faz em sua tese de doutoramento [Ard14]. As camadas de *hardware* e sistema operacional foram as mais estudadas por serem de mais baixo nível e, portanto, mais diretamente relacionadas com o consumo energético.

O foco deste trabalho é na camada de aplicação, que só recentemente passou a ser explorada. Fizemos 4 experimentos sobre esta camada com 2 objetivos principais:

1. Expandir e validar resultados de outros pesquisadores para o nosso sistema.
2. Encontrar oportunidades de aumentar a eficiência energética.

Tanto os capítulos quanto suas subdivisões seguem uma sequência lógica. Assim, para melhor compreensão do trabalho, sugerimos a leitura na ordem apresentada. Porém, como os capítulos de revisão são superficiais, o leitor familiarizado com os conceitos poderá focar diretamente no experimentos sem maiores problemas. A organização dos capítulos é a seguinte:

1. Pequena revisão sobre computação verde e eficiência energética nos Capítulos 2 e 3.
2. Apresentação de nosso sistema de medição no Capítulo 4.
3. Especificação dos experimentos e análises dos seus resultados no Capítulo 5.
4. Recapitulação e conclusão no capítulo 6.



## Capítulo 2

# Computação Verde

Sistemas computacionais impactam no meio ambiente desde sua fabricação até muito depois de seu descarte. A produção de componentes eletrônicos utiliza recursos naturais escassos. O uso de um sistema computacional consome energia e gera gases do efeito estufa. O descarte, como Sthiannopkao nota em [SW13], na maior parte do mundo é feito indevidamente, acumulando lixo e substâncias tóxicas.

Ao estudo e desenvolvimento de técnicas para minimizar o impacto ambiental de sistemas computacionais, chamamos Computação Verde. É interdisciplinar pois une esforços políticos, acadêmicos e industriais. Os principais focos da pesquisa nessa área são:

- reciclagem e descarte de hardware;
- projeto e fabricação de componentes;
- regulamentações e métricas;
- uso eficiente de recursos computacionais.

Como as duas primeiras são exclusivas da engenharia, não trataremos delas neste texto. Nas seções seguintes, mostraremos as principais regulamentações, critérios considerados para as métricas, e guias para o uso eficiente de recursos.

### 2.1 Regulamentações

Regulamentações associadas à Computação Verde são um meio político de impor restrições sobre o impacto ambiental do uso e descarte de sistemas computacionais. Tem grande potencial, pois faz com que a competitividade de uma empresa dependa do investimento desta em computação verde.

Na data de escrita deste trabalho, os Estados Unidos e a União Europeia têm as regulamentações mais fortes. Como Harmon e Auseklis notam em [HA09], as normas estadunidenses focam em eficiência energética, enquanto as europeias, na produção e no descarte de componentes. São elas:

- *Waste Electrical and Electronic Equipment Directive* (WEEEED)  
Lixo eletrônico é qualquer equipamento eletrônico cuja vida útil chegou ao fim [PRZ+02]. Antes diretiva e lei na União Europeia a partir de 2003, a WEEEED impõe a responsabilidade do lixo eletrônico sobre os produtores dos componentes.
- *Restriction of Hazardous Substances* (RoHS)  
Junto à diretiva anterior, passou a ser lei na União Europeia a partir de 2003. Como o nome diz, restringe o uso de algumas substâncias nocivas ao meio ambiente na produção de componentes.

- *Electronic Product Environmental Assessment Tool* (EPEAT)  
Avalia, através de uma série de critérios, computadores de mesa, *notebooks* e monitores. Oferece uma ferramenta para que potenciais compradores possam comparar os equipamentos. Desde 2007, agências federais norte-americanas são obrigadas a comprar equipamentos que seguem esta norma.
- *Energy Star 4.0*  
Revisão mais recente do padrão *Energy Star* que regulamenta o desempenho energético de computadores de mesa, *notebooks* e monitores nos vários estados de operação do *hardware*.

## 2.2 Métricas e *benchmarks*

Métricas são cruciais em computação verde por dois motivos principais. Primeiro, porque fornecem uma ferramenta de avaliação e comparação de sistemas. Segundo, porque todo desenvolvimento ou aprimoramento de técnicas verdes sempre visa melhorar o desempenho sob alguma métrica. Wang e Khan [WK13] apresentam uma revisão do trabalho em métricas para computação verde e consideram os seguintes critérios:

- Emissão de gases do efeito estufa
- Umidade
- Temperatura
- Potência e energia
- Combinações dos critérios acima

Em geral, para comparar os desempenhos de sistemas diferentes sob uma determinada métrica, precisamos especificar uma certa operação ou carga de trabalho. A essa operação e carga de trabalho, chamamos *benchmark*. Alguns dos *benchmarks* mais comumente usados em computação verde são:

- SPECpower
- JouleSort
- Linpack

## 2.3 Uso eficiente de recursos computacionais

Historicamente, o desenvolvimento da computação foi focado em performance. Desde a teoria de complexidade de algoritmos até a construção de parques de servidores dedicados à demanda sempre crescente de usuários.

Com os computadores crescendo em tamanho e em sendo cada vez mais utilizados, os gastos com a energia consumida por sistemas computacionais pode fazer com que a sua operação economicamente inviável para uma organização. Isto coloca uma forte barreira no aumento da performance e, por isso, começou a se estudar técnicas para utilizar estes sistemas consumindo menos energia.

Nosso trabalho se enquadra nessa categoria pois queremos aumentar a eficiência energética de um sistema através de software. Por isso, julgamos importante que o leitor conheça as técnicas em eficiência energética, dedicando o próximo capítulo inteiramente a esta revisão técnica.

## Capítulo 3

# Eficiência Energética

Neste capítulo, apresentamos uma revisão das principais iniciativas para economizar energia nas camadas de *hardware*, sistema operacional e aplicação. Após a leitura deste material introdutório, esperamos que o leitor possa entender melhor nosso trabalho e o que ele representa na pesquisa em eficiência energética.

### 3.1 Hardware e Firmware

Os dispositivos eletrônicos são os responsáveis diretos pela dissipação de energia. Assim, os primeiros estudos em eficiência energética foram direcionados a eles. Este capítulo mapeia as principais inovações técnicas que permitem que os dispositivos usem energia mais eficientemente. Naturalmente, todas elas precisam de apoio do BIOS.

Como estamos sob o ponto de vista da ciência da computação, não tratamos da evolução da implementação física específica dos componentes, apenas mostramos as interfaces, especificações ou padrões utilizados pelos fabricantes de *hardware*.

Seguiremos a taxonomia proposta por Beloglazov et al. em [BBL<sup>+</sup>11]. Para uma revisão dos algoritmos associados a cada uma das inovações que descreveremos, o leitor pode consultar o trabalho de Albers [Alb10].

#### 3.1.1 Desativação Dinâmica de Componentes

Como o próprio nome diz, a ideia para economizar energia é desligar componentes de hardware que não estão sendo utilizados. No estudo de eficiência energética, este é o modelo mais simples imaginável: um sistema com dois estados.

A solução do problema da desativação dinâmica seria trivial se não houvesse restrições ao modelo: deligue se estiver inativo, ligue se for requisitado. Porém, devemos considerar que:

- Há um custo razoável de energia associado à alimentação e estabilização de potência, ao fazer a transição do estado desligado para o ligado.
- Não se sabe, a priori, qual será a demanda do sistema no futuro

Assim, apesar de ser uma ideia extremamente simples, resolver este problema pode ser bem complicado. Para isso, utilizam-se técnicas preditivas com algoritmos *online* e teoria de processos estocásticos.

#### 3.1.2 *Advanced Power Management*

Apesar de os algoritmos para resolver o problema da desativação dinâmica chegarem a uma boa competitividade<sup>1</sup>, a eficiência energética é limitada pela granularidade dos estados. Como há

---

<sup>1</sup>Como em teoria de algoritmos *online*.

apenas dois estados, o intervalo de tempo necessário para que seja tomada a decisão de desativar o sistema pode ser muito grande.

A solução trivial para este problema é aumentar o número de estados. E esta é justamente a proposta da norma *Advanced Power Management* (APM) elaborada pelas empresas Intel e Microsoft em 1992 e atualizada em 1996. A APM especifica estados para dispositivos e estados para o sistema.

Segundo a versão 1.2 da norma APM [IC96], os estados especificados para dispositivos são mostrados na Tabela 3.1. Os estados do sistema são mostrados na Tabela 3.2.

Estados	Características
<i>Device On</i>	Dispositivo ligado e totalmente funcional.
<i>Device Power Managed</i>	Dispositivo ligado mas com funcionalidade reduzida.
<i>Device Low Power</i>	Dispositivo inoperante e em baixo consumo, Pronto para ser ativado rapidamente..
<i>Device Off</i>	Dispositivo totalmente desligado.

**Tabela 3.1:** Estados dos dispositivos sob a norma APM v1.2

Estados	Características
<i>Full On</i>	Sistema totalmente operante e todos os componentes ligados.
<i>APM Enabled</i>	Sistema operando com a APM gerenciando o uso dos dispositivos quando necessário.
<i>APM Standby</i>	Sistema pode estar inoperante, com os dispositivos operando em estados de baixo consumo energético. Volta rapidamente ao estado acima por requisição do usuário.
<i>APM Suspend</i>	Sistema inoperante, com o <i>clock</i> da CPU parado. Maioria dos dispositivos desligados porém com o estado anterior gravado.
<i>Off</i>	Sistema totalmente inoperante, todos os dispositivos desligados.

**Tabela 3.2:** Especificação dos estados do sistema pela APM v1.2

Os algoritmos utilizados são muito parecidos com os usados para desativação dinâmica de componentes e, muitas vezes, generalizações triviais.

### 3.1.3 Regulagem Dinâmica de Frequência e Voltagem

Na seção anterior, vimos que os dispositivos em geral podem ter estados de menor consumo energético e com algoritmos inteligentes para a transição entre estados pode-se economizar energia.

Agora, trataremos especificamente do processador, que possui estados de menor consumo energético mas ainda operantes. À possibilidade de configurar a frequência e voltagem de operação do processador, chamamos Regulagem Dinâmica de Frequência e Voltagem<sup>2</sup>. Usaremos a abreviação, do inglês *Dynamic Voltage and Frequency Scaling*, DVFS, para manter a compatibilidade entre as referências.

Quando se faz DVFS, há dois objetivos fundamentais: otimização para performance e otimização para consumo energético. E em geral, o objetivo do usuário é dado por uma combinação linear entre os dois.

O problema de decidir qual o par de frequência e voltagem ótimo não é simples. Alguns dos problemas que uma solução deve tratar são, como Beloglazov et. al referenciam em [BBL<sup>+</sup>11]:

<sup>2</sup>Também pode-se encontrar variações como Escalonamento Dinâmico de Frequência e Tensão.

- Nem sempre a performance é melhor para frequências mais altas (ex: *Memory Bound* e *IO Bound*<sup>3</sup>).
- A arquitetura dos processadores é muito complexa, tornando difícil modelar a frequência ótima em função do objetivo do usuário.
- A frequência do processador impacta no escalonamento de tarefas. Assim, uma redução em sua frequência de operação pode trazer consequências indesejáveis.

## 3.2 Sistema Operacional

Na seção passada revisamos soluções implementadas no BIOS e apoiadas pelo *hardware* para economizar energia. Apesar de aumentarem muito a eficiência energética de um sistema, o fato de as soluções serem implementadas em *firmware* implica em vários problemas. Dentre eles, apontamos alguns principais<sup>4</sup> abaixo.

- O *firmware* possui poucas informações sobre como o usuário está utilizando o sistema. Técnicas que levam em conta os perfis do usuário podem ter desempenho melhor.
- Cada fabricante define a própria implementação dos gerenciadores.
- A complexidade dos algoritmos e técnicas para eficiência energética é limitada pelo BIOS.
- Os esforços gastos no desenvolvimento de técnicas para evitar desperdício de energia são multiplicados pelos desenvolvedores de BIOS.
- Correções de *bugs* nos gerenciadores podem ser muito caras, pois necessitam uma atualização do BIOS.

Esta coleção de problemas sugere que o controle dos dispositivos deve ser passado para uma camada acima. É isso o que propõe a norma ACPI, como veremos na próxima seção.

### 3.2.1 *Advanced Configuration and Power Interface*

Para resolver os problemas apresentados na introdução, surge em 1996 a especificação *Advanced Configuration and Power Interface* (ACPI) [HP04]. Este padrão substitui o APM colocando o gerenciamento e monitoramento de energia sob o controle do sistema operacional.

Como a APM, a ACPI também especifica uma série de estados, tanto para os dispositivos, quanto para o sistema, chamados de globais. Porém, o processador recebe tratamento especial, com estados específicos para ele. Além disso, define estados de performance, que indicam dispositivos operantes mas em velocidades ou frequências reduzidas.

Não colocaremos as descrições mais detalhadas dos estados como fizemos para a APM pois ocuparia muito espaço e o resto do texto não depende das especificações da ACPI, mas sim da ideia geral desta norma.

Para conhecer a ACPI mais profundamente, o leitor pode consultar diretamente a especificação em [HP04]. Para ver como a ACPI é implementada num sistema UNIX, no caso o FreeBSD, pode-se consultar o artigo de Watanabe em [Wat02].

---

<sup>3</sup>Estes termos não possuem tradução fácil e têm significados suficientemente precisos em inglês. Por isso, não foram traduzidos.

<sup>4</sup>Extraídos do website da norma ACPI [Int11], que veremos adiante.

### 3.2.2 Gerenciadores para uso do processador

Na seção de *hardware*, vimos que através da técnica *DVFS*, pode-se configurar o processador para trabalhar em diferentes frequências. Agora, pela implantação da *ACPI*, o sistema operacional tem a capacidade de fazer *DVFS* observando a demanda do usuário. Para isso, utiliza os chamados gerenciadores ou governadores de frequência.

Como exemplo, a Tabela 3.3 mostra os gerenciadores do processador no Linux. As descrições foram extraídas da documentação do *kernel* do Linux, por Brodowski em [Bro13].

Modo do Gerenciador	Características
<i>Performance</i>	Força o processador a usar a frequência de <i>clock</i> mais rápida disponível. É estática, independente da demanda. Assim, é o que gasta mais energia.
<i>Powersave</i>	Ao contrário da primeira, esta força o processador a trabalhar na frequência mais baixa disponível. Também é estática e é a que menos gasta energia.
<i>Ondemand</i>	Gerencia dinamicamente a frequência do processador de acordo com a demanda. Quando a carga é alta, usa a maior frequência disponível, e quando aquela é baixa, usa a menor. A performance pode ser afetada se as trocas de frequências do processador forem muito constantes
<i>Userspace</i>	Permite programas rodando com permissões do sistema configurar as frequências.
<i>Conservative</i>	Como o <i>ondemand</i> , mas variando entre frequências menos extremas.

Tabela 3.3: Gerenciadores de uso do processador no Linux

## 3.3 Aplicação

Conforme as técnicas nas camadas de *hardware* e sistema operacional foram avançando, o consumo implicado pelas aplicações começa a ficar mais aparente. Assim, o estudo de eficiência energética na camada de aplicação é o mais recente. Apresentamos algumas destas técnicas nas próximas seções.

### 3.3.1 Recomendações de alto nível aos usuários

Estas são recomendações gerais que podem ser feitas a qualquer usuário. Em geral, são encontradas em artigos sobre computação verde que buscam atingir leitores sem conhecimento técnico avançado. Como exemplo, citaremos algumas recomendações por Murugesan em [Mur08]:

- usar os gerenciadores de consumo energético;
- desligar o sistema quando não estiver sendo utilizado;
- usar protetores de tela.

Estas recomendações, se seguidas pelos funcionários de uma organização, podem economizar quantidade significativa de energia.

### 3.3.2 Recomendações às gerências de TI

Como na seção anterior, são recomendações de alto nível. Porém, agora são direcionadas aos gerentes e administradores de setores de tecnologia de informação, portanto, com maior conheci-

mento técnico. Tanto no já citado trabalho de Mugesan [Mur08] quanto no de Harmon [HA09], encontramos guias desse tipo. Algumas delas são:

- melhorar a infraestrutura dos *data centers*;
- implantar gestão de carga térmica;
- utilizar virtualização nos servidores;
- utilizar serviços de computação em nuvem.

### 3.3.3 Guias de desenvolvimento

Na data de escrita deste trabalho, a grande maioria dos trabalhos sobre eficiência energética na camada de aplicação se enquadra nesta categoria. Pela forte correlação entre energia consumida e tempo de execução, há uma série de recomendações para eficiência energética baseadas em técnicas de otimização para performance.

Como exemplo, citaremos algumas recomendações para o desenvolvedores de *software* encontradas em trabalhos publicados pela Intel [Ams10] e pela Universidade de Amsterdã [SA11] :

- fazer *lazy loading* de módulos e bibliotecas;
- usar algoritmos com menor complexidade;
- balancear a carga entre processos;
- evitar o uso de *byte-code*;
- reduzir a redundância de dados;
- usar as otimizações do compilador;
- usar bibliotecas de alto desempenho;
- diminuir a complexidade computacional;
- usar as caches eficientemente;
- aplicações adaptáveis.

A grande maioria destas recomendações são encontradas na literatura de otimização para desempenho. Porém, muitas destas recomendações não têm validação empírica para eficiência energética. Ainda, a forma como plataformas diferentes consomem energia é muito variável, como Bunse et al. mostram em [BHMR09]. Assim, cada uma dessas recomendações precisa ser validada para diferentes plataformas.

### 3.3.4 Código dependente de contexto

Em alguns casos, normalmente quando o software é de aplicação específica, a equipe de desenvolvimento sabe antecipadamente em que ambiente um aplicativo será executado. Quando isso acontece, pode-se otimizar o sistema desenvolvido para aquele ambiente.

O uso de informações como memória RAM disponível, tamanho das caches, *clock* da CPU, tamanho da *pipeline* e quaisquer outras relativas às capacidades e velocidades dos dispositivos para desenvolver software mais eficiente é chamado desenvolvimento dependente de contexto.

Um exemplo claro de como pode-se conseguir desempenhos muito melhores através da contextualização é no desenvolvimento para dispositivos móveis. Bunse et al. observa em [BHMR09] o InsertionSort, ordenando vetores de tamanhos entre 0 e 1000, sendo o algoritmo com melhor

desempenho energético, apesar de não ser o com melhor desempenho temporal, em dispositivos móveis.

Um conceito relacionado ao de código dependente de contexto é o de aplicações adaptáveis. Aplicações são ditas adaptáveis quando mudam seu comportamento dinamicamente de acordo com os requisitos do usuário ou propriedades do sistema. Para o sucesso de uma aplicação adaptável, os desenvolvedores devem implementar diferentes comportamentos para os diferentes contextos possíveis. Aplicações adaptáveis são sugeridas por Bunse et al. [BHRM09] e Ardito em [Ard13] para aumentar a eficiência energética de software.

## Capítulo 4

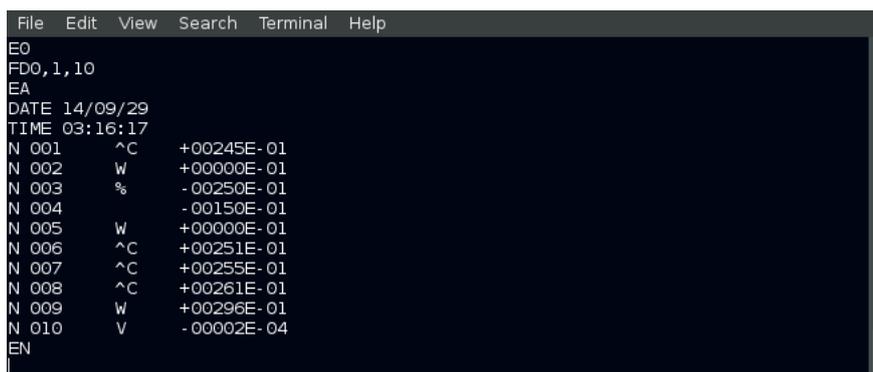
# Metodologia de medição

Para desenvolver o estudo empírico deste trabalho, precisamos de meios para medir a energia consumida por aplicações. Neste pequeno capítulo, primeiro apresentamos a máquina com o sistema que permite medir seu consumo energético, e depois explicamos como tratamos estes dados para obter o consumo de aplicações.

### 4.1 Sobre o sistema utilizado

O sistema utilizado foi gentilmente emprestado pelo Professor Hermes Senger, da Universidade Federal de São Carlos, e seu grupo. Tal sistema é composto por uma máquina, à que nos foi concedido o acesso por SSH, cuja fonte de alimentação é ligada a um sensor de potência que envia os valores para o servidor de dados MW100 Yokogawa. A taxa máxima de amostragem que conseguimos é de 2 valores a cada segundo.

Terminais sob a rede da UFSCar podem requerer dados do MW100 por duas interfaces: Web e Telnet. O MW100 também oferece, ao administrador, a possibilidade de gravar os valores durante certo intervalo. Porém, como nosso acesso é restrito, escrevemos uma simples biblioteca estática para encapsular as requisições Telnet e permitir que gravássemos os dados para análise futura.



```
File Edit View Search Terminal Help
EO
FDO,1,10
EA
DATE 14/09/29
TIME 03:16:17
N 001 ^C +00245E-01
N 002 W +00000E-01
N 003 % -00250E-01
N 004 -00150E-01
N 005 W +00000E-01
N 006 ^C +00251E-01
N 007 ^C +00255E-01
N 008 ^C +00261E-01
N 009 W +00296E-01
N 010 V -00002E-04
EN
```

Figura 4.1: Acesso por Telnet

A máquina de testes possui as seguintes configurações:

- RAM: 4GB
- CPU: Intel Core i5-2400 @ 3.1Ghz
- SO: Ubuntu (kernel 3.2.0) 64 bits

## 4.2 Gravando o consumo de potência

Como foi dito, não temos o direito de requerer gravações ao sistema MW100. Assim, escrevemos a biblioteca estática<sup>1</sup> `mw100_recorder` que oferece ao usuário comum a possibilidade de utilizar a própria máquina para fazer requisições de última medição e gravar os valores lidos.

A interface que a biblioteca oferece é a seguinte:

```
1 void mw100_recorder_init(char mw100ip4[], int mw100port,
2                          double time_interval, int channel);
3 void mw100_recorder_start(int verbose);
4 void mw100_recorder_stop(int verbose);
5 void mw100_get_recorded_info(mw100_record_info *p_user_info);
```

O funcionamento da biblioteca é extremamente simples como pode ser visto pela descrição, em alto nível, de cada função:

- `void mw100_recorder_init()`:  
Inicializa o gravador passando como parâmetros o IP do servidor MW100, a porta sob a qual roda o Telnet, o intervalo aproximado de tempo entre requisições, e o canal do MW100 a que está conectado o sensor de potência.
- `void mw100_recorder_start()`:  
Começa uma gravação e, dependendo do parâmetro passado, imprime os valores na saída padrão. A gravação é feita através de um *thread* criado especificamente para fazer requisições e gravar os valores lidos.
- `void mw100_recorder_stop()`:  
Finaliza uma gravação, terminando o *thread* que havia sido criado.
- `void mw100_get_recorded_info()`:  
Sobrescreve os valores do elemento passado por referência com os da medição que acabou de ser feita. Na estrutura de dados específica para conter estas informações, temos os dados de potência média, energia total consumida e tempo total de medição.

Para testar a biblioteca, desenvolvemos uma ferramenta chamada `powerdump` que, passados um intervalo de tempo em segundos e um arquivo, usa a biblioteca para obter os valores de potência e escrevê-los no arquivo passado.

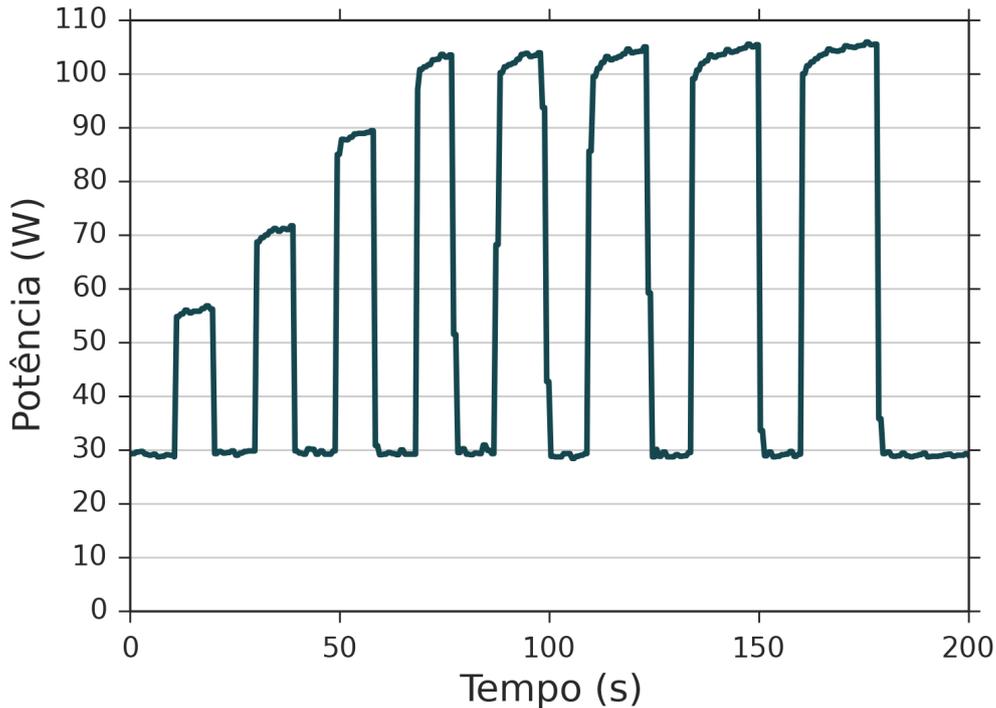
Utilizando esta ferramenta, determinamos os picos de consumo da máquina, executamos o seguinte:

- Para  $n = 1, 2, \dots, 8$ , faça:
  - espere 10 segundos
  - rode, paralelamente,  $n$  processos que calculam as 30.000 casas decimais de  $\pi$

Os valores obtidos nesta gravação podem ser vistos na Figura 4.2.

---

<sup>1</sup>Note que esta biblioteca está muito longe de ser o foco do trabalho e foi escrita apenas por problemas de acesso. Assim, este capítulo não traz nada parecido com uma documentação, apenas simples descrições para que o leitor possa entender melhor nosso sistema de medição.



**Figura 4.2:** Consumo de potência obtido pela biblioteca para execuções paralelas de uma aplicação

E como a CPU da máquina possui 4 núcleos, estes resultados não surpreendem.

### 4.3 Medindo o consumo de aplicações

Usando a biblioteca descrita anteriormente, criamos uma simples ferramenta chamada **energyanalyser**. Esta ferramenta roda várias vezes uma aplicação com seus argumentos e calcula seu consumo médio de potência, energia, dentre outras informações de consumo.

Para calcular o consumo de uma aplicação, a ferramenta começa a execução com uma medição do consumo total, sem a execução do comando, por um intervalo longo para a CPU (em torno de 10 segundos). Tendo calculado este consumo como base, ao final este consumo é desconsiderado da medição.

Escrita em C, roda em linha de comando e recebe os argumentos:

- Formato de saída, que pode ser **h**, indicando que deve ser legível para uma pessoa, ou **d**, indicando que deve permitir fácil extração de informações para um programa analisar (ex. gerador de gráficos).
- Número de vezes que o comando passado pelo usuário será rodado para ter seu consumo energético médio calculado.
- Comando sobre o qual o usuário pretende fazer as medições.

Um exemplo de sua execução é dado abaixo:

```
File Edit View Search Terminal Help
[thales@ubuntu:~]$ energy_analyser h 10 pidigits 30000
[*] Command: pidigits 30000 > /dev/null
[*] Baseline Power Consumption: 30.7690
[*] Iteration   Time (s)   Av. Power (W)   Total Energy (J)   User Energy (J)
[.] 0          8.8029    53.2802         469.0206           198.1645
[.] 1          8.8038    52.1860         459.4364           188.5516
[.] 2          8.7932    53.0875         466.8114           196.2527
[.] 3          8.8105    54.3431         478.7912           207.7007
[.] 4          8.8016    53.4883         470.7803           199.9656
[.] 5          8.8098    52.9990         466.9108           195.8426
[.] 6          8.8166    52.3797         461.8100           190.5328
[.] 7          8.8033    53.0138         466.6953           195.8274
[.] 8          8.8033    55.7379         490.6783           219.8093
[.] 9          8.8236    53.6816         473.6653           202.1719
[*] Statistics
[.] Mean       : 199.4819
[.] Min       : 188.5516
[.] Max       : 219.8093
[.] Var       : 72.6545
[thales@ubuntu:~]$
```

Figura 4.3: Exemplo de execução do energyanalyser

## Capítulo 5

# Experimentos na camada de aplicação

Apresentaremos os experimentos que fizemos para encontrar oportunidades de economia de energia na camada de aplicação. Após sua conclusão, o leitor verá que:

- Aplicações de diferentes naturezas de utilização de recursos possuem diferentes perfis de consumo de potência.
- Nem sempre o algoritmo mais rápido é o que consome menos energia.
- Pequenas diferenças na implementação de um algoritmo podem implicar em diferenças na potência e energia consumidas.
- Fixados os executáveis, há um número ótimo de execuções paralelas destes programas para minimizar o consumo energético.

### 5.1 Perfis de Consumo de Potência

Queremos explorar o consumo de potência implicado pela execução de uma aplicação. Para isso, analisaremos o perfil de representantes de cada tipo de processo abaixo:

- *CPU bound*
- *Memory bound*
- *IO bound*

Nossos três objetivos são:

- Testar a granularidade de nosso sistema de medição.  
Pode ser que nosso sistema, por sua baixa taxa de amostragem, seja muito grosseiro para detectar diferenças nos consumos de aplicações diferentes. O caso tratado por este experimento é extremo, onde as aplicações tem diferentes naturezas e, se mesmo nesse caso não conseguirmos detectar diferenças, todo o estudo ficará comprometido.
- Entender melhor como uma aplicação consome potência.  
Não encontramos nenhum material que comparasse o consumo de potência de aplicações. O mais próximo disso que encontramos foi a tese de doutoramento de Ardito [Ard14], em que o autor comparou o perfil de consumo de potência de diferentes servidores.
- Encontrar oportunidades de melhorar a eficiência energética.  
Após a análise, devemos ter resultados sugerindo possibilidades de técnicas para aumentar a eficiência energética de aplicações.

Assim sendo, apesar de ser um experimento muito simples, os resultados serão utilizados nas interpretações dos próximos experimentos e no desenvolvimento de um algoritmo que troca performance por eficiência energética no segundo experimento.

### 5.1.1 Especificação do experimento

Para simular tarefas de diferentes naturezas, utilizamos a ferramenta de código aberto *SysBench* [Kop04]. Pode-se ver na Tabela 5.1 quais as linhas de comando utilizadas para os testes.

<i>CPU bound</i>	<code>sysbench -test=cpu -cpu-max-prime=20000 run</code>
<i>Memory bound</i>	<code>sysbench -test=memory -memory-block-size=2G run</code>
<i>IO bound</i>	<code>sysbench -test=fileio -file-total-size=24G -file-test-mode=rndrw -max-time=50 -max-requests=0 run</code>

Tabela 5.1: Linha de comando para cada tipo de tarefa

Também queremos observar o comportamento de cada tipo de tarefa quando executada paralelamente com outras do mesmo tipo. Assim, executamos  $i$  processos paralelos, para  $i = 1, 2, \dots, 8$ , para cada tipo de tarefa.

Para cada execução, gravamos o consumo de potência e do processador através da ferramenta `powerdump`. Para um determinado tipo de tarefa, utilizamos a mesma gravação para os processos paralelos fazendo intervalos de 10s a cada incremento no valor de  $i$ . Pelos gráficos, o leitor não deverá encontrar dificuldades em perceber quantos processos foram utilizados.

### 5.1.2 Resultados obtidos

A Figura 5.1 mostra o consumo de potência para processos *CPU Bound* rodando paralelamente.

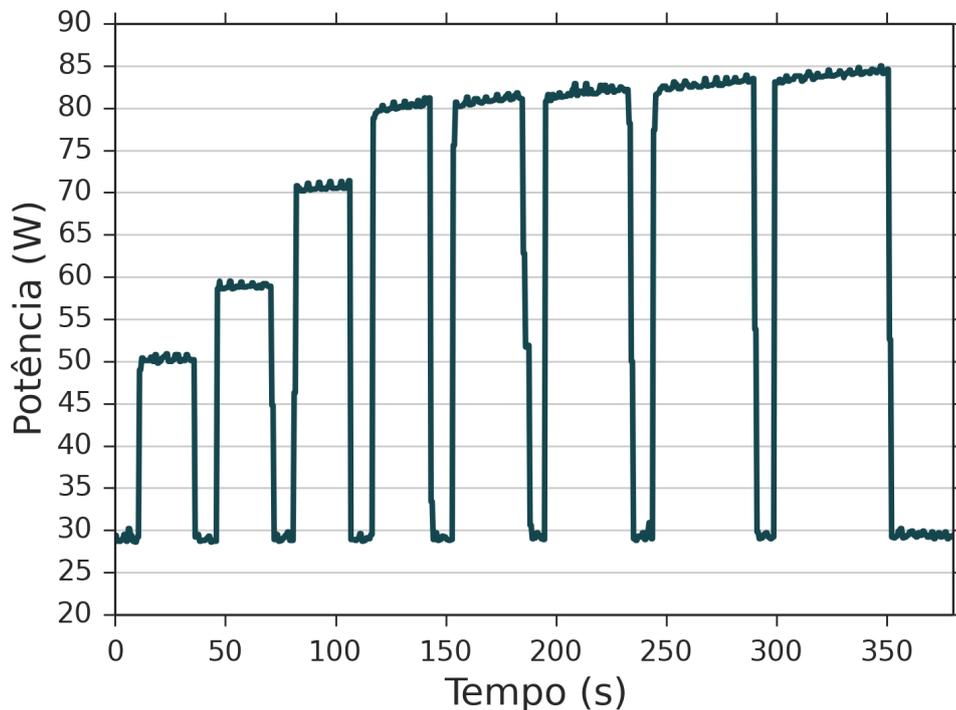
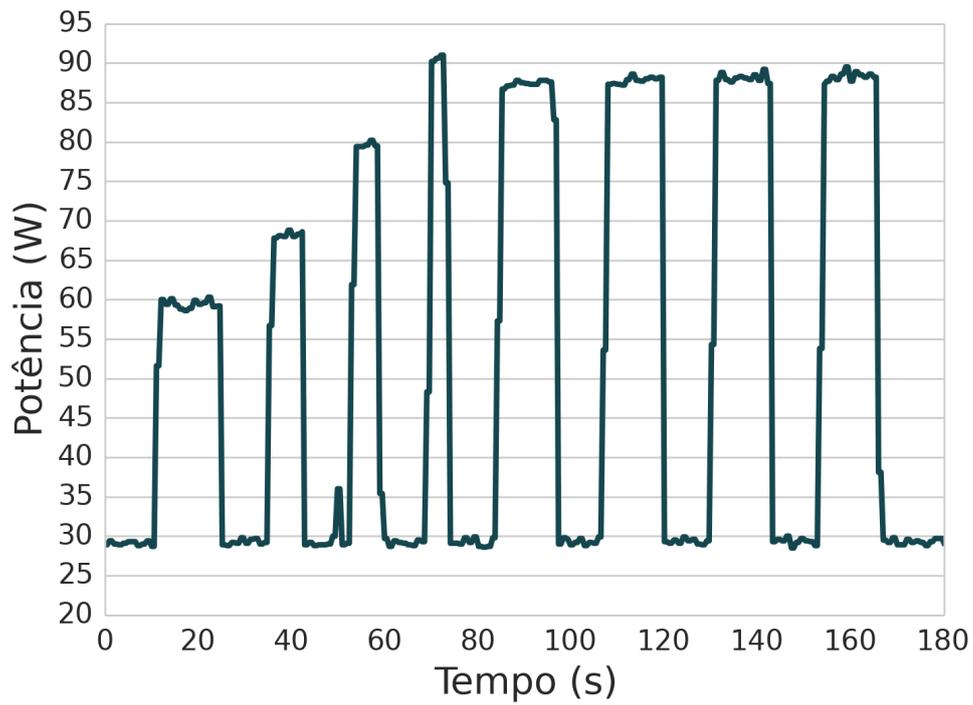


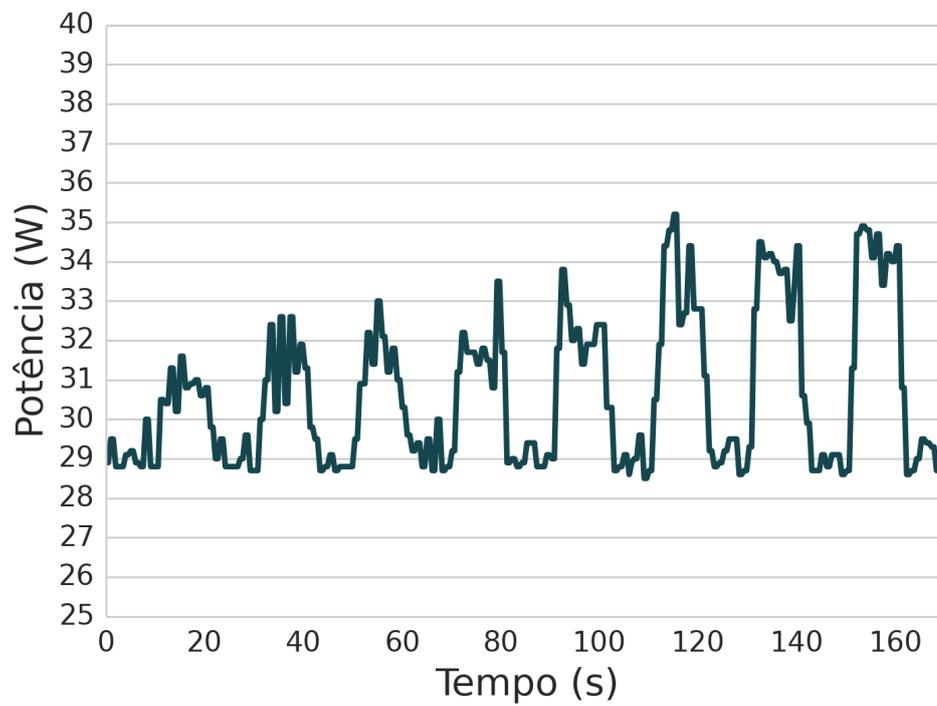
Figura 5.1: Perfil do consumo de potência para uma tarefa CPU bound

A Figura 5.2 mostra o consumo de potência para processos *Memory Bound* rodando paralelamente.

A Figura 5.3 mostra o consumo de potência para uma aplicação *IO Bound* rodando paralelamente.



**Figura 5.2:** Perfil do consumo de potência para uma tarefa Memory bound



**Figura 5.3:** Perfil do consumo de potência para uma tarefa IO bound

### 5.1.3 Análise dos resultados

A letra  $n$  indicará o número de processos paralelos.

O comportamento observado dos processos *CPU Bound* não é surpreendente. Até utilizarmos os quatro do processador núcleos ( $n \in \{1, 2, 3, 4\}$ ), não há grande diferença no tempo de execução. Nestes casos, apenas o consumo de potência cresce pois, a cada passo, passamos a usar um núcleo que estava inativo. Para  $n > 4$ , as execuções começam a demorar mais pois não há como aumentar o processamento. O que é interessante, é o aparente crescimento do consumo de potência, próximo de 5W, quando vamos de  $n = 5$  a  $n = 8$ . Este aumento no consumo parece ser de responsabilidade, provavelmente devido às trocas de contexto. Em nosso último experimento, usaremos esta observação para tentar encontrar uma oportunidade no agendamento paralelo de processos.

Sobre o comportamento dos processo *Memory Bound*, vemos diretamente que os consumos são maiores que os dos processos *CPU Bound*. Isso ocorre pois, neste caso temos a soma do consumo do processador com a da memória. Vemos que o comportamento observado não é tão uniforme quanto no caso *CPU Bound*. O tempo de acesso à memória e a implementação do modo *Memory Bound* pelo *sysbench* podem ser as causas destas diferenças. A diferença entre processos que usam a memória de modos diferentes serão exploradas nos experimentos sobre algoritmos.

Para os processos *IO Bound*, vemos que é ainda menos uniforme que no caso *Memory Bound*. Novamente, uma possível causa é a latência no acesso ao disco. Vemos rapidamente que o consumo é bem menor do que nos outros casos, com a maior perturbação sendo de pouco menos de 7W. Esta pequena diferença deve ser reflexo do pouco trabalho, além do que já estava considerado no consumo base do sistema, que o disco rígido faz para realizar as operações. Também há pouco uso da *CPU* neste caso, que passa a maior parte do tempo em espera de operações de Entrada e Saída.

Com estas observações, pudemos ver que nossos sistema de medição parece ser suficiente para comparar o consumo energético de aplicações. Mostramos alguns fatores que influenciam no consumo de potência. E finalmente, conseguimos encontrar algumas oportunidades.

## 5.2 Escolha de Algoritmos

Esta seção é baseada no trabalho de Bunse et al. [BHMR09] em que os autores avaliam o desempenho energético de alguns dos algoritmos de ordenação mais comuns em dispositivos embarcados. Como estes sistemas são muito limitados, o número de entradas a serem ordenadas não passou de 1000. Sua proposta é, principalmente, responder às seguintes questões:

- O consumo energético é dependente do desempenho temporal?
- Qual o impacto do tipo de dados no consumo energético?

Após medir o consumo energético dos algoritmos, comparam sua complexidade com o desempenho temporal, e com o consumo energético. Apesar de o tempo de execução ser bem aproximado pela complexidade, isso não ocorre para o consumo de energia. Observaram, contra nossa intuição, o *insertion sort* sendo o algoritmo mais energeticamente eficiente. Segundo os autores, isso ocorre por ser o que consome menos memória.

Para examinar o impacto do tipo de dados, compararam inteiros com representações em ponto flutuante (*floats*). A diferença foi muito grande, com a ordenação de *floats* consumindo mais energia. Segundo os autores, isto ocorreu pois, além de *floats* consumirem mais memória, os processadores dos sistemas de testes não possuíam uma *Floating Point Unit*, sendo esta emulada por um software.

Em nosso trabalho, atacamos somente a primeira questão. Como os autores de [BHMR09], comparamos algoritmos de ordenação, porém como nosso sistema é mais complexo, estamos interessados em ordenar entradas de tamanhos bem maiores. Isso faz com que no nosso caso não

haja esperanças para o *insertion sort*, com seu baixíssimo desempenho temporal. Assim, com base em algoritmos conhecidos, construiremos um algoritmo que consome menos potência e veremos que em alguns casos, observaremos este algoritmo consumindo menos energia mesmo perdendo em desempenho.

### 5.2.1 Especificação do experimento

Iremos estudar os seguintes algoritmos clássicos de ordenação:

- HeapSort
- MergeSort
- QuickSort

E como candidato a algoritmo com melhor eficiência energética, usaremos uma variação esparsa do BucketSort com o InsertionSort em lista ligada sendo utilizado a cada vez que um número é adicionado a um dos *buckets*. Chamaremos este algoritmo de SparseBucketSort. Escolhemos o SparseBucketSort como candidato pois tem natureza de consumo muito diferente dos outros algoritmos de ordenação.

Ainda, consideraremos 3 versões do SparseBucketSort.

- SparseBucketSort<sub>0,5</sub>
- SparseBucketSort<sub>1,0</sub>
- SparseBucketSort<sub>1,5</sub>

Assim, se  $n$  denotar o tamanho da entrada a ser ordenada pelo algoritmo SparseBucketSort <sub>$\alpha$</sub> , o BucketSort será feito usando-se  $\alpha \times n$  *buckets*.

Implementamos os algoritmos em C, para ordenação de `doubles`. Fizemos 30 amostras para cada algoritmo, sobre um vetor com valores uniformemente distribuídos entre 0 e `RAND_MAX`. Após a ordenação, verifica-se se o vetor foi ordenado. Os consumos para criação do vetor e verificação de corretude da ordenação não foram considerados na medição.

### 5.2.2 Resultados obtidos

A Figura 5.4 mostra o consumo médio de potência dos algoritmos estudados.

Na Figura 5.5, pode-se ver como foi o desempenho temporal dos algoritmos.

E o consumo energético de cada algoritmo pode ser visto na Figura 5.6.

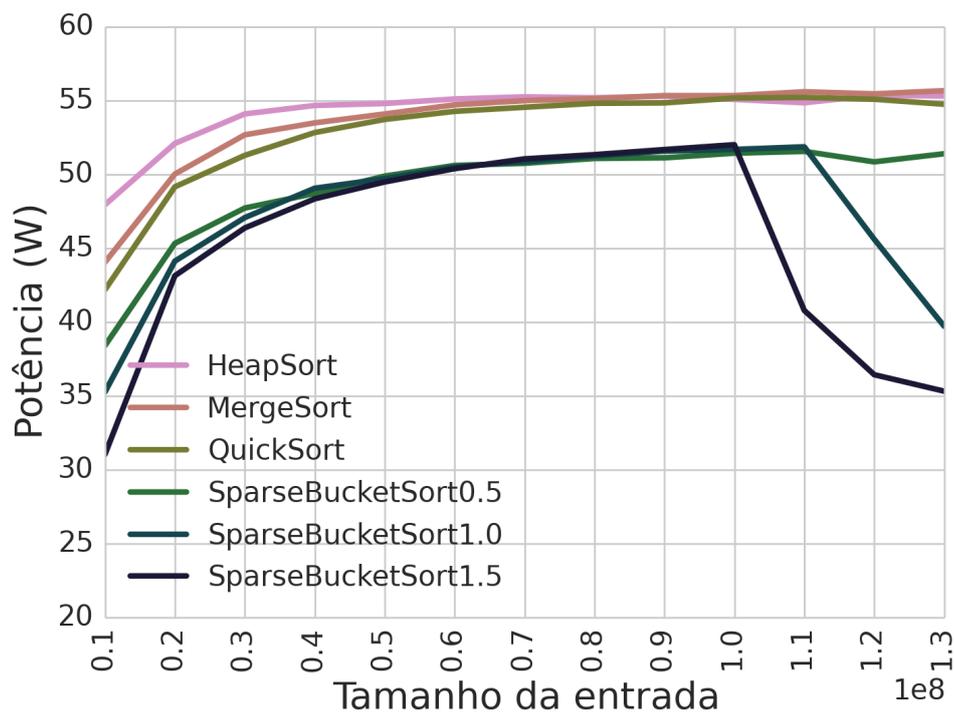


Figura 5.4: Potência consumida pelos algoritmos para diferentes tamanhos de entrada

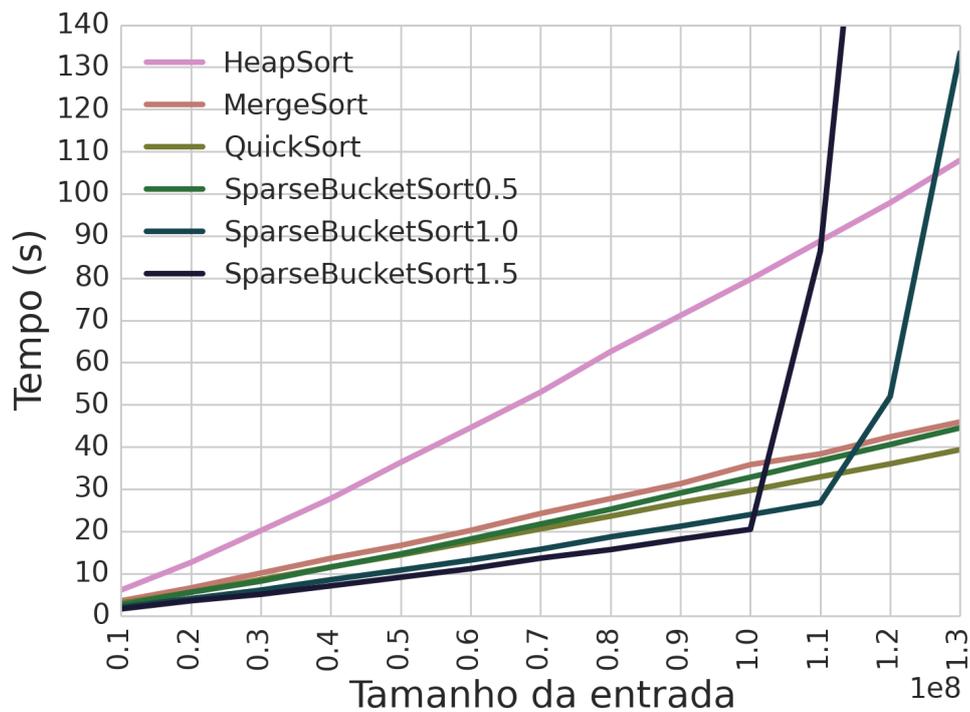


Figura 5.5: Tempo de execução dos algoritmos para diferentes tamanhos de entrada

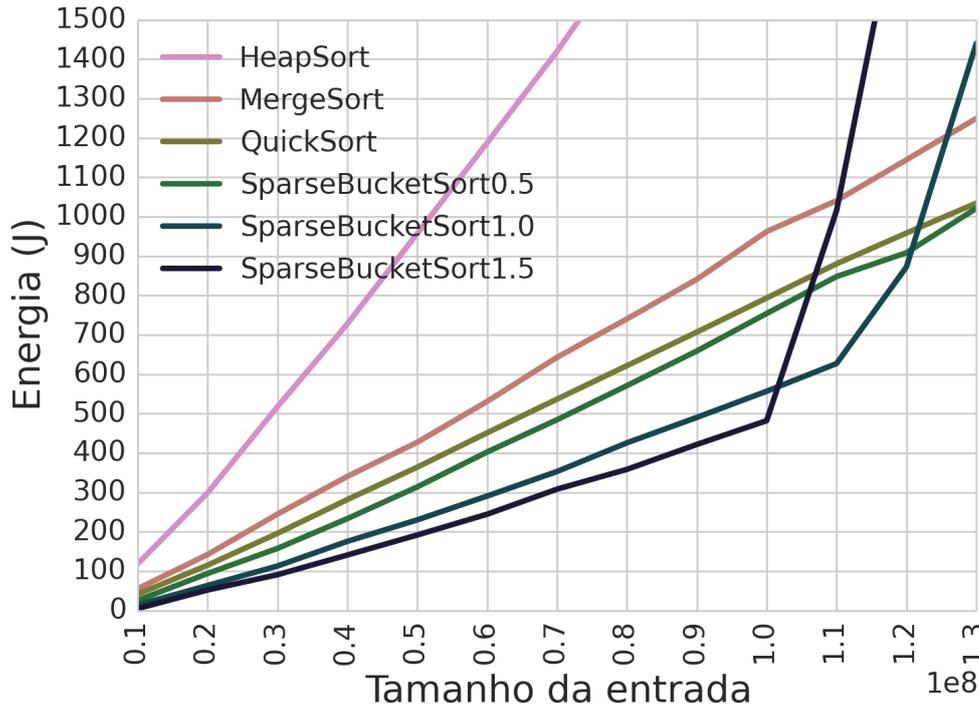


Figura 5.6: Energia consumida pelos algoritmos para diferentes tamanhos de entrada

### 5.2.3 Análise dos resultados

Primeiro, vemos pela Figura 5.4 que o SparseBucketSort de fato parece consumir menos potência do que os outros. Interessante notar que o rápido decréscimo dos consumos para o SparseBucketSort<sub>1.5</sub> a partir de  $n = 1.0 \times 10^8$  e para o SparseBucketSort<sub>1.0</sub> a partir de  $n = 1.1 \times 10^8$  acontece pois estes esgotam a RAM do sistema e passam a usar o *swap*. Lembrando do que foi visto sobre os perfis de consumo de potência, esse decréscimo faz sentido, pelo menor consumo de potência de aplicações *CPU Bound*.

Sobre o desempenho temporal:

- O SparseBucketSort<sub>1.5</sub> teve o melhor desempenho temporal até esgotar a RAM. Usando o *swap*, teve desempenho melhor que o HeapSort para  $n = 1.1 \times 10^8$ . Perdeu em desempenho para MergeSort com uma diferença relativa da ordem de 100%.
- O SparseBucketSort<sub>1.0</sub> teve o melhor desempenho temporal para  $n = 1.1 \times 10^8$ . Usando o *swap*, perdeu em desempenho quando  $n = 1.2 \times 10^8$  para SparseBucketSort<sub>0.5</sub>, QuickSort e MergeSort.
- O SparseBucketSort<sub>0.5</sub> perdeu em desempenho, em todos os casos observados, para o QuickSort. Chegou a ter uma diferença da ordem de 10% no desempenho comparado ao QuickSort.

Sobre o consumo energético:

- O SparseBucketSort<sub>1.5</sub> teve consumo energético pouco melhor do que o MergeSort para  $n = 1.1 \times 10^8$ .
- O SparseBucketSort<sub>1.0</sub> é o que teve o menor consumo energético para  $n = 1.2 \times 10^8$ .
- O SparseBucketSort<sub>0.5</sub> consumiu menos energia, em todos os casos observados, que o QuickSort.

Com estas observações, mostramos que é possível construir um algoritmo com o objetivo de diminuir o consumo de potência como alternativa ao desenvolvimento baseado em performance.

Porém, é importante notar que este estudo foi feito como prova de conceito, pois não nos preocupamos com uma comparação rigorosa entre os algoritmos em situações extremas nem com a escalabilidade do SparseBucketSort.

Pudemos ver claramente que nem sempre o algoritmo de melhor performance é o que consome menos energia. Isso sugere, como Bunse et al. indicaram em [BHMR09], que aplicações adaptáveis são promissoras na busca por eficiência energética. Como exemplo de aplicação direta, uma aplicação que precisa ordenar uma entrada poderia escolher qual algoritmo usar dentro dos apresentados, de acordo com uma combinação entre desempenho e consumo energético passado pelo usuário.

### 5.3 Eliminação de *Code Smells*

Esta seção é baseada no trabalho de Vetro et al. [VAPM13], em que os autores definem *energy code smells* como padrões de código que afetam a eficiência energética de aplicações. Tentam, então, responder a duas perguntas principais:

- Quais *code smells*<sup>1</sup> são *energy code smells*?
- *Code smells* que impactam no consumo de energia também impactam na performance?

Para isso, os autores implementaram, em C++, funções com e sem alguns *code smells* detectáveis pelos softwares *CppCheck* e *FindBugs*, para análise estática de código. Estes dois grupos de código foram executados num sistema embarcado, sem sistema operacional, e comparados em relação ao seu consumo energético e desempenho temporal.

Após analisar dos resultados, os autores concluíram que *code smells* têm impacto, na potência média, da ordem de micro Watts e não encontraram impacto no tempo de execução. Ao final do trabalho, o ambiente de testes é apontado como principal limitação do estudo. Assim, indicam a necessidade de serem feitos estudos sobre outras plataformas.

Utilizando um experimento parecido, queremos validar os resultados para nosso sistema de testes, utilizando a linguagem C. Nesta seção, chamaremos de *código sujo* aquele que possui um *code smell*, e de *código limpo*, aquele que não. As variáveis estudadas foram o tempo de execução (T), a potência média (P), e a energia consumida **pela aplicação** (E).

#### 5.3.1 Especificação do experimento

Consideramos os *code smells* na tabela abaixo:

<i>Code Smell</i>	Descrição
<i>Dead Local Store</i>	Variáveis guardam valores que não serão utilizados novamente.
<i>Non Short Circuit</i>	Operador lógico na expressão não é de curto circuito.
Parameter By Value	Parâmetro de função passado por valor, não referência.
<i>Redundant Function Call</i>	Chamada de função sobre os mesmo parâmetros a cada iteração.
<i>Repeated Conditional</i>	Bloco condicional dentro de outro sobre mesma expressão.
<i>Self Assignment</i>	Atribuição do valor de uma variável à ela mesma.

**Tabela 5.2:** Code smells *considerados*

<sup>1</sup>Não encontramos tradução simples e concisa. Como é bem definido em [Fow97], deixamos no idioma original.

Dentre eles, apenas o *Redundant Function Call* não foi considerado pelos autores de [VAPM13]. Além disso, todos os outros são aqueles indicados no artigo como *energy code smells*.

Chamamos limpo, o código sem *code smells* e sujo, aquele com o respectivo *code smell*. Escrevemos código limpo e sujo para cada padrão. Todos os códigos foram compilados utilizando o gcc (v.4.6.3) sem otimizações para evitar que o próprio compilador fizesse a eliminação e não pudéssemos identificar seu impacto.

Abaixo, um exemplo para o *code smell Non Short Circuit*. Temos o código com o *code smell* à esquerda, e depois de sua eliminação, à direita.

```

1 #include <stdlib.h>
2
3 void non_short_circuit();
4
5 int main(int argc, char *argv[]) {
6     long i, n;
7
8     n = atoi(argv[1]);
9     for (i = 0; i < n; i++)
10         non_short_circuit();
11     return 0;
12 }
13
14 void non_short_circuit() {
15     int a, b;
16
17     a = rand();
18     b = rand();
19     if ((a < b) & (b > a)) {
20         a += b;
21         return;
22     }
23     b += a;
24 }
```

```

1 #include <stdlib.h>
2
3 void wo_non_short_circuit();
4
5 int main(int argc, char *argv[]) {
6     long i, n;
7
8     n = atoi(argv[1]);
9     for (i = 0; i < n; i++)
10         wo_non_short_circuit();
11     return 0;
12 }
13
14 void wo_non_short_circuit() {
15     int a, b;
16
17     a = rand();
18     b = rand();
19     if ((a < b) && (b > a)) {
20         a += b;
21         return;
22     }
23     b += a;
24 }
```

Como chamadas de funções podem ser muito rápidas, para fazer medições sobre o consumo de funções é necessário fazer várias iterações sobre suas chamadas a cada execução do programa. Para isso, os *code smells* foram implementados em funções e o número de chamadas dessas funções é dado pelo argumento passado em linha de comando na execução do programa. Este número é diferente para cada padrão testado, dependente do tempo de execução da função. Embora, evidentemente, seja o mesmo número para as versões limpas e sujas de um mesmo código.

Com  $n = 50$  medições através do *energyanalyser*, conseguimos os resultados apresentados na próxima seção.

### 5.3.2 Resultados obtidos

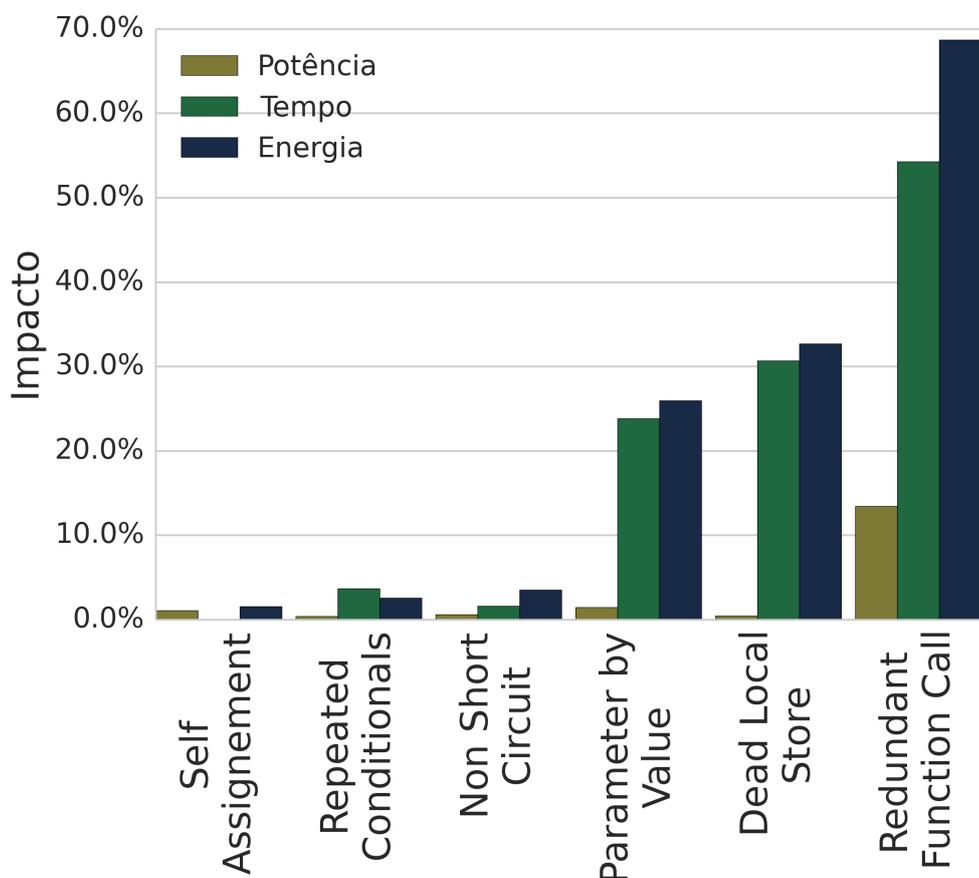
Tabela 5.3 contém os valores das médias amostrais de cada variável estudada. Nesta seção T indica o tempo, P a potência média<sup>2</sup> e E a energia consumida, medidas em segundos, Watts e Joules, respectivamente. Os índices S e L indicam as medições sobre, respectivamente, os códigos sujos e limpos.

<sup>2</sup>Assim,  $\bar{P}$  indicará a médias das potências médias observadas.

<i>Code Smell</i>	Sujo			Limpo		
	$\bar{T}_S$ (s)	$\bar{P}_S$ (W)	$\bar{E}_S$ (J)	$\bar{T}_L$ (s)	$\bar{P}_L$ (W)	$\bar{E}_L$ (J)
Dead Local Store	7.7319	50.2840	155.8870	5.3567	50.0569	104.8748
Non Short Circuit	9.8608	52.7283	222.2495	9.7011	52.3865	214.3383
Redundant Call	8.7104	52.3006	190.6049	3.9796	45.2733	59.6092
Repeated Conditionals	9.8446	52.7815	219.2489	9.4782	52.5596	213.5707
Parameter by Value	11.1594	53.4761	257.4827	8.5002	52.6891	190.6035
Self Assignment	9.5107	53.5184	218.6024	9.5105	52.9153	215.0994

Tabela 5.3: Médias amostrais das variáveis estudadas

A Figura 5.7 mostra o impacto, em cada variável, da eliminação de cada *code smell*.

Figura 5.7: Impacto médio da retirada de cada *code smell* nas variáveis de interesse

Ainda, a Tabela 5.4 contém os cálculos dos impactos apresentados na Figura 5.7.

### 5.3.3 Análise dos resultados

Pela Tabela 5.4, podemos ver que, exceto pelos padrões *Redundant Call* e *Self Assignment*, a diferença no tempo de execução parece ser a principal causa da diminuição do consumo energético. Este se opõe ao que foi encontrado pelos autores de [VAPM13].

Também diferente do encontrado em [VAPM13], a diferença de potência encontrada foi, pelo menos, da ordem de, pelo menos,  $10^{-3}$  Watts, não  $10^{-6}$ , como o artigo mostra. Porém esta diferença era esperada pois o sistema embarcado utilizado em [VAPM13] tinha um nível de consumo muito menor.

Além das diferenças apresentadas entre as duas configurações, uma das possíveis causas destas diferenças é o pequeno número de repetições das funções que os autores do artigo utilizaram, de

<i>Code Smell</i>	$\frac{T_S - T_L}{T_S}$	$\frac{P_S - P_L}{P_S}$	$\frac{E_S - E_L}{E_S}$
<i>Dead Local Store</i>	30.72%	0.45%	32.72%
<i>Non Short Circuit</i>	1.62%	0.65%	3.56%
<i>Redundant Call</i>	54.31%	13.44%	68.73%
<i>Repeated Conditionals</i>	3.72%	0.42%	2.59%
<i>Parameter by Value</i>	23.83%	1.47%	25.97%
<i>Self Assignment</i>	0.00%	1.13%	1.60%

**Tabela 5.4:** Tabela de impacto da eliminação de cada code smell

1.000.000, enquanto nossos números de repetição vão de 40.000.000 a 1.000.000.000.

De qualquer forma, o impacto de *code smells* na eficiência energética de uma aplicação é da ordem de nJ, que é quase imperceptível. Assim, o estudo da eliminação de *code smells* a fim de economizar energia não parece ser muito promissor, porém pode melhorar o entendimento de como uma aplicação pode gastar energia.

Estas diferenças de resultados também mostram que os resultados sobre eficiência energética encontrados para dispositivos móveis ou embarcados podem não valer para computadores pessoais e servidores. Novamente, a validação de resultados para diferentes plataformas é justificada.

## 5.4 Agendamento de Processos

Nesta seção, estudaremos como o escalonamento de processos pode impactar na eficiência energética de nosso sistema. Em particular, queremos saber como, para uma certa tarefa dada por vários processos, a escolha do número máximo de processos paralelos impacta no consumo de energia.

Importante lembrar que, teoricamente, o agendamento de processos é serviço do sistema operacional. Porém, como nossos privilégios são limitados na máquina de testes, este estudo foi feito na camada de aplicação.

### 5.4.1 Especificação do experimento

Consideramos, como tarefa, a execução de 32 processos do programa `pidigits 30000`. A aplicação `pidigits` faz parte do projeto *The Computer Language Benchmarks Game* [?]. Para mostrar que o agendamento de processos impacta na eficiência energética, usaremos o `energyanalyser` com 50 observações para cada  $n = 1, 2, \dots, 32$ , sobre o procedimento descrito abaixo:

- Execute os 32 processos, porém permitindo que no máximo  $n$  processos rodem paralelamente.

Realizamos este procedimento com a ferramenta GNU Parallel [Tan11].

Com isso, esperamos encontrar diferenças significativas no consumo energético para cada valor de  $n$ . Ainda, se isto ocorrer, tentaremos encontrar um  $n$  ótimo, que minimize o consumo energético total.

### 5.4.2 Resultados obtidos

Primeiramente, pode-se ver como a potência varia em função de  $n$  na Figura 5.8.

A Figura 5.9 mostra que o  $n$  ótimo para performance observado foi de  $n = 7$ .

O consumo energético descontando o consumo do sistema pode ser visto na Figura 5.10. Este gráfico sugere que  $n = 4$  é ótimo para energia. Porém, como trata-se de um agendamento, o objetivo é minimizar o consumo de todo o sistema. Este consumo total é mostrado na Figura 5.11. Neste caso, o  $n$  ótimo observado foi  $n = 7$ .

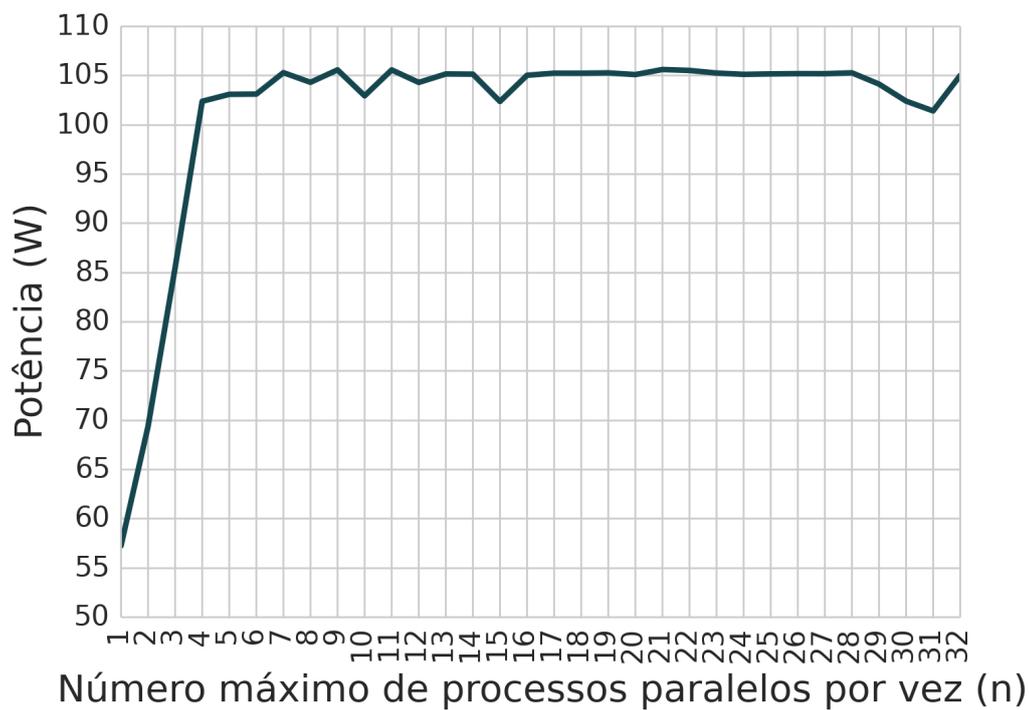


Figura 5.8: Potência média consumida pelo sistema para cada  $n$

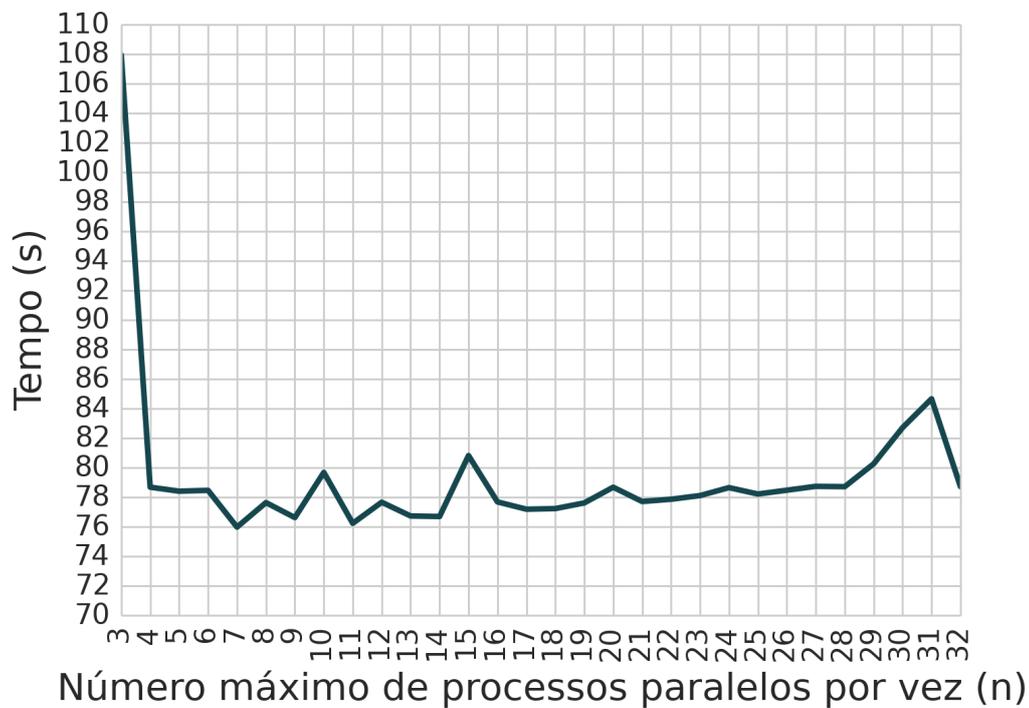


Figura 5.9: Tempo médio de execução da tarefa para cada  $n$



Figura 5.10: Energia média consumida pela tarefa para cada  $n$

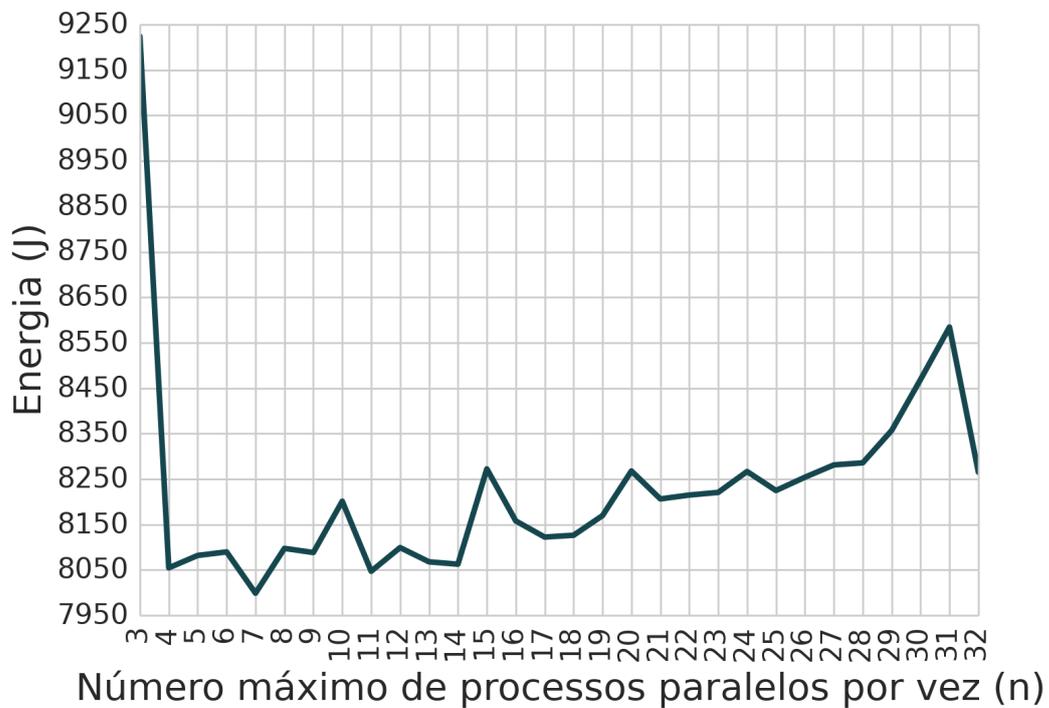


Figura 5.11: Energia média consumida pelo sistema para cada  $n$

Seja  $\bar{E}(i)$  o consumo energético total utilizando  $n = i$ , a Tabela 5.5 mostra a melhora pela utilização de  $n = 7$ .

$n$	$\frac{\bar{E}(n)-\bar{E}(7)}{\bar{E}(7)}$	$n$	$\frac{\bar{E}(n)-\bar{E}(7)}{\bar{E}(7)}$	$n$	$\frac{\bar{E}(n)-\bar{E}(7)}{\bar{E}(7)}$	$n$	$\frac{\bar{E}(n)-\bar{E}(7)}{\bar{E}(7)}$
1	103.51%	9	1.12%	17	1.54%	25	2.82%
2	35.59%	10	2.53%	18	1.59%	26	3.19%
3	15.32%	11	0.60%	19	2.13%	27	3.53%
4	0.70%	12	1.26%	20	3.36%	28	3.59%
5	1.04%	13	0.86%	21	2.59%	29	4.48%
6	1.14%	14	0.80%	22	2.70%	30	5.88%
7	0.00%	15	3.42%	23	2.77%	31	7.32%
8	1.23%	16	1.99%	24	3.35%	32	3.33%

**Tabela 5.5:** Tabela de melhora pela utilização de  $n = 7$  para cada  $n$

### 5.4.3 Análise dos resultados

Os gráficos apresentados sugerem que o escalonamento impacta no consumo energético. Como esperado, o consumo é muito maior quando o fato de o sistema ser paralelo é mal explorado, ou seja,  $n \in \{1, 2, 3\}$ . Além disso, observamos vários escalonamentos não triviais ( $n \notin \{1, 32\}$ ) que melhoram o consumo energético.

Os máximos locais foram exatamente os mesmos para o desempenho temporal e consumo energético total, enquanto os mínimos locais só não são iguais pois  $n = 4$  não é mínimo local para o tempo. Isto indica que deve valer a forte relação entre performance temporal e eficiência energética também para o escalonamento de processos. Por outro lado, também sugere que podemos esperar casos especiais em que pode-se trocar performance por economia energética.

Vimos que  $n = 7$ , deve ser utilizado se quisermos minimizar o consumo energético do sistema, apesar de  $n = 4$  ser o que resulta em melhor consumo pela tarefa. Este é um caso particular de *race to idle*, em que é preferível rodar uma tarefa sobrecarregando um pouco o processador para terminar o quanto antes. Assim, como a tarefa é finalizada mais rapidamente para  $n = 7$ , o consumo energético base do sistema no decorrer de sua execução é menor do que o aquele para  $n = 4$ . Esta é uma possibilidade de melhora pois muitas vezes é possível minimizar o consumo base finalizando processos e desligando componentes de hardware que não contribuem para nossa tarefa. Isto também mostra como é útil medir o consumo das aplicações separadamente do consumo do sistema na busca por oportunidades de otimização energética.

Finalmente, apesar de todo o experimento ser feito somente na camada de aplicação, conseguimos encontrar oportunidades de otimização do consumo energético para o escalonamento. Isso indica ser interessante investir em técnicas de escalonamento a fim de implementar no *kernel* do sistema um agendador de processos alternativo para eficiência energética.

## Capítulo 6

# Conclusões

Fizemos uma rápida introdução à computação verde para justificar o estudo em eficiência energética.

Revisamos os estudos em eficiência energética no *hardware* e sistema operacional, para mostrar ao leitor a posição que as técnicas na camada de aplicação ocupam.

No experimento sobre perfis de consumos de potência, mostramos que um sistema de medição simples pode ser utilizado para medir o consumo de aplicações. Além disso, pudemos ver como o software realmente é responsável pelo consumo de potência e de energia de um sistema computacional.

Quando estudamos a escolha de algoritmos no consumo energético, pudemos usar o conhecimento adquirido no primeiro experimento para escolher um algoritmo que consumisse menos potência que os outros clássicos. Vimos que nem sempre o algoritmo mais rápido é o mais recomendado. Assim, reforçamos as conclusões de trabalhos de outros autores de que aplicações adaptáveis são promissoras em eficiência energética na aplicação.

Avaliando o impacto de *code smells* no consumo energético de aplicações, pudemos ver novamente que resultados para sistemas embarcados ou dispositivos móveis não são em geral estendíveis para computadores pessoais e servidores. Também vimos que, pelo pequeno impacto da ordem de nJ, não parecem ser uma oportunidade real de melhoria de consumo energético. Porém, continua sendo boa prática de programação eliminá-los.

Vimos que o escalonamento paralelo pode impactar no consumo energético. Como havíamos visto no experimento sobre perfis de consumo de potência, com muitos processos paralelos, começa a haver custos energéticos significativos pelas trocas de contexto. Este custo nem sempre é perceptível na mesma ordem de grandeza na performance, o que pode fazer com que o número ótimo de processos paralelos seja diferente para performance e energia.

Este foi um trabalho de conclusão de graduação, com pouco tempo para seu desenvolvimento. Assim, o estudo teve caráter exploratório em que favorecemos a clareza de exposição em relação ao rigor estatístico. Futuramente, poderemos expandir este estudo com uma análise quantitativa sem muitas dificuldades. Finalmente, conseguimos atingir nossos objetivos propostos na introdução através de nossos experimentos.



# Referências Bibliográficas

- [Alb10] Susanne Albers. Energy-efficient algorithms. *Communications of the ACM*, 53(5):86–96, 2010. 5
- [Ams10] University Amsterdam. Best practices for energy efficient software, Setembro 2010. [http://wiki.cs.vu.nl/green\\_software/index.php/Best\\_practices\\_for\\_energy\\_efficient\\_software](http://wiki.cs.vu.nl/green_software/index.php/Best_practices_for_energy_efficient_software). 9
- [Ard13] Luca Ardito. Energy aware self-adaptation in mobile systems. Em *Proceedings of the 2013 International Conference on Software Engineering*, páginas 1435–1437. IEEE Press, 2013. 10
- [Ard14] Luca Ardito. *Energy-aware Software*. Tese de Doutorado, Politecnico di Torino, 2014. 1, 15
- [BBL<sup>+</sup>11] Anton Beloglazov, Rajkumar Buyya, Young Choon Lee, Albert Zomaya et al. A taxonomy and survey of energy-efficient data centers and cloud computing systems. *Advances in Computers*, 82(2):47–111, 2011. 5, 6
- [BHMR09] Christian Bunse, Hagen Hopfner, Essam Mansour e Suman Roychoudhury. Exploring the energy consumption of data sorting algorithms in embedded and mobile environments. Em *Mobile Data Management: Systems, Services and Middleware, 2009. MDM'09. Tenth International Conference on*, páginas 600–607. IEEE, 2009. 9, 18, 22
- [BHRM09] Christian Bunse, Hagen Höpfner, Suman Roychoudhury e Essam Mansour. Choosing the "best" sorting algorithm for optimal energy consumption. Em *ICSOFT (2)*, páginas 199–206, 2009. 10
- [Bro13] Dominik Brodowski. Cpu frequency and voltage scaling code in the linux tm kernel, Novembro 2013. <https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt>. 8
- [Fow97] Martin Fowler. *Refactoring: Improving the design of existing code*, 1997. 22
- [HA09] Robert R Harmon e Nora Auseklis. Sustainable it services: Assessing the impact of green computing practices. Em *Management of Engineering & Technology, 2009. PICMET 2009. Portland International Conference on*, páginas 1707–1717. IEEE, 2009. 3, 9
- [HP04] Intel Hewlett-Packard. Microsoft, phoenix, and toshiba. advanced configuration and power interface specification, 2004. 7
- [IC96] Microsoft Corporation Intel Corporation. Advanced power management apm bios interface specification, Fevereiro 1996. <http://intel-vintage-developer.eu5.org/IAL/POWERMGM/APMV12.PDF>. 6
- [Int11] Intel. Acpi overview, Fevereiro 2011. [http://www.acpi.info/presentations/ACPI\\_Overview.pdf](http://www.acpi.info/presentations/ACPI_Overview.pdf). 7

- [Kop04] Alexey Kopytov. Sysbench: a system performance benchmark. 2004. <http://sysbench.sourceforge.net>. 16
- [Mur08] San Murugesan. Harnessing green it: Principles and practices. *IT professional*, 10(1):24–33, 2008. 1, 8, 9
- [PRZ<sup>+</sup>02] Jim Puckett, Mary Ryan, Bill Zude, Staci Simpson, Tony Hedenrick e Basel Action Network. *Exporting harm: the high-tech trashing of Asia*. Basel Action Network, 2002. 3
- [SA11] B Steigerwald e Abhishek Agrawal. Developing green software. *Intel White Paper*, 2011. 9
- [SW13] Suthipong Sthiannopkao e Ming Hung Wong. Handling e-waste in developed and developing countries: Initiatives, practices, and consequences. *Science of the Total Environment*, 463:1147–1153, 2013. 3
- [Tan11] Ole Tange. Gnu parallel—the command-line power tool. *The USENIX Magazine*, 36(1):42–47, 2011. 25
- [VAPM13] Antonio Vetro, Luca Ardito, Giuseppe Procaccianti e Maurizio Morisio. Definition, implementation and validation of energy code smells: an exploratory study on an embedded system. Em *ENERGY 2013, The Third International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*, páginas 34–39, 2013. 22, 23, 24
- [Wat02] Takanori Watanabe. Acpi implementation on freebsd. Em *USENIX Annual Technical Conference, FREENIX Track*, páginas 121–131, 2002. 7
- [WK13] Lizhe Wang e Samee U Khan. Review of performance metrics for green data centers: a taxonomy study. *The journal of supercomputing*, 63(3):639–656, 2013. 4