UNIVERSIDADE DO ESTADO DA BAHIA

DEPARTAMENTO DE CIÊNCIAS EXATAS E DA TERRA

CURSO DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

MATEUS PENA MACHADO DE JESUS

AVALIAÇÃO DA EFICIÊNCIA ENERGÉTICA DAS ESTRUTURAS DE REPETIÇÃO PARA A CONSTRUÇÃO DE SOFTWARES VERDES

SALVADOR

2022

MATEUS PENA MACHADO DE JESUS

AVALIAÇÃO DA EFICIÊNCIA ENERGÉTICA DAS ESTRUTURAS DE REPETIÇÃO PARA A CONSTRUÇÃO DE SOFTWARES VERDES

Monografia apresentada ao curso de Sistemas de Informação do Departamento de Ciências Exatas e da Terra da Universidade do Estado da Bahia - UNEB, como requisito à obtenção do grau de bacharel em Sistemas de Informação.
Área de Concentração: Ciência da Computação

Orientador: Ernesto de Souza Massa Neto

SALVADOR

2022

FICHA CATALOGRÁFICA
Sistema de Bibliotecas da UNEB

P397a Pena Machado de Jesus, Mateus

Avaliação da eficiência energética das estruturas de repetição para a construção de Softwares Verdes / Mateus Pena Machado de Jesus. - Salvador, 2022.

63 fls.

Orientador(a): Ernesto de Souza Massa Neto.

Inclui Referências

TCC (Graduação - Sistemas de Informação) - Universidade do Estado da Bahia. Departamento de Ciências Exatas e da Terra. Campus I. 2022.

1.Computação Verde. 2.Programação. 3.Eficiência Energética.

CDD: 604

MATEUS PENA MACHADO DE JESUS

AVALIAÇÃO DA EFICIÊNCIA ENERGÉTICA DAS ESTRUTURAS DE REPETIÇÃO PARA A CONSTRUÇÃO DE SOFTWARES VERDES

Monografia apresentada ao curso de Sistemas de Informação do Departamento de Ciências Exatas e da Terra da Universidade do Estado da Bahia - UNEB, como requisito à obtenção do grau de bacharel em Sistemas de Informação. Área de Concentração: Ciência da Computação

Aprovada em:

BANCA EXAMINADORA

Ernesto de Souza Massa Neto (Orientador)
Universidade do Estado da Bahia – UNEB

Carlos Helano Aquino do Nascimento
Universidade do Estado da Bahia – UNEB

Eduardo Manuel de Freitas Jorge
Universidade do Estado da Bahia – UNEB

AGRADECIMENTOS

Agradeço ao professor Ernesto Massa por todas as orientações que possibilitaram o desenvolvimento deste trabalho, sobretudo por se tratar de uma área bastante inovadora. Agradeço também a professora Débora Chaves que sempre se colocou à disposição para ajudar a revisar o texto, diversas vezes dando valiosas dicas que contribuíram para aumentar a qualidade dessa produção.

RESUMO

A temática de sustentabilidade em computação tem ganhado cada vez mais destaque, tendo em vista a crescente preocupação com o futuro das próximas gerações, o que tem despertado o olhar para a necessidade de atenuação dos impactos negativos causados pelo homem ao meio ambiente. Diferentes estudos foram conduzidos nos últimos anos abordando a chamada Computação Verde, ressaltando a importância do uso dos computadores de forma consciente e alinhada com as necessidades ecológicas. Este trabalho objetivou realizar um estudo sobre a Computação Verde pautado na metodologia experimental acerca do desempenho energético dos softwares, salientando através de evidências quantitativas o impacto no consumo de energia do sistema computacional. Os resultados obtidos com os experimentos aqui realizados atestam um relacionamento entre as decisões diversas que permitem codificar softwares logicamente equivalentes e os comportamentos referentes à eficiência energética.

Palavras-chave: Sustentabilidade. Computação Verde. Necessidades Ecológicas. Software.

ABSTRACT

The theme of sustainability in computing has been gaining more and more prominence, in view of the growing concern with the future of the next generations, which has raised awareness of the need to mitigate the negative impacts caused by man to the environment. Different studies have been conducted in the last few years approaching the so-called Green Computing, highlighting the importance of using computers in a conscious way and aligned with ecological needs. This work aimed to conduct a study on Green Computing based on the experimental methodology about the energy performance of software, highlighting through quantitative evidence the impact on energy consumption of the computer system. The results obtained with the experiments carried out here attest to a relationship between the various decisions that allow the coding of logically equivalent software and the behavior regarding energy efficiency.

Keywords: Sustainability. Green Computing. Ecological Needs. Software.

LISTA DE FIGURAS

LISTA DE CÓDIGOS-FONTE

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
CPU	Central Processing Unit
CSV	Comma Separated Values
GPU	Graphics Processing Unit
HPC	Computação de Alto Desempenho
IA	Inteligência Artificial
JAR	Java Archive
JVM	Java Virtual Machine
ONU	Organização das Nações Unidas
PHP	PHP Hypertext Preprocessor
PID	Process Identifier
RAPL	Running Average Power Limit
SBC	Sociedade Brasileira de Computação
TI	Tecnologia da Informação
UML	Unified Modeling Language

SUMÁRIO

1 INTRODUÇÃO

Em abril de 1987, a Organização das Nações Unidas (ONU) conduziu uma comissão especial para pôr em perspectiva a relação existente entre o meio ambiente e o desenvolvimento das sociedades humanas, sob diferentes aspectos. No relatório derivado desta comissão, intitulado de "Nosso Futuro Comum", a médica norueguesa Gro Harlem Brudtland define o desenvolvimento sustentável como sendo o desenvolvimento que atende as necessidades atuais sem comprometer a possibilidade das futuras gerações de atenderem às suas próprias necessidades (BRUDTLAND, 1991). As recomendações feitas pela comissão colocaram o assunto em destaque na agenda pública mundial, amadurecendo o olhar da humanidade sobre as questões ambientais.

Mais recentemente, no ano de 2018, a Sociedade Brasileira de Computação (SBC) realizou a 38ª edição de seu congresso anual, cujo assunto principal foi a relação existente entre a computação e a sustentabilidade, objetivando compartilhar e gerar conhecimento sobre como novas tecnologias podem criar e manter condições para o progresso da humanidade em harmonia com o meio ambiente. A revista publicada neste congresso coloca que temas como o controle da emissão de gás carbônico na atmosfera, o consumo sustentável de recursos naturais e de eletricidade possuem bastante relevância na atualidade, fazendo parte de debates envolvendo ativistas, grupos ambientais e governamentais, pesquisadores, membros da indústria e sociedade como um todo (ALMEIDA et al., 2018). O mesmo texto ainda introduz que, devido ao avanço da capacidade computacional e o seu impacto na sociedade, essa discussão também começou a fazer parte da área de ciência da computação, estabelecendo a Computação Verde como um recente tema correlato que visa incentivar a utilização da Tecnologia da Informação (TI) com uma preocupação com o meio ambiente (ALMEIDA et al., 2018).

A computação moderna se alicerça em atividades desempenhadas de forma conjunta por elementos de hardware e de software, sendo que as atividades de hardware estão diretamente relacionadas ao consumo de eletricidade, uma vez que de forma geral estes componentes possuem certas especificações de tensão e de corrente elétrica para funcionarem adequadamente. As operações desempenhadas pelo computador são direcionadas pelos softwares, que se forem

mal implementados, podem eliminar quaisquer características sustentáveis incorporadas pelo hardware.

É natural que o foco de estudos sobre a melhoria da eficiência energética em termos de hardwares tenha sido alvo de maior interesse ao longo do tempo, em detrimento de áreas como a eficiência energética de software. Existem várias soluções orientadas a hardware, mas o envolvimento do Software Verde ainda não está bem amadurecido, já que o software tem efeito indireto sobre o meio ambiente, mesmo sendo aquele que gerencia e opera o hardware subjacente (SALAM; KHAN, 2015). Sendo assim, o software é corresponsável pelo consumo de mais ou menos energia por um sistema computacional. Por isso, é razoável supor que a adoção de determinadas técnicas e práticas na construção de softwares conduzem a melhores resultados que outras no tocante ao consumo energético e uma descrição da arquitetura do produto de software, complementada com medições de consumo de energia, pode ajudar a direcionar os esforços da Computação Verde, contribuindo com a utilização de algoritmos energeticamente eficientes.

Este trabalho, portanto, teve por objetivo apresentar a relação existente entre a execução de softwares e o consumo de energia do sistema computacional, ressaltando possíveis abordagens iterativas para o desenvolvimento de software em contribuição para a Computação Verde. Desejou-se demonstrar através de evidências quantitativas o impacto positivo para o meio ambiente propiciado pela economia na alimentação elétrica da computação que é habilitada pela adoção de práticas de programação que permitem construir softwares logicamente equivalentes, porém mais eficientes no tocante ao uso de eletricidade por parte dos sistemas computacionais.

Em concordância com tal objetivo, inicialmente buscou-se estabelecer um planejamento para os experimentos, delimitando as características a serem observadas nos programas. Depois foi realizada a implementação dos programas utilizados nos experimentos, fazendo uso de diferentes abordagens iterativas de codificação. Após isso, utilizando-se de softwares para a avaliação de consumo de energia para obter medições em tempo de execução, os experimentos foram executados repetidamente. De posse destas medições, foi feita a avaliação do benefício da adoção de determinadas abordagens de codificação.

Além desta introdução, este material está organizado da seguinte maneira: o capítulo 2 subsequente apresenta uma contextualização mais detalhada sobre a área de Computação Verde, o capítulo 3 descreve melhor o trabalho, esclarecendo a metodologia, os capítulo 4 apresenta os experimentos, os testes e os resultados obtidos. Por fim, o capítulo 5 discorre acerca das

considerações finais sobre os resultados obtidos e sobre as perspectivas para trabalhos futuros.

2 SUSTENTABILIDADE E A COMPUTAÇÃO

Até o início da década de 1970, o meio ambiente era visto como uma fonte inesgotável de recursos e de matérias primas a qual o ser humano acreditava poder usufruir sem nenhum critério de preservação (IGNACIO, 2020). Contudo, os progressivos impactos negativos causados pela exploração desenfreada do homem ao ecossistema já naquela época começava a trazer à tona as severas consequências que a própria humanidade estaria fadada a carregar no longo prazo. Rios poluídos, florestas sendo destruídas pela chuva ácida, poluição atmosférica nas grandes cidades, secas em lagos e rios começavam a colocar em questão a visão de que os recursos naturais seriam ilimitados e que o homem poderia sempre usufruir deles de forma descuidada. Diante deste cenário, foi em 1972 que aconteceu a primeira conferência internacional na qual diversos países do globo se reuniram para discutir as questões ambientais.

Promovida pela ONU, a Conferência de Estocolmo de 1972 contou com a presença de 113 países, diversas organizações internacionais, ONGs, observadores e jornalistas inclinados a identificar, a proteger, a conservar, a valorizar e a transmitir para as gerações futuras a importância do patrimônio cultural e natural (IGNACIO, 2020). Embora nenhum acordo concreto tenha sido concluído em Estocolmo, ficou compreendido a partir daquele momento que a exploração exacerbada e livre de juízo dos recursos naturais pela humanidade acabariam ocasionando um esgotamento ambiental. O relatório derivado da Conferência de Estocolmo, intitulado de "Declaração sobre o Meio Ambiente Humano", apresentou ao mundo os princípios que norteiam a perspectiva ecológica para o desenvolvimento humano, defendendo que os recursos naturais necessitam de gestão adequada para não serem esgotados, uma vez que estes precisam estar disponíveis para as gerações futuras (BEZERRA, 2020).

Outro momento de grande relevância histórica ocorreu em 1987, quando a ONU promoveu a Comissão Mundial sobre o Meio Ambiente, na qual foram discutidas questões sobre o desenvolvimento econômico frente a responsabilidade ecológica. Esta comissão também ficou conhecida como "Comissão Brudtland"(ou ainda "Relatório Brudtland") (GUITARRARA, 2022), pois foi derivado desta comissão o documento intitulado de "Nosso Futuro Comum"(BRUDTLAND, 1991) que trouxe diversas considerações acerca do desenvolvimento

sustentável. Nele a ex-primeira ministra e médica norueguesa Gro Harlem Brudtland apresenta ao mundo a sua visão de desenvolvimento sustentável como sendo aquele em que sociedade contemporânea atende às suas necessidades sem comprometer o futuro das próximas gerações. Estabeleceu-se, assim, a definição de desenvolvimento sustentável (GUITARRARA, 2022).

Realizada em 1992, na cidade do Rio de Janeiro no Brasil, a conferência Eco-92 colocou o assunto ambiental na agenda pública de uma maneira inovadora, tendo sido um importante avanço da humanidade no tocante à sua relação com o planeta. Duas décadas após a primeira conferência em Estocolmo, a Eco-92 contou com a representação de 179 países, além da efetiva participação das ONGs, dos movimentos sociais e de representantes da sociedade civil no Fórum Global (IGNACIO, 2020). A Eco-92 é considerada uma das principais conferências ambientais realizadas em nível global, em razão da sua grande repercussão e formalização de documentos importantes como a Agenda-21, nos quais foram estabelecidas políticas e ações de responsabilidade ambiental (CAMPOS, 2022).

2.1 A COMPUTAÇÃO VERDE

A história da computação está repleta de avanços movidos pelo desejo de criar dispositivos que possuíssem, além da superior capacidade de processamento, dimensões menores que os existentes, fossem menos dispendiosos do ponto de vista energético e menos emitentes de calor. Houve uma época em que os circuitos eram conhecidos como tubos de vácuo, e a memória era conhecida como tambores magnéticos e os computadores eram enormes, usavam grande quantidade de eletricidade e liberavam um calor insuportável até mesmo para a própria máquina, causando maus funcionamentos (SHAH; SOOMRO, 2015). Em seguida, com a evolução da tecnologia e a criação dos interruptores eletrônicos, os computadores passaram a ser baseados em transistores e diodos. Essa segunda geração dos computadores inovou sendo mais rápida, barata, menor e mais eficiente, fazendo com que os computadores também se tornassem mais confiáveis. Contudo, a fabricação dos computadores, desde já, estava bastante ligada à manipulação de elementos químicos nocivos para a saúde humana e a do meio ambiente.

Não diferente das preocupações gerais acerca da conservação do meio ambiente diante da necessidade de progresso das sociedades humanas, na ciência da computação também emergiram temáticas de pesquisas visando dar apoio a gestão adequada dos recursos naturais através da busca por formas cada vez melhores de se fazer uso dos computadores na era da

informação (SAHA, 2014). Houve, portanto, um expressivo aumento das preocupações e das inquietações relacionadas à temática da sustentabilidade na área da computação, que ficou conhecida como a Computação Verde ou TI verde.

Observa-se que a princípio uma significativa preocupação com a gestão dos nocivos resíduos gerados pela cadeia de fabricação dos hardwares, assim como a obsolescência em ritmo acelerado que é vista na área de tecnologia e que está ligada ao tratamento indevido de lixo eletrônico. Os contínuos avanços tecnológicos levaram a um acúmulo de muito lixo eletrônico (PUTRI et al., 2015). Conforme os autores colocam, os dispositivos antigos contêm materiais tóxicos que são potencialmente nocivos e quando eles são descartados a maioria destes dispositivos acabam em aterros sanitários ou são exportados para países em desenvolvimento sob o pretexto de reciclagem. Eles afirmam ainda que o fluxo de lixo eletrônico para a Indonésia tem diminuído a idade dos aterros sanitários e causado problemas de saúde entre os catadores de lixo.

Surgiram também preocupações relacionadas ao crescente aumento dos custos com a sustentação de centros de processamento de dados, tanto dentro das empresas quanto em provedores de nuvem, no tocante ao consumo da energia demandada pela maior necessidade por comunicação de sistemas distribuídos e serviços de TI. Diminuir o uso de energia dos centros de dados é uma tarefa desafiadora e complexa, mas há no mundo moderno uma necessidade de modelos de computação em nuvem verde para controlar remotamente os centros de dados e de servidores, tornando-os mais eficientes em termos energéticos e economicamente confiáveis (AGRAWAL et al., 2020). Segundo dados trazidos pelos autores, 17% do total da emissão de carbono causada pela tecnologia é devida a centros de dados e a eletricidade necessária para operar esses centros de dados é de quase 30 bilhões de watts. Estes servidores desperdiçam 90% da energia que utilizam porque funcionam em plena capacidade durante todo o dia.

Outra perspectiva relevante dentro da temática de dispêndio energético da computação é a sustentabilidade em termos de software - ou Softwares Verdes -, que começou a ganhar relevância entre a comunidade científica da computação, demonstrando um amplo espaço para novas contribuições apresentando o relacionamento que existe entre o trabalho de desenvolvimento de softwares com as questões ambientais.

2.2 SOFTWARES VERDES

Apesar da energia ser consumida diretamente pelo hardware, as operações realizadas neste são direcionadas pelo software e podem eliminar quaisquer características sustentáveis incorporadas ao hardware, sugerindo que o software é o principal responsável pelo o consumo de energia do sistema (JAGROEP et al., 2016). Contudo, mesmo a temática da eficiência energética de softwares pode ainda ser analisada sob ênfases distintas.

2.2.1 O impacto da inteligência artificial

No campo da Inteligência Artificial (IA), a razão para o seu sucesso se deve principalmente à convergência entre a Computação de Alto Desempenho (HPC) e o uso de algoritmos de Machine Learning. Além disso, estudos já apontam que os esforços para enfrentar as mudanças climáticas e as questões ambientais poderiam ter um impacto negativo devido às necessidades de alta energia para as aplicações de IA, especialmente se forem usadas fontes não neutras de carbono (SILVA, 2021a). Os algoritmos de IA são caros de se treinar e de se desenvolver, tanto em capacidade computacional, quanto para o meio ambiente, devido ao acentuado consumo de energia (que demanda emissão de carbono) requerida pelos ambientes computacionais. A maioria das publicações relacionadas a Machine Learning não relatam regularmente as métricas de energia ou de emissão de carbono (HENDERSON et al., 2020). Para eles, a razão pela qual muitas pesquisas não relatam estas métricas se deve à complexidade da coleta de carbono. A coleta de métricas de emissão de carbono requer a compreensão das emissões das redes de energia, o registro das informações de energia fornecidas pela *Graphics Processing Unit (GPU)* e pela *Central Processing Unit (CPU)*, e a navegação entre diferentes ferramentas para realizar estas tarefas.

2.2.2 O impacto dos jogos de computador

Outra importante ênfase relacionada aos softwares dentro da Computação Verde é a dos impactos ambientais no que tange o consumo de recursos de eletricidade e de emissão de carbono relacionados à complexidade de jogos de computador. Como forma de entretenimento, as pessoas simplesmente gostam de jogar em seus computadores e isso os coloca em evidência tanto quanto qualquer outro assunto relacionado ao estudo de Softwares Verdes.

A dinâmica do jogo de computador gira em torno de cálculos matemáticos e de

renderização de quadros (AHMED; SHARMA, 2014). O jogo é controlado por um laço de repetição, que exibe o próximo quadro na ocorrência de um evento. A taxa de renderização dos quadros varia em cada nível do jogo e depende de muitos fatores como topologia da tela, movimentação física dos personagens, complexidade, regras da ciência física, etc. Como a complexidade de cada nível sucessivo é maior que a do nível anterior, a taxa de geração de quadros também varia em cada nível sucessivo.

Uma regra básica no projeto de jogos de computador é que os usuários preferem taxas altas de geração de quadros e por isso a maioria das aplicações de jogos são projetadas para maximizar as taxas de quadros sem nenhuma consideração em relação ao uso de recursos ou consumo de energia. Foi feita uma investigação empírica no intuito de descobrir se um aumento sucessivo no nível da complexidade de um jogo de computador aumenta ou não o consumo de energia, sendo constatado uma mudança máxima de 22%, e média mínima de 10%, no consumo de energia em qualquer nível sucessivo em todos os jogos observados (AHMED; SHARMA, 2014).

2.2.3 Projetos de softwares verdes

Outros trabalhos focam nos projetos de softwares, com intuito de ajudar todas as partes interessadas na construção dos produtos de Software Verde. Há interesses que se concentram no sistema operacional, ajudando a controlar o consumo de energia das aplicações, assim como preocupações levantadas acerca da importância de se ter a disposição ferramentas para facilitar a mensuração dos efeitos do software sobre o meio ambiente e o efeito dos ambientes de desenvolvimento sobre o software em termos de eficiência energética. Para resolver as questões relacionadas ao desenvolvimento de software sustentável de acordo com as necessidades atuais dos clientes, há uma necessidade veemente de integrar os princípios da engenharia verde com métodos ágeis para produzir softwares sustentáveis (RASHID; KHAN, 2016).

Um aplicativo mal codificado pode ser ineficiente e realizar inúmeras operações desnecessárias e dispendiosas, mas os desenvolvedores também enfrentam certas dificuldades para determinarem se seu aplicativo está sendo codificado de forma ineficiente no que diz respeito ao consumo de energia (LI et al., 2013). Diante disso, o trabalho de Li e colegas propõe um artefato técnico para sobrepor visualmente as informações de consumo de energia

sob código fonte de um software (PINTO; CASTOR, 2017). Quando instalado no ambiente de desenvolvimento, este artefato colore as linhas do código fonte com as diferentes quantidades de energia consumidas. As linhas azuis descrevem um baixo consumo de energia, enquanto que as linhas vermelhas indicam alto consumo de energia. Esta técnica de visualização é de granulação fina e funciona no nível do código fonte.

Alguns estudos se centraram na identificação de fatores de riscos críticos ligados ao desenvolvimento de Softwares Verdes, bem como riscos ligados a projetos *multi-sourcing*. O *multi-sourcing* pode ser definido como um paradigma moderno de terceirização, que se utiliza de múltiplos fornecedores para o desenvolvimento de software em um período de tempo mais curto (SALAM; KHAN, 2017). O *multi-sourcing* também é um paradigma a nível global para o desenvolvimento de software, objetivando alta qualidade a um custo e tempo mínimos. Independentemente da importância do desenvolvimento de Software Verde, poucas pesquisas empíricas foram conduzidas sobre a identificação e gestão de fatores de risco no desenvolvimento de Software Verde e sustentável neste âmbito (SALAM; KHAN, 2017).

2.2.4 Codificação de softwares verdes

As boas práticas orientadas ao software propiciam a utilização dos recursos computacionais de forma eficiente, atenuando a necessidade do provisionamento de recursos em ambientes de produção juntamente com as atualizações de softwares. Dessa forma, uma maneira adequada de evitar a escalada exagerada de hardware, majorando o consumo de eletricidade, é com o melhor uso de recursos computacionais e a otimização energética também no nível dos códigos executáveis que irão efetivamente entrar em operação nos ambientes de produção.

O mapeamento sistemático construído por Welter e colegas visou apresentar quais temáticas estão sendo realizadas nas pesquisas sobre métricas verdes em TI, direcionando para a engenharia e organização de softwares (WELTER et al., 2014). Ainda que uma quantidade considerável de métricas específicas para data centers ou service centers tenham sido observadas, os autores constataram que a preocupação direcionada ao consumo de energia elétrica e sua devida mensuração é quase unânime nas linhas de pesquisa. Em 93,88% das pesquisas selecionadas por eles foram propostas ou utilizadas métricas sobre energia elétrica. Já as métricas para cálculo da emissão de carbono (também denominado de gás do efeito estufa) aparecem em 42,86% dos estudos e mensurações de uso ou performance em 36,73% dos estudos.

Aspectos verdes também podem ser introduzidos durante o processo de desenvolvimento de software, assim como na análise e projeto do software. Diversas investigações têm sido feitas sobre os benefícios e as dificuldades de se produzir programas de computador mais alinhados com as expectativas verdes e sobre quais abordagens são cabíveis. Uma das formas que a TI poderia ajudar a preservar o meio ambiente com a redução da emissão de carbono é através da produção de softwares energeticamente mais eficientes (OLAOLUWA et al., 2015). Diante da necessidade de investigação científica sobre a medição da energia realmente consumida ao se executar uma aplicação, o Joulemeter foi a ferramenta de medição selecionada porque oferecia a capacidade de medir o uso de energia de softwares residentes em um computador e outros sistemas de TI e por isso foi utilizado para estudar o consumo de energia ao executar um código escrito na linguagem *PHP Hypertext Preprocessor (PHP)* (OLAOLUWA et al., 2015). O painel de controle da ferramenta Joulemeter pôde ser usado naquele contexto para visualizar o consumo de energia do computador e também acompanhar o uso de energia de aplicações específicas.

No contexto da construção de Softwares Verdes, os compiladores possuem um importante papel, pois também são responsáveis pela geração de códigos executáveis mais eficientes e podem ser uma boa fonte de otimização energética a nível de software. Logo, o compilador verde conservador de energia é aquele que efetivamente gera os códigos executáveis otimizados para conservação de energia, fazendo o trade-off entre a conservação de energia e o desempenho (CHANDA et al., 2017). Estes compiladores utilizam técnicas de compilação verde para gerar em tempo de compilação os códigos executáveis conservadores de energia, transformando e reformulando códigos binários pela aplicação de estratégias verdes. A conservação de energia e o desempenho são objetivos conflitantes e o tempo de compilação de um programa é aumentado quando são empregadas estratégias verdes, contudo o compilador verde apresentado foi capaz de conservar os ciclos de relógio em *CPU* em cerca de 30% a 40%.

3 DESCRIÇÃO DO PROJETO

Partindo do princípio de que é possível melhorar a eficiência energética geral de um sistema computacional através da melhoria da eficiência individual de cada software em execução nele, foi levantada a possibilidade de que a utilização de diferentes estruturas lógicas para a implementação de softwares também conduzem a resultados diferentes no tocante ao consumo energético. Ao avaliar a eficiência em termos de tempo de processamento e de consumo de memória é bastante oportuno que sejam realizados experimentos práticos que evidenciam o comportamento da implementação de determinados algoritmos em ambientes diversos de execução. Semelhantemente neste trabalho foi almejado a obtenção resultados concretos e comprobatórios acerca do consumo de eletricidade em termos de software, servindo de alicerce para elaboração de conclusões satisfatórias.

Como apresentado nas seções anteriores, este estudo sobre os Softwares Verdes aproxima-se das preocupações crescentes sobre métodos eficientes para calcular de forma precisa o consumo energético dos softwares, pois são eles corresponsáveis pelo consumo geral de eletricidade por parte dos computadores modernos. Entretanto, a carência de ferramentas e conhecimentos para entender o consumo energético do software e escrever códigos energeticamente mais eficientes é um desafio presente (NOUREDDINE, 2022).

Dentre as abordagens usuais para a mensuração de energia, salientam-se duas principais classificações: as baseadas em inspeção direta das grandezas físicas e as baseadas na utilização de *Application Programming Interface (API)* de hardwares para a coleta de dados. A primeira, além de necessitar de instrumentação específica para medição, oferece complexidade consideravelmente maior para a avaliação individual de softwares em execução e por isso está fora do foco deste trabalho.

Por outro lado, a segunda classificação diz respeito ao uso de *API*, bibliotecas e ferramentas de software para acessar, através de chamadas aos recursos do sistema, as informações de consumo energético coletadas pelos hardwares. Neste contexto, a maioria das ferramentas disponíveis utilizam a interface *Running Average Power Limit (RAPL)* da Intel, seja diretamente dos registros ou através da implementação do kernel Linux usando a interface powercap

(NOUREDDINE, 2022).

3.1 MENSURAÇÃO DE CONSUMO DE ENERGIA

Em virtude da falta de disposição de equipamento composto por um processador da fabricante Intel, no contexto deste trabalho foi utilizada a interface *RAPL* através da sua implementação do kernel Linux, conhecida como Power Capping Framework ou apenas interface powercap. A interface powercap é uma consistente interface entre o kernel e o espaço do usuário que permite que os drivers de limitação de energia exponham as configurações ao espaço do usuário de maneira uniforme.

A interface powercap do Linux expõe os dispositivos de limitação de energia ao espaço do usuário na forma de uma árvore de objetos. Os objetos no nível da raiz da árvore representam diferentes métodos de limitação de energia (KERNEL. . . , 2022). Assim, através desta interface, é possível coletar da árvore de objetos as informações necessárias para a mensuração do consumo de energia do sistema computacional.

3.2 MONITORAMENTO COM O POWERJOULAR E JOULARJX

O PowerJoular é uma proposta de ferramenta capaz de monitorar e fornecer informações sobre o consumo de energia da *CPU* e *GPU* de computadores como um todo ou de um processo específico em execução (NOUREDDINE, 2022). Seu objetivo é fornecer uma ferramenta de monitoramento de energia atualizada, fácil de usar e compatível com múltiplas plataformas, como servidores x86/x64 e dispositivos de placa única ARM, para possibilitar a análise do consumo de energia do software.

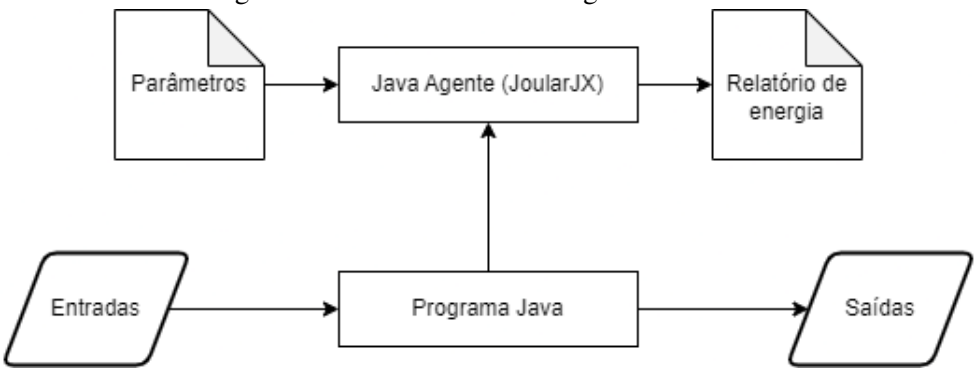
A fim de fornecer uma ferramenta de baixo impacto, o PowerJoular foi escrito em Ada, uma vez que tal linguagem está constantemente classificada entre as linguagens de programação mais eficientes em termos energéticos (NOUREDDINE, 2022). O PowerJoular detecta a configuração do computador e dos módulos suportados, e fornece dados de potência em conformidade. Para a *GPU*, o PowerJoular verifica se a NVIDIA SMI está instalada e a utiliza para verificar se o monitor de energia da *GPU* suporta a leitura do consumo de energia da GPU, a cada segundo. No caso da *CPU*, o PowerJoular utiliza os dados de potência da Intel *RAPL*, através da interface powercap no Linux, lendo os objetos de sistema apropriados. Finalmente,

as leituras de potência de todos os componentes suportados são agregadas para fornecer um consumo geral de energia do sistema (NOUREDDINE, 2022).

O PowerJoular pode monitorar o consumo de energia da *CPU* de um processo individual, fornecendo seu *Process Identifier (PID)* em tempo de execução. Se um *PID* for monitorado, seus dados de energia serão exibidos no terminal e serão armazenados em um arquivo do tipo *Comma Separated Values (CSV)* distinto, enquanto a energia do dispositivo é salva independentemente. Ao final de cada sessão de monitoramento, PowerJoular exibe o consumo total de energia em Joules da sessão e do *PID* monitorado (NOUREDDINE, 2022).

Para trabalhar em conjunto com a ferramenta PowerJoular a mesma literatura propõe o JoularJX, um componente de software capaz de se integrar ao PowerJoular e monitorar a execução de aplicações sob a *Java Virtual Machine (JVM)* (NOUREDDINE, 2022). O agente Java inicia automaticamente o PowerJoular com os parâmetros apropriados, isto é, monitorando o programa Java através do seu *PID* e gerando ao final da sua execução um relatório com os dados de energia em um arquivo *CSV*. O diagrama da figura 1 seguinte resume esquematicamente este processo.

Figura 1 – Funcionamento do Agente JoularJX.



Fonte: Autoria Própria.

A utilização da *CPU* é monitorada a cada segundo, e para cada thread do programa Java, e o consumo de energia é alocado de acordo. Um segundo laço de repetição de monitoramento detecta a cada 10 milissegundos, para cada thread, qual método está sendo executado atualmente observando o primeiro método na sua pilha de execução, e então o consumo de energia em Joules é alocado estatisticamente para cada método (NOUREDDINE, 2022). Conforme demonstrado no diagrama da figura 1 anterior, também é possível fornecer parâmetros para o agente JoularJX, a fim de fazê-lo monitorar um pacote, ou uma classe, ou até mesmo um método

específico do programa Java.

Assim, diante das características destacadas das ferramentas, foi considerado pertinente para a implementação dos experimentos idealizados neste trabalho fazer o uso do PowerJoular/JoularJX e da linguagem de programação Java, nas versões 1.0 e 17 respectivamente. Para coletar, mensurar e analisar o consumo de energia dos algoritmos, eles foram implementados de acordo com as especificações da linguagem Java e compilados no intuito de se produzir um artefato executável *Java Archive (JAR)* capaz de ser monitorado pelo agente JoularJX.

3.3 ESTRUTURAS DE PROGRAMAÇÃO

Com a seleção das linguagens Java para a implementação dos programas de teste, visto a disponibilidade das ferramentas mencionadas para a medição de consumo de energia, faz-se necessário ressaltar as principais estruturas de dados presentes na linguagem, bem como suas respectivas estratégias para iteração, que são comumente utilizadas ao representar os algoritmos. Em concordância com o objetivo deste trabalho, primeiro será apresentada as diferentes possibilidades de se representar e processar os dados, pois este conhecimento permite que diferentes maneiras de se construir programas sejam posteriormente exploradas. Por fim, vale reforçar também que, apesar de não ser uma abordagem inerente a uma linguagem específica de programação, em muitas ocasiões existe a possibilidade de se utilizar funções recursivas para trabalhar as diferentes estruturas, mas esta abordagem não será utilizada nos experimentos idealizados para este trabalho.

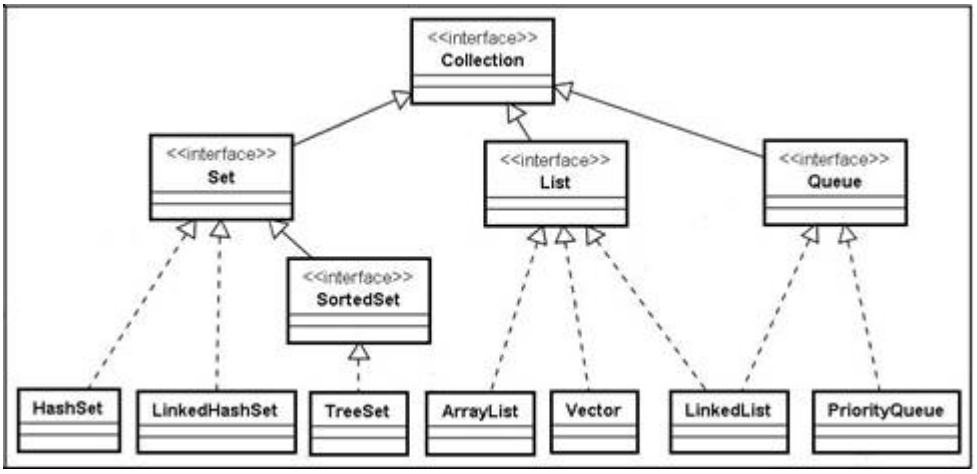
3.3.1 Estruturas da linguagem Java

O Java é uma linguagem bastante ligada aos conceitos da orientação a objetos, o que significa que seus tipos de dados são representados por classes que obedecem contratos bem definidos por interfaces. Na linguagem é encontrada uma ampla variedade de tipos e subtipos de dados que podem acomodar diferentes necessidades técnicas para a elaboração de programas, mas o objetivo deste texto não é detalhar exaustivamente cada um destes tipos e necessidades. Aqui, por outro lado, se busca salientar as principais estruturas encontradas no Java, observando suas características e diferenças.

Para entender as estruturas presentes na linguagem Java é necessário entender a organização do Collections Framework. O Java possui estruturas para representar arranjos de

valores e outras para arranjos de chave e valor. O Java possui uma cadeia de herança de interfaces e classes para representar estes tipos de dados. Esta cadeia de herança é o Collections Framework propriamente dito e o uso de qualquer um dos seus componentes implica na necessidade de importação do pacote correspondente. Dentre os tipos que representam arranjos de valores, na figura 2 seguinte é encontrado um diagrama *Unified Modeling Language (UML)* da hierarquia de componentes.

Figura 2 – Heranças da interface Collection.



Fonte: Retirada de (JAVA. . . , 2010)

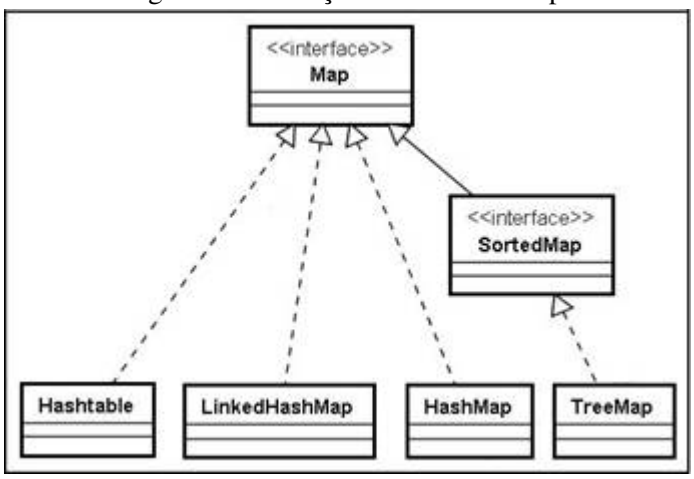
A interface Collection está no topo da hierarquia e, apesar de não existir implementação direta dessa interface, ela define as operações básicas para as coleções, como adicionar, remover, esvaziar, etc (JAVA. . . , 2010). Imediatamente abaixo da interface Collection são encontradas três interfaces derivadas que representam tipos de dados semelhantes, mas ligeiramente diferentes: as listas (List), os conjuntos (Set) e as filas (Queue). As listas em Java consistem de estruturas não ordenadas e sem tamanho máximo previamente definido, capazes de acomodar elementos duplicados. Os conjuntos são estruturas bastante semelhantes as listas, mas não podem acomodar elementos duplicados (JAVA. . . , 2010).

As filas em Java, por outro lado, são estruturas organizadas em função de um critério que regulamenta a entrada e a saída dos seus elementos, estabelecendo por exemplo que o primeiro elemento a entrar será o primeiro a sair dela (SILVA, 2021b). Na parte inferior do mesmo diagrama 2 são encontradas algumas das diferentes classes que correspondem às implementações das listas, dos conjuntos e das filas.

A hierarquia da Collections Framework tem uma segunda árvore para descrever classes e interfaces relacionadas a mapas, que não são derivadas de Collection (JAVA. . . , 2010).

Mapa é a nomenclatura usual da linguagem para descrever estruturas de chave e valor. Os mapas permitem que existam valores duplicados, desde que as chaves sejam únicas. A hierarquia de mapas do Java pode ser observada no diagrama *UML* da figura 3 seguinte, na qual as implementações em classes também estão posicionadas na parte inferior.

Figura 3 – Heranças da interface Map.



Fonte: Retirada de (JAVA..., 2010)

No Java são encontradas as tradicionais instruções de repetição: While, Do-While e For. O código 1 seguinte demonstra a sintaxe de declaração destas estruturas, ilustrando também entre as linhas 15 e 17 uma segunda declaração de instrução de For, denominada de instrução For Aprimorado (For Enhanced) (DEITEL; DEITEL, 2017).

Código-fonte 1 – Laços de repição em Java

```
1   while (conditional) {
2       // Code...
3   }
4
5   do {
6       // Code...
7   } while(conditional);
8
9   // For Classic
10  for (start; conditional; step) {
11      // Code...
12  }
13
14  // For Enhanced
15  for (element : collection) {
16      // Code...
17  }
```

Conforme observado no código, o For Aprimorado (For Enhanced em inglês) é uma instrução de repetição de fato otimizada para facilitar o trabalho com os tipos de dados derivados de Collection que foram apresentados anteriormente. A instrução For Aprimorado itera pelos elementos de uma Collection sem necessitar do uso de um contador, entretanto ela só pode ser utilizada para obter os elementos desta estrutura, não podendo ser usada para modificar estes elementos (DEITEL; DEITEL, 2017).

A partir da versão 8, as Collections do Java passaram a contar com um ForEach que é capaz de realizar a leitura dos seus elementos (NOLETO, 2010). Isso decorre do fato da interface Collection ter passado a herdar da interface Iterable, que possui a implementação padrão do método. O código 2 seguinte ilustra o uso do método ForEach, da sua forma mais completa à mais simplificada.

Código-fonte 2 – O método ForEach do Java 8.

```
1  import java.util.function.Consumer;
2  import java.util.List;
3
4  public class MyClass {
5      public static void main(String args[]) {
6          List<Integer> list = List.of(1, 2, 3);
7
8          // ForEach full statement.
9          list.forEach(
10             new Consumer<Integer>() {
11                 @Override
12                 public void accept(Integer element) {
13                     System.out.println(element);
14                 }
15             });
16
17         // ForEach simplified statement.
18         list.forEach(element -> {
19                 System.out.println(element);
20             });
21
22         // ForEach totally simplified statement.
23         list.forEach(System.out::println);
24     }
25 }
```

Naturalmente que a implementação padrão do ForEach contém uma instrução For

Aprimorado, tal como observado no código 3 seguinte, mas seu contrato exige o fornecimento de um objeto derivado da implementação da interface Consumer. Também é possível que seja fornecido uma implementação própria para o ForEach, com configurações e preferências internas do sistema operacional, permitindo que o código tenha a possibilidade de ser otimizado (NOLETO, 2010). Assim, o uso do ForEach estaria mais ligado a uma boa prática para organização de código que a uma prática de otimização de desempenho.

Código-fonte 3 – O método forEach da interface Iterable.

```
1   public interface Iterable<T> {
2       Iterator<T> iterator();
3
4       default void forEach(Consumer<? super T> action) {
5           Objects.requireNonNull(action);
6           for (T t : this) {
7               action.accept(t);
8           }
9       }
10  }
```

A interface Iterable garante também a disponibilidade de um método "iterator", que quando chamado irá devolver um objeto de Iterator para a coleção em questão. O Iterator será do mesmo tipo T da Collection que o originou e oferecerá os métodos para facilitar a leitura de seus valores: hasNext, next. Enquanto o primeiro verifica se existe um elemento a ser lido da estrutura e retorna um valor booleano, o segundo método recupera o valor do elemento em questão e avança o cursor de controle. O código 4 seguinte ilustra a utilização do Iterator para percorrer a mesma lista vista no código 2 e escrever o mesmo resultado na saída padrão.

Código-fonte 4 – O método iterator da interface Iterable.

```
1   import java.util.Iterator;
2   import java.util.List;
3
4   public class MyClass {
5       public static void main(String args[]) {
6           List<Integer> list = List.of(1, 2, 3);
7
8           Iterator<Integer> iter = list.iterator();
9           while (iter.hasNext()) {
10              System.out.println(iter.next());
11          }
12      }
13  }
```

Finalmente, compete trazer outro relevante acréscimo que o Java teve em sua versão 8: os fluxos (Streams). Os fluxos são objetos das classes que implementam a interface Stream e permitem realizar tarefas sobre coleções de elementos, muitas vezes a partir de um array ou de um objeto Collection (DEITEL; DEITEL, 2017). Com fluxos é possível realizar as operações de filter, distinct, map, reduce e etc, que geram sob determinada regra uma nova Collection a partir dos valores de outra.

Fluxos são fáceis de paralelizar, permitindo que os programas se beneficiem de um melhor desempenho em sistemas multiprocessados (DEITEL; DEITEL, 2017) e por isso também podem influenciar o perfil energético do programa. A interface Collection possui duas implementações padrão de métodos que facilitam trabalhar com a estrutura através de fluxos: o "stream"e o "parallelStream". Conforme demonstrado no código 5 seguinte, a única diferença entre ambos consiste no fato do processamento em paralelo estar ativado no método "parallelStream"através do segundo parâmetro booleano verdadeiro. Com os fluxos também é possível iterar a estrutura Collection, fazendo uso de métodos como o ForEach.

Código-fonte 5 – Os métodos Stream da interface Collection.

```
1   public interface Collection<E> extends Iterable<E> {
2       default Stream<E> stream() {
3           return StreamSupport.stream(
4               spliterator(), false);
5       }
6
7       default Stream<E> parallelStream() {
8           return StreamSupport.stream(
9               spliterator(), true);
10      }
11  }
```

4 TESTES E RESULTADOS

Este capítulo tem o objetivo de descrever em detalhes os diversos experimentos realizados neste trabalho a fim de atestar o impacto energético da computação ao meio ambiente também a nível de software. O texto subsequente está organizado em seções que descrevem cada experimento realizado, nas quais é possível encontrar a especificação geral do teste, contendo a apresentação do propósito do programa, o fluxograma de seu algoritmo, a enumeração das características principais em análise e as diferentes estratégias de codificação. Na respectiva seção encontram-se também os resultados colhidos após executar repetidamente as estratégias, com a apresentação de gráficos comparativos. É desejado que aqui se faça claro ao leitor o quão próximas estão as características energéticas de um programa das decisões tomadas durante a sua codificação. Os experimentos, portanto, consistem de pequenas implementações de programas através de diferentes estratégias de codificação, isto é, utilizando-se das diferentes estruturas lógicas presentes na linguagem que foram salientadas no capítulo 3 e seus códigos fonte podem ser visualizados de forma completa nos apêndices deste material e também no repositório do GitHub[1].

Conforme descrito no capítulo 3 anterior, os experimentos conduzidos neste trabalho foram feitos em um notebook com o sistema operacional Linux, devido aos requisitos demandados pela ferramenta de mensuração de energia JoularJX. Além disso, tal máquina é composta por uma *CPU* AMD Ryzen™ 5 4600H, que é formada por 6 núcleos físicos e 12 threads. A frequência básica de operação deste processador é de 3GHz e ele possui dois níveis de memória cache de 3MB e 8MB, respectivamente. A máquina também é composta por 32GB de memória RAM, com dois módulos de 16GB do tipo DDR4, e com frequências de 3200MHz, operando em dual-channel. Para os testes, o notebook foi mantido na alimentação elétrica e com a configuração padrão de utilização de energia do Linux que equilibra o consumo energético e o desempenho máximo do processador. Por fim, durante as execuções, com exceção dos processos do sistema operacional, apenas os experimentos, as ferramentas de mensuração de energia e os ambientes de desenvolvimento foram mantidos em operação no computador.

Vale reforçar também que cuidados foram tomados na condução dos testes e na

[1] Disponível em: <https://github.com/mateuspena/uneb.tcc-experimentos-java>. Acesso em: 17 dez, 2022.

apresentação dos resultados no intuito de se evitar que dados distorcidos prejudiquem a reflexão posterior. Primordialmente foi estabelecido que os programas idealizados para os experimentos deveriam ser capazes de exigir esforço computacional de tal maneira que suas execuções tivessem a duração de pelo menos alguns segundos, pois isso ajudaria a evitar que a ferramenta de mensuração de energia capturasse grandezas físicas demasiadamente pequenas ou ínfimas. Da mesma forma foi cuidado para que fossem minimizadas as diferenças entre as estratégias de codificação, evitando que uma estratégia tivesse excessiva vantagem sobre outra no tocante à alocação de estruturas de dados em memória, uma vez que isso possivelmente iria favorecer seu perfil energético.

Determinou-se também que cada estratégia deveria ser executada repetidamente em processos diferentes, desassociando-se do código do programa a repetição dos testes, para minimizar a influência do contexto de execuções passadas nas subsequentes. Além disso, para obter um conjunto de dados satisfatoriamente representativo, estabeleceu-se que cada estratégia deveria ser executada no mínimo 100 vezes, uma vez que os resultados nestas 100 repetições se mantiveram razoavelmente uniformes e Em razão do limitado espaço de tempo a disposição para executar todos os testes, coletar e consolidar os dados, obtendo ainda um conjunto de amostragem representativo, não se julgou necessário uma quantidade maior de repetições.

4.1 CONVERSOR DE IMAGENS

O propósito do conversor de imagens é receber uma imagem colorida e produzir outra equivalente em escala cinza, preservando a sua resolução. Em vista da popularidade desta funcionalidade, diversos são os exemplos de soluções de softwares comerciais ou de código aberto que implementam esta funcionalidade. Muitas bibliotecas de código fonte para processamento de imagens também possuem rotinas prontas para converter os pixels de uma imagem para a escala cinza, contudo, neste trabalho não é desejado fazer o uso de bibliotecas de terceiros, salvo em casos de acentuada necessidade. Dessa forma, a operação de conversão dos pixels fez parte da implementação do experimento.

Aqui não é desejado detalhar exaustivamente a teoria sobre o processamento de imagens, uma vez que este não é o foco do trabalho, mas faz sentido elucidar em linhas gerais como se dá o processo de conversão em questão. A conversão de um pixel colorido em seu equivalente em escala cinza consiste caracteristicamente da aplicação de uma função matemática.

A imagem é representada como uma matriz de pixels com duas dimensões: a altura e a largura. Cada pixel é obtido de uma coordenada (X, Y) da matriz, onde X é um valor entre 0 e a largura da imagem e Y um valor entre 0 e a altura da imagem.

Os pixels da imagem são decompostos em uma tuplas RGB, isto é, três valores de números inteiros entre 0 e 255 que representam respectivamente uma nuance das cores vermelha, verde e azul. Cada tupla então é usada como entrada para a função matemática de conversão supracitada, que aplica a seguinte fórmula matemática:

$$Y = 0.2989R + 0.5870G + 0.1140B \tag{4.1}$$

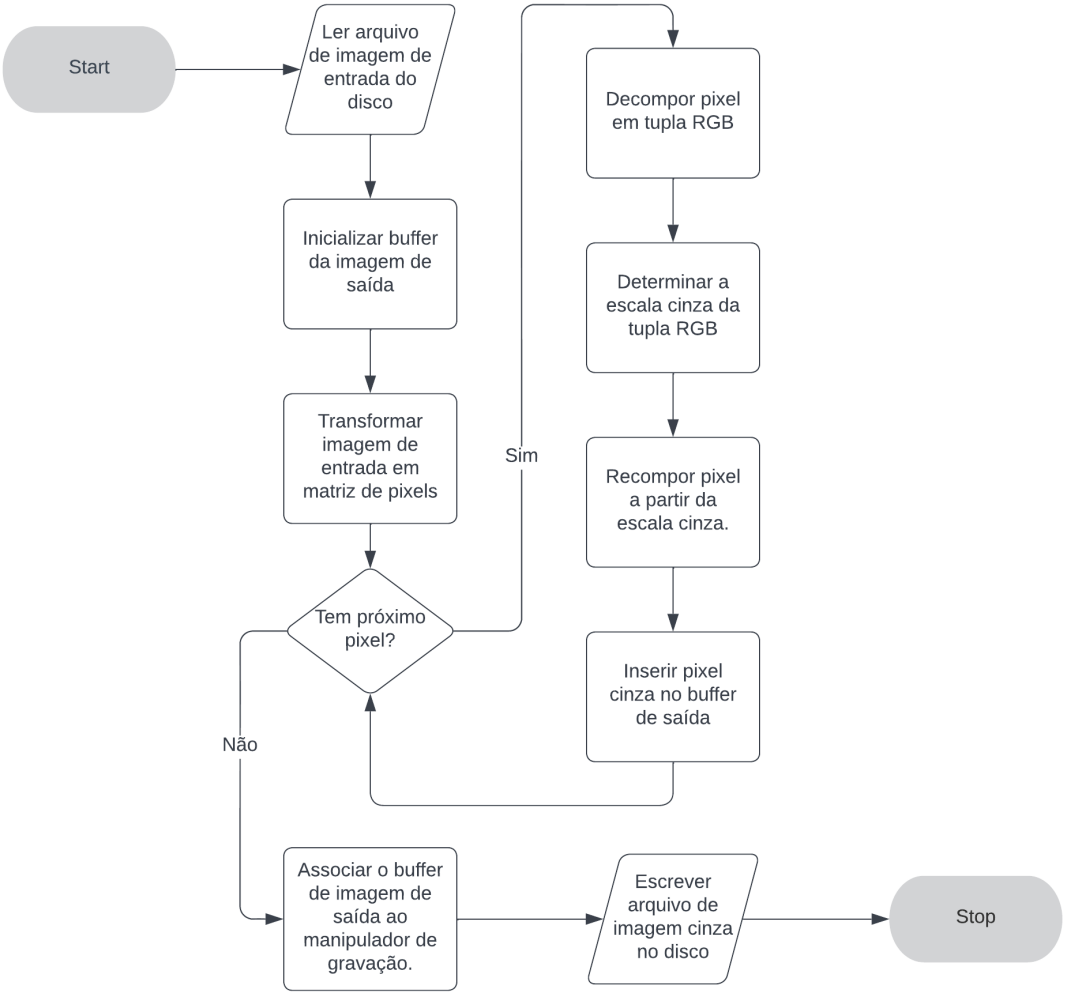
O retorno Y da função matemática 4.1, que é um número decimal, é arredondado para um valor inteiro que é usado para remontar uma tupla RGB. Nesta nova tupla, o mesmo valor Y é atribuído como o vermelho, o verde e o azul. A tupla é então usada para criar um novo objeto de pixel e ele é associado a mesma coordenada (X, Y) do pixel colorido na nova matriz em escala cinza. Por fim, após todos os pixels serem convertidos, a nova matriz é usada para criar uma nova imagem de saída em escala cinza. A figura 4 seguinte demonstra o fluxograma deste algoritmo.

É possível observar que o procedimento em si não pode ser classificado como de alta complexidade, visto a simplicidade da operação aritmética realizada que é o coração deste programa. Entretanto, tal operação é repetida muitas vezes para converter todos os pixels da imagem fornecida e isso exige certo custo computacional que pode ser mensurado e avaliado.

Conforme mencionado anteriormente, os experimentos conduzidos neste trabalho deveriam durar pelo menos alguns segundos e exigir certo esforço computacional e por isso, para este programa, foi fornecida uma imagem com a maior resolução padronizada possível: a 16K. Imagens nesta resolução possuem largura de 15.360 pixels e altura de 8.640 pixels, totalizando uma matriz com 132.710.400 de pixels. A imagem selecionada está no formato JPG e possui o tamanho de 35 MBytes aproximadamente. Conforme visto na figura 5 seguinte, a imagem original fornecida também é majoritariamente constituída por tons diferentes do cinza e do preto.

Como este experimento consiste de um programa essencialmente repetitivo, no qual cada pixel da imagem fornecida é lido e convertido tal como demonstrado no fluxograma, o

Figura 4 – Fluxograma do conversor de imagens.



Fonte: Autoria própria.

desempenho energético do programa foi avaliado sob o uso das diferentes estruturas de repetição da linguagem.

Foram implementadas cinco classes para realizar a conversão através de cinco estruturas repetitivas diferentes: a ForClassicConverter, a ForEnhancedConverter, a ForEachConverter, a ForEachParallelConverter e a IteratorConverter. Todas estas classes satisfazem o contrato definido pela interface IGrayConverter, que exige a implementação de um método chamado "convert". Este método deve receber um objeto de BufferedImage da imagem fornecida e retornar outro BufferedImage da imagem convertida. É neste método que estão todos os passos necessários para a conversão da imagem fornecida com a abordagem da estratégia em questão e, por este motivo, este teve o seu desempenho que foi avaliado. A imagem convertida pelo programa pode ser observada na figura 6 seguinte.

Figura 5 – Imagem fornecida para o conversor.

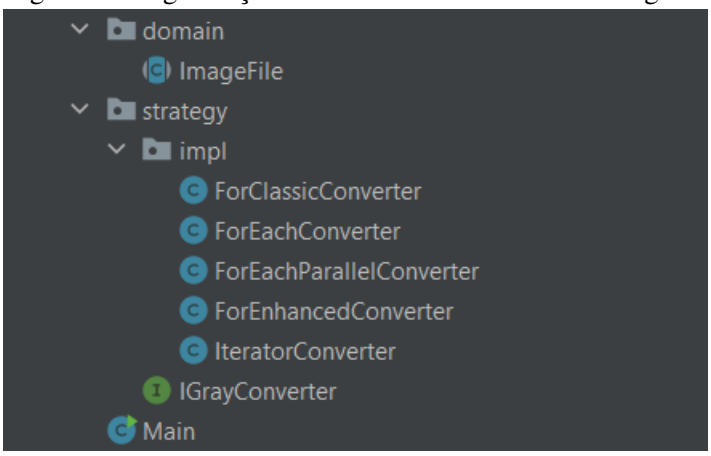
Fonte: Retirada de (POWERFULL. . . , 2022)

Figura 6 – Imagem convertida para o conversor.

Fonte: Autoria própria.

Há também uma classe Main, que é o ponto de entrada do programa. Esta classe, além de instanciar a estratégia em análise e invocar o seu método "convert", contém também as operações de entrada e saída que não são avaliadas no âmbito deste experimento. A organização destes componentes e de seus pacotes pode ser observada na figura 7 seguinte.

Figura 7 – Organização de classes do conversor de imagens.

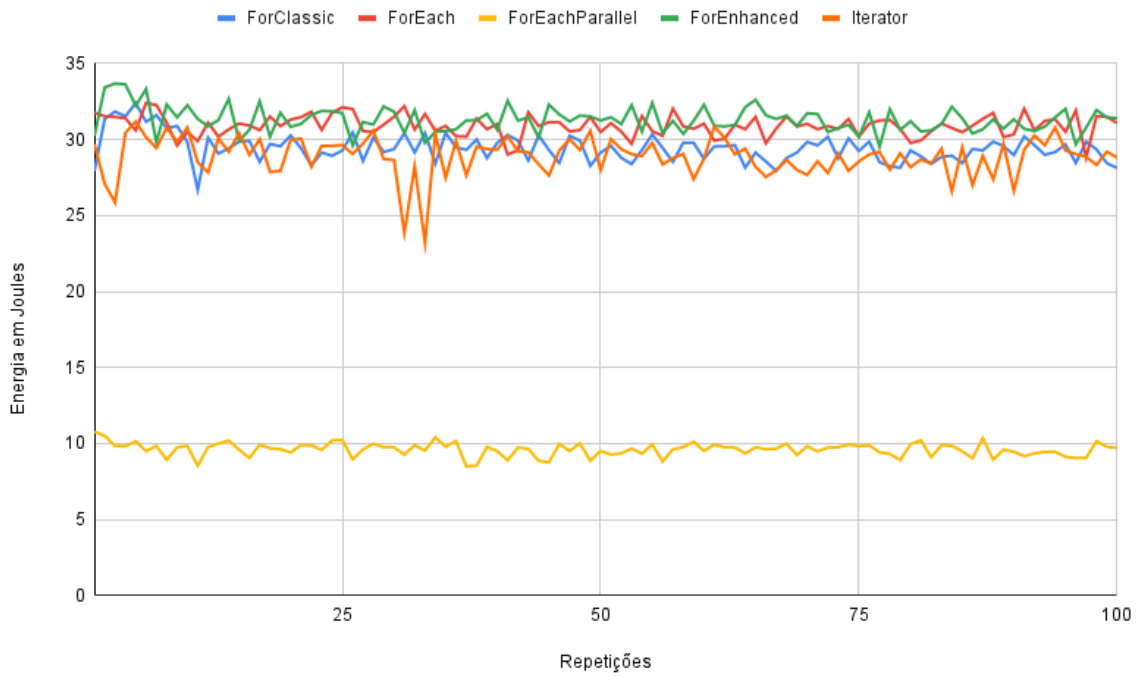


Fonte: Autoria própria.

4.1.1 Resultados do conversor de imagens

Após a execução das estratégias de conversão repetidamente, tal como mencionado anteriormente, foi possível mensurar o desempenho energético do programa com o auxílio do JoularJX. A figura seguinte 8 ilustra o gráfico com desempenho energético em cada repetição da execução com as estratégias.

Figura 8 – Consumos de energia do conversor de imagens.



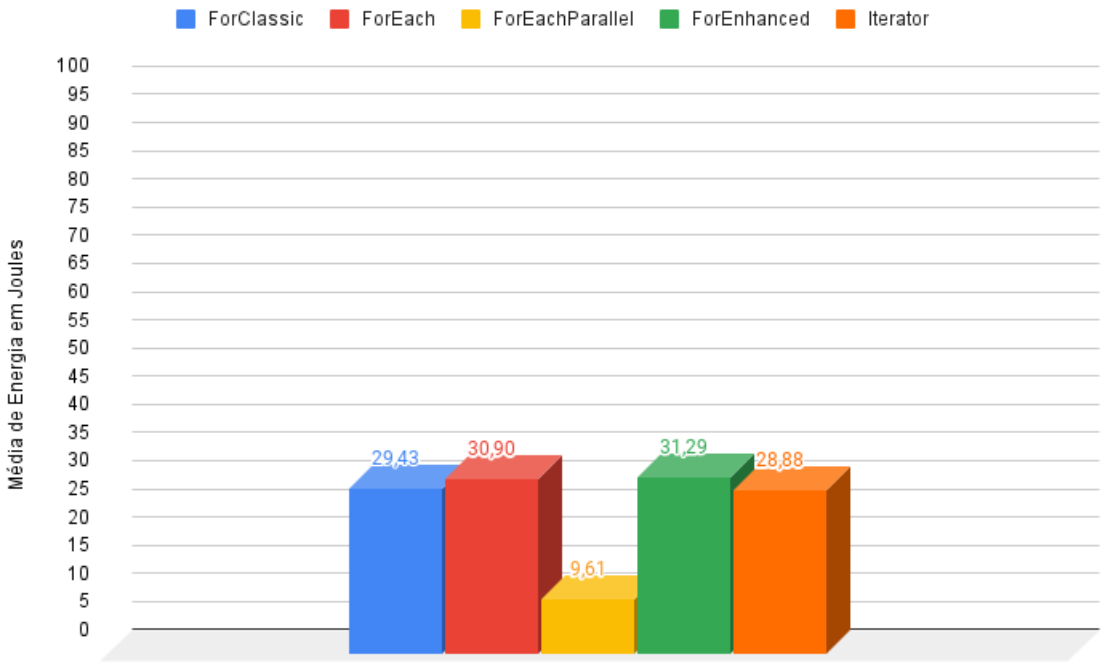
Fonte: Autoria própria.

Como é possível observar no gráfico, as implementações do conversor de imagens

com uso do For Clássico (azul), For Aprimorado (vermelho), ForEach (amarelo) e Iterator (laranja) obtiveram resultados semelhantes, oscilando próximo de 30 Joules. Vale pontuar que, com exceção do Iterator, que em pontuais execuções teve o desempenho energético abaixo dos 25 Joules, estas estratégias também não demonstraram muitas variações bruscas. O uso do ForEachParallel no conversor de imagens, por outro lado, teve o perfil energético mais econômico dentre todas as estratégias avaliadas, variando próximo aos 10 Joules aproximadamente. Da mesma forma, não foi possível observar grandes surtos no consumo energético da abordagem com o ForEachParallel ao longo de todas as suas execuções.

O gráfico da figura 9 seguinte consolida estes dados e apresenta a média dos consumos de energia em Joules de cada uma das estratégias, elucidando suas diferenças de comportamento.

Figura 9 – Consumo médio de energia do conversor de imagens.

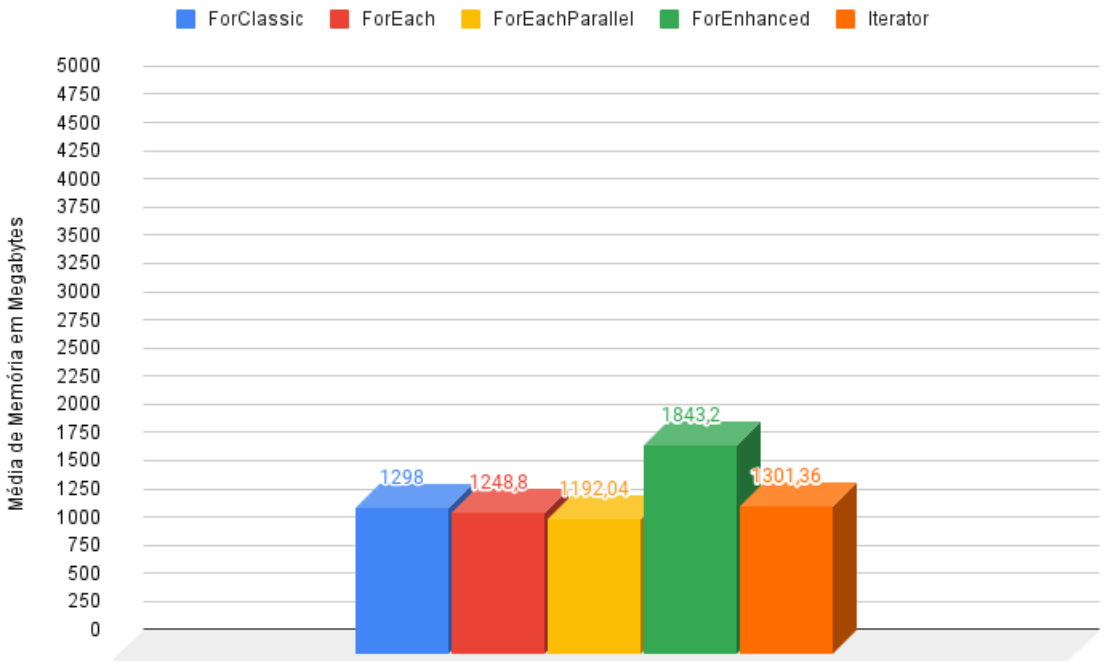


Fonte: Autoria própria.

É possível observar no gráfico anterior que o ForEachParallel está em primeiro lugar, tendo o consumo energético mais econômico de todas as estratégias, isto é, em média cerca de um terço da energia utilizada para realizar a mesma atividade de conversão que as demais estratégias. Mesmo sendo baseado em chamadas de implementações de métodos de algum tipo de estrutura de dados, o ForEachParallel consegue até mesmo superar instruções primitivas como

o For Clássico e o For Aprimorado. Por este motivo, e também para enriquecer a análise, vale a pena entender o consumo de recursos destas estratégias de implementação do conversor sob outras perspectivas. O gráfico na figura 10 seguinte retrata o consumo médio de memória em Megabytes de cada estratégia.

Figura 10 – Consumo médio de memória do conversor de imagens.

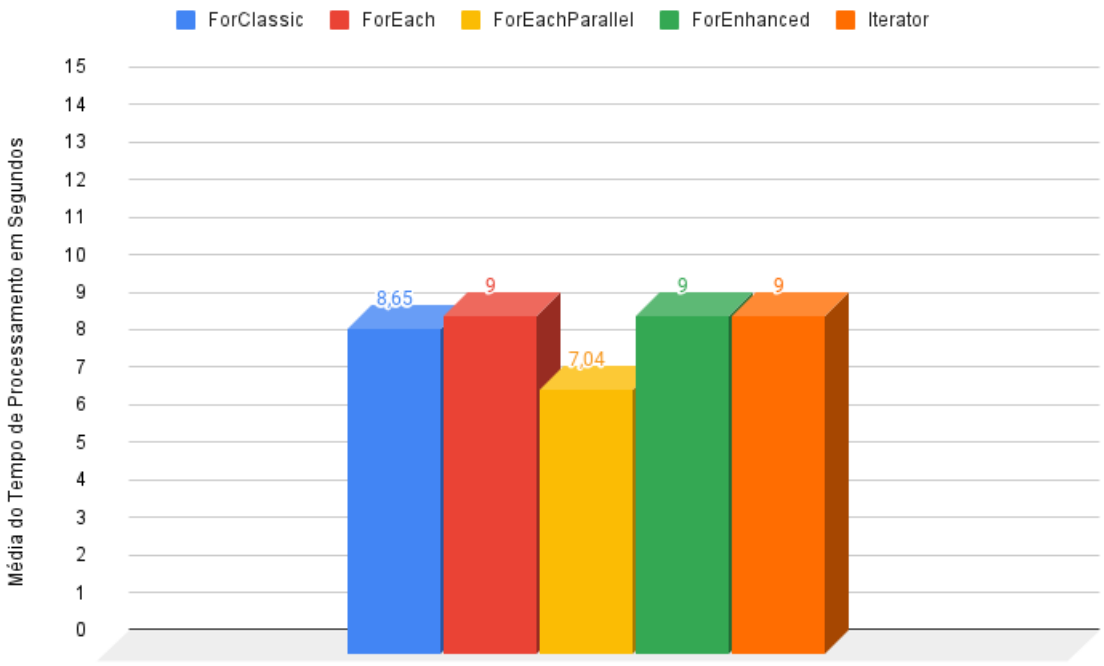


Fonte: Autoria própria.

Em relação a utilização da memória principal, observa-se que quase todas as estratégias apresentaram uma alocação de dados bastante semelhante, em média. Observa-se que o ForEachParallel, em relação ao consumo de memória, também foi o mais econômico, mas seus resultados em média não se distanciaram tanto assim das outras estratégias. O destaque nesta análise, contudo, vai para o For Aprimorado, que teve um consumo de memória em média cerca de 46% maior que a média das demais estratégias. Isso indica que o consumo de energia e o de memória não estão sempre diretamente relacionados.

Por fim, no que se refere ao desempenho do conversor de imagens, também foi oportuno fazer uma análise do tempo de processamento demandado, no intuito de ajudar a ampliar a compreensão sobre os perfis energéticos. O gráfico na figura 11 seguinte apresenta os tempos médios em segundos do processamento das estratégias.

Figura 11 – Consumo médio de tempo do conversor de imagens.



Fonte: Autoria própria.

Conforme evidenciado no gráfico, o tempo médio de *CPU* do ForEachParallel foi cerca de 21% menor que a média dos tempos das demais estratégias. Dentre as demais estratégias, ainda que o For Clássico tenha obtido uma leve economia, não houve um grande destaque e ambas conseguiram desempenhar a atividade de conversão da imagem fornecida com tempos bastante semelhantes. Nota-se, portanto, que o tempo de processamento teve influência maior sobre o perfil energético do programa, ainda que não se possa afirmar que a variável temporal esteja direta e unicamente relacionada ao perfil energético de um programa.

4.2 CALCULADORA DE FATORIAIS

A necessidade idealizada para o segundo experimento consiste em receber um arquivo de texto contendo um número inteiro por linha e produzir outro arquivo de saída contendo o número fatorial na respectiva linha. Considerando-se que o número fatorial de um determinado inteiro X consiste da multiplicação de todos os inteiros entre 1 e X, e que conceitualmente o fatorial de 0 é igual a 1, o arquivo fornecido continha apenas inteiros maiores ou igual a 0. Assim, o processo de cálculo do fatorial de um inteiro X traduz-se em um processo de repetição computacional, no qual se determina cada inteiro entre 1 e X através de uma operação de incremento ou decremento. Ainda que, por vezes, este algoritmo possa ser representado com a

abordagem recursiva, este não é o foco deste experimento. Aqui objetiva-se o processo iterativo, fazendo uso das mesmas estratégias de repetição utilizadas para o experimento com o conversor de imagens.

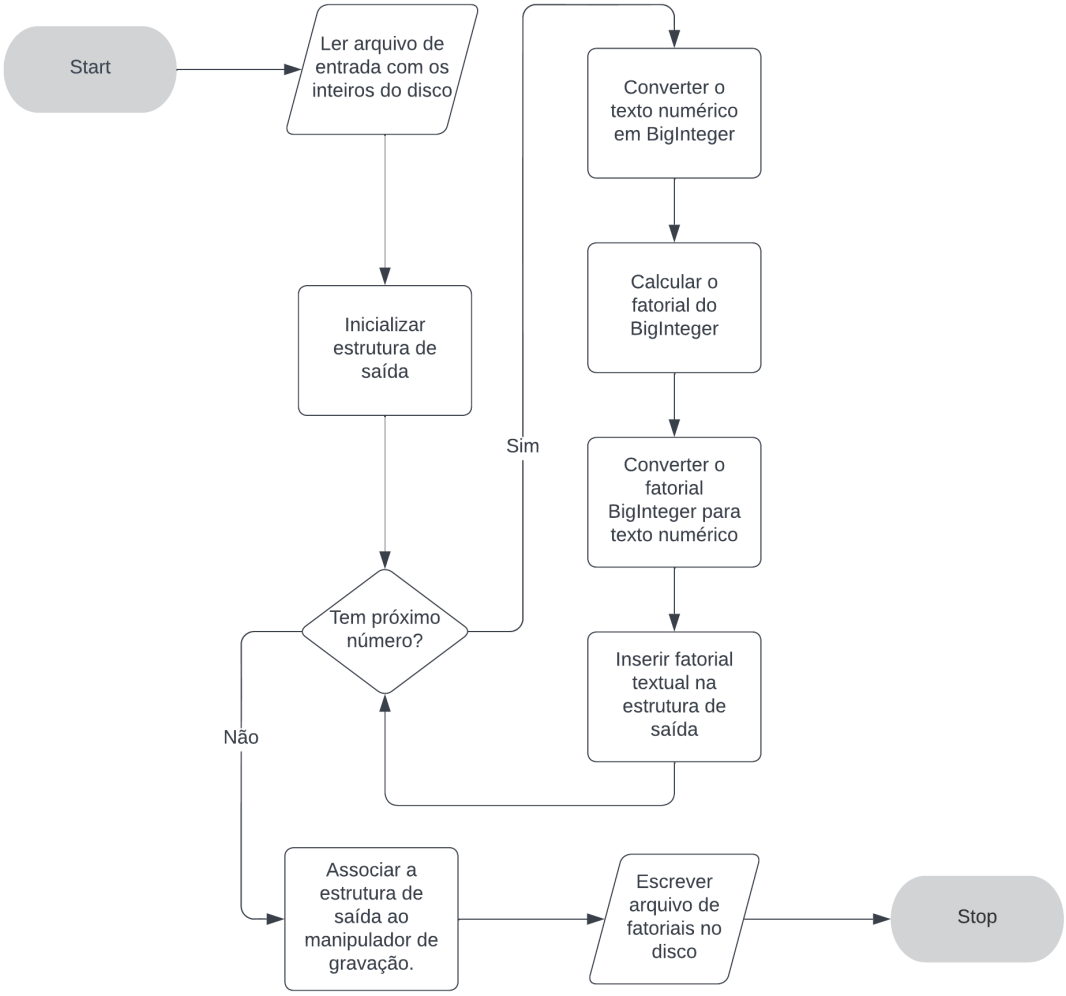
Para realizar a execução deste experimento, um arquivo de texto apropriado foi construído contendo 3 milhões de números inteiros entre 0 e 100, um em cada linha. Vale salientar que, por conta de limitações tecnológicas, o maior número inteiro possível de ser representado com os tipos primitivos “int” e “long” na linguagem Java são 2.147.483.647 e 9.223.372.036.854.775.807, respectivamente, e isso inviabiliza utilizar estes tipos para calcular o fatorial de números inteiros maiores ou iguais a 21, uma vez que o fatorial deste número é igual a 51.090.942.171.709.440.000. É para situações semelhantes a esta que a linguagem Java dispõe da classe BigInteger e ela foi utilizada para trabalhar os números do arquivo de entrada fornecido para este experimento.

O algoritmo deste programa, então, resume-se em ler o arquivo fornecido e transformar o seu conteúdo em uma estrutura iterável na qual em cada passo se possa recuperar uma linha, ou melhor, um número inteiro. Na sequência, a estrutura iterável será repassada para o método de cálculo fatorial de uma das estratégias, que irá recuperar cada linha textual contendo um número, convertê-la para um BigInteger e, por sua vez, passá-lo para o método que irá calcular o seu fatorial. O método que calcula o fatorial, portanto, recebe um BigInteger e retorna outro BigInteger. Cada fatorial retornado é colocado em uma estrutura iterável de saída, que é o retornado pela estratégia. A estrutura de saída é, então, repassada para o método de escrita que irá produzir o arquivo de saída contendo os fatoriais da estrutura em cada linha. A figura 12 seguinte apresenta o fluxograma deste algoritmo.

Tanto o processo de leitura do arquivo fornecido e inicialização da estrutura iterável de entrada, quando o processo de escrita da estrutura de saída no arquivo produzido não são alvos do estudo sobre o desempenho energético levantado neste experimento. A análise de consumo de energia limita-se ao trabalho realizado pela estratégia de cálculo selecionada, que faz o uso de uma das abordagens de repetição para processar cada número fornecido.

Foram implementadas cinco classes para realizar o cálculo do fatorial através de cinco estruturas repetitivas diferentes: a ForClassicCalculator, a ForEachCalculator, a ForEachParallelCalculator, ForEnhancedCalculator e a IteratorCalculator. Todas estas classes satisfazem o contrato definido pela interface IFactorialCalculator, que exige a implementação de um método

Figura 12 – Fluxograma da calculadora de fatoriais.

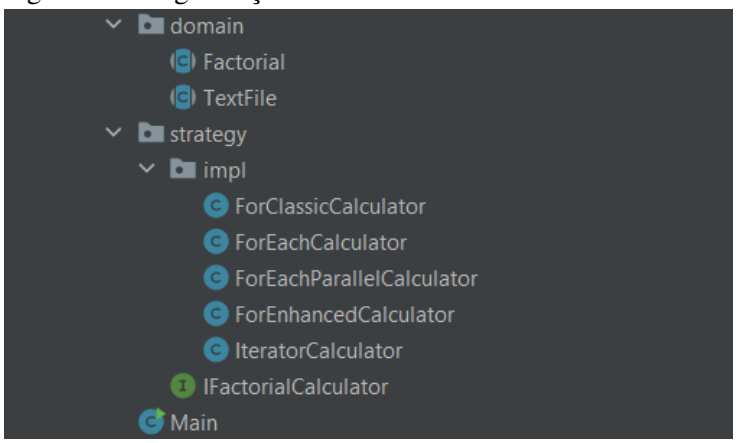


Fonte: Autoria própria.

chamado "calculateAll". Este método deve receber uma lista de String contendo as linhas do arquivo de números fornecido e retornar outra lista de String com os fatoriais calculados.

Há também uma classe Main, que é o ponto de entrada do programa. Esta classe, além de instanciar a estratégia em análise e invocar o seu método "calculateAll", contém também as operações de entrada e saída. A organização destes componentes e de seus pacotes pode ser observada na figura 13 seguinte e o código fonte completo pode ser visualizado nos apêndices deste material.

Figura 13 – Organização de classes da calculadora de fatoriais.

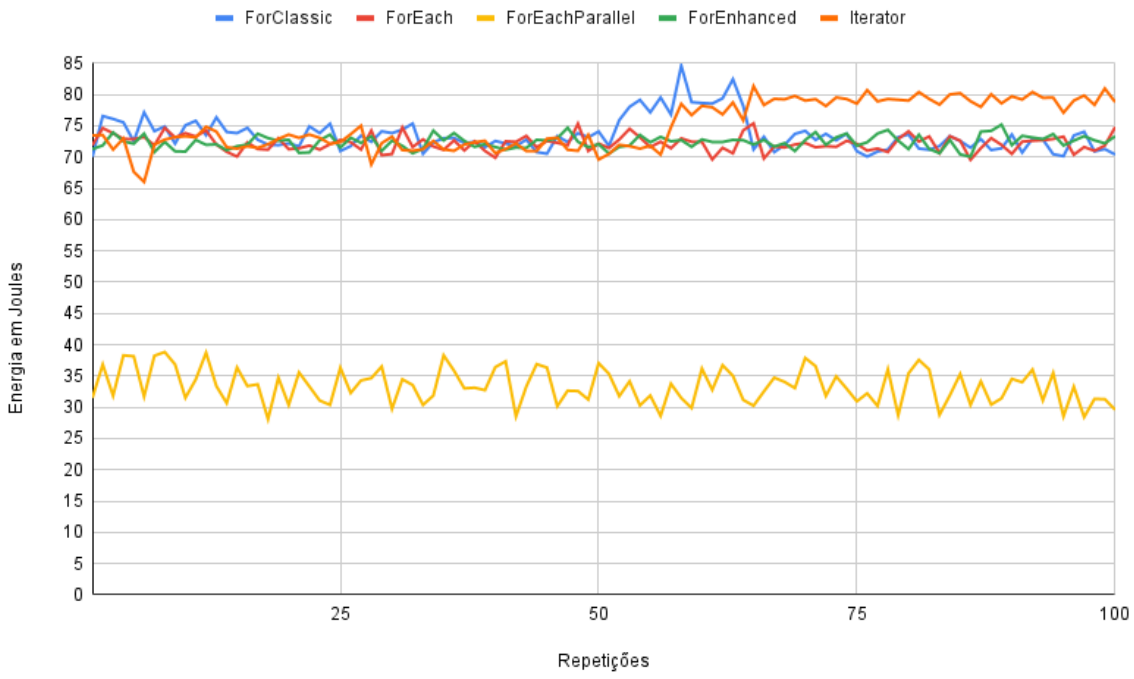


Fonte: Autoria própria.

4.2.1 Resultados da calculadora de fatoriais

Com o auxílio do JoularJX foi possível obter os dados de consumo de energia da calculadora de fatoriais. O gráfico observado na figura 14 seguinte apresenta o consumo de cada uma das 100 vezes em que o programa foi executado para cada estratégia.

Figura 14 – Consumos de energia do calculadora de fatoriais.

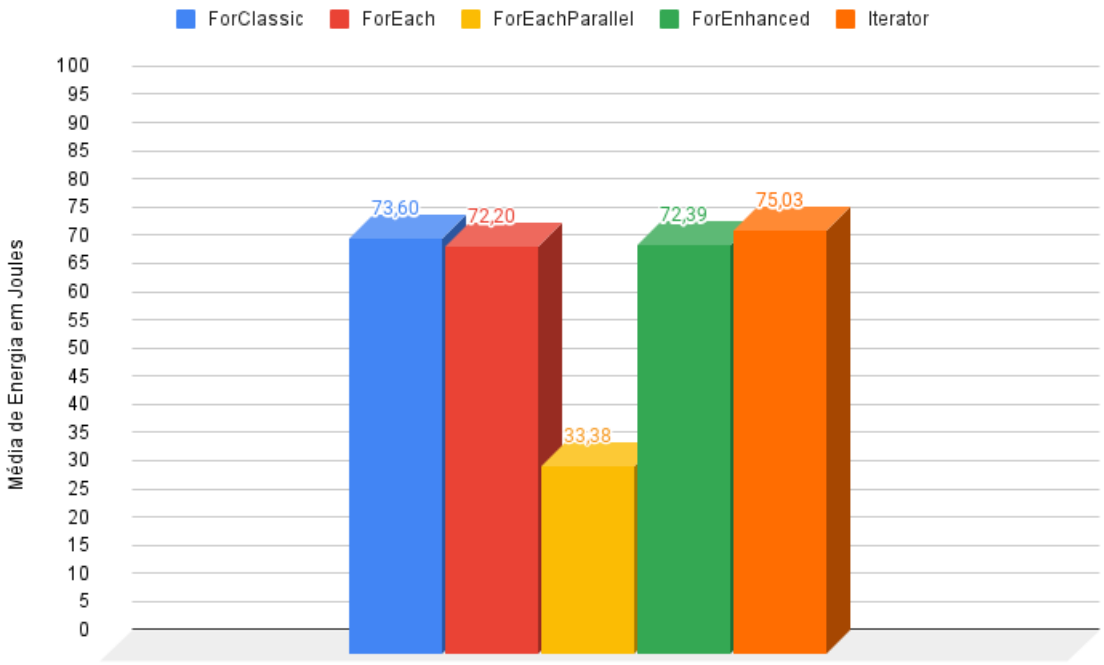


Fonte: Autoria própria.

Semelhantemente neste caso de estudo as implementações com o For Clássico (azul), For Aprimorado (vermelho), ForEach (amarelo) e Iterator (laranja) também obtiveram resultados

semelhantes, com valores próximos a 75 Joules. Aqui o For Aprimorado e o ForEach nitidamente tiveram resultados estáveis, enquanto que o For Clássico e o Iterator a partir de determinado ponto apresentaram surtos no consumo energético. E finalmente o ForEachParallel apresentou o melhor resultado em termos energéticos também na calculadora de fatoriais, com valores próximos a 35 Joules. O gráfico da figura 15 seguinte consolida os dados de energia das execuções e apresenta a média dos consumos em Joules de cada uma das estratégias.

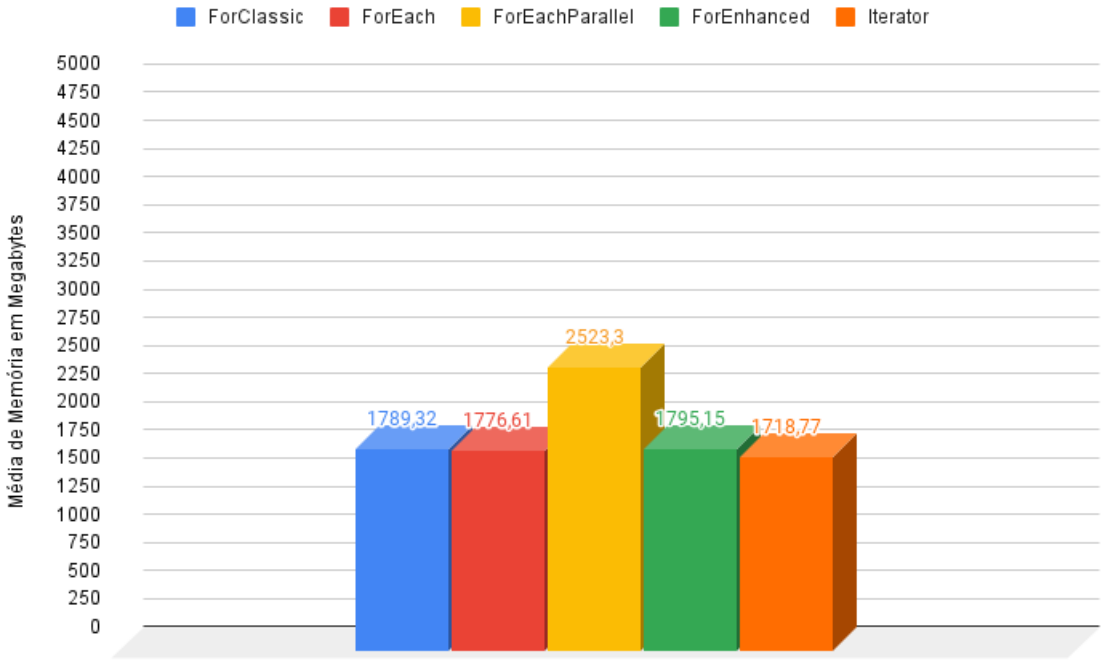
Figura 15 – Consumos médio de energia do calculadora de fatoriais.



Fonte: Autoria própria.

O ForEachParallel, conforme observado, ficou novamente em primeiro lugar, tendo o menor consumo de energia em média das estratégias para realizar o cálculo dos números fatoriais. Diferente do conversor de imagens, o Iterator apresentou um consumo médio de 75 Joules e foi a abordagem mais energeticamente dispendiosa, apesar de não se distanciar tanto das demais. Em termos de consumo energético, portanto, o ForEachParallel teve um desempenho cerca de 54% melhor que as demais estratégias. O gráfico na figura 16 seguinte retrata o consumo médio de memória em Megabytes de cada estratégia.

Figura 16 – Consumos médio de memória do calculadora de fatoriais.

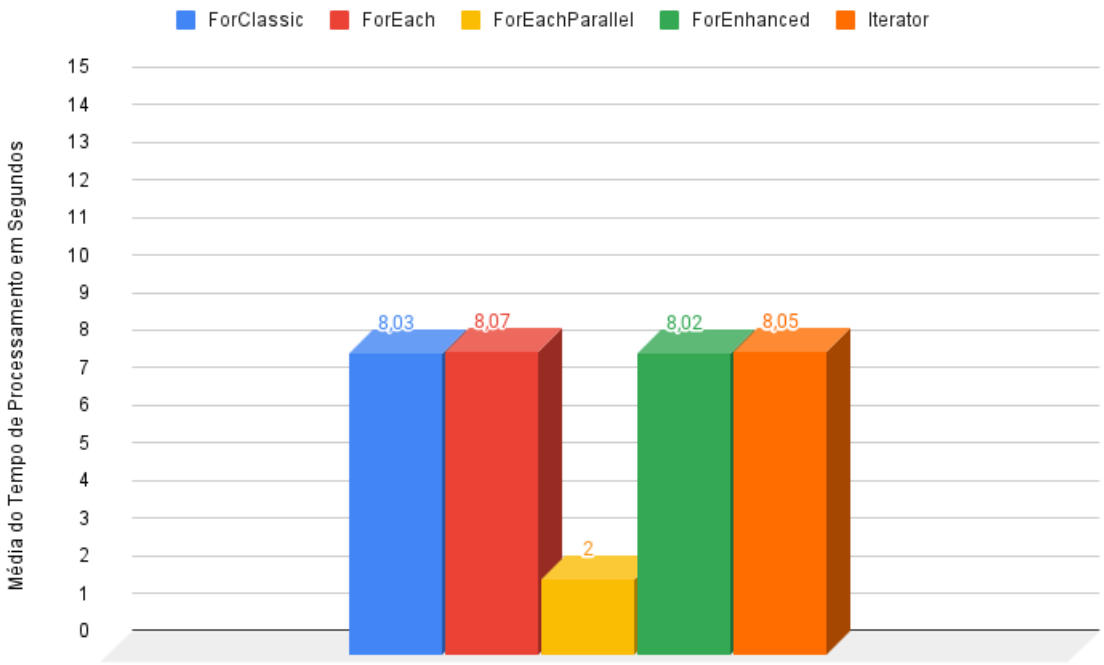


Fonte: Autoria própria.

Em relação ao consumo de memória, o gráfico evidencia que, em média, o ForEachParallel apresentou o maior consumo dentre todas as estratégias, alocando 42,56% a mais que a média das demais abordagens. Por fim, também foi feita a análise do tempo de processamento demandado pelas estratégias da calculadora de fatoriais. O gráfico na figura 17 seguinte apresenta os tempos médios em segundos do processamento das estratégias.

O gráfico demonstra que o tempo médio de *CPU* do ForEachParallel foi cerca de 75% menor que a média dos tempos das demais estratégias. Neste experimento também não houve um grande destaque para as demais estratégias, que desempenharam a atividade de cálculo de fatoriais com tempos bastante semelhantes. Foi constatado da mesma forma que o tempo de processamento teve influência maior sobre o perfil energético do programa, ainda que não se possa afirmar que a variável temporal esteja unicamente relacionada ao perfil energético do programa.

Figura 17 – Consumos médio de tempo do calculadora de fatoriais.



Fonte: Autoria própria.

5 CONSIDERAÇÕES FINAIS

Os resultados obtidos neste trabalho apontam que determinadas decisões que são tomadas durante o processo de codificação conduzem a um resultado final mais ou menos eficiente no sentido da utilização de eletricidade do sistema computacional. Como visto com o programa conversor de imagens e com a calculadora de fatoriais, as abordagens com o ForEachParallel foram mais econômicas no dispêndio energético que as demais abordagens. O ForEachParallel, que não foi a mais econômica estratégia em alocação de memória, foi a mais rápida em tempo de *CPU* em ambos os casos.

De fato, ainda que a análise do tempo efetivamente gasto em *CPU* detenha destacada importância, é um equívoco afirmar que a variável temporal influencia direta e unicamente a questão energética, já que uma maior quantidade de dados em memória também influencia o perfil energético geral do programa. Em ambos os experimentos, o For Clássico foi o segundo melhor colocado em termos de tempo demandado de *CPU*, mas não foi o segundo melhor em termos de eficiência energética, ainda que as diferenças sejam relativamente pequenas.

As abordagens For Clássico e For Aprimorado, mesmo sendo abordagens baseadas em instruções primitivas da linguagem Java, não conseguiram economizar o uso de energia ao realizar as atividades propostas. Já o Iterator e o ForEach, que são abordagens baseadas em métodos, tiveram resultados próximos aos do For Clássico e do For Aprimorado. As abordagens baseadas em métodos como o ForEach, por exemplo, na prática exigem o fornecimento de uma instância de uma implementação da interface Consumer, ainda que muitas vezes esta declaração possa ser encontrada nas formas variantes mais simplificadas que foram demonstradas no capítulo 3. Isso significa que para cada repetição executada para processar uma estrutura de Collection, como as listas, há também a necessidade de alocação de mais recursos para acomodar estes objetos criados e descartados dinamicamente a cada passo da repetição.

Como visto no experimento da calculadora de fatoriais, ainda que o ForEachParallel tenha sido a abordagem que mais alocou bytes em memória, ela também foi a abordagem que consumiu menos energia para realizar a atividade. Vale reforçar que um programa que geralmente aloca mais dados em memória tem um maior fluxo de dados entrando e saindo da

CPU para serem utilizados em operações lógicas ou aritméticas e isso sustentaria a consequência de que este fator contribuiria diretamente para uma elevação do consumo de energia. Contudo, os dados obtidos com este experimento, e que estão em linha com o do conversor de imagens, apontam que o consumo de energia e o consumo de memória não estão diretamente relacionados, isto é, o aumento do consumo de energia não se sustenta unicamente em um eventual aumento proporcional de alocação de memória.

Vale reforçar que, ainda que a diferença entre as médias de consumo de energia seja de algumas dezenas de Joules, tal economia obtida com a simples troca de algumas linhas de código pode proporcionar uma notável economia de eletricidade se for considerado que os softwares normalmente são hospedados em ambientes de produção com a finalidade de desempenhar repetidamente a mesma tarefa centenas de milhões de vezes em um período de um ano, por exemplo. Softwares mais eficientes demandam menos esforço computacional, conforme demonstrado nos testes, e contribuem para que a escalada desenfreada por máquinas cada vez mais poderosas seja mitigada.

5.1 TRABALHOS FUTUROS

Este trabalho foi realizado através da busca de alternativas para as diversas limitações que foram encontradas ao longo do seu desenvolvimento e, por isso, a questão da eficiência energética de softwares no âmbito da Computação Verde pode ser explorada em trabalhos futuros que busquem preencher as lacunas que aqui não puderam ser preenchidas. Devido a disposição de hardware com arquitetura de processador AMD Ryzen, foi necessário realizar os experimentos de mensuração energética através da interface powercap fornecida pelo sistema Linux. Será que os programas se comportam energeticamente da mesma maneira em sistemas com arquitetura Intel moderna, capazes de fornecer acesso ao RAPL diretamente? Será que a execução em sistemas operacionais Windows geraria os mesmos resultados? Será que outras bibliotecas de monitoramento de energia iriam produzir resultados mais ou menos próximos aos que foram obtidos aqui? Será que os mesmos programas que aqui foram implementados na linguagem Java teriam comportamentos diferentes se fossem implementados, por exemplo, em Python? Será que linguagens interpretadas, como o PHP moderno, são energeticamente mais dispendiosas? O presente trabalho não responde estas questões mas as aponta como caminhos futuros para o amadurecimento do tema.

REFERÊNCIAS

AGRAWAL, N.; SAINI, J.; WANKHEDE, P. Review on green cloud computing: A step towards saving global environment. *International Journal of Engineering Research Technology (IJERT)*, 2020.

AHMED, F.; SHARMA, M. Computer games complexity: A reality check of environmental footprint. *International Parallel Conferences on Researches in Industrial and Applied Sciences*, 2014.

ALMEIDA, E. et al. Computação brasil. *Revista da Sociedade Brasileira de Computação*, 2018.

BEZERRA, J. *Conferência de Estocolmo*. 2020. Disponível em: <https://www.todamateria.com.br/conferencia-de-estocolmo/>.

BRUDTLAND, G. H. *Nosso Futuro Comum*. [S.l.]: Editora da Fundação Getúlio Vargas, 1991. v. 2.

CAMPOS, M. *ECO-92*. 2022. Disponível em: <https://mundoeducacao.uol.com.br/geografia/eco92.htm>.

CHANDA, P.; MODAK, S.; MUKHERJEE, P. K. Scope and issues in green compiler. *International Research Journal of Computer Science (IRJCS)*, 2017.

DEITEL, P.; DEITEL, H. *Java Como Programar*. [S.l.]: Editora da Fundação Getúlio Vargas, 2017. v. 10.

GUITARRARA, P. *ECO-92*. 2022. Disponível em: <https://brasilescola.uol.com.br/geografia/eco-92.htm>.

HENDERSON, P. et al. Towards the systematic reporting of the energy and carbon footprints of machine learning. *Journal of Machine Learning Research 21*, 2020.

IGNACIO, J. *ECO-92: o que foi a conferência e quais foram seus principais resultados?* 2020. Disponível em: <https://www.politize.com.br/eco-92/>.

JAGROEP, E. et al. Extending software architecture views with an energy consumption perspective. *Computing 99*, 2016.

JAVA Collections: Como utilizar Collections. 2010. Disponível em: <https://www.devmedia.com.br/java-collections-como-utilizar-collections/18450>.

KERNEL. Power Capping Framework. 2022. Disponível em: <https://www.kernel.org/doc/html/latest/power/powercap/powercap.html>.

LI, D. et al. Calculating source line level energy information for android applications. *International Symposium on Software Testing and Analysis (ISSTA)*, 2013.

NOLETO, C. *Java forEach: como usar o enhanced-for loop*. 2010. Disponível em: <https://blog.betrybe.com/java-foreach/>.

NOUREDDINE, A. Powerjoular and joularjx: Multi-platform software power monitoring tools. *International Conference on Intelligent Environments*, 2022.

OLAOLUWA, P. O.; KOR, A.-L.; PATTINSON, C. Php single and double quotes: Does it make a difference to energy consumption. *International Sustainable Ecological Engineering Design for Society (SEEDS)*, 2015.

PINTO, G.; CASTOR, F. Energy efficiency: A new concern for application software developers. *Communications of the ACM 60(12)*, 2017.

POWERFULL Saitama One-Punch Man (2560x1080) Resolution Wallpaper. 2022. Disponível em: <https://wallpapersden.com/powerfull-saitama-one-punch-man-wallpaper/2560x1080/>.

PUTRI, N. et al. E-waste handling in dki jakarta private higher education institution. *Journal of Theoretical and Applied Information Technology*, 2015.

RASHID, N.; KHAN, S. U. Identification of practices for the development of green and sustainable software using agile methods in gsd projects: A systematic literature review protocol. *Journal of Multidisciplinary Engineering Science and Technology (JMEST)*, 2016.

SAHA, B. Green software. *International Journal of Computer Trends and Technology (IJCTT)*, 2014.

SALAM, M.; KHAN, S. U. Systematic literature review protocol for green software multi-sourcing with preliminary results. *Pakistan Academy of Sciences*, 2015.

SALAM, M. A.; KHAN, S. U. Risks mitigation practices for multi-sourcing vendors in green software development. *Pakistan Academy of Sciences*, 2017.

SHAH, H.; SOOMRO, T. R. Perspectives of green computing. *SZABIST-Dubai Working Paper Series*, 2015.

SILVA, G. D. *Performance and Energy efficiency Analysis of Machine Learning algorithms Towards Green AI: a case study of decision tree algorithms*. 51 f. Monografia (Graduação) — National Laboratory for Scientific Computing, Petrópolis, RJ - Brazil, 2021.

SILVA, W. *Estruturas de Dados: Listas, Filas, Pilhas, Conjuntos, Árvores e Hash Tables*. 2021. Disponível em: <https://growthdev.com.br/programacao/estruturas-de-dados-listas-filas-pilhas-conjuntos-arvores-e-tabela-espelhadas/>.

WELTER, M.; BENITTI, F. B. V.; THIRY, M. Green metrics to software development organizations, a systematic mapping. *XL Latin American Computing Conference (CLEI)*, 2014.

APÊNDICES

APÊNDICE A – Código-fonte do conversor de imagens

Os blocos de código-fonte abaixo demonstram o código fonte completo do programa conversor de imagens coloridas em escala cinza analisado no capítulo 4.

```
1  package io.github.mateuspena.app1;
2
3  import io.github.mateuspena.app1.strategy.IGrayConverter;
4  import io.github.mateuspena.app1.strategy.impl.*;
5
6  import java.awt.image.BufferedImage;
7  import java.io.IOException;
8  import java.lang.management.ManagementFactory;
9  import java.util.concurrent.TimeUnit;
10
11 import static io.github.mateuspena.app1.domain.ImageFile.readImage;
12 import static io.github.mateuspena.app1.domain.ImageFile.writeImage;
13
14 public class Main {
15   public static void main(String[] args)
16       throws IOException, InterruptedException {
17     IGrayConverter strategy = new ForClassicConverter();
18
19     BufferedImage in = readImage(args[0]);
20     writeImage("converted-image.jpg", strategy.convert(in));
21
22     System.out.println("* Time elapsed: " + timeElapsedInSeconds());
23     System.out.println("* Memory usage: " + memoryUsageInMegabytes());
24     TimeUnit.SECONDS.sleep(1);
25     System.exit(0);
26   }
27
28   private static long timeElapsedInSeconds() {
29     return ManagementFactory.getThreadMXBean().getCurrentThreadCpuTime()
30       / (1000 * 1000 * 1000);
31   }
32
33   private static long memoryUsageInMegabytes() {
34     return ManagementFactory.getMemoryMXBean().getHeapMemoryUsage().getUsed()
35       / (1024 * 1024);
36   }
37 }
```

```java
package io.github.mateuspena.app1.domain;

import javax.imageio.ImageIO;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;

public abstract class ImageFile {
  public static BufferedImage readImage(String filepath)
      throws IOException {
    return ImageIO.read(new File(filepath));
  }

  public static void writeImage(String filepath, BufferedImage image)
      throws IOException {
    ImageIO.write(image, "jpg", new File(filepath));
  }
}
```

```java
package io.github.mateuspena.app1.strategy;

import java.awt.*;
import java.awt.image.BufferedImage;

public interface IGrayConverter {
  BufferedImage convert(BufferedImage input);

  default Color grayColor(final Color color) {
    int gray =
        (int) ((0.2989 * color.getRed())
                + (0.5870 * color.getGreen())
                + (0.1140 * color.getBlue()));
    return new Color(gray, gray, gray);
  }
}
```

```java
package io.github.mateuspena.app1.strategy.impl;

import io.github.mateuspena.app1.strategy.IGrayConverter;

import java.awt.*;
import java.awt.image.BufferedImage;

public class ForClassicConverter implements IGrayConverter {
  @Override
  public BufferedImage convert(BufferedImage input) {
    final int height = input.getHeight();
    final int width = input.getWidth();
    final BufferedImage output =
      new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);

    final int size = height * width;
    for (int index = 0; index < size; index++) {
      final int x = index % width;
      final int y = index / width;
      final int pixel = input.getRGB(x, y);

      final Color color = new Color(pixel);
      final Color grayColor = grayColor(color);

      output.setRGB(x, y, grayColor.getRGB());
    }

    return output;
  }
}
```

```java
package io.github.mateuspena.app1.strategy.impl;

import io.github.mateuspena.app1.strategy.IGrayConverter;

import java.awt.*;
import java.awt.image.BufferedImage;
import java.util.stream.IntStream;

public class ForEachConverter implements IGrayConverter {
  @Override
  public BufferedImage convert(BufferedImage input) {
    final int height = input.getHeight();
    final int width = input.getWidth();
    final BufferedImage output =
      new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);

    final int size = height * width;
    IntStream.range(0, size)
        .forEach(
            index -> {
              final int x = index % width;
              final int y = index / width;
              final int pixel = input.getRGB(x, y);

              final Color color = new Color(pixel);
              final Color grayColor = grayColor(color);
              output.setRGB(x, y, grayColor.getRGB());
            });

    return output;
  }
}
```

```java
package io.github.mateuspena.app1.strategy.impl;

import io.github.mateuspena.app1.strategy.IGrayConverter;

import java.awt.*;
import java.awt.image.BufferedImage;
import java.util.stream.IntStream;

public class ForEachParallelConverter implements IGrayConverter {
  @Override
  public BufferedImage convert(BufferedImage input) {
    final int height = input.getHeight();
    final int width = input.getWidth();
    final BufferedImage output =
      new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);

    final int size = height * width;
    IntStream.range(0, size)
        .parallel()
        .forEach(
            index -> {
              final int x = index % width;
              final int y = index / width;
              final int pixel = input.getRGB(x, y);

              final Color color = new Color(pixel);
              final Color grayColor = grayColor(color);
              output.setRGB(x, y, grayColor.getRGB());
            });

    return output;
  }
}
```

```java
package io.github.mateuspena.app1.strategy.impl;

import io.github.mateuspena.app1.strategy.IGrayConverter;

import java.awt.*;
import java.awt.image.BufferedImage;
import java.util.stream.IntStream;

public class ForEnhancedConverter implements IGrayConverter {
  @Override
  public BufferedImage convert(BufferedImage input) {
    final int height = input.getHeight();
    final int width = input.getWidth();
    final BufferedImage output =
      new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);

    final int size = height * width;
    for (int index : IntStream.range(0, size).toArray()) {
      final int x = index % width;
      final int y = index / width;
      final int pixel = input.getRGB(x, y);

      final Color color = new Color(pixel);
      final Color grayColor = grayColor(color);
      output.setRGB(x, y, grayColor.getRGB());
    }

    return output;
  }
}
```

```java
package io.github.mateuspena.app1.strategy.impl;

import io.github.mateuspena.app1.strategy.IGrayConverter;

import java.awt.*;
import java.awt.image.BufferedImage;
import java.util.Iterator;
import java.util.stream.IntStream;

public class IteratorConverter implements IGrayConverter {
  @Override
  public BufferedImage convert(BufferedImage input) {
    final int height = input.getHeight();
    final int width = input.getWidth();
    final BufferedImage output =
      new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);

    final int size = height * width;
    final Iterator<Integer> iter = IntStream.range(0, size).iterator();
    while (iter.hasNext()) {
      final int index = iter.next();

      final int x = index % width;
      final int y = index / width;
      final int pixel = input.getRGB(x, y);

      final Color color = new Color(pixel);
      final Color grayColor = grayColor(color);
      output.setRGB(x, y, grayColor.getRGB());
    }

    return output;
  }
}
```

APÊNDICE B – Código-fonte da calculadora de fatoriais

Os blocos de código-fonte abaixo demonstram o código fonte completo do programa da calculadora de fatoriais analisado no capítulo 4.

```
1  package io.github.mateuspena.app2;
2
3  import io.github.mateuspena.app2.strategy.IFactorialCalculator;
4  import io.github.mateuspena.app2.strategy.impl.ForClassicCalculator;
5
6  import java.io.IOException;
7  import java.lang.management.ManagementFactory;
8  import java.util.List;
9  import java.util.concurrent.TimeUnit;
10
11 import static io.github.mateuspena.app2.domain.TextFile.readLines;
12 import static io.github.mateuspena.app2.domain.TextFile.writeLines;
13
14 public class Main {
15   public static void main(String[] args)
16       throws IOException, InterruptedException {
17     IFactorialCalculator strategy = new ForClassicCalculator();
18
19     List<String> numbers = readLines(args[0]);
20     writeLines("converted-numbers.txt", strategy.calculateAll(numbers));
21
22     System.out.println("* Time elapsed: " + timeElapsedInSeconds());
23     System.out.println("* Memory usage: " + memoryUsageInMegabytes());
24     TimeUnit.SECONDS.sleep(1);
25     System.exit(0);
26   }
27
28   private static long timeElapsedInSeconds() {
29     return ManagementFactory.getThreadMXBean().getCurrentThreadCpuTime()
30       / (1000 * 1000 * 1000);
31   }
32
33   private static long memoryUsageInMegabytes() {
34     return ManagementFactory.getMemoryMXBean().getHeapMemoryUsage().getUsed()
35       / (1024 * 1024);
36   }
37 }
```

```java
package io.github.mateuspena.app2.domain;

import java.math.BigInteger;

public abstract class Factorial {
  public static BigInteger calculate(String number) {
    return calculate(Long.parseLong(number));
  }

  public static BigInteger calculate(int number) {
    return calculate((long) number);
  }

  public static BigInteger calculate(long number) {
    return calculate(BigInteger.valueOf(number));
  }

  public static BigInteger calculate(BigInteger number) {
    BigInteger bigOne = BigInteger.ONE;

    BigInteger f = bigOne;
    for (BigInteger n = number; n.compareTo(bigOne) > 0; n = n.subtract(bigOne)) {
      f = f.multiply(n);
    }

    return f;
  }
}
```

```java
package io.github.mateuspena.app2.domain;

import java.io.*;
import java.util.List;

public abstract class TextFile {
  public static List<String> readLines(String filepath) throws IOException {
    BufferedReader reader = new BufferedReader(new FileReader(filepath));
    List<String> lines = reader.lines().toList();
    reader.close();

    return lines;
  }

  public static void writeLines(String filepath, List<String> lines) throws IOException {
    BufferedWriter writer = new BufferedWriter(new FileWriter(filepath));
    writer.write(String.join(System.lineSeparator(), lines));
    writer.close();
  }
}
```

```java
package io.github.mateuspena.app2.strategy;

import java.util.List;

public interface IFactorialCalculator {
  List<String> calculateAll(List<String> numbers);
}
```

```java
package io.github.mateuspena.app2.strategy.impl;

import io.github.mateuspena.app2.domain.Factorial;
import io.github.mateuspena.app2.strategy.IFactorialCalculator;

import java.util.ArrayList;
import java.util.List;

public class ForClassicCalculator implements IFactorialCalculator {
  @Override
  public List<String> calculateAll(List<String> numbers) {
    final List<String> factorials = new ArrayList<>();

    int size = numbers.size();
    for (int i = 0; i < size; i++) {
      factorials.add(Factorial.calculate(numbers.get(i)).toString());
    }

    return factorials;
  }
}
```

```java
package io.github.mateuspena.app2.strategy.impl;

import io.github.mateuspena.app2.domain.Factorial;
import io.github.mateuspena.app2.strategy.IFactorialCalculator;

import java.util.ArrayList;
import java.util.List;

public class ForEachCalculator implements IFactorialCalculator {
  @Override
  public List<String> calculateAll(List<String> numbers) {
    final List<String> factorials = new ArrayList<>();

    numbers.forEach(number -> factorials.add(Factorial.calculate(number).toString()));

    return factorials;
  }
}
```

```java
package io.github.mateuspena.app2.strategy.impl;

import io.github.mateuspena.app2.domain.Factorial;
import io.github.mateuspena.app2.strategy.IFactorialCalculator;

import java.util.ArrayList;
import java.util.List;

public class ForEachParallelCalculator implements IFactorialCalculator {
  @Override
  public List<String> calculateAll(List<String> numbers) {
    final List<String> factorials = new ArrayList<>();

    numbers.parallelStream()
        .forEach(number -> factorials.add(Factorial.calculate(number).toString()));

    return factorials;
  }
}
```

```java
package io.github.mateuspena.app2.strategy.impl;

import io.github.mateuspena.app2.domain.Factorial;
import io.github.mateuspena.app2.strategy.IFactorialCalculator;

import java.util.ArrayList;
import java.util.List;

public class ForEnhancedCalculator implements IFactorialCalculator {
  @Override
  public List<String> calculateAll(List<String> numbers) {
    final List<String> factorials = new ArrayList<>();

    for (String number : numbers) {
      factorials.add(Factorial.calculate(number).toString());
    }

    return factorials;
  }
}
```

```java
package io.github.mateuspena.app2.strategy.impl;

import io.github.mateuspena.app2.domain.Factorial;
import io.github.mateuspena.app2.strategy.IFactorialCalculator;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class IteratorCalculator implements IFactorialCalculator {
  @Override
  public List<String> calculateAll(List<String> numbers) {
    final List<String> factorials = new ArrayList<>();

    Iterator<String> iter = numbers.iterator();
    while (iter.hasNext()) {
      String number = iter.next();
      factorials.add(Factorial.calculate(number).toString());
    }

    return factorials;
  }
}
```