



UNIVERSIDADE DO MINHO
Departamento de Informática

Projeto de Engenharia Informática
Eficiência Energética das Linguagens de Programação

Feito por:

Dinis Gonçalves Estrada (PG53770)
Emanuel Lopes Monteiro da Silva (PG53802)
Simão Pedro Cunha Matos (PG54239)



PG53770



PG53802



PG54239

November 29, 2024
Ano Letivo 2024/25

Conteúdo

1	Introdução	3
2	Objetivos do Trabalho	3
3	Especificações de Hardware e Software	3
3.1	Versões dos Benchmarks utilizados	4
3.2	Versões das Linguagens Utilizadas	4
3.3	Escolha dos PowerCaps	5
3.4	Obtenção de Resultados	5
4	Programas Utilizados	6
4.1	Rust	6
4.2	Java	7
4.3	Python	7
4.4	Haskell	8
4.5	Ruby	8
4.6	C	9
5	Tratamento de Outliers e Dados obtidos	9
5.1	Colunas do CSV	9
5.2	Tratamento de Outliers	10
5.3	Dados Obtidos	10
5.3.1	Tipo de Dados	10
5.3.2	Correlação	11
5.3.3	Descrição de Valores	12
6	Análise de resultados	13
6.1	Análise geral de cada Linguagem	13
6.2	Consumo mínimo de cada Linguagem	16
6.2.1	C	16
6.2.2	Haskell	16
6.2.3	Java	17
6.2.4	Python	17
6.2.5	Ruby	18
6.2.6	Rust	18
6.3	Análise de consumo de cada programa	19
6.4	Energia vs Tempo	22
6.5	Comparação entre linguagens	23
6.6	Impacto no consumo total	23
7	Link do Projeto	23
8	Conclusão	24
9	Apêndice	25

Lista de Figuras

1	Especificações do Sistema Operativo/Máquina.	4
2	Exemplo de um Outlier	10
3	Exemplo de um Outlier relativo ao <i>Core</i>	10
4	Tipo dos dados	10
5	Tabela da Correlação	11
6	Correlação em Haskell	11
7	Descrição do conjunto dos datasets	12
8	Distruição dos valores de Core e Package (em Joules) por PowerLimit	13
9	Distruição dos valores de tempo (em segundos) por PowerLimit	14
10	Distruição dos valores de Temperatura (em °C) por PowerLimit	15
11	Valores de Package (em Joules) e Tempo (em segundos) comparados à execução não limitada- C	16
12	Valores de Package (em Joules) e Tempo (em segundos) comparados à execução não limitada-Haskell	16
13	Valores de Package (em Joules) e Tempo (em segundos) comparados à execução não limitada-Haskell	17
14	Valores de Package (em Joules) e Tempo (em segundos) comparados à execução não limitada-Python	17
15	Valores de Package (em Joules) e Tempo (em segundos) comparados à execução não limitada-Ruby	18
16	Valores de Package (em Joules) e Tempo (em segundos) comparados à execução não limitada-Rust	18
17	Comparação entre os diferentes programas- C	19
18	Comparação entre os diferentes programas- Haskell	19
19	Comparação entre os diferentes programas- Java	20
20	Comparação entre os diferentes programas- Ruby	20
21	Comparação entre os diferentes programas- Python	21
22	Comparação entre os diferentes programas- Rust	21
23	Comparação Energia vs Tempo	22
24	Energia vs Tempo - Regressão Linear	22
25	Programa omitido na comparação- Python	25
26	Programas omitidos na comparação- Rust	25

1 Introdução

No âmbito da Unidade Curricular Projeto de Informática foi nos proposta a investigação sobre a Eficiência Energética de determinadas Linguagens de Programação. Ao longo deste relatório, vamos então explicar como foram obtidas as medições de consumo energético de cada linguagem de programação, posteriormente a análise que fizemos sobre cada uma e por fim uma conclusão sobre o trabalho e qual a melhor opção energética para cada uma das linguagens.

2 Objetivos do Trabalho

Ao longo da resolução deste trabalho utilizamos vários benchmarks de várias linguagens e vários limites de uso de energia, para ver o impacto destes limites nos programas e de que maneira a linguagem seria afetada. Pretendemos, portanto, descobrir qual é a linguagem que é mais afetada pelas limitações de energia e como é que isso afeta o desempenho geral do programa. Para não cairmos na falácia de um único caso poder ser um outlier, testamos várias vezes cada programa e vários programas. Assim, teremos resultados mais fidedignos e portanto resultados e análises mais coerentes. Para aplicar a limitação de energia ao processador foi utilizado um programa chamado RAPL, fornecido pelo professor, que nos permitiu então recolher vários dados relativos às execuções dos programas- Foram escolhidas, as 6 seguintes linguagens/benchmarks devido à sua grande utilização nos dias de hoje e consequentemente as que são mais relevantes numa investigação como esta.

- Haskell - NoFib
- Java - Dacapo
- Python - Pyperformance
- C - Parsec
- Ruby - Benchmark Suite de Ruby
- Rust - N/A (Rosetta Code)

3 Especificações de Hardware e Software

Uma limitação deste programa RAPL, é apenas pudermos utiliza-lo em máquinas que possuem um processador Intel, diminuindo assim o número de máquinas disponíveis. Para além disto, uma das maneiras que utilizámos para que pudéssemos ter resultados consistentes foi a utilização sempre da mesma máquina. Na imagem abaixo vemos as várias características do sistema onde foram feitas as medições, entre as quais destacamos:

- Sistema Operativo : Ubuntu 24.04.1 LTS
- CPU : Intel i7-9750H 4.5 GHz
- Memória RAM : 16GB

```

emanuel@emanuel-Predator-PH315-52: ~
-----
OS: Ubuntu 24.04.1 LTS x86_64
Host: Predator PH315-52 V1.12
Kernel: 6.8.0-48-generic
Uptime: 24 mins
Packages: 1976 (dpkg), 13 (snap)
Shell: bash 5.2.21
Resolution: 1920x1080
DE: GNOME 46.0
WM: Mutter
WM Theme: Adwaita
Theme: Yaru-bark-dark [GTK2/3]
Icons: Yaru-bark [GTK2/3]
Terminal: gnome-terminal
CPU: Intel i7-9750H (12) @ 4.500GHz
GPU: Intel CoffeeLake-H GT2 [UHD Graphics 630]
GPU: NVIDIA GeForce GTX 1660 Ti Mobile
Memory: 2636MiB / 15827MiB

```

Figure 1: Especificações do Sistema Operativo/Máquina.

3.1 Versões dos Benchmarks utilizados

As versões dos benchmarks utilizadas são:

- Dacapo : 23.11-chopin (8 de Novembro de 2023)
- Pyperformance : 1.10 (22 de Outubro de 2023)
- NoFib : 7ffecc8115865fea9995a951091e6ff23cf8ca3a¹ (Janeiro de 2024)
- Parsec : 3.0 (7 de Julho de 2022)
- Ruby Benchmark Suite : 314ae79² (15 de Novembro de 2013)
- Rust : N/A (Rosetta Code)

3.2 Versões das Linguagens Utilizadas

As versões das linguagens/interpretadores utilizadas foram as seguintes:

- Java : 21.0.1 (17 outubro de 2023 LTS)
- Python : 3.10.12 (20 Novembro de 2023)
- Haskell : 9.4.7 (versão do GHC)
- C : 13.2.0 (versão do gcc)
- Ruby : 3.2.3 (2024-01-18 revision 52bb2ac0a6)
- Rust : 1.75.0 (versão do rustc 82e1608df 2023-12-21)

¹id do último commit no gitlab

²id do último commit no github

3.3 Escolha dos PowerCaps

A ferramenta utilizada por nós, o RAPL, permite limitar o consumo de energia por parte do CPU. Este limite serve para melhor termos a performance não só da nossa máquina bem como das ferramentas consoante as diferentes características. Para isto, tivemos de escolher diferentes PowerCaps dependendo da nossa máquina. Portanto, tivemos que encontrar uma maneira para escolher os PowerCaps. Inicialmente começamos por executar o algoritmo de fibonacci em python e em C e repararmos que tanto numa linguagem como noutra o limite 4. Por isso, tomamos este valor como ponto de referência e escolhemos os valores mais próximos. Uma vez que estamos a testar linguagens de *low level* e *high level* abrimos um pouco mais o nosso range de limites para que pudéssemos visualizar a performance destes dois tipos de linguagens consoante o aumento do *PowerCap* ficando então com os seguintes valores:

- -1 (sem qualquer limite)
- 1 2 3 4 5 6 7 8 9 10 15 20 (limites intermédios para ver a performance das linguagens)

3.4 Obtenção de Resultados

Após as escolhas dos PowerCaps e dos programas fizemos uma script (run.sh) para executar a medição de energia, de memória e temporal dos Benchmarks. Executamos 10 vezes cada programa/problema do benchmark. Escolhemos este valor para ao mesmo tempo ser fiável (a influência de *outliers* é diluída com o aumento do número de execuções) e não muito demorado, preferindo mais apostar na diversidade de programas (veremos na análise de resultados mais informações). Como para cada linguagem/benchmark executa à volta de 10 programas/problemas e cada programa executa 10 vezes para os 13 PowerCaps cada linguagem vai gerar um CSV de cerca de 1100 linhas (em que a primeira delas é de cabeçalho). É de mencionar que a parte de Python foi a mais demorada, seguida da de Java, Haskell, C, Ruby e por último Rust. Apesar de os benchmarks na script estarem todos seguidos, eles foram medidos em 3 fases (uma para cada linguagem), todos nas mesmas condições: durante a noite depois de o computador estar desligado antes 2 horas. As 3 fases significam que cada uma das Linguagens/Benchmarks foi medido uma por cada noite. Assim garantimos não só uma maior uniformização de experiências entre linguagem, mas também resultados menos comprometidos por, por exemplo, atividades externas, como a luz diária possivelmente aumentar a temperatura. Decidiu-se manter assim no run.sh tudo junto para ilustrar como poderia obter-se os resultados todos de uma vez, que infelizmente só não o fizemos assim devido ao tempo que iria demorar.

4 Programas Utilizados

Vamos agora explicar a escolha dos programas para cada uma das linguagens. É importante realçar que muitos destes programas foram escolhidos por diversas razões, estando entre elas:

- Havia programas que não funcionavam corretamente com a versão da linguagens instaladas na máquina onde foram testado. Um exemplo disto é o *Cassandra*, que apenas funcionava com Java 17. Outro exemplo eram os programas *parallel* do Haskell.
- Conhecimento e curiosidade sobre alguns dos programas, como o Kafka.
- Devido a não conseguirmos realisticamente testar todos os programas, nem acrescentaria muito ao nosso trabalho, os restantes foram escolhidos consoante medições rudimentares feitas antes da obtenção final dos resultados. Estas medições não foram feitas muito rigorosamente, porque apenas tinham o objetivo de ajudar na escolha entre alguns programas disponíveis.

4.1 Rust

Uma vez que não existia um benchmark suite para Rust como para as outras linguagens presentes optamos por selecionar algoritmos conhecidos pela sua complexidade. Para que houvesse algum rigor relativamente aos inputs de cada programa decidimos utilizar os mesmos que outros papers já feitos para assim também podermos ter a possibilidade de comparação.

Programa	Descrição	Input
MergeSort	Ordenar uma coleção de inteiros utilizando o algoritmo merge sort	10k números inteiros aleatórios
QuickSort	Ordenar uma coleção de inteiros utilizando o algoritmo quick sort	10k números inteiros aleatórios
Hailstone	Gerar a sequência hailstone para números específicos	*Rosetta
Fibonacci	Calcular o número de Fibonacci	fib(47)
Ackermann	Calcular a Função de Ackermann	*Rosetta
N-Queens Problem	Resolver o problema n-queens	12-rainhas
100-doors	Resolver o problema das 100 portas	*Rosetta
Remove duplicates	Remover elementos duplicados de uma sequência	2^{17} elementos aleatórios
Sieve of Eratosthenes	Executar o algoritmo que encontra os números primos até um dado inteiro	10k

Table 1: Descrições dos Benchmarks em Rust

4.2 Java

Relativamente ao Java, como foi dito devido a esta linguagem ser bastante conhecida pelo suporte a versões antigas, alguns destes programas não funcionavam na versão 21. A maioria funcionava com Java 17, mas esta amplitude ia ser absorvida pelas melhorias entre o 17 e o 21.

Programa	Descrição
Avrora	Simulação e análise de programas para microcontroladores AVR
Batik	Ferramenta para processamento e renderização de imagens SVG
BioJava	Framework para computação biológica
Eclipse	Ambiente de Desenvolvimento Integrado (IDE)
Spring	Framework de desenvolvimento de aplicações empresariais em Java
Tomcat	Servidor web para execução de Servlets Java e JSPs
GraphChi	Framework para computação em grafos
JME (jMonkeyEngine)	Framework para desenvolvimento de jogos
Jython	Implementação de Python para a plataforma Java
Kafka	Sistema de mensagens distribuído

Table 2: Descrições dos Benchmarks em Java

4.3 Python

Já o pyperformance tinha um total de 84 programas para benchmark. Decidimos então escolher 10. Primeiramente, o pyperformance possui vários grupos e dentro destes grupos decidimos escolher o grupo apps que contém os programas mais exaustivos temporalmente e energicamente e também mais um ou dois de cada um dos grupos async, templates e math.

Programa	Descrição
Chameleon	Um motor de templates para Python
Docutils	Processa documentação em texto simples para formatos úteis como HTML ou LaTeX
HTML5lib	Uma biblioteca para análise de HTML e XHTML em Python
Deepcopy	Copia objetos de forma recursiva em Python
Tornado_HTTP	Lida com pedidos HTTP assíncronos
N-body	Simula a interação gravitacional de um grupo de corpos
JSON.dumps	Serializa objetos Python para o formato JSON
PiDigits	Calcula dígitos do número Pi
Async_tree	Gere e executa tarefas assíncronas numa estrutura em árvore
Django_template	Um motor de templates para construir páginas web dinâmicas usando Django

Table 3: Descrições dos Benchmarks em Python

4.4 Haskell

Para haskell foi a escolha foi mais arbitrária. Tentamos ter uma representação mais ou menos uniforme dos vários grupos de benchmark, como por exemplo o grep e compress do real, o sorting do spectral, o rfib do imaginary e o fannkuch-redux do shootout. Temos de mencionar que tal como no caso do Java havia programas que não conseguimos utilizar como os programas do grupo parallel. É também de mencionar o facto de estes serem programas muito menos exigentes do que os das outras linguagens de programação.

Programa	Descrição
spectral/sorting	Realiza análise espectral e operações de ordenação
real/grep	Simula o comando ‘grep’ para pesquisa de padrões em texto
real/compress	Comprime dados utilizando Haskell
real/compress2	Uma implementação alternativa de compressão de dados
real/gg	Realiza transformações de dados baseadas em grafos
real/rsa	Executa encriptação e decriptação RSA
imaginary/rfib	Calcula números de Fibonacci utilizando um algoritmo recursivo
shootout/binary-trees	Avalia a criação e manipulação de árvores binárias
shootout/fannkuch-redux	Executa o benchmark Fannkuch-redux
shootout/spectral-norm	Calcula a norma espectral de uma matriz

Table 4: Descrições dos Benchmarks em Haskell

4.5 Ruby

Relativamente a Ruby encontramos um benchmark suite na plataforma GitHub bastante utilizado pela comunidade. E por isso limitamos nos a utilizar os benchmark padrão propostos pelo mesmo.

Programa	Descrição
app_objects	Benchmark focado na instanciação e manipulação de objetos.
app_fib	Calcula números de Fibonacci recursivamente para testar o desempenho das chamadas de função.
app_mandelbrot	Gera um fractal de Mandelbrot, testando a eficiência computacional para números complexos.
app_sieve	Implementa o Crivo de Eratóstenes para encontrar números primos até um dado número inteiro.
app_strconcat	Concatena strings repetidamente para avaliar o desempenho do manuseio de strings.
app_tak	Executa a função Tak, testando a profundidade de recursividade e ramificação.
app_tarai	Calcula a função Tarai (Tak com memorização), testando a recursividade otimizada.
app_whileloop	Executa um simples loop while para testar o desempenho da iteração e controlo de fluxo.

Table 5: Descrições dos Benchmarks em Ruby

4.6 C

Por último, C é o que apresenta apenas 6 programas ao invés dos habituais 10 como nos outros benchmarks. O motivo deste número reduzido é que este benchmark serve tanto para C como C++ e por isso os programas que ficaram por usar deste benchmark foram aqueles que eram exclusivamente a direcionados para C++.

Programa	Descrição
blackscholes	Precificação de opções financeiras usando o modelo Black-Scholes.
canneal	Algoritmo de otimização baseado em recozimento simulado.
streamcluster	Resolve o problema de clustering online.
swaptions	Precificação de derivativos de taxas de juros usando Monte Carlo.
freqmine	Mineração de itemsets frequentes em bases de dados transacionais.
fluidanimate	Simula dinâmica de fluidos usando Hidrodinâmica de Partículas Suavizadas (SPH).

Table 6: Descrições dos Benchmarks em C

5 Tratamento de Outliers e Dados obtidos

5.1 Colunas do CSV

A seguir encontra-se uma descrição sobre cada coluna presente nos vários CSVs obtidos:

- **Language:** Linguagem de programação do Benchmark dos programas.
- **Program:** Nome do programa/problema que correu;
- **Package:** Consumo de energia da socket inteira (o consumo de todos os cores, GPU e componentes externas aos cores) (em Joules).
- **Core:** Consumo de energia de todos os cores e caches dos mesmos (em Joules).
- **GPU:** Consumo de energia pelo GPU.
- **DRAM:** Consumo de energia pela RAM.
- **Time:** Tempo de execução do problema/programa (em ms).
- **Temperature:** Temperatura média em todos os cores (em °C);
- **Memory:** Total de memória gasta durante a execução do programa (em KBytes);
- **PowerLimit:** Limite de potência dos cores (em Watts).

5.2 Tratamento de Outliers

A colheita dos dados é sempre afetada devido à existência de *outliers*, tal acontece pois o *RAPL* pode originar erros. A imagem seguinte é um exemplo de um outlier.

```
Java, biojava, 20,204.874389648437500000, 187.475463867187500000, , , 10271, 54.8, 970028
Java, biojava, 20,-261938.806030273437500000, 187.805480957031250000, , , 10292, 49.0, 804004
Java, biojava, 20,200.618408203125000000, 183.582031250000000000, , , 10057, 48.7, 779036
```

Figure 2: Exemplo de um Outlier

Para eliminar os *outliers* de cada CSV, originado por cada linguagem, iterou-se sobre cada programa e cada *PowerCap* e ordenou-se por *Package* (consumo total de energia) e eliminou-se os 2 maiores e os 2 menores valores (dando um total 6 valores para cada conjunto de programa e *PowerCap*).

Isto verificou-se para todas as linguagens exceto para Python que apresentou outliers também na coluna do *Core*, e portanto decidiu-se para cada par programa e *PowerCap* ordenar por *Package* e remover o maior e os 2 menores valores e em seguida ordenou-se por *Core* e removeu-se o maior e os 2 menores valores, tal decisão permitiu-nos não remover demasiadas linhas do CSV de python.

Em seguida temos um exemplo de *outlier* na coluna do *Core* no CSV de python.

```
Python, docutils, 20,4528.222961425781250000, 4176.769958496093750000, , , 227140, 54.5, 62648
Python, docutils, 20,4557.385681152343750000, -257939.960083007812500000, , , 228577, 56.0, 62740
Python, docutils, 20,4564.994873046875000000, 4211.328491210937500000, , , 228944, 53.3, 62772
```

Figure 3: Exemplo de um Outlier relativo ao *Core*

5.3 Dados Obtidos

5.3.1 Tipo de Dados

Os tipos de dados presentes em cada CSV é o seguinte:

Language	object
Program	object
PowerLimit	int64
Package	float64
Core	float64
GPU	object
DRAM	object
Time	int64
Temperature	float64
Memory	int64

Figure 4: Tipo dos dados

É importante mencionar que como não se obteve valores de GPU e de DRAM em nenhum CSV e por isso decidiu-se remove-los do dataset.

5.3.2 Correlação

A imagem a seguir apresenta a tabela de correlação dos 6 datasets, correspondentes a cada linguagem, juntos.

	PowerLimit	Package	Core	Time	Temperature	Memory
PowerLimit	1.000000	0.018028	0.050043	-0.082472	0.333724	0.002434
Package	0.018028	1.000000	0.974398	0.745993	0.226182	0.129884
Core	0.050043	0.974398	1.000000	0.577270	0.303725	0.124267
Time	-0.082472	0.745993	0.577270	1.000000	-0.080920	0.088233
Temperature	0.333724	0.226182	0.303725	-0.080920	1.000000	0.242034
Memory	0.002434	0.129884	0.124267	0.088233	0.242034	1.000000

Figure 5: Tabela da Correlação

Vemos uma correlação significativa entre a temperatura e o *PowerLimit*.

Ao analisar cada um dos datasets em separado, vemos uma poluição de uns aos outros. Por exemplo, na linguagem Haskell, vemos alguns atributos com correlações diferentes:

	PowerLimit	Package	Core	Time	Temperature	Memory
PowerLimit	1.000000	0.092745	0.260253	-0.417257	0.528636	-0.022550
Package	0.092745	1.000000	0.930061	0.326694	0.407936	0.018240
Core	0.260253	0.930061	1.000000	-0.042731	0.658377	-0.004098
Time	-0.417257	0.326694	-0.042731	1.000000	-0.585719	0.057742
Temperature	0.528636	0.407936	0.658377	-0.585719	1.000000	-0.043648
Memory	-0.022550	0.018240	-0.004098	0.057742	-0.043648	1.000000

Figure 6: Correlação em Haskell

Como iremos tratar cada linguagem em separado estes problemas serão diluídos.

5.3.3 Descrição de Valores

Está representada aqui na tabela mais dados sobre a combinação dos 6 datasets:

	PowerLimit	Package	Core	Time	Temperature	Memory
count	3874.000000	3874.000000	3874.000000	3.874000e+03	3874.000000	3.874000e+03
mean	6.846154	127.583489	97.276551	1.925401e+04	37.362855	1.479241e+05
std	5.559441	382.687209	311.135884	6.947331e+04	5.708968	3.024399e+05
min	-1.000000	0.040710	0.021790	2.000000e+00	28.800000	8.960000e+02
25%	3.000000	0.089249	0.076187	1.000000e+01	34.300000	2.048000e+03
50%	6.000000	40.467255	25.373810	4.783000e+03	35.800000	7.424000e+03
75%	9.000000	133.045685	93.866241	1.878075e+04	39.000000	5.580800e+04
max	20.000000	5365.804077	5032.109680	1.109980e+06	67.700000	1.995604e+06

Figure 7: Descrição do conjunto dos datasets

6 Análise de resultados

6.1 Análise geral de cada Linguagem

De forma a perceber como o uso de powercaps afeta cada linguagem de programação, começamos por analisar os resultados obtidos para cada uma delas. O objetivo principal é inferir como podemos minimizar o consumo de energia e, com base nessas conclusões, comparar as diferentes linguagens entre si.

Para tal, calculamos a média dos resultados por PowerLimit, agregando todas as execuções dos programas. Desta forma, garantimos que o peso de cada execução é uniformemente distribuído, evitando que o desempenho superior ou inferior de um único programa distorça a análise dos restantes.

É importante referir que os programas escritos em C, Rust e Ruby apresentam tempos de execução significativamente mais curtos, o que, apesar de ser uma vantagem pela rapidez na obtenção de resultados, pode aumentar o ruído nas medições. Além disso, a curta duração limita a estabilização de fatores como temperatura e consumo energético, tornando as medições menos consistentes. Em contraste, programas mais longos, como os de Python ou Java, permitem medições mais estáveis, já que o sistema tem tempo suficiente para atingir um estado mais uniforme.

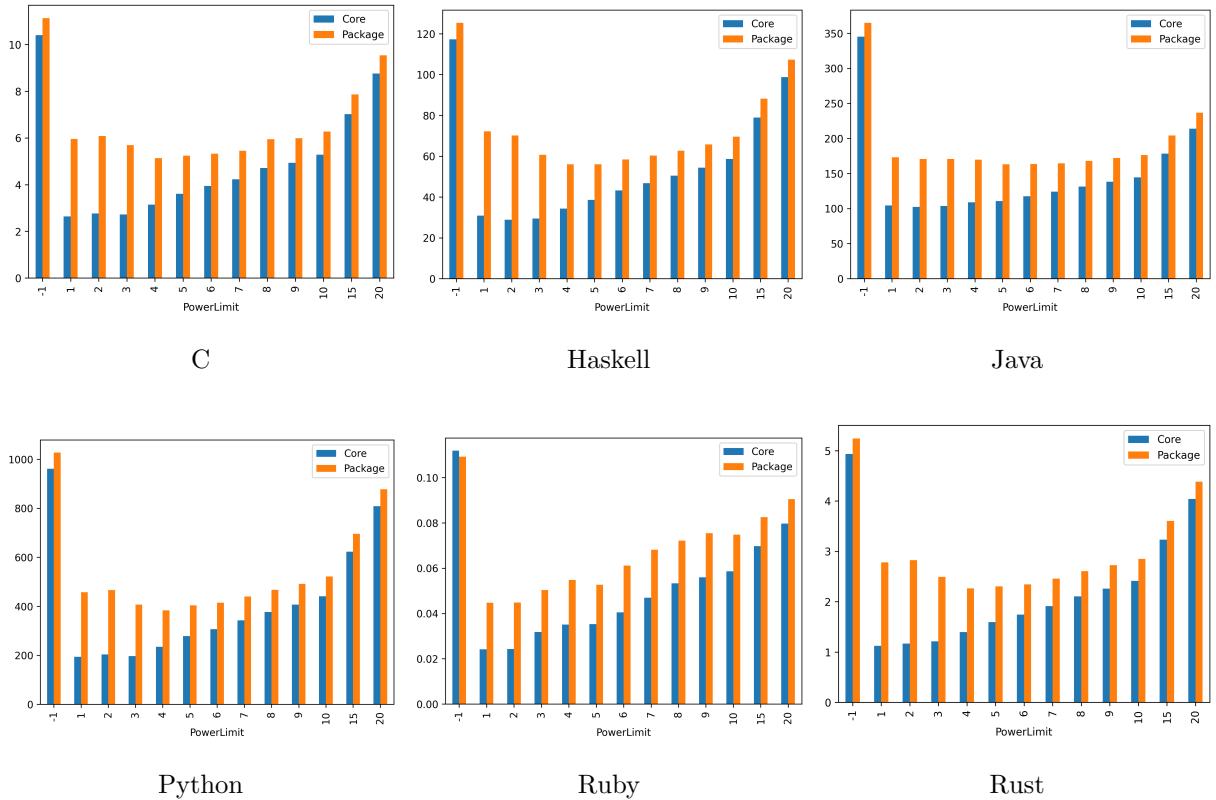


Figure 8: Distribuição dos valores de Core e Package (em Joules) por PowerLimit

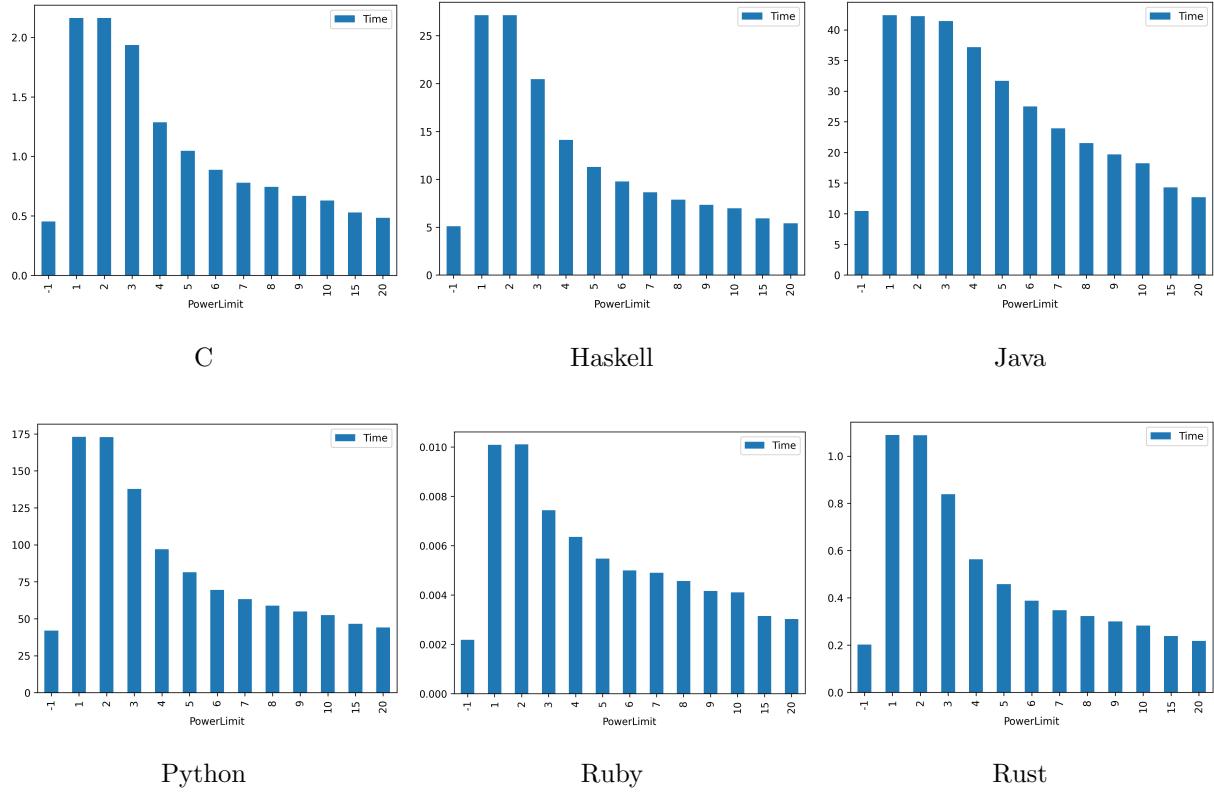


Figure 9: Distribuição dos valores de tempo (em segundos)por PowerLimit

Como seria de esperar, para todas as linguagens, a execução que não limita o processador foi a pior em termos de consumo de energia e a melhor em termos de tempo.

Quanto às execuções onde limitamos o processador podemos verificar que, quanto menor o valor de PowerLimit:

- Menor será a energia consumida pelo processador e, de um modo geral, a energia total.
- Maior será o tempo gasto na execução.

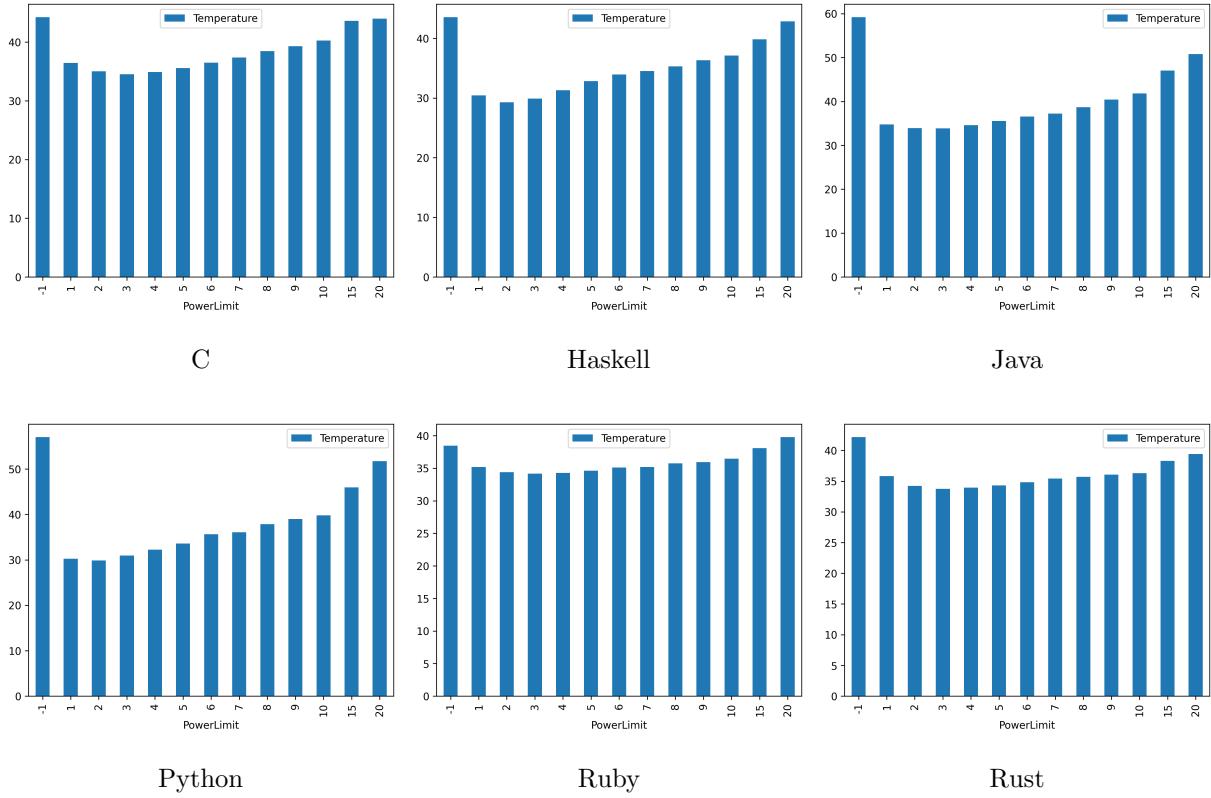


Figure 10: Distribuição dos valores de Temperatura (em °C) por PowerLimit

É possível observar que a temperatura é diretamente influenciada pela potência disponível do processador. Com PowerLimits mais altos, o hardware dissipava mais calor. Algumas linguagens apresentam uma maior diferença entre as temperaturas, e acreditamos que isso esteja relacionado com a duração das execuções, como referido anteriormente.

6.2 Consumo mínimo de cada Linguagem

6.2.1 C

PowerLimit	Package	Dif Package (%)	Time	Dif Temporal
-1	11.128157	100.000000	0.453472	1.000000
1	5.955098	53.513790	2.162500	4.768760
2	6.084023	54.672336	2.162306	4.768331
3	5.695574	51.181644	1.936472	4.270322
4	5.137872	46.170014	1.286167	2.836263
5	5.237111	47.061804	1.047750	2.310505
6	5.325261	47.853939	0.887194	1.956447
7	5.448785	48.963946	0.778472	1.716692
8	5.941201	53.388905	0.743889	1.640429
9	5.985482	53.786824	0.667111	1.471118
10	6.269833	56.342062	0.629944	1.389158
15	7.857964	70.613350	0.529333	1.167289
20	9.536257	85.694843	0.483889	1.067075

Figure 11: Valores de Package (em Joules) e Tempo (em segundos) comparados à execução não limitada- C

O valor de PowerLimit que minimiza a energia gasta para C é 4. Neste caso, o valor de package é 46,17% do valor base (sem powercap), o que resulta em uma redução de 53,83% no consumo de energia. Em contrapartida, o tempo de execução aumentou em 2,83 vezes.

6.2.2 Haskell

PowerLimit	Package	Dif Package (%)	Time	Dif Temporal
-1	125.314375	100.000000	5.101367	1.000000
1	72.171014	57.591968	27.141317	5.320401
2	70.182827	56.005408	27.140333	5.320208
3	60.634400	48.385830	20.456050	4.009916
4	56.001513	44.688818	14.112533	2.766422
5	56.063071	44.737941	11.283467	2.211852
6	58.380085	46.586902	9.782850	1.917692
7	60.272232	48.096822	8.650983	1.695817
8	62.650005	49.994269	7.864067	1.541561
9	65.803632	52.510841	7.339883	1.438807
10	69.559067	55.507652	6.981617	1.368578
15	88.225521	70.403352	5.903533	1.157245
20	107.311312	85.633681	5.391250	1.056825

Figure 12: Valores de Package (em Joules) e Tempo (em segundos) comparados à execução não limitada-Haskell

Para Haskell, a configuração de PowerLimit mais eficiente em termos de consumo de energia é 4. Desta forma, o valor de package corresponde a 44,69% do valor base, o que representa uma economia de 55,31%. O tempo de execução foi 2,77 vezes maior em comparação com o valor base.

6.2.3 Java

PowerLimit	Package	Dif Package (%)	Time	Dif Temporal
-1	364.758634	100.000000	10.439700	1.000000
1	172.950969	47.415182	42.385150	4.059997
2	170.591030	46.768195	42.218783	4.044061
3	170.726019	46.805203	41.420633	3.967608
4	169.624711	46.503275	37.191583	3.562515
5	163.077517	44.708336	31.689933	3.035521
6	163.470474	44.816067	27.500167	2.634191
7	164.411004	45.073917	23.935500	2.292738
8	168.087423	46.081821	21.503450	2.059777
9	172.172087	47.201648	19.670850	1.884235
10	176.212289	48.309285	18.215417	1.744822
15	204.134487	55.964264	14.309617	1.370692
20	236.813318	64.923293	12.697083	1.216231

Figure 13: Valores de Package (em Joules) e Tempo (em segundos) comparados à execução não limitada-Haskell

O PowerLimit ideal para minimizar a energia gasta em Java é 5, onde o package representa 44,71% do valor base, proporcionando uma redução de 55,29%. Contudo, o aumento no tempo de execução foi de 3,04 vezes.

6.2.4 Python

PowerLimit	Package	Dif Package (%)	Time	Dif Temporal
-1	1027.150407	100.000000	42.029550	1.000000
1	456.897949	44.482088	172.966500	4.115355
2	466.275928	45.395097	172.832150	4.112158
3	407.180782	39.641787	137.812325	3.278939
4	383.556348	37.341790	96.953250	2.306788
5	403.540739	39.287405	81.412675	1.937034
6	414.045058	40.310071	69.502850	1.653666
7	439.831749	42.820579	63.218475	1.504144
8	467.517773	45.515999	58.750550	1.397839
9	491.045988	47.806629	54.817300	1.304256
10	521.533699	50.774813	52.373750	1.246117
15	695.673772	67.728520	46.505450	1.106494
20	877.223618	85.403619	44.049250	1.048054

Figure 14: Valores de Package (em Joules) e Tempo (em segundos) comparados à execução não limitada-Python

Para Python, o valor de PowerLimit que reduz a energia consumida de forma mais eficiente é 4. Nessa configuração, o valor de package é 37,34% do valor sem powercap, resultando numa economia de 62,66%. O aumento no tempo de execução foi de 2,31 vezes.

6.2.5 Ruby

PowerLimit	Package	Dif Package (%)	Time	Dif Temporal
-1	0.109244	100.000000	0.002188	1.000000
1	0.044682	40.900679	0.010083	4.609524
2	0.044847	41.051994	0.010104	4.619048
3	0.050383	46.119912	0.007437	3.400000
4	0.054747	50.114651	0.006354	2.904762
5	0.052706	48.246482	0.005479	2.504762
6	0.061176	55.999674	0.005000	2.285714
7	0.068106	62.343301	0.004896	2.238095
8	0.072206	66.095934	0.004562	2.085714
9	0.075413	69.031462	0.004167	1.904762
10	0.074841	68.507676	0.004104	1.876190
15	0.082545	75.560160	0.003146	1.438095
20	0.090471	82.815173	0.003021	1.380952

Figure 15: Valores de Package (em Joules) e Tempo (em segundos) comparados à execução não limitada-Ruby

Em Ruby, a melhor configuração de PowerLimit é 1, com o valor de package a alcançar 40,9% do valor base. Isso implica um ganho de 59,1% na economia de energia. O tempo de execução aumentou 4,6 vezes.

6.2.6 Rust

PowerLimit	Package	Dif Package (%)	Time	Dif Temporal
-1	5.240846	100.000000	0.202815	1.000000
1	2.782483	53.092253	1.089056	5.369704
2	2.826546	53.933011	1.088444	5.366691
3	2.491996	47.549509	0.839000	4.136779
4	2.264133	43.201672	0.563407	2.777940
5	2.301569	43.915985	0.458185	2.259131
6	2.342941	44.705393	0.388056	1.913349
7	2.456946	46.880724	0.347815	1.714938
8	2.607059	49.745005	0.323185	1.593499
9	2.727613	52.045273	0.300333	1.480825
10	2.851865	54.416128	0.283222	1.396457
15	3.605440	68.795011	0.238889	1.177867
20	4.380993	83.593238	0.218074	1.075237

Figure 16: Valores de Package (em Joules) e Tempo (em segundos) comparados à execução não limitada-Rust

O PowerLimit que proporciona a maior redução no consumo de energia em Rust é 4, com o valor de package atingindo 43,2% do valor base. Essa configuração resulta em uma economia de 56,8%. O aumento do tempo de execução foi de 2,78 vezes.

6.3 Análise de consumo de cada programa

Decidimos também analisar de que forma o valor de PowerLimit afeta a execução individual de cada programa. Por questões de facilidade de visualização, para Python e Rust, omitimos alguns programas, gráficos referentes a essas execuções encontram-se em apêndice.

Após realizar os testes, concluímos que para todas as linguagens, as tendências anteriormente observadas se repetem, o que reforça a forte relação entre a limitação de potência, o consumo de energia e o desempenho dos programas.

De maneira geral, conforme o PowerLimit é reduzido, há uma diminuição do consumo de energia, o que está diretamente ligado à redução da potência disponível para o processador. No entanto, essa diminuição na energia consumida leva, geralmente, a um aumento no tempo de execução, como observado nas diferentes linguagens de programação. Esse comportamento é consistente independentemente da linguagem, indicando que a relação entre limitação de potência e desempenho é uma característica comum à grande maioria dos testes realizados.

Porém, existem alguns casos, como `json.dumps` do `pyperformance`, em que não há uma alteração significativa no tempo de execução, apesar da redução no consumo de energia.

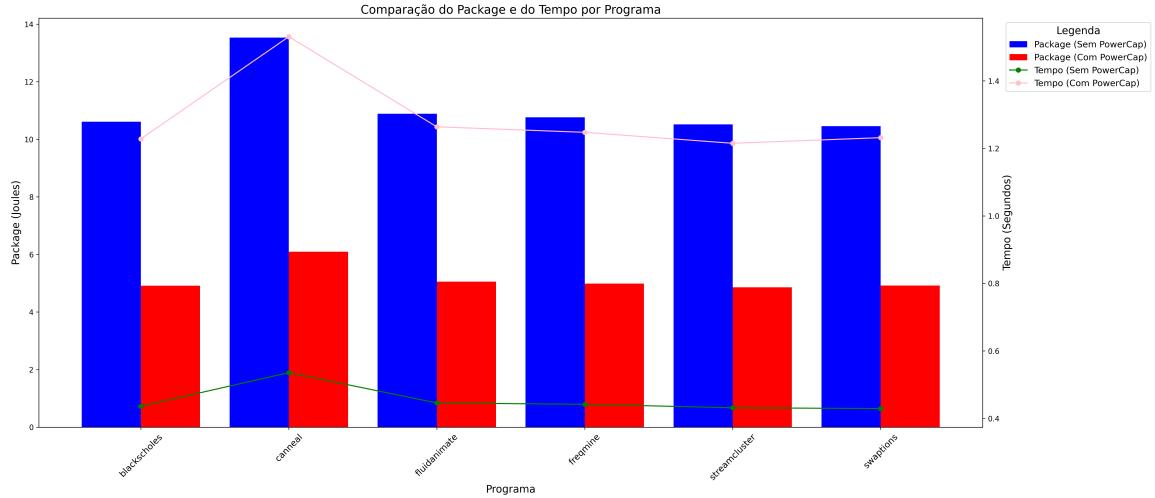


Figure 17: Comparação entre os diferentes programas- C

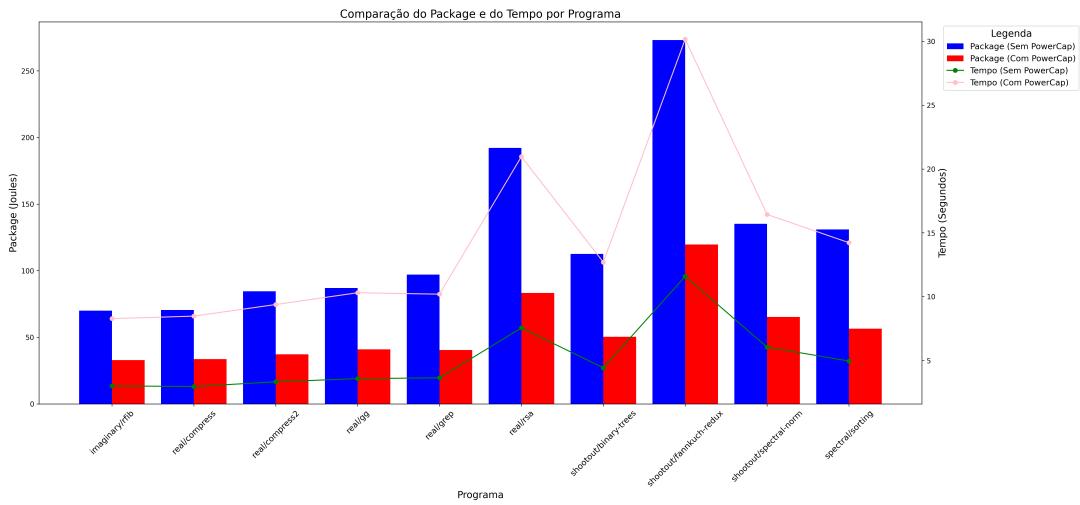


Figure 18: Comparação entre os diferentes programas- Haskell

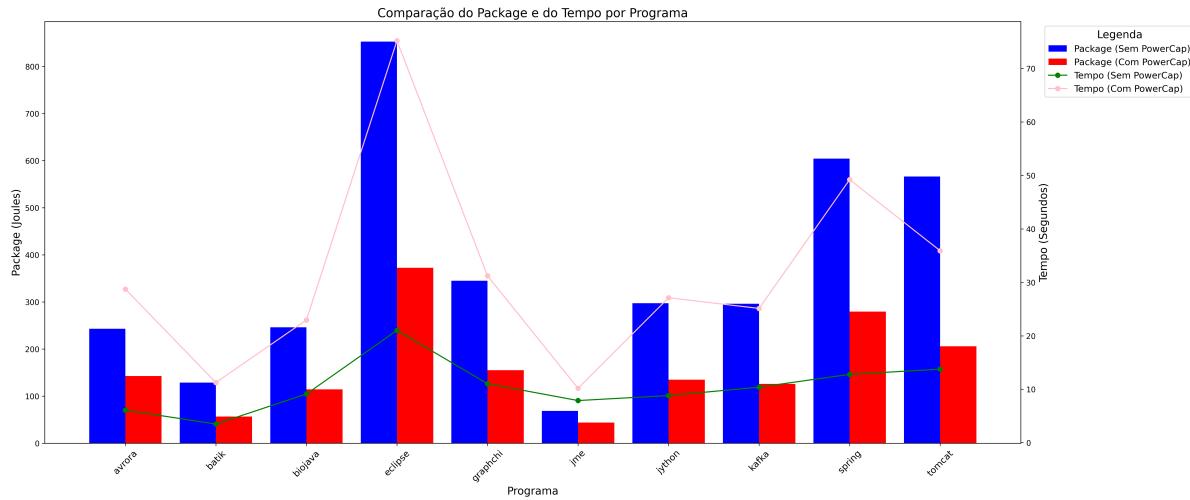


Figure 19: Comparação entre os diferentes programas- Java

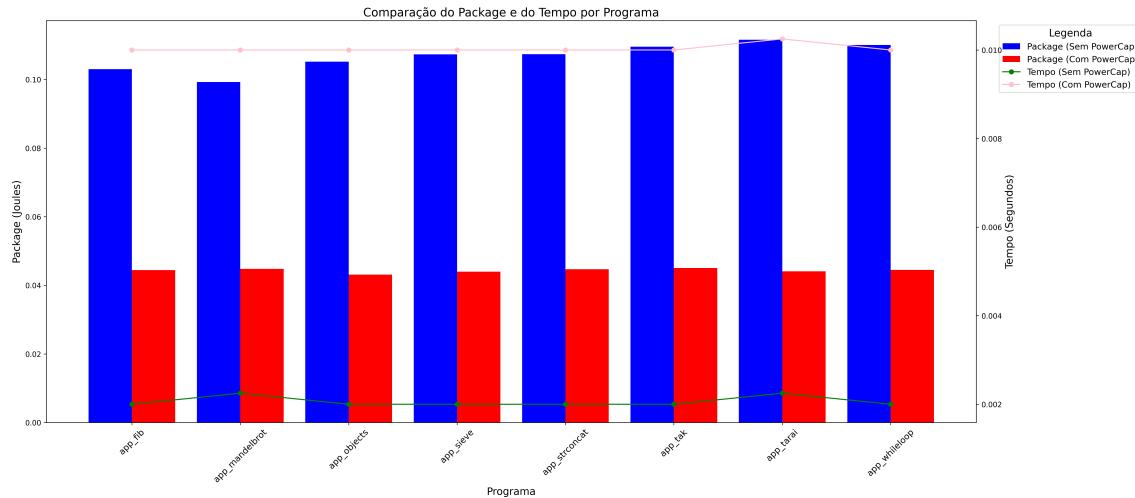


Figure 20: Comparação entre os diferentes programas- Ruby

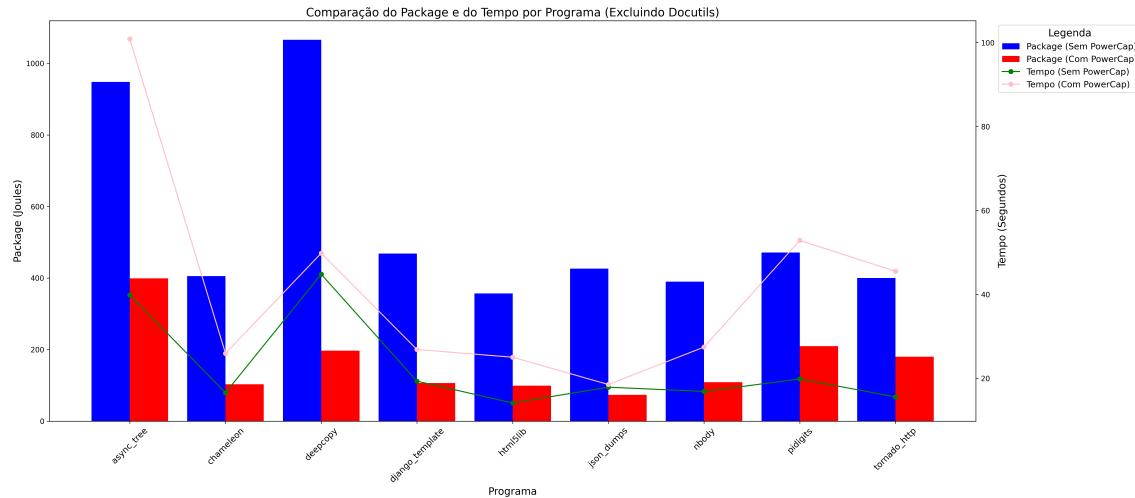


Figure 21: Comparação entre os diferentes programas- Python

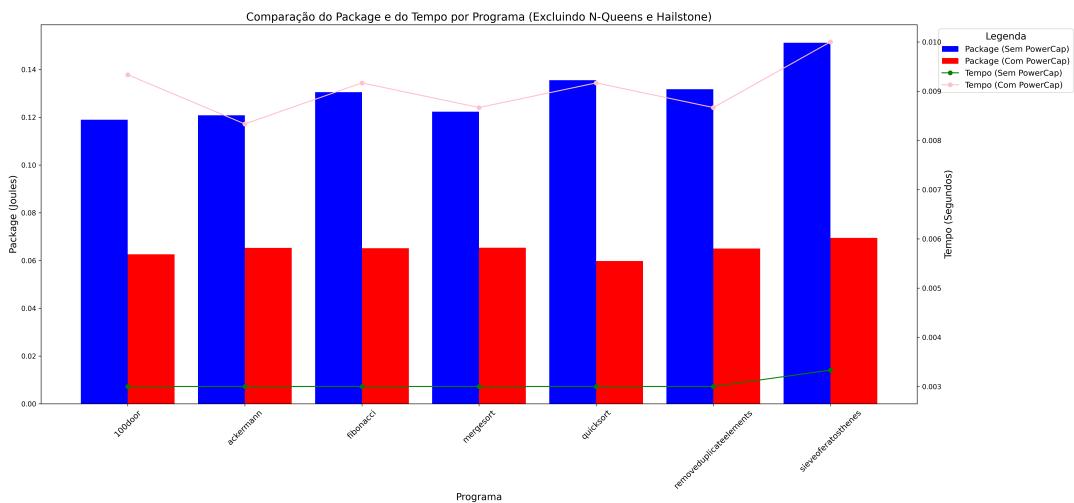


Figure 22: Comparação entre os diferentes programas- Rust

6.4 Energia vs Tempo

O gráfico a seguir representa a energia total consumida por cada programa de cada linguagem, com e sem *PowerCap*, em função do tempo.

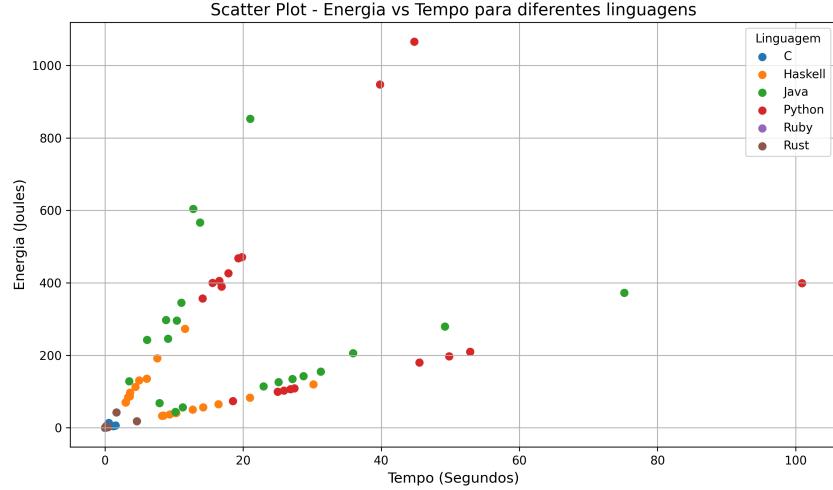


Figure 23: Comparação Energia vs Tempo

Pelo gráfico retira-se que os programas de Python e Java consomem mais energia ao contrário de C e Ruby que além de consumirem menos também são executados em muito menos tempo.

O gráfico seguinte representa duas regressões lineares, uma para os programas executados com *PowerCap* e outra para os programas sem *PowerCap*.

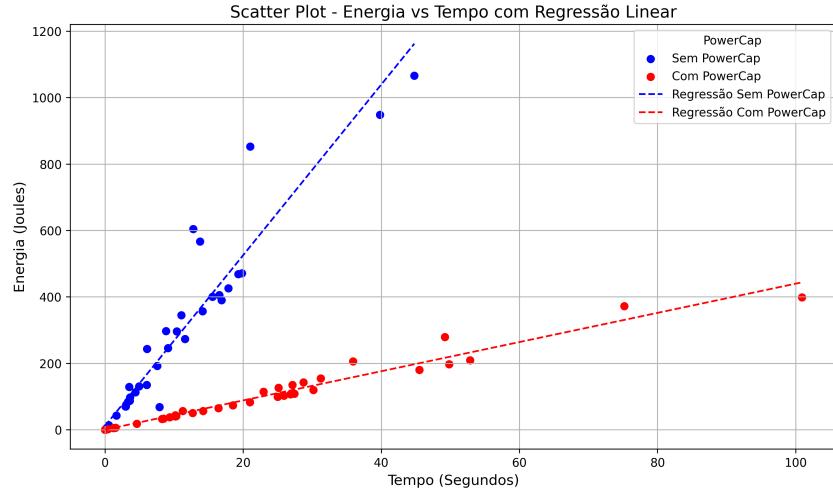


Figure 24: Energia vs Tempo - Regressão Linear

Retiram-se várias conclusões relativamente ao gráfico, a mais evidente é que à medida que o tempo de execução aumenta, o consumo de energia sem *PowerCap* cresce de forma muito mais acentuada do que com *PowerCap*.

Isto indica que o uso de *PowerCap* é eficaz em controlar e limitar o consumo energético ao longo do tempo, especialmente em execuções mais longas. Sem o *PowerCap*, o crescimento do consumo é quase exponencial, enquanto com o *PowerCap* o aumento é linear e muito mais moderado. Esta diferença destaca a importância de implementar o *PowerCap* em sistemas onde a eficiência energética e o controlo do consumo são cruciais.

6.5 Comparação entre linguagens

Linguagem	Ganho de Energia (%)	Aumento do Tempo de Execução (vezes)
Python	62.66	2.31
Ruby	59.1	4.60
Java	55.29	3.04
Haskell	55.31	2.77
Rust	56.80	2.78
C	53.83	2.83

Table 7: Comparação de linguagens com base no ganho de economia de energia e aumento do tempo de execução.

A tabela apresentada compara seis linguagens de programação com base no ganho de economia de energia e no aumento do tempo de execução resultante da aplicação de powercaps. Os valores de ganho de energia mostram a redução percentual no consumo em relação ao valor base (sem powercap), enquanto a coluna aumento do tempo de execução indica quantas vezes o tempo de execução aumentou em comparação ao valor base.

Observa-se que a linguagem Python proporciona o maior ganho de energia, com uma economia de 62,66%, e também o menor aumento no tempo de execução (2,31 vezes). Em contraste, a linguagem Ruby apresenta um ganho de energia de 59,1%, mas o aumento no tempo de execução é significativamente mais alto (4,60 vezes). Outras linguagens, como Haskell, Java, Rust e C, seguem com ganhos de energia em torno de 53% a 56%, e aumentos no tempo de execução entre 2,7 e 3 vezes.

6.6 Impacto no consumo total

	Energia Total (J)	Tempo de Execução Total (s)
Sem PowerCap	71179	2641
Com PowerCap	28796	6703
Diferença	-42383	+4062

Table 8: Comparação da energia e tempo totais com e sem powercaps.

Por fim, comparámos todas as execuções de todos os programas nos cenários com e sem PowerCap. Sem PowerCap, o consumo de energia total foi de 71179 joules, enquanto com PowerCap esse valor foi reduzido para 28796 joules, representando uma economia de 59,5%.

Em contrapartida, o tempo total de execução aumentou de 2641 segundos para 6703 segundos, ou seja, cerca de 2,54 vezes. Estes resultados destacam o impacto dos PowerCaps na eficiência energética, reduzindo significativamente o consumo de energia, mas prolongando o tempo necessário para completar as execuções.

7 Link do Projeto

Todo o material realizado pode ser encontrado no GitHub do grupo juntamente com as instruções de como recriar os resultados obtidos aqui .

8 Conclusão

Ao longo deste trabalho, foi possível analisar o impacto da eficiência energética em diferentes linguagens de programação, utilizando diversos benchmarks e limites de consumo energético impostos ao processador. Através das medições realizadas, identificámos tendências claras entre o consumo energético e o tempo de execução das linguagens, bem como o efeito positivo do PowerCap no controlo do consumo de energia.

Concluímos que o uso de PowerCap se mostrou eficaz para reduzir significativamente o consumo energético, com ganhos variando entre as linguagens testadas. Embora o aumento no tempo de execução seja uma consequência inevitável, verificámos que, em muitos casos, o compromisso entre desempenho e economia de energia pode ser vantajoso, especialmente em cenários onde a eficiência energética é priorizada.

Por fim, este estudo reforça a importância de considerar a eficiência energética como um fator determinante na escolha de uma linguagem de programação, especialmente em contextos como computação sustentável, dispositivos móveis ou sistemas com restrições de consumo. Não há uma "melhor" linguagem universal, mas sim linguagens mais ou menos adequadas dependendo dos objetivos específicos. O uso de ferramentas como o PowerCap e a avaliação de métricas como consumo energético e desempenho são essenciais para tomar decisões informadas.

9 Apêndice

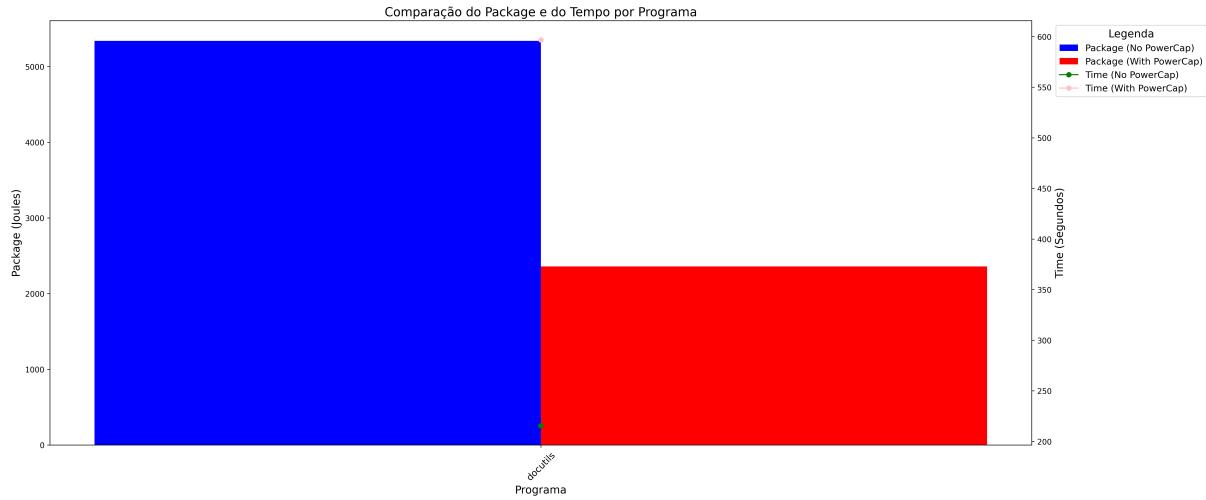


Figure 25: Programa omitido na comparação- Python

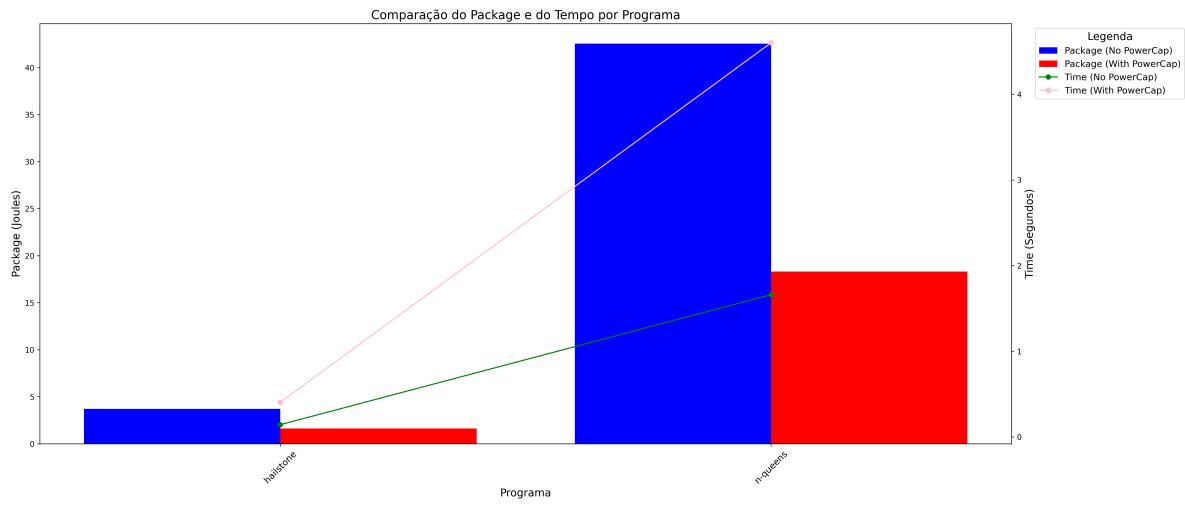


Figure 26: Programas omitidos na comparação- Rust