Universidade do Minho
Departamento de Informática

Desenvolvimento, Validação e Manutenção de Software

Experimentação em Engenharia de Software

Efeito do PowerCap em diversas linguagens de programação

Elaborado por:
João Afonso Alvim Oliveira Dias de Almeida - pg53902@alunos.uminho.pt
Simão Oliveira Alvim Barroso - pg54236@alunos.uminho.pt
Simão Pedro Cunha Matos - pg54239@alunos.uminho.pt

Ano Letivo 2023/24
22 de maio de 2024

Índice

0 Introdução

O presente relatório diz respeito ao trabalho prático da Unidade Curricular de **Experimentação em Engenharia de Software** do perfil de Desenvolvimento, Validação e Manutenção de Software.

Ao longo deste trabalho, vamos explicar tudo relacionado à resolução do mesmo, incluindo:

- A explicação do trabalho e dos seus objetivos
- A estrutura do ficheiro `.zip` entregue.
- Uma secção de como replicar os resultados, onde se inclui as versões utilizadas das bibliotecas e linguagens, entre outros.
- A explicação das nossas resoluções das fases 1 e 2 destes trabalho.

É de realçar que grande parte do que for esclarecido sobre a fase 1 é uma continuação dos `slides` entregues no TP1.

0.1 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 últimos nos programas e de que maneira a linguagem afeta isso. Pretendemos, portanto, descobrir qual a linguagem é mais afetada pelas limitações de energia e como é que isso afeta. Para não cairmos na falácia de um único caso, testamos várias vezes cada programa e vários programas. Assim, teremos resultados mais fidedignos.

Foi utilizado um programa chamado `RAPL`, apresentado pelos professores nas aulas, que nos permitiu recolher vários dados relativos às execuções dos programas.

Foram escolhidas, e tal como pedido no enunciado, as 3 seguintes linguagens/*benchmarks*:

- Haskell - NoFib
- Java - Dacapo
- Python - Pyperformance

0.2 Estrutura do `.zip` de entrega

No início do trabalho, foram fornecidas as pastas Powercap, RAPL, RAPLCap e Utils.

Adicionalmente, adicionamos a pasta Language, que contém todos os programas destinados a realizar benchmarks.

Também incluímos a pasta Notebook, a qual contém vários notebooks com a análise de dados.

Além disso, temos os arquivos *measurements_nome da linguagem.csv*, que contêm as informações de cada medição para cada linguagem. Esses arquivos estão sem nenhum tipo de tratamento. O README também contém informações sobre como executar o programa. Os ficheiro bash `runs` correspondem a:

- `runSetup.sh` : Um script que instala tudo o necessário para a execução destes programas.
- `run.sh` : Script usada para recolher os dados relativos às 3 linguagens de programação
- `run2.sh` : Script usada para recolher os dados das versões python 3.6 e 3.8

0.3 Como fazer para replicar os resultados?

Nesta secção vamos explicar como será possível replicar os resultados.

0.3.1 Hardware

Devido às limitações do próprio RAPL, apenas pudemos utilizar estas medições num dos computadores de um dos elementos do grupo.

Na imagem ao em baixo vemos as várias características do sistema onde foram feitas as medições, entre as quais destacamos:

- **Sistema Operativo :** Ubuntu 22.04.4 LTS
- **CPU :** Intel i7-12700H
- **Memória RAM :** 16GB

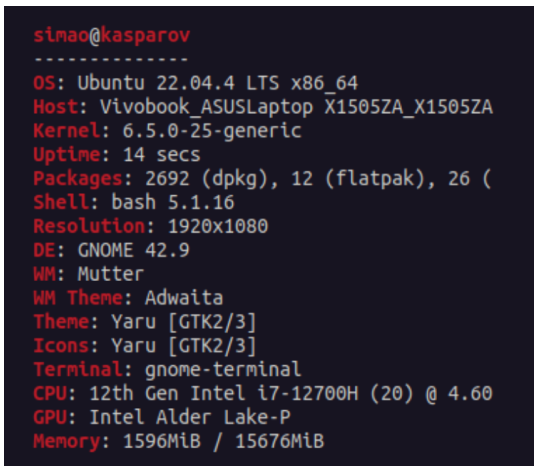

Figura 1: Especificações

0.3.2 Versões dos Benchmarks utilizados

As versões dos benchmarks utilizadas são:

- **Dacapo :** 23.11-chopin (lançado a 8 de Novembro de 2023)
- **Pyperformance :** 1.10 (lançado a 22 de Outubro de 2023)
- **NoFib :** 7ffecc8115865fea9995a951091e6ff23cf8ca3a (Janeiro de 2024)[1]

[1] como não encontramos mais informações deixamos aqui o id do último commit no gitlab

0.3.3 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)

0.3.4 Configurações / Notas necessárias

Ao longo da resolução deste trabalho, tivemos alguns erros que tivemos de corrigir para

1. Instalar o pyperformance com sudo pip install pyperformance
2. Instalar o haskell-libs através do package manager apt.
3. Caminhos absolutos para o compilador e interpretador do Haskell (ghc e ghci).
4. Para mais informações, ler o README.md contido no zip

0.3.5 Escolha dos PowerCaps

A ferramenta utilizada por nós, o RAPL, permite limitar o consumo de energia por parte do CPU. Ora este limite serve para melhor vermos a performance não só do nosso computador bem como das ferramentas consoante as diferentes características.

Para isto, tivemos de escolher diferentes PowerCaps dependendo do nossos computadores. Portanto, tivemos que encontrar uma maneira para escolher os PowerCaps. Baseamo-nos no trabalho feito aquando da primeira entrega, sobre os algoritmos de ordenação, para encontrar os 3 primeiros valores.

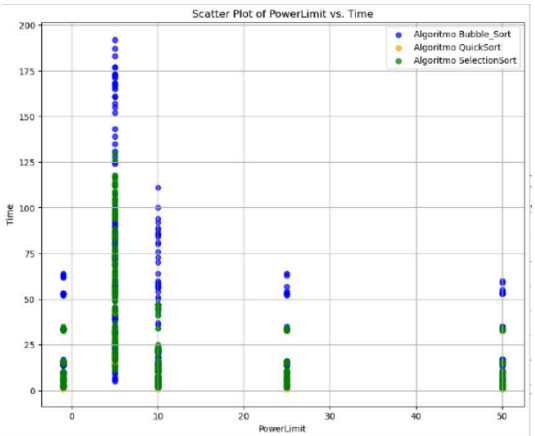

Figura 2: Resultados da 1ª entrega.

O 3 valores seguintes basearam-se nos especificações do processador.
Os valores escolhidos (em Watts) foram:

- **-1 :** sem limite (opção para ver a, **em princípio**, melhor execução temporal do programa)
- **5 :** Valor retirado da 1ª entrega

- **10 :** Valor retirado da 1ª entrega
- **35 :** Valor apontado pela Intel como mínimo em Watts para que o processador funcione de forma adequado.
- **45 :** Valor apontado pela Intel como normal Watts para que o processador funcione de forma adequado.
- **115 :** Valor apontado pela Intel como máximo Watts para que o processador funcione de forma adequado.

É de mencionar também que uma abordagem extra e mencionada pelos professores para encontrar um bom PowerCap seria a utilização de um programa que calcule uma sequência fibonnaci.

Outro aspeto relevante a mencionar é que para os resultados da primeira fase utilizamos os 6 PowerCaps, mas no da segunda fase apenas utilizamos os limites **-1** e **5**, uma vez que são estes que melhores resultados dão.

1 Fase 1: Monitorização da performance de linguagens de programação

Esta primeira fase foi entregue no dia 22 de março; assim este capítulo é um aprofundamento do que foi explicado nesses *slides*.

1.1 Obtenção dos resultados

Como dito anteriormente, nesta primeira fase usamos **6 PowerCaps**; Utilizamos **3 benchmarks/linguagens**; e para cada linguagem escolhemos **10 programas**. Cada um desses progrmas foi executado **10 vezes**. Logo para cada linguagem temos um `dataset` de 601 linhas (uma de cabeçalho).

1.2 Escolha dos programas

Vamos agora explicar a escolha dos programas para cada uma das linguagens.

É de realçar que muitos destes programas foram escolhidos por diversas razões, estando entre elas:

1. Havia programas que não funcionavam corretamente com a versão da linguagens instalada no computador onde isto foi testado. Um exemplo disto é o Cassandra, que apenas funcionava com Java 17. Outro exemplo eram os programas parallel do Haskell.

2. Conhecimento e curiosidade sobre alguns do programas, como o Kafka.

3. 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 muito rigorosamente feitas, mas apenas tinham o objetivo de ajudar na escolha entre alguns programas.

1.2.1 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.

Os programas escolhidos foram os seguintes:

- avrora
- batik
- biojova
- eclipse
- kafka

- spring
- tomcat
- graphchi
- jme
- jython

1.2.2 Python

Já o pyperformance tinha um total de 84 programas para **benchmark**. Decidimos então escolher 10. Primeiramente, o **pyperformance** possui vários grupos (para listar basta fazer `pyperfomance list_groups`).

Dentro destes grupos decidimos escolher :

- O grupo `apps` que contém os programas mais exaustivos temporalmente e energicamente
- 1/2 de cada um dos grupos : **async**, **templates**, **math**

Assim temos uma representação dos vários benchmarks.

Os programas escolhidos foram os seguintes:

- chameleon
- docutils
- html5lib
- 2to3
- tornado_http
- nbody
- json_dumps
- pidigits
- async_tree
- django_template

1.2.3 Haskell

Por último a escolha dos programas em *haskell* foi a 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 pôr a funcionar 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.

Os programas escolhidos foram os seguintes:

- Grep

- Compress
- Compress2
- gg
- RSA
- Rfib
- binary-trees
- fannkuch-redux
- spectral-norm
- sorting

1.3 Como foram obtidos os dados?

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 **10 programas/problemas** e cada programa **executa 10 vezes** para os **6 PowerCaps** cada linguaguem vai gerar um CSV de **601 linhas** (uma delas é de cabeçalho).

É de mencionar que a parte de Python foi a mais demorada, seguida da de Java e por último a de Haskell.

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 ia demorar.

1.4 Dados Obtidos e Tratamento de Outliers

Como dito anteriormente, obtivemos 3 CSV cada um com 601 linhas.

Decidimos manter separado para simplificar uma vez que o objetivo é comparar as linguagens no geral e não em particular (não há nenhum programa/problema que seja comum a linguagens).

Estes dados obtidos foram tratados de seguida no ficheiro `tratamento.ipynb`. Mais à frente vamos referir outros notebooks que usamos para outros tratamentos de dados.

Os dados tinham ausência das colunas de GPU e DRAM pelo que as eliminamos.

Passamos também o valor de tempo para segundos, que consistia em dividir por 1000.

1.4.1 Colunas do CSV

As colunas dos CSVs resultantes são as seguintes:

- **Language :** Linguagem de programação do Benchmark dos problemas/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)
- **Core :** Consumo de energia de todos cores e caches dos mesmos.
- **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 ºC);
- **Memory :** Total de memória gasta durante a execução do programa (em KBytes);
- **PowerLimit :** PowerCap dos cores (em Watts).

É necessário mencionar, tal como foi dito na teórica, que existem algumas colunas que não conseguiram ser medidas, como é o caso da coluna **GPU** e **DRAM**.

1.4.2 Tratamento de Outliers

Como podemos ver na imagem abaixo, a colheita de dados é sempre sujeita à existência de vários *outliers*.

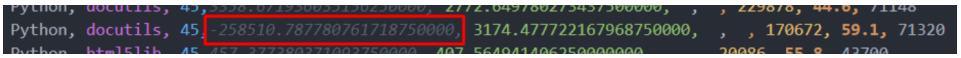

Figura 1.1: Exemplos de Outliers

Para combater isto fizemos 2 coisas:

- Primeiro para cada programa e para cada *PowerCap* ordenar por *Package* (consumo total de energia) e eliminar os 2 maiores e os 2 menores valores. (Ficando cada programa para cada *PowerCap* com 6 valores)
- Para alguma da análise fizemos o seguinte:

  Pegamos nessas 6 linhas de cada programa e calculamos a média, tendo assim 1 linha para cada *PowerCap* de cada Programa, ficando o `CSV` com 61 linhas (6 *PowerCaps* * 10 Programas + 1 de cabeçalho). Assim podemos analisar cada programas
- Para outras situações utilizamos um *dataframe* com a média de todos os programas por *PowerLimit*, CSV com 7 linhas (1 cabeçalho + 6 *PowerLimit*). Assim podemos analisar a linguagem como um todo.

1.4.3 Dados obtidos

Pegando nos vários `notebooks` presentes na pasta `notebooks` do `zip`.

Tipo de dados

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

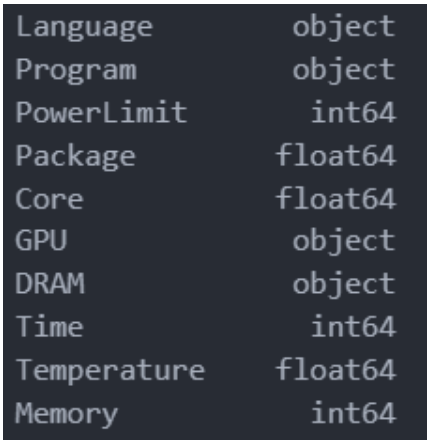

Figura 1.2: Tipo dos dados

É de mencionar que em nenhum dataset tivemos valores para o GPU e para DRAM, pelo que na análise os eliminamos do dataset.

Correlação

A tabela de correlação dos 3 `datasets` juntos é:

	PowerLimit	Package	Core	Time	Temperature	Memory
PowerLimit	1.000000	-0.001306	-0.042007	-0.083554	0.411436	-0.005609
Package	-0.001306	1.000000	-0.017660	-0.004208	-0.019407	0.010855
Core	-0.042007	-0.017660	1.000000	-0.023784	0.001359	0.009968
Time	-0.083554	-0.004208	-0.023784	1.000000	-0.135121	-0.020559
Temperature	0.411436	-0.019407	0.001359	-0.135121	1.000000	0.156521
Memory	-0.005609	0.010855	0.009968	-0.020559	0.156521	1.000000

Figura 1.3: Tabela de Correlação

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

No entanto e verificando o de cada um dos `datasets` separados, vemos uma poluição de uns aos outros.

Vendo por exemplo a de Haskell, vemos alguns atributos com correlações diferentes:

	PowerLimit	Package	Core	Time	Temperature	Memory
PowerLimit	1.000000	0.222589	0.304471	-0.284757	0.507204	0.103252
Package	0.222589	1.000000	0.968801	0.318723	0.392359	0.137643
Core	0.304471	0.968801	1.000000	0.090730	0.564471	0.197011
Time	-0.284757	0.318723	0.090730	1.000000	-0.629547	-0.205830
Temperature	0.507204	0.392359	0.564471	-0.629547	1.000000	0.311404
Memory	0.103252	0.137643	0.197011	-0.205830	0.311404	1.000000

Figura 1.4: Correlação em Haskell.

Portanto temos de ter cuidado aquando de formos retirar conclusões. Como em principio os vamos tratar em separado cada linguagem, estes problemas são diluídos.

Descrição dos Valores

Apresentamos aqui mais dados sobre os dados obtidos:

	PowerLimit	Package	Core	Time	Temperature	Memory
count	1800.000000	1800.000000	1800.000000	1800.000000	1800.000000	1.800000e+03
mean	34.833333	162.933848	103.390752	21136.285000	48.882333	2.872868e+05
std	39.425065	6125.578029	6128.785014	45625.486075	6.530551	4.512732e+05
min	-1.000000	-258510.787781	-259030.248047	2069.000000	37.000000	5.120000e+03
25%	5.000000	77.055420	60.837082	5503.750000	42.900000	6.080000e+03
50%	22.500000	153.757660	115.658508	9368.500000	50.900000	4.376800e+04
75%	45.000000	316.406494	262.030045	17734.000000	53.400000	4.669410e+05
max	115.000000	3804.868530	3331.643921	432469.000000	67.800000	1.816748e+06

Figura 1.5: Describe dos 3 datasets juntos.

Clustering

O `clustering` é uma técnica bastante utilizada para dividir `datasets`. Com este intuito, tentamos pegar nos valores dos `datasets`, juntamos todos, eliminamos colunas vazias e as colunas de texto, como a de linguagens.

Assim, retiramos, a coluna que queremos prever que é a de linguagens/*benchmarks*.

Assim, e utilizando a técnica aprendida nas aulas, descobrimos o seguinte:

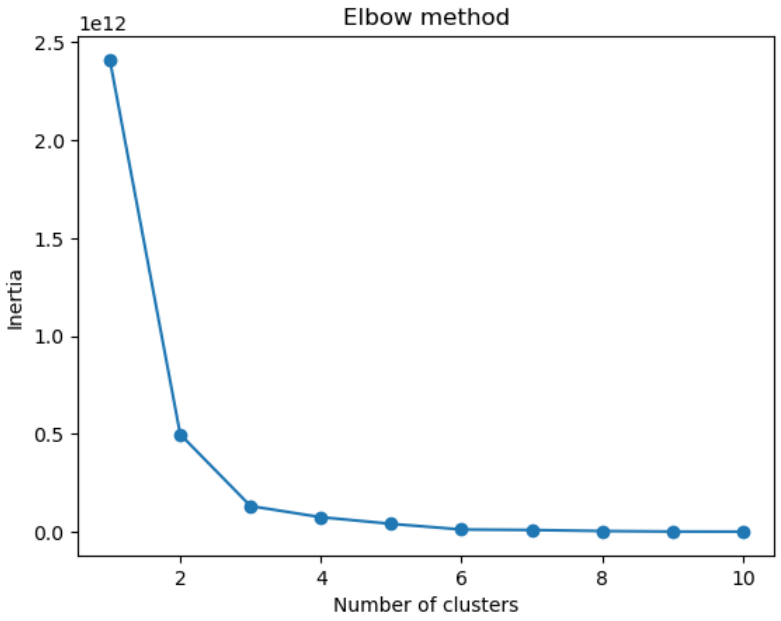

Figura 1.6: Método do Cotovelo.

Assim, e tal como expectável, conseguimos dividir o `dataset` em 3, cada um correspondente a uma linguagem.

Vemos aqui a divisão em 3 `clusters`, tendo em conta o tempo gasto e a energia gasta (`Package`).

Verificamos uma clara divisão em 3 clusters, cada um representativo de uma linguagem.

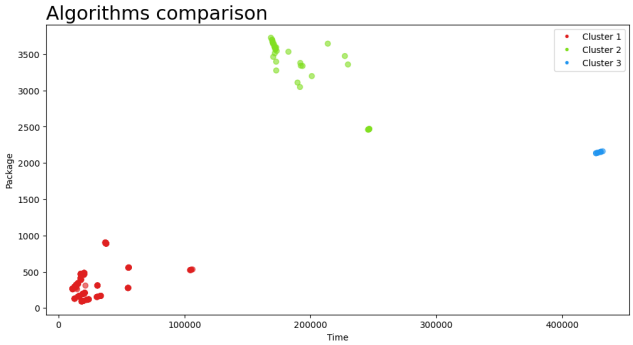

Figura 1.7: Divisão entre clusters.

No entanto, vemos que esta técnica não resultou muito bem, principalmente devido às escolhas dos programas.

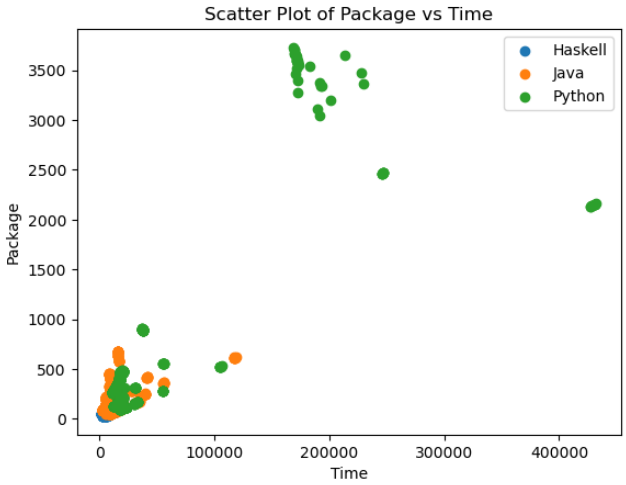

Figura 1.8: Divisão que supostamente se deveria obter.

Decision Trees

A árvore de decisão do nosso modelo é:

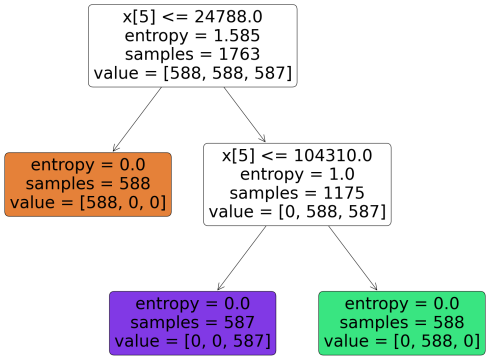

Figura 1.9: Decision Tree

2 Fase 2: Análise dos Resultados

Nesta secção vamos explicar os resultados obtidos nas alíneas anteriores. É de realçar que alguma desta análise já foi feita para a entrega do `tp1`, uma vez que essa análise foi bastante profunda.

No entanto, esta análise vai ser mais profunda em dois aspetos:

1. O facto de termos feito extra de comparar a performance em diferentes versões do `python`.
2. Utilizarmos os mecanismos e técnicas ensinada pelo professor Paulo Azevedo nas aulas teóricas e práticas.

Sobre este último ponto é importante mencionar que apesar de termos em atenção o dado nas aulas, não aplicamos um dos métodos para tratamento de *outliers*, uma vez que achamos que a nossa maneira, em cima descrita, é suficiente para esta análise, até porque retirando o ocasional valor, não havia grandes diferentes entre os valores observados.

2.1 Análise por linguagem

Decidimos, primeiramente, fazer uma análise por linguagem, para verificar quais são as tendências dentro de cada linguagem/*benchmark*.

Para tratamento de dados, utilizamos o *notebook* `tratamento.ipynb` nas alíneas seguintes. Este *notebook* tem imensos gráficos e tentativas de fazer gráficos que programamos.

Nas alíneas seguintes utilizamos um método de fazer a média por *PowerLimit* de todos os programas. Assim, o peso de uma determinada execução é nivelada pelas outras; também faz com que a execução maior de um dos programas não afeta os outros. Assim, e como vamos ver nas imagens em baixo, temos um total de uma tabela com 7 linhas, onde uma é cabeçalho e cada uma das 6 seguintes é um dos *PowerLimit* com os valores em média de cada um dos programas todos juntos.

Vamos focar a nossa análise principalmente nos *PowerLimit* -1 (sem limite) e no *PowerLimit* 5 (5 watt).

2.1.1 Java

De forma representar o *dataframe* com os dados tratados em cima referenciado, temos a seguinte tabela:

PowerLimit	Package	Core	Time	Temperature	Memory
-1	250.80283406575523	218.61824645996094	7.604866666666666	54.24166666666666	786042.3333333333
5	200.70226338704427	92.9109883626302	37.01058333333333	40.641666666666666	842075.6
10	155.06514689127604	97.51852416992188	15.808850000000001	43.385000000000005	809169.1333333333
35	221.38602600097656	188.13086446126303	7.93065	53.64666666666667	803159.8
45	238.10235087076825	205.5230244954427	7.7081	54.23166666666666	783649.2
115	244.88507995605468	212.36087036132812	7.807666666666667	53.89833333333333	790990.6666666667

Figura 2.1: Tabela resultante Java

Como esperávamos a melhor performance em termos de tempo (menos tempos) e pior em termos de energia gasta (maior valor do Package) é a que não põe limite ao consumo de energia (-1).

Vemos também uma tendência que quanto menor o *PowerCap* em geral maior é a temperatura. Isto deve-se a o *PowerCap* colocar um limite na energia gasta que leva a que leva a que o processador para não ultrapassar isso não pode correr em temperaturas muito grandes, uma vez que temperaturas mais altas levam a um gasto de energia maior.

Em termos de memória não vemos uma variação muito grande. No entanto, vemos as partes mais lentas a gastar mais memória. O que cremos ser algo mais estranho, uma vez que em teoria os mesmos programas iriam utilizar a mesma quantidade de memória. Mas uma vez que não é o foco do trabalho, não pensamos muito nestes casos, tendo como principal argumento para isto o *PowerCap* afetar ferramentas como a `cache`.

Vemos aqui a distribuição dos vários atributos pelos vários PowerLimit. É de realçar as conclusões acima mencionadas.

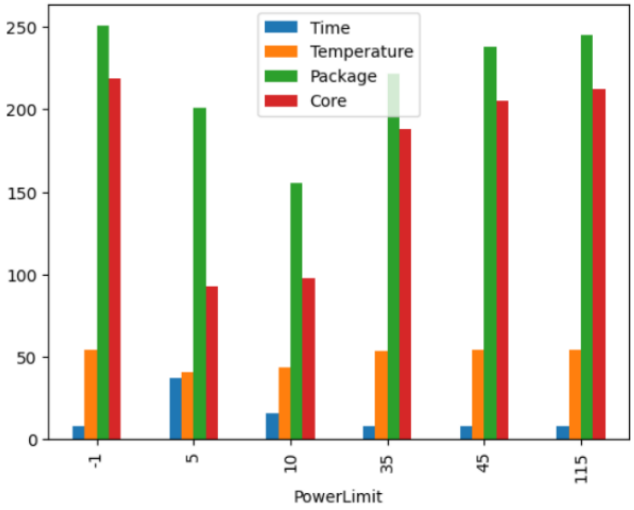

Figura 2.2: Distribuição dos valores por PowerLimit

Como vemos na tabela em baixo, uma poupança de 19,97% de energia vai levar a um ganho de 387% e aumento do tempo de execução do programa por cerca de 5 vezes.

Nota: estes valores de ganho foram calculados com o método `pct_change()` do pandas.

	PowerLimit	Package	Ganho Package (%)	Time	Ganho Temporal
0	-1	250.802834	NaN	7.604867	NaN
1	5	200.702263	-19.976078	37.010583	3.866697
2	10	155.065147	-22.738715	15.808850	-0.572856
3	35	221.386026	42.769688	7.930650	-0.498341
4	45	238.102351	7.550759	7.708100	-0.028062
5	115	244.885080	2.848661	7.807667	0.012917

Figura 2.3: Ganho Temporal e de Package de cada um dos limites para o seguinte

Vemos aqui a confirmação de que quanto menor o limite menos energia gasta. No entanto vimos uma situação a estranhar, o facto de o limite 10 gastar mais energia no total do que o limite 5, mas a energia do core ser maior no 10. Assumimos portanto que isto poderia ser algo externo aos *benchmarks*, como algum processo extra lançado durante as execuções do limite 5.

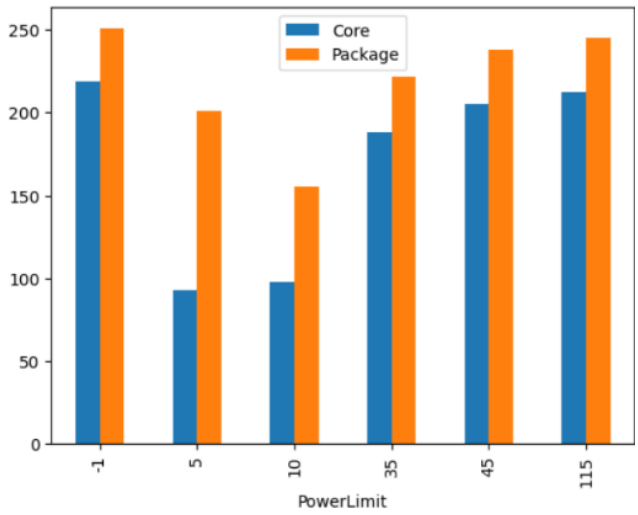

Figura 2.4: Distribuição do Core e do Package por PowerCap

Vemos novamente, no gráfico em baixo, a correlação entre maior temperatura maior valor do *package*.

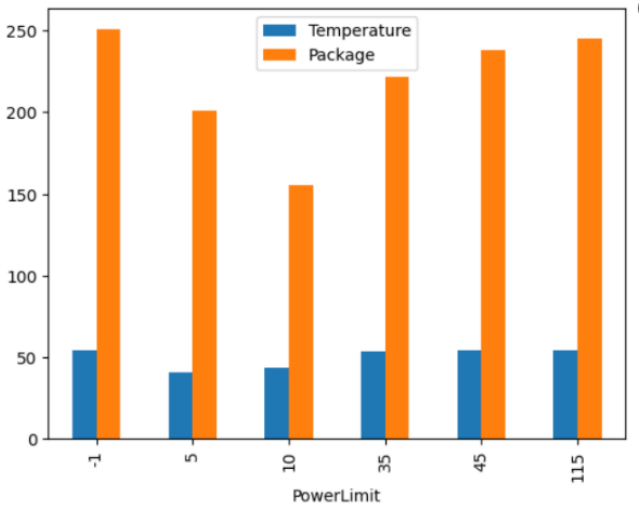

Figura 2.5: Distribuição do Package e da Temperatura por PowerCap

Vemos o previsto que quanto maior o limite ao consumo de energia mais tempo vai demorar a executar os programas.

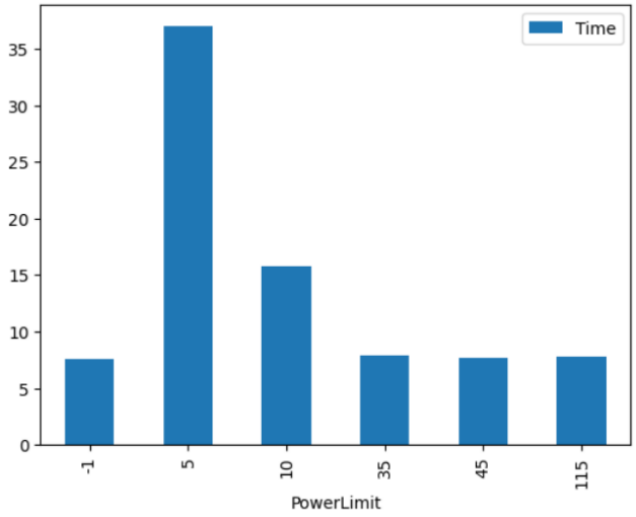

Figura 2.6: Distribuição Temporal por PowerCap

Neste último vemos um gasto de memória mais ou menos distribuído, no entanto vemos que o limite afeta ligeiramente a memória gasta.

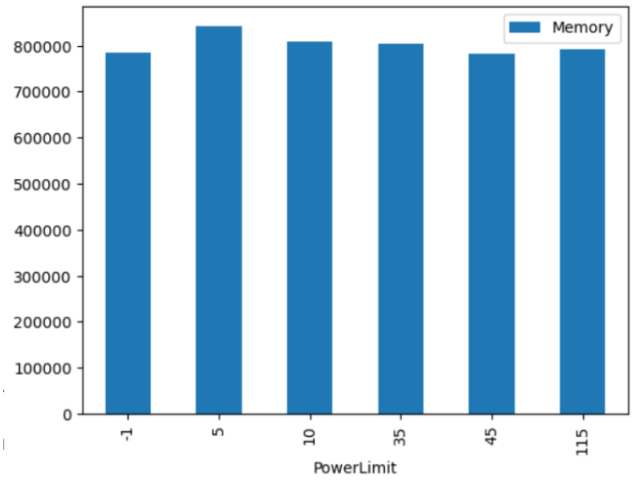

Figura 2.7: Gasto de Memória por PowerCap

Análise por programa

Decidimos também fazer uma análise por programa, para ver algumas tendências, se havia algum particularmente notável, entre outras razões.

Nesta comparação entre os programas testados em Java, vemos claramente uma forte influência do Eclipse que é o programa que não só consome mais energia como também o que demora mais tempo em ambas as versões.

Vemos que o tempo do eclipse foi muito afetado pelo *PowerCap*, embora o seu consumo não tenha sido afetado da mesma maneira.

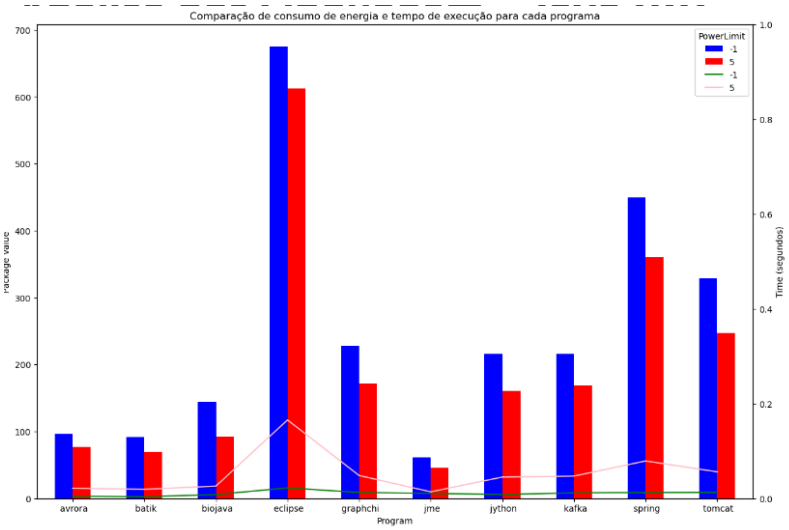

Figura 2.8: Comparação temporal e em Package entre os vários programas para os limites -1 e 5.

Nota : As linhas são os tempos e as barras são os valores dos *packages*, para cada um dos *PowerCap*.

2.1.2 Python

Assim como no Java, aqui verificamos que altura onde se gastou mais energia, menos tempo e onde se obteve maior temperatura do processador foi sem PowerLimit.

Por outro lado, o menor dos PowerLimit (5 Watts) é onde se gastou mais tempo e menos energia, com menor temperatura.

Assim sendo, é nestes 2 que vamos focar a nossa análise, como feito para o Java.

PowerLimit	Package	Core	Time	Temperature	Memory
-1	751.3114512125651	657.2998148600261	33.44858333333333	53.85666666666666	52329.13333333334
5	378.33116658528644	193.89121805826824	75.64361666666666	38.39	52337.53333333334
10	455.0203043619792	336.875498453776	45.376783333333336	44.32666666666667	52346.066666666666
35	743.5906768798828	649.9462493896484	33.78793333333333	53.473333333333336	52322.066666666666
45	728.1011515299479	630.749681599935	35.57695	53.09166666666666	52324.86666666667
115	720.3532287597657	623.5578440348307	35.70705	53.14333333333333	52312.46666666667

Figura 2.9: Tabela resultante Python

Aqui vemos as tendências verificadas ao longo da resolução deste trabalho:

- *PowerCap* mais baixo implica mais tempo e menos energia gasta.
- PowerCap a -1 significa mais energia gasta e menos tempo

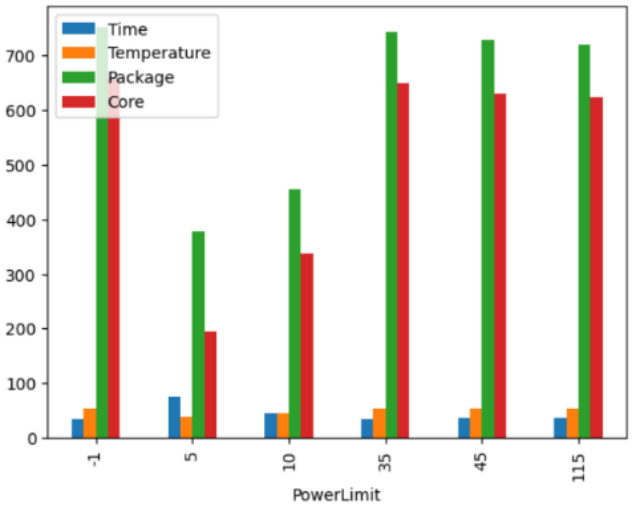

Figura 2.10: Distribuição dos valores por PowerLimit

Como vemos na tabela em baixo, uma poupança de 49,64% de energia vai levar a um ganho de 126% (aumento do tempo de execução do programa por cerca de 2.26 vezes).

Nota: estes valores de ganho foram calculados com o método `pct_change()` do pandas.

	PowerLimit	Package	Ganho Package (%)	Time	Ganho Temporal
0	-1	751.311451	NaN	33.448583	NaN
1	5	378.331167	-49.643897	75.643617	1.261489
2	10	455.020304	20.270373	45.376783	-0.400124
3	35	743.590677	63.419230	33.787933	-0.255392
4	45	728.101152	-2.083071	35.576950	0.052948
5	115	720.353229	-1.064127	35.707050	0.003657

Figura 2.11: Ganho Temporal e de Package de cada um dos limites para o seguinte

Tal como visto no Java, o PowerCap a 5 leva ao menor gasto de energia.

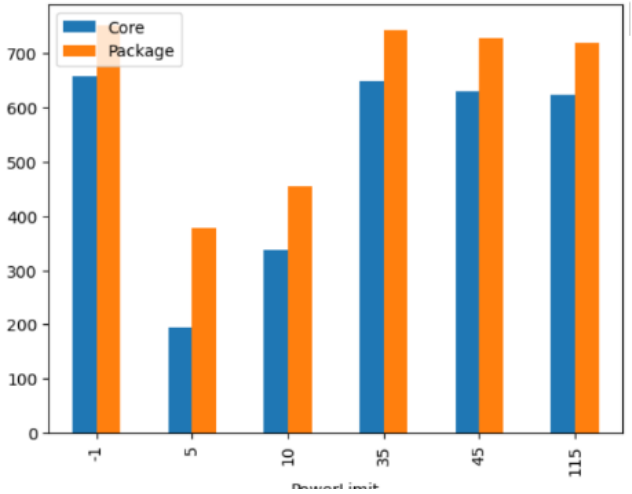

Figura 2.12: Distribuição do Core e do Package por PowerCap

Vemos a correlação entre a temperatura e o gasto de energia. Maior gasto de energia leva a maior temperatura.

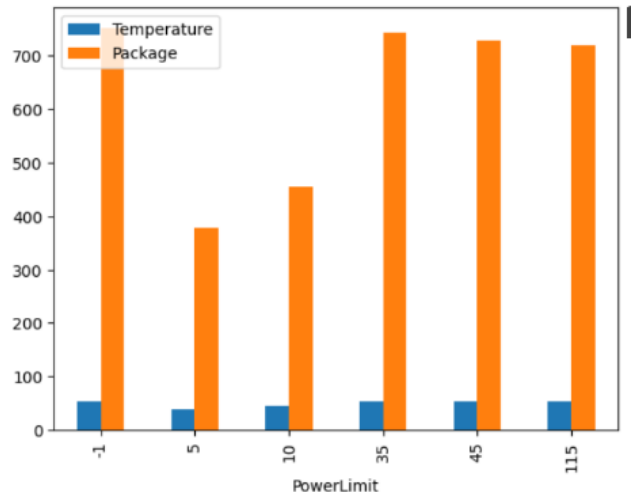

Figura 2.13: Distribuição do Package e da Temperatura por PowerCap

Vemos que o mais rápido foi o sem limite e o limite mais baixo (valor do PowerCap menor) leva à maior execução temporal.

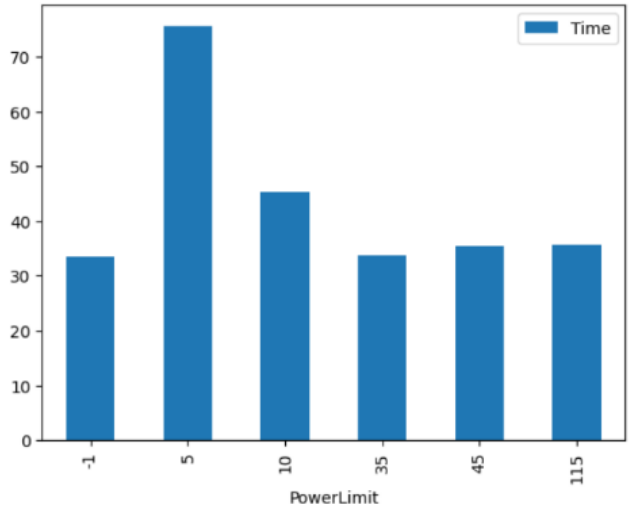

Figura 2.14: Distribuição Temporal por PowerCap

O gasto de memória foi mais ou menos constante ao longo de todas as execuções.

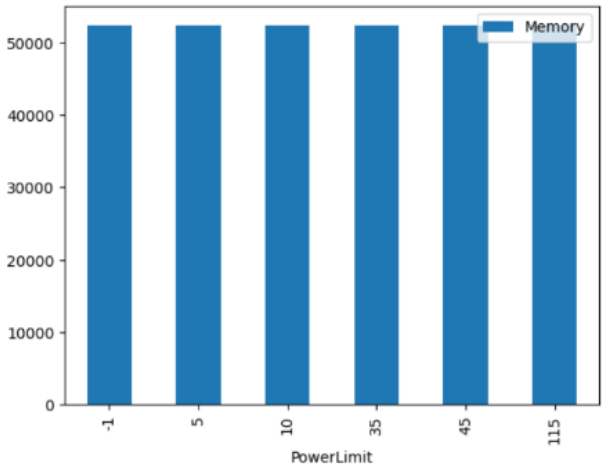

Figura 2.15: Gasto de Memória por PowerCap

Análise por programa

Tal como caso do Java vemos a existência de um programa muito acima de todos os outros quer seja em termos de gasto de energia quer seja em termos de tempo.

Vemos também que o *PowerCap* houve uma diminuição drástica do consumo de energia, acompanhada de uma subida de tempo não tão drástica.

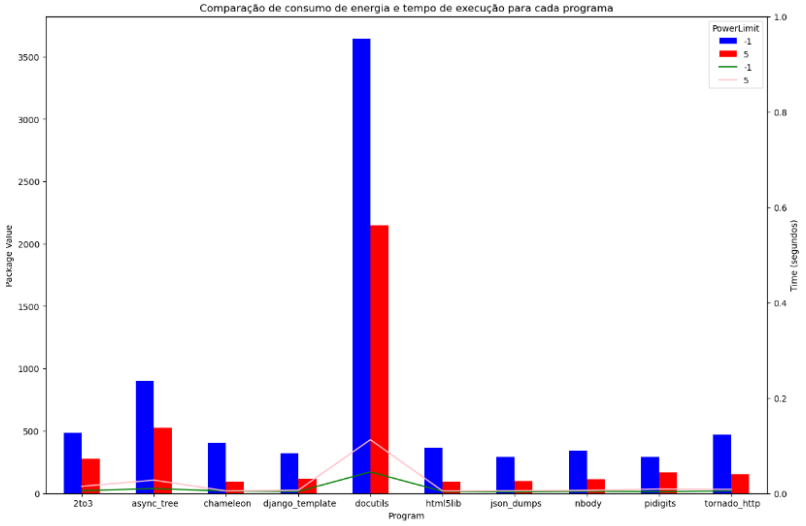

Figura 2.16: Comparação temporal e em Package entre os vários programas para os limites -1 e 5.

2.1.3 Haskell

Tal como visto anteriormente e era previsível, o *PowerCap* teve um efeito de diminuir a energia gasta e aumentar o tempo de execução.

E, tal como dantes, vamos concentrar a nossa análise nos valores de -1 e 5.

PowerLimit	Package	Core	Time	Temperature	Memory
-1	92.66651814778646	81.63722737630209	4.201333333333333	50.875	6061.333333333334
5	54.98665161132813	28.581238810221357	11.0021	38.48166666666667	5928.0
10	61.46668701171875	46.172584025065106	6.110016666666667	43.25666666666667	6008.0
35	84.38891906738282	72.29192810058593	4.645300000000001	49.345	6072.0
45	92.91887003580729	82.20967712402344	4.233316666666667	51.754999999999995	6061.333333333334
115	93.05833841959635	82.46624247233072	4.190133333333334	51.84166666666666	6040.0

Figura 2.17: Tabela resultante Haskell

Vemos que uma poupança de 40,66% de energia vai levar a um ganho de 161% de tempo. Ou seja, o programa demorou cerca de 2.61 vezes mais.

Nota: estes valores de ganho foram calculados com o método `pct_change()` do pandas.

	PowerLimit	Package	Ganho Package (%)	Time	Ganho Temporal
0	-1	92.666518	NaN	4.201333	NaN
1	5	54.986652	-40.661792	11.002100	1.618716
2	10	61.466687	11.784743	6.110017	-0.444650
3	35	84.388919	37.292122	4.645300	-0.239724
4	45	92.918870	10.107904	4.233317	-0.088688
5	115	93.058338	0.150097	4.190133	-0.010201

Figura 2.18: Ganho Temporal e de Package de cada um dos limites para o seguinte.

Vemos a mesma tendência verificável no `python`, pelo que, novamente, focamos a nossa análise em -1 e 5.

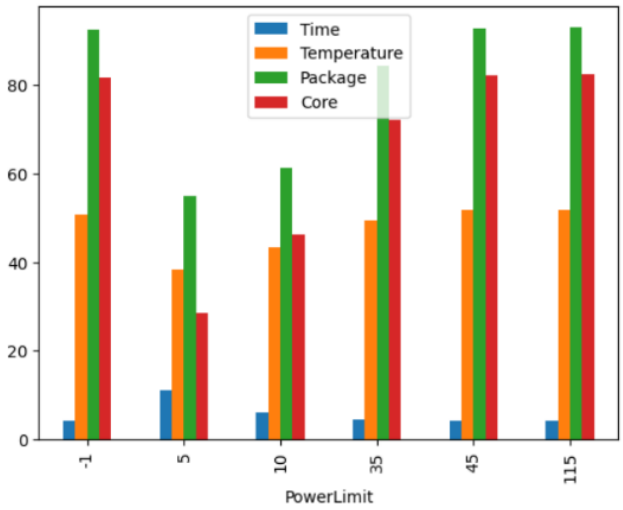

Figura 2.19: Distribuição dos valores por PowerLimit.

A evolução do consumo de energia mantém-se relativamente dentro do esperado e verificado nas outras linguagens.

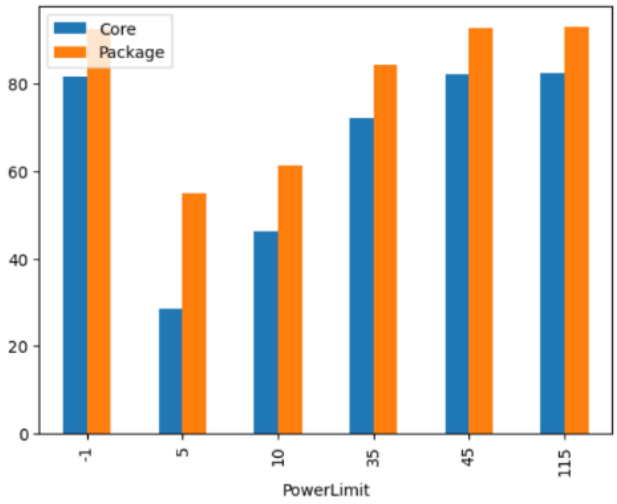

Figura 2.20: Distribuição do Core e do Package por PowerCap

O mesmo acontece neste diagrama. No entanto é de verificar a temperatura mais ou menos constante. Isto deve-se ao facto de estes programas serem menos exigentes do que nos outros *benchmarks*.

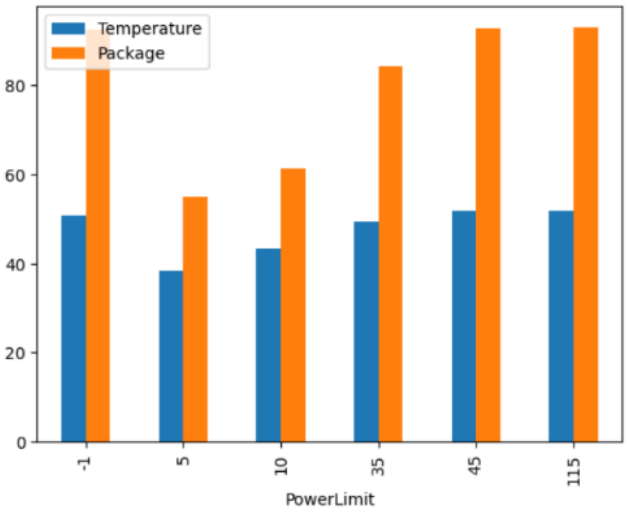

Figura 2.21: Distribuição do Package e da Temperatura por PowerCap.

Tal como os anteriores, se queremos maximizar a performance não pomos nenhum limite.

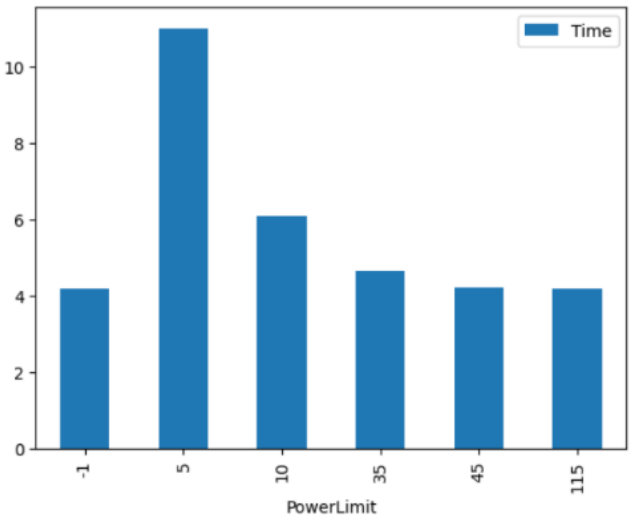

Figura 2.22: Distribuição Temporal por PowerCap.

Memória gasta constante, assim como é de esperar.

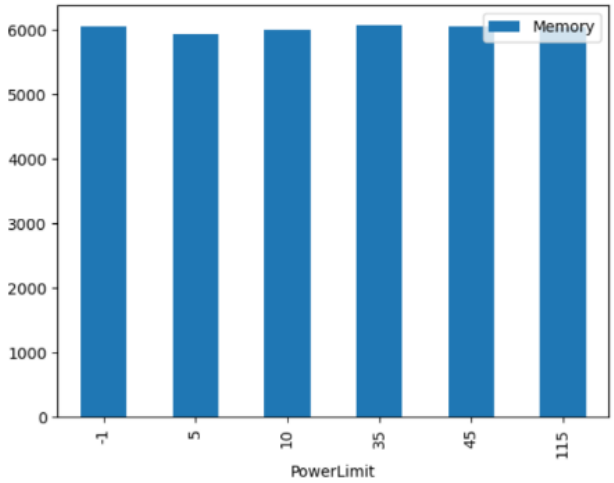

Figura 2.23: Gasto de Memória por PowerCap.

Análise por programa

Aos contrário das outras duas, nestas houve uma distribuição muito menos uniforme do tempo.

Vemos também que a maioria dos programas executados tem forte tendência matemática/científica.

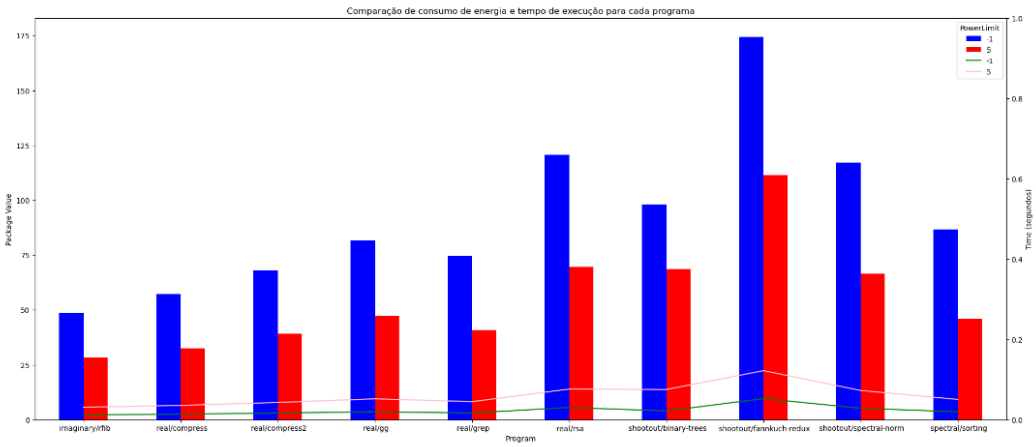

Figura 2.24: Comparação temporal e em Package entre os vários programas para os limites -1 e 5.

2.2 Comparações entre linguagens

Ao comparamos as três linguagens, confirmamos o enunciado no artigo científico demonstrado pelo professor durante as aulas da Unidade Curricular.

Verificamos que, e embora não estejamos a analisar os mesmos programas nas diferentes linguagens, o Python foi o que gastou mais energia.

Em segundo lugar, não muito distante está o Haskell, que embora tenha gasto dos 3 menos energia, foi também o que demorou menos tempo, devido à complexidade dos programas.

Em primeiro ficou o Java, que gastou mais energia que o Haskell mas teve programas mais complexos.

2.2.1 Em qual linguagem teve o PowerLimit menor e maior impacto?

A linguagem em que teve menor impacto foi o Java: com powerCap levou a uma poupança de apenas cerca de 20% mas levou o programa a demorar cerca de 3,5 vezes mais tempo. Podemos então concluir que em termos energéticos o Java é a mais eficiente.

A linguagem em que o Power Cap teve mais impacto foi o Python já que permitiu diminuir em metade o consumo de energia e demorou cerca de 2,26 mais. Com isto podemos ver que o Python consome muita energia.

No meio ficou o Haskell, que com poupança menor do que o Python (cerca de 40%) e de ganho temporal de 161%.

Como dito anteriormente, para esta análise utilizamos principalmente os PowerCap -1 e 5.

É também feita a ressalva que apesar de termos feito esta análise, ela só é possível porque estamos a fazer em geral para uma linguagem. Esta seria melhor implementada e mais justa se utilizássemos mais programas e os mesmo programas para as diferentes linguagens. Infelizmente, devido à natureza dos *benchmarks*, isso não é possível.

2.3 Extra : Diferentes versões de python

Como extra, foi nos sugerido duas ideias para melhor analisarmos este problema:

- Utilização de um compilador para python, como o `codon`.
- Diferentes versões de compiladores ou interpretadores.

Relativamente ao primeiro é de mencionar que não o escolhemos, uma vez que `pyperformance` já faz por si uma compilação, utilizando o `cython`. Apesar de podermos ver a diferente entre os 2, esta seria mais fútil do que a opção 2.

Relativamente ao segundo ponto, entre as várias linguagens, era mais fácil a utilização deste mecanismo para o `python`, uma vez que existem várias formas de fazer isto, entre as quais o próprio `pyperformance` tem uma maneira de especificar a versão, o que não resultou no nosso caso, ou a utilização de `conda envs`, que funcionou para nós.

É necessário mencionar que para esta parte apenas utilizamos os limites -1 e 5 para a medição. Assim focamos a nossa análise nestes 2 limites. Tomamos esta decisão para facilitar, uma vez que os restantes dos dados demoravam muito tempo a ser recolhidos e não traziam muita vantagens.

Para além da versão `3.10` que medimos na realização da 1ª fase, também medimos as versões `3.6.15` e `3.8`. Infelizmente, os resultados do 3.8 ficaram corrompidos devido a erro do grupo enquanto estavam a ser medidos. Por isto, apenas comparamos as versões 3.6 e 3.10. O tratamento dos dados destas medições estão no *notebook* `comp_python.ipynb`.

	PowerLimit	Versão Python	Package	Core	Time	Temperature	Memory
0	-1	Python 3.10	751.311451	657.299815	33.448583	53.856667	52.329133
1	-1	Python 3.6	418.318221	266.445597	59.878356	42.014375	53.838042
2	5	Python 3.10	378.331167	193.891218	75.643617	38.390000	52.337533
3	5	Python 3.6	402.315726	206.265395	79.895583	42.401667	54.040200

Figura 2.25: Dataframe tratado final.

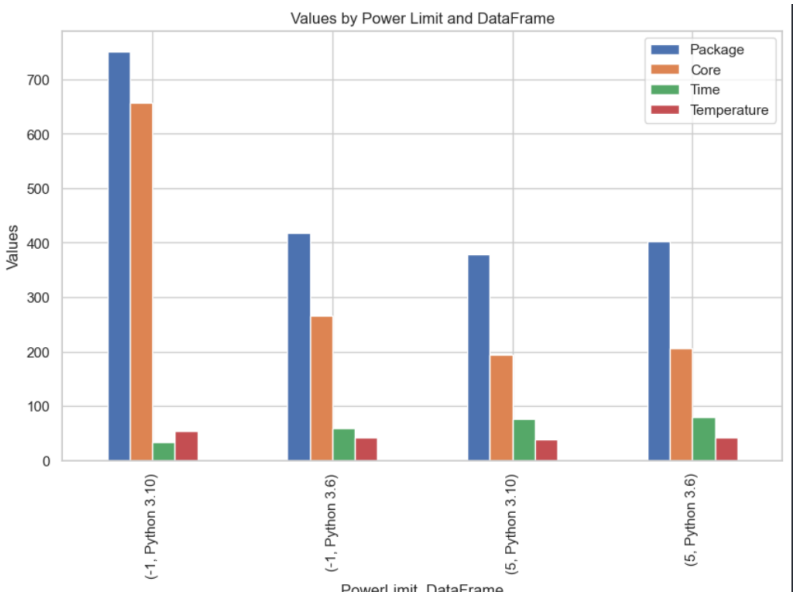

Figura 2.26: Gráficos comparação

É de mencionar que os dois foram medidos mais ou menos nas mesmas condições, durante a noite, depois do computador estar desligado durante 2 horas.

Ao analisarmos os dois gráficos, vemos claramente a correlação *package* temperatura.

Vemos também uma diferença significativa entre o Python 3.10 e o 3.6. Denotamos as seguintes:

1. Para a parte sem limite (PowerLimit a -1):
    - Verificamos que o Python 3.10 gasta muito mais energia que o Python 3.6 mas também demora bem menos tempo. Signfica então que em termos de gastos energéticos a evolução das versões do Python tornaram-no pior, mas melhoraram substancialmente em termos de tempos.
2. Para a parte com limite a 5 W (PowerLimit a 5):
    - Vemos aqui um impacto positivo da evolução das versões do Python uma vez que a versão 3.10 gasta menos tempo e menos energia em média do que a versão 3.6, assinalando um avanço positivo nas duas versões.

Relativamente à memória manteve-se mais ou menos a mesma, mas vemos uma ligeira descida do uso da memória nas versões mais recentes do *python*.

Portanto conclui-se o que previámos no início, quanto melhor a versão de um interpretador melhor a sua performance.

Também temos que fazer a ressalva que poderíamos ter ido mais a fundo nesta questão, mas cremos que como extra apenas serve para nos dar mais introspeção de trabalho futuro nesta área.

3 Conclusão

Ao longo deste trabalho aprendemos sobre benchmarks, sobre medições de energia e sobre várias linguagens. Aprendemos também sobre um lado que não costumamos ver das linguagens de programação, como a eficiência energética e o efeito desta no tempo.

No entanto achamos que há vários aspetos a melhorar, como:

- Aumentar a utilização de outros benchmarks de outras linguagens (infelizmente a maioria dos que encontramos seriam para medir programas feitos por nós e não já predefinidos).
- Utilizar o codon para melhorar a performance do python

Cremos que fizemos um bom trabalho e ficamos satisfeitos.