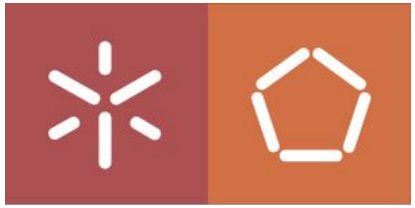
EES: Monitorização da performance de linguagens de programação

Trabalho Prático 1
Experimentação em Engenharia de Software - SDVM

13 Junho de 2024

Trabalho Feito por:



Universidade do Minho
Escola de Engenharia

- João Afonso Alvim Oliveira Dias de Almeida (pg53902)
- Simão Oliveira Alvim Barroso (pg54236)
- Simão Pedro Cunha Matos (pg54239)

Introdução e Objetivos do Trabalho

Este trabalho é da UC de Experimentação em Engenharia de Software. Nele utilizamos benchmarks em várias linguagens de programação sobre vários programas/problemas bem conhecidos e diferentes consoante a linguagem.

Assim, monitorizamos a performance das várias linguagens de programação.

Utilizamos 3 benchmarks/linguagens : Haskell (nofib), Java (Dacapo) e Python (Pyperformance).

Utilizamos este benchmarks para ver o impacto da limitação do consumo de energia nestas 3 linguagens.

Introdução e Objetivos do Trabalho (II)

Estes slides estão organizados em 4 partes :

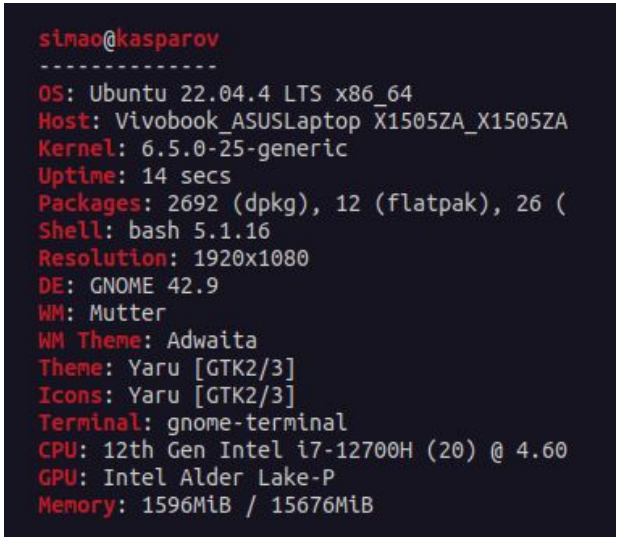
- a primeira com as versões e hardware utilizado, para facilitar a replicação dos resultados;
- uma segunda com a recolha e tratamento que foi dado aos dados;
- uma terceira com análise desses dados;
- por último uma conclusão e uma secção crítica do nosso trabalho.

Versões

Hardware onde foram feitas as medições:

Na imagem ao lado 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**



Versões dos Benchmarks utilizados:

- **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)*

*como não encontramos mais informações deixamos aqui o id do último commit no gitlab

Versão das linguagens/interpretadores

- **Java** : 21.0.1 (17 outubro de 2023 LTS)
- **Python** : 3.10.12 (20 Novembro de 2023)
- **Haskell** : 9.4.7 (versão do GHC)

Configurações / notas necessárias

Devido a alguns erros ao executar nos benchmarks do python e do haskell (o Dacapo funcionou bem à primeira), tivemos de fazer o seguinte:

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

Tratamento e Medição dos dados

Escolha dos PowerCap

Os *power caps* (em Watts) escolhidos para este trabalho foram os seguintes:

- **-1** : sem limite (opção para ver a, **em princípio**, melhor execução temporal do programa)
- **5**
- **10**

Explicação no slide seguinte

- **35**
- **45**
- **115**

Valores apontados pela Intel* como mínimo, normal e máximo Watts para que o processador funcione de forma adequado.

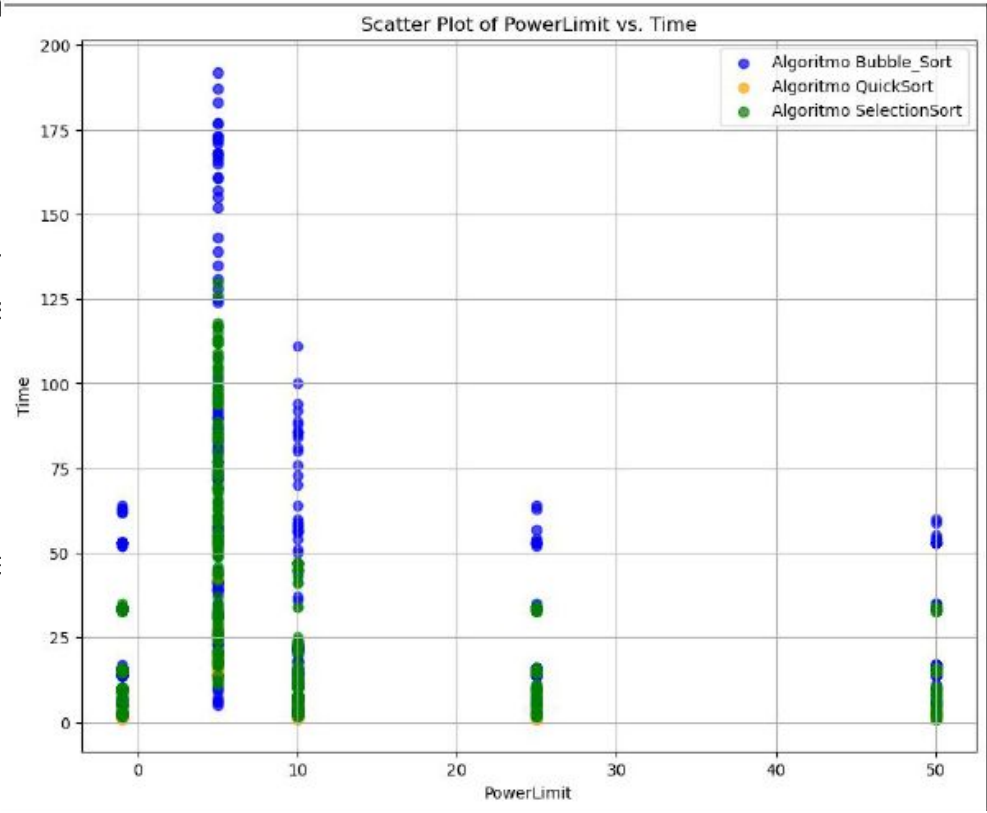
*Especificações Processador

Explicação PowerCaps 5 e 10

Como vimos no trabalho anterior (imagem
utilização do powerCap em 5 e 10 traz res
interessantes para analisar.

Por este motivo decidimos utilizar estes 2

Outra maneira de medir seria a utilização
(exercício exemplo), mas seria redundante



Escolha dos Programas

Para **cada linguagem**/benchmark escolhemos **10 programas/problemas**.

A escolha fundamentou-se numa primeira análise dos programas disponíveis nos benchmarks. Como seria impossível fazer todos, devido à complexidade temporal acrescido ao powerCap, optámos por uma seleção limitada em cada. Cada linguagem devido à complexidade e ao benchmark tem características diferente, mas tentamos escolher o mais diverso possível dentro de cada linguagem, seja por tempo seja por ações que faz no sistema.

Escolha dos Programas (II) - Java

Havia alguns programas em Java que não estavam disponíveis para a versão de Java que o sistema onde estávamos a testar tinha, como por exemplo o cassandra.

Decidimos então escolher programas que já tínhamos ouvido falar como kafka, spring, tomcat e eclipse.

Os seguintes foram decididos por uma escolha informal através da medição de vários programas com powerCap a -1 e escolher alguns com valores menores e outros maiores quer sejam temporais quer sejam de energia e de temperatura.

- avrora
- batik
- biojova
- eclipse
- kafka
- spring
- tomcat
- graphchi
- jme
- jython

Escolha dos Programas (III) - Python

Já o pyperformance tinha um total de 84 programas para benchmark. Decidimos então escolher 10. Primeiramente, o pyperformance possui vários grupos (para listar pyperfomance list_groups). Dentro destes grupos decidimos escolher :

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

Assim temos uma representação dos vários benchmarks.

Foram escolhidos em particular estes desse grupo pela mesma razão dos de java.

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

Escolha dos Programas (IV) - 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 no caso dos programas no grupo parallel.

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

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

Como foram obtidos os dados (II)

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**.

Assim garantimos não só uma maior uniformização de experiências entre linguagem prometia também resultados menos comprometidos por, por exemplo, atividades externas (como a luz diária possivelmente aumentar)

As 3 fases significam que **cada uma das Linguagens/Benchmarks** foi medido **uma por cada noite**.

Decidiu-se manter assim no run.sh tudo junto para ilustrar como poderia obter-se os resultados todos de uma vez (só não o fizemos assim devido ao tempo que ia demorar).

Dados Obtidos

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**.

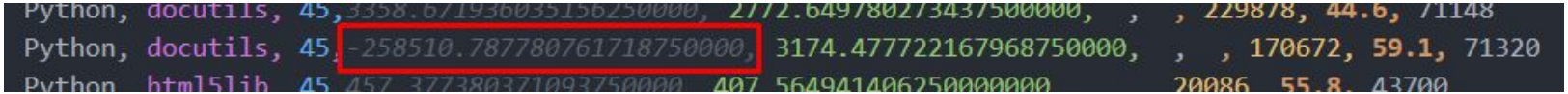
Os dados tinham ausência das colunas de GPU e DRAM pelo que as eliminamos.

Passamos também o valor de tempo para segundos (dividir por 1000).

Explicação das colunas do CSV

- **Language** : Linguagem de programação do Benchmark dos problemas/programas.
- **Program** : Nome do programa/problema que correy;
- **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 °C);
- **Memory** : Total de memória gasta durante a execução do programa (em KBytes);
- **PowerLimit** : Power cap dos cores (em Watts).

Tratamento de Outliers



Como podemos ver em cima, a **colheita de dados** é sempre **sujeita** à existência de vários **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)
- Pegamos nessas 6 linhas e **calculamos a média**, tendo assim 1 linha para cada PowerCap de cada Programa (ficando o CSV com 61 linhas)
- Para algumas situações utilizamos um data frame com a média de todos os programas (CSV com 7 linhas) e outras vezes não. Estes primeiros é para analisar a linguagem como um todo, o segundo é para analisar problemas dentro da mesma linguagem.

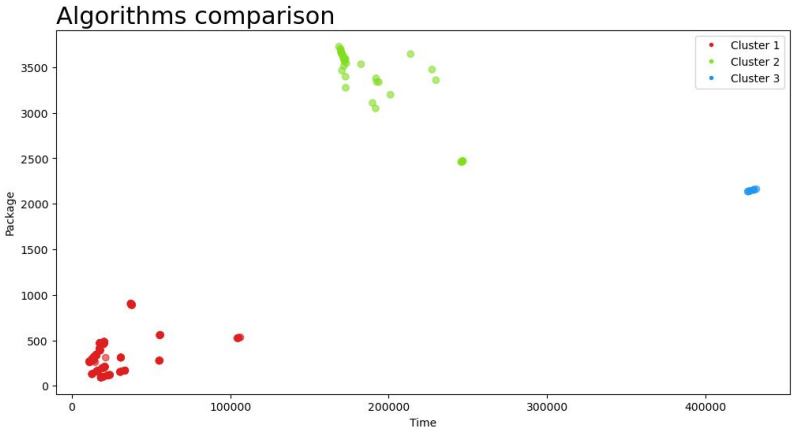
Correlação e Tipos de dados

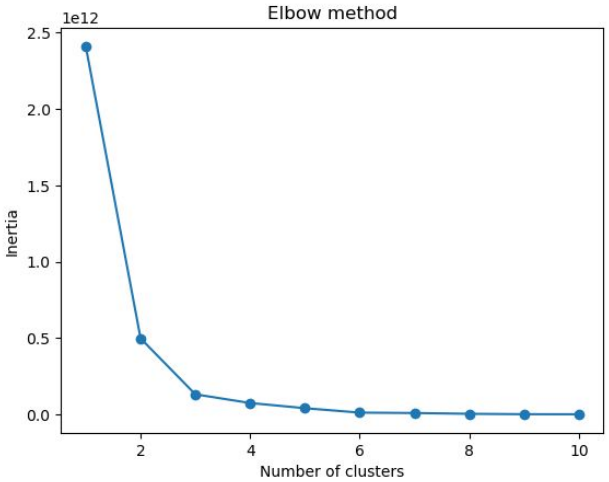
	PowerLimit	Package	Core	Time	Temperature	Memory
PowerLimit	1.000000	-0.001306	-0.042007	-0.083554	0.411436	-0.005609
Package	-0.001306	1.000000	-0.017660	-0.004208	-0.019407	0.010855
Core	-0.042007	-0.017660	1.000000	-0.023784	0.001359	0.009968
Time	-0.083554	-0.004208	-0.023784	1.000000	-0.135121	-0.020559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

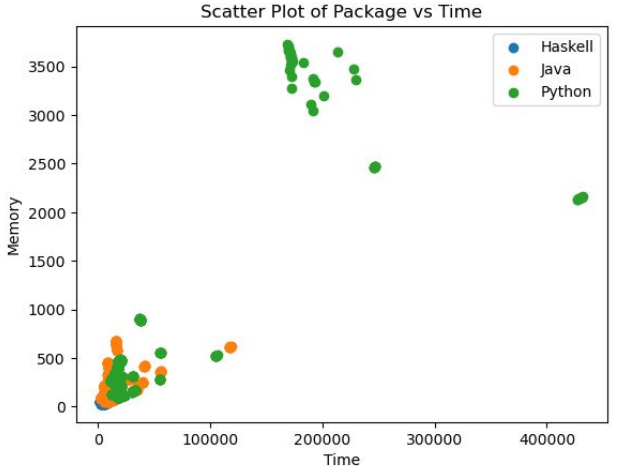
	PowerLimit	Package	Core	Time	Temperature	Memory
PowerLimit	1.000000	0.222589	0.304471	-0.284757	0.507204	0.103252
Package	0.222589	1.000000	0.968801	0.318723	0.392359	0.137643
Core	0.304471	0.968801	1.000000	0.090730	0.564471	0.197011
Time	-0.284757	0.318723	0.090730	1.000000	-0.629547	-0.205830
Temperature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

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

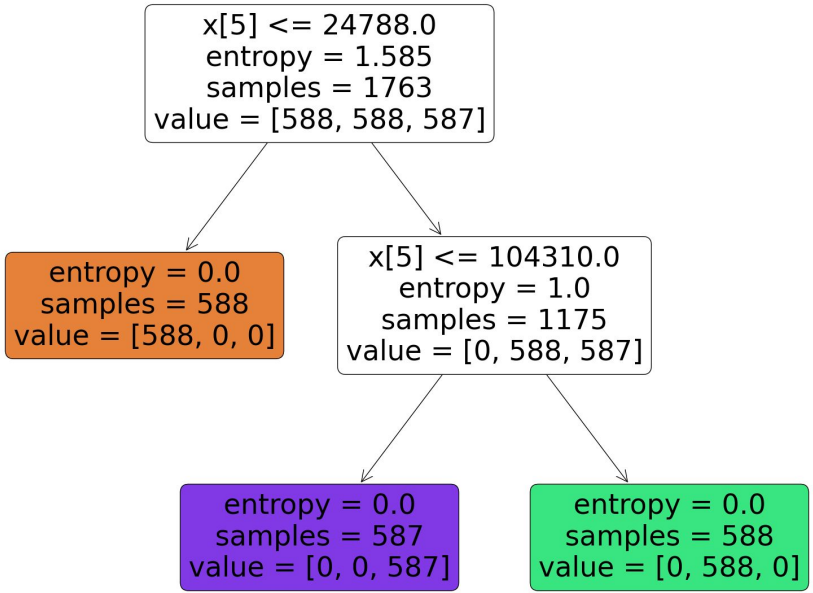
Clustering


Algorithms comparison
Cluster 1
Cluster 2
Cluster 3
Package
Time
0
500
1000
1500
2000
2500
3000
3500
0
100000
200000
300000
400000


Elbow method
1e12
Inertia
Number of clusters
0.0
0.5
1.0
1.5
2.0
2.5
2
4
6
8
10


Scatter Plot of Package vs Time
Haskell
Java
Python
Memory
Time
0
500
1000
1500
2000
2500
3000
3500
0
100000
200000
300000
400000

Decision Tree



Análise dos Resultados

Análise por Linguagem - Java

Depois do tratamento dos dados feito, onde foi feita a média por powerlimit dos programas todos juntos vimos como esperávamos que a melhor performance em tempos e pior em termos de energia foi sem limite (-1).

Vemos também uma tendência que quanto menor o powerCap em geral maior é a temperatura.

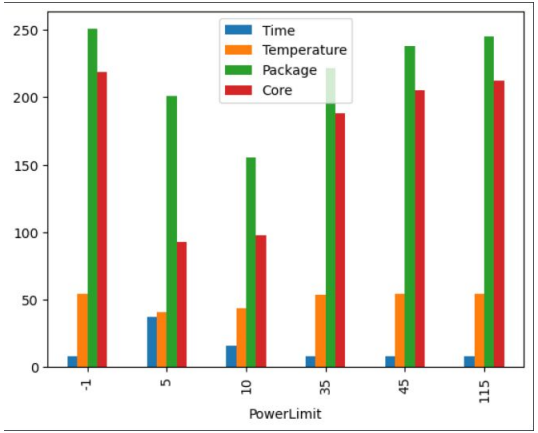
Em termos de memória não vemos uma variação muito grande. No entanto, vemos as partes mais lentas a gastar mais memória.

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

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

Análise por Linguagem - Java (II)

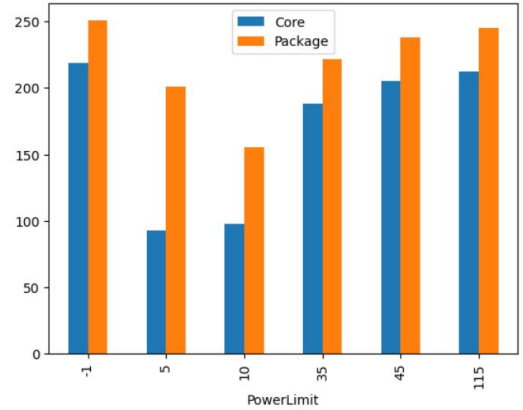
Como vimos na tabela anterior e neste gráfico de barras, uma poupança de 19,97% de energia vai levar a um ganho de 387% e aumento do tempo de execução do programa por cerca de 5 vezes.

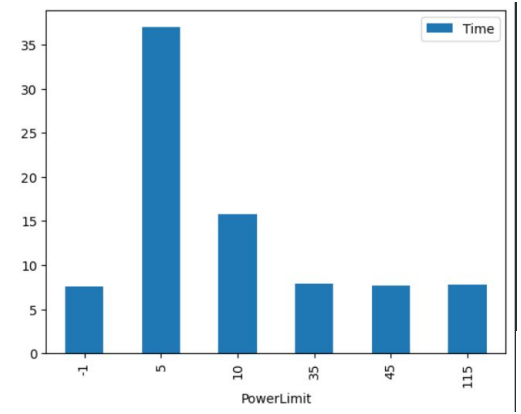


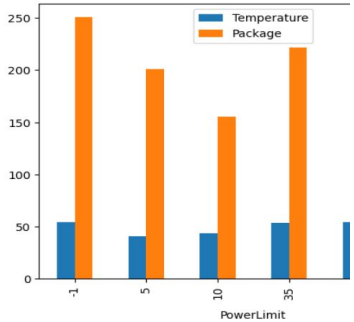
	PowerLimit	Package	Ganho Package (%)	Time	Ganho Temporal
0	-1	250.802834	NaN	7.604867	NaN
1	5	200.702263	-19.976078	37.010583	3.866697
2	10	155.065147	-22.738715	15.808850	-0.572856
3	35	221.386026	42.769688	7.930650	-0.498341
4	45	238.102351	7.550759	7.708100	-0.028062
5	115	244.885080	2.848661	7.807667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

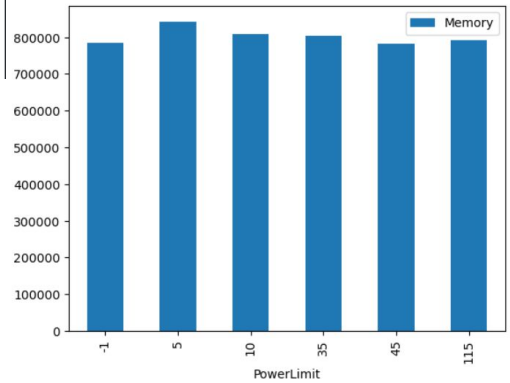
Nota: estes valores de ganho foram calculados com o método pct_change() do pandas.

Análise por Linguagem - Java (III)









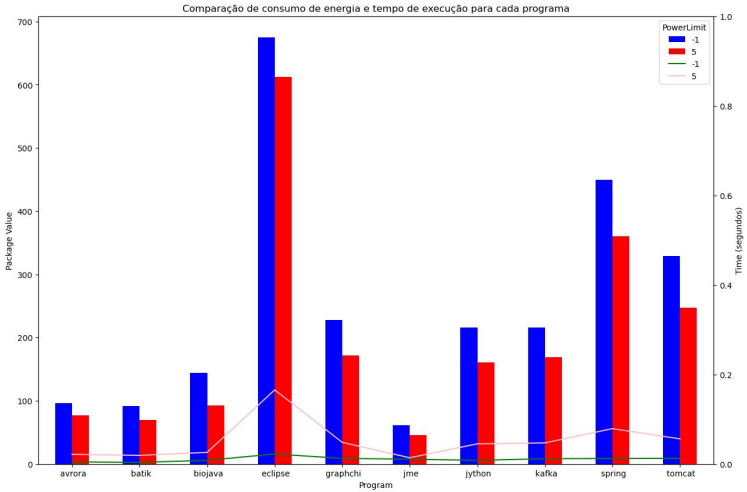
Temos neste slide, as comparações dos vários aspetos medidos para cada PowerLimit.

Análise por Linguagem - Java (IV)
Análise por programa

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

Nesta comparação entre os programas testados em Java, vemos claramente uma forte influência do Eclipse que é o programa que não consome mais energia como 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.



Análise por Linguagem - Python

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

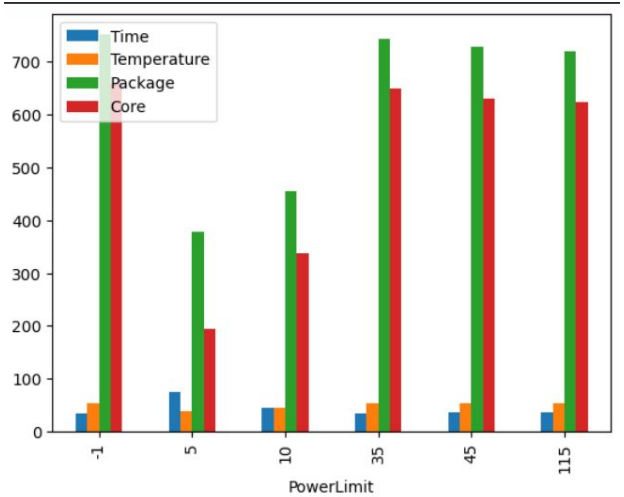
Assim como no Java, aqui verificamos que altura onde se gastou mais energia menos tempo e onde se obteve maior temperatura do processador foi sem PowerLimit.

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.

Análise por Linguagem - Python (II)

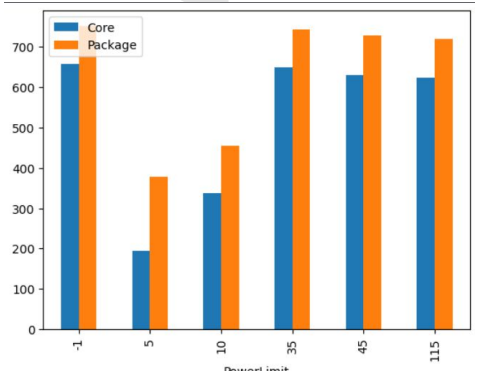
Como vimos na tabela anterior, neste gráfico de barras e na tabela em baixo, uma poupança de 49,64% de energia vai levar a um ganho de 126% (aumento do tempo de execução do programa por cerca de 2.26 vezes).

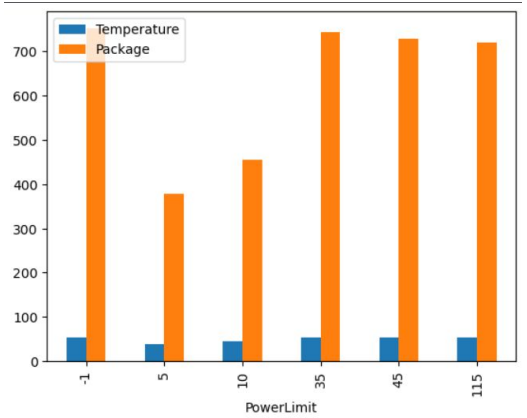


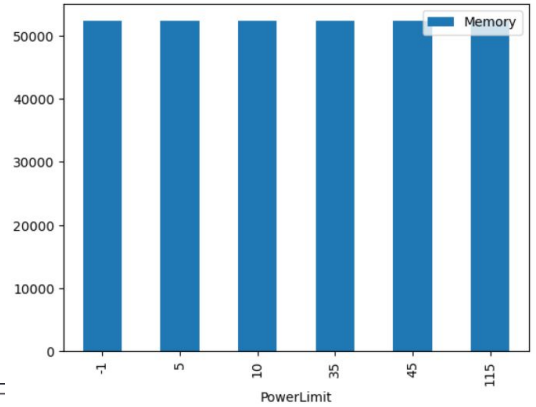
	PowerLimit	Package	Ganho Package (%)	Time	Ganho Temporal
0	-1	751.311451	NaN	33.448583	NaN
1	5	378.331167	-49.643897	75.643617	1.261489
2	10	455.020304	20.270373	45.376783	-0.400124
3	35	743.590677	63.419230	33.787933	-0.255392
4	45	728.101152	-2.083071	35.576950	0.052948
5	115	720.353229	-1.064127	35.707050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

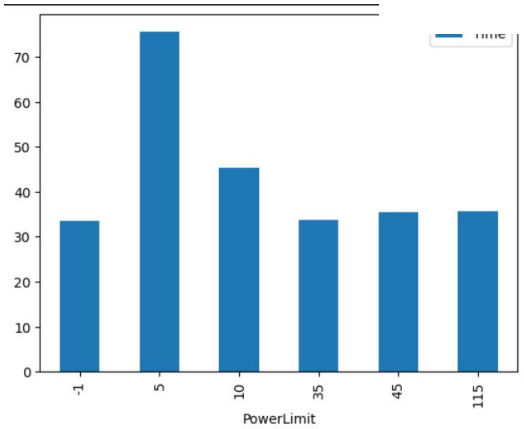
Nota: estes valores de ganho foram calculados com o método pct_change() do pandas.

Análise por Linguagem - Python (III)







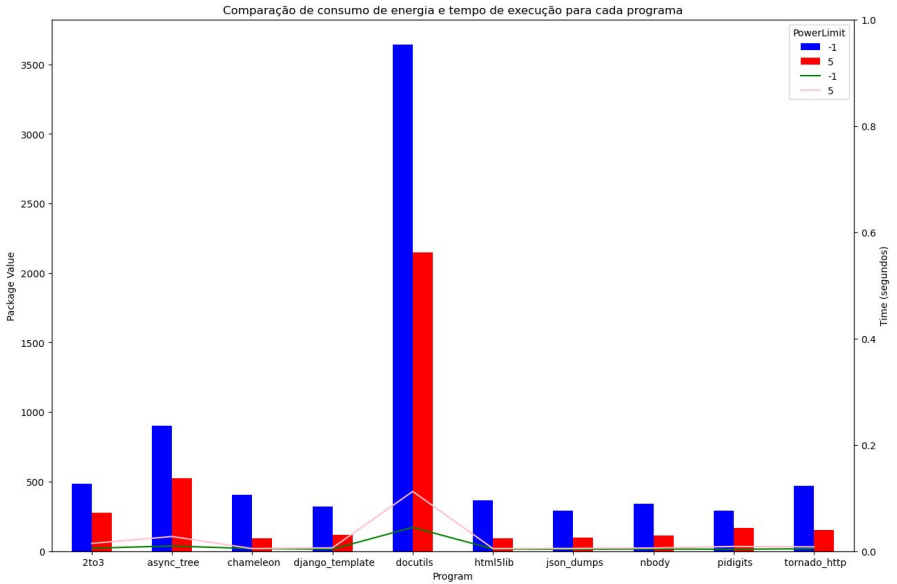


Temos neste slide, as comparaçõ

Análise por Linguagem - Python (IV)
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 power cap houve uma diminuição drástica do consumo de energia, acompanhada de uma subida de tempo não tão drástica.



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

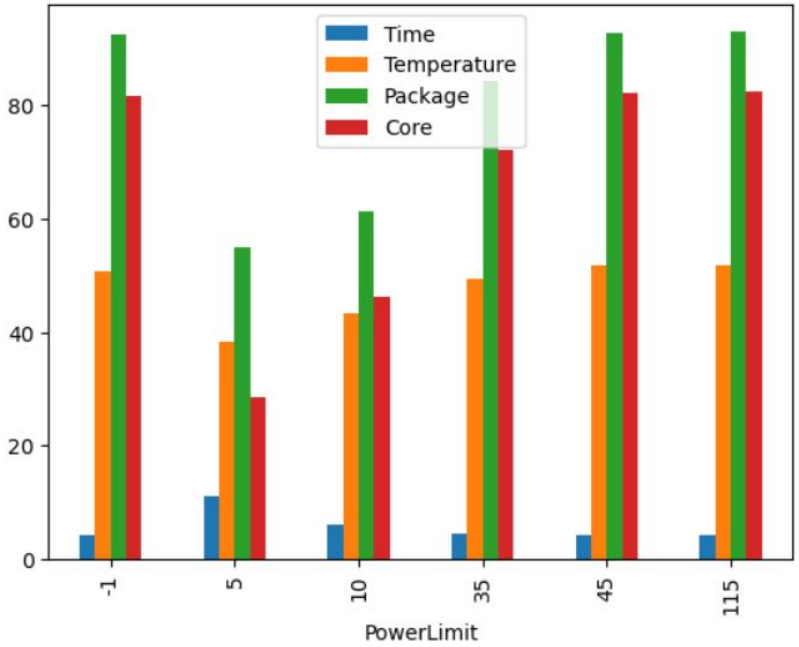
Análise por Linguagem - Haskell

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

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

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

Análise por Linguagem

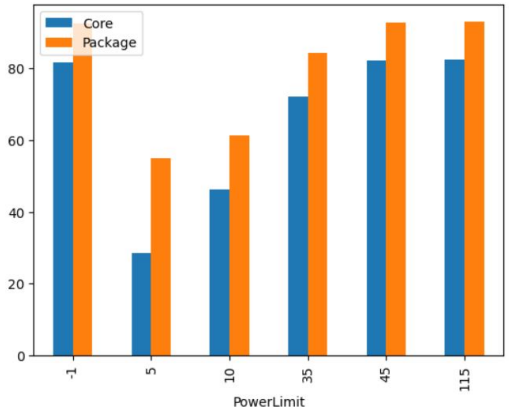


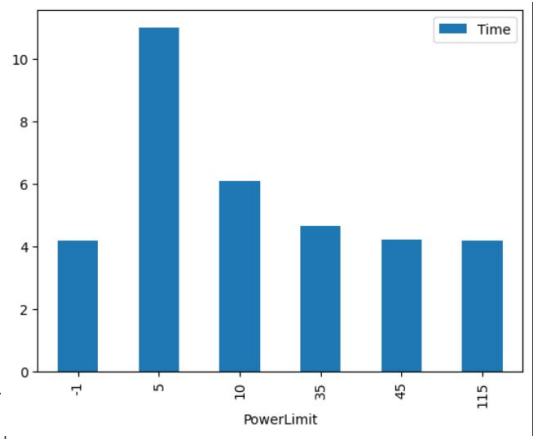
Vemos que uma poupança de 40,66% de energia vai levar a um ganho de 161% de tempo. Ou seja, o programa demorou cerca de 2.61 vezes mais.

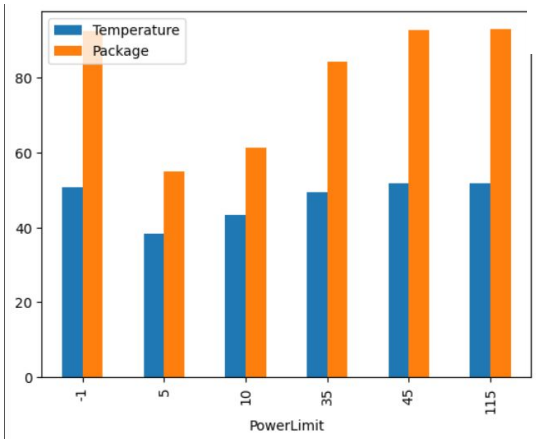
	PowerLimit	Package	Ganho Package (%)	Time	Ganho Temporal
0	-1	92.666518	NaN	4.201333	NaN
1	5	54.986652	-40.661792	11.002100	1.618716
2	10	61.466687	11.784743	6.110017	-0.444650
3	35	84.388919	37.292122	4.645300	-0.239724
4	45	92.918870	10.107904	4.233317	-0.088688
5	115	93.058338	0.150097	4.190133	-0.010201

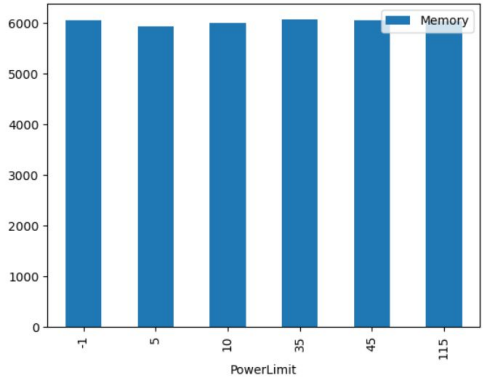
É de notar nesta tabela como nas outras o ganho é relativo ao powerlimit anterior.

Análise por Linguagem - Haskell (III)









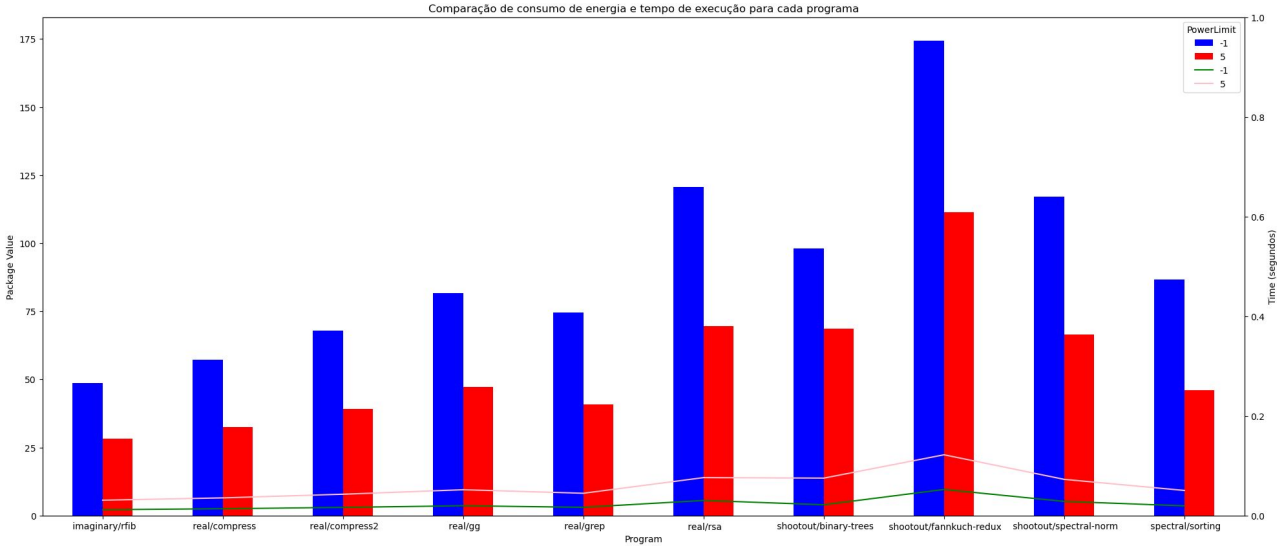
Temos neste slide, as comparações dos vários aspetos medidos para cada PowerLimit.

Análise por Linguagem - Haskell (IV)
Análise por programa

Nota : As linhas são os tempos e as barras são os valores dos packages, para cada um dos power cap. Tudo em média.

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.



Análise de Resultados
Comparação 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.

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 \textit{benchmarks}, isso não é possível.

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%.

Extra : Diferentes versões de python

	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

- Verificamos que o Python 3.10 gasta muito mais energia que o Python 3.6 mas também demora bem menos tempo. Significa então que em termos de gastos energéticos a evolução das versões do Python tornaram-no pior, mas melhoraram substancialmente em termos de tempos.
- 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.

Conclusões e Aspetos a melhorar

Conclusões

Ao longo deste trabalho aprendemos sobre benchmarks, sobre medições de energia e sobre várias linguagens.

Aprendemos também sobre um lado que não costumamos a ver das linguagens de programação como a eficiência energética e o efeito desta no tempo.

Creemos que fizemos um bom trabalho e ficamos satisfeitos.

Aspetos a Melhorar

Entre os vários aspetos a melhorar estão:

1. 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)
2. Utilizar o condon para melhorar a performance do python (ser compilado em vez de interpretado.)*

*embora o pyperformance use cython que já ajuda bastante nisto

Notas do Zip de envio

Não colocamos os vários benchmarks nos zips, uma vez que são bastante pesados.

Estrutura muito semelhante à demonstrada nas aulas práticas.

Contém um README.md com informação sobre como correr o programa.

Para correr as medições do programa existia uma script chamada **run.sh e run2.sh (está melhor explicado no relatorio).**

Muito Obrigado!

João Alvim (pg53902)

Simão Barroso (pg54236)

Simão Matos (pg54239)

13 Junho de 2024