Universidade do Minho

4 º ano do Mestrado Integrado em Engenharia Informática

Métodos Formais em Engenharia de Software

Análise e Teste de Software

CaparicaPost

Análise e Otimização de Consumo de Energia de uma Aplicação Java

Diogo José Linhares Couto – 71604
Gil Gonçalves – 67738
Pedro Silva - 67751

Introdução

Neste presente relatório é explicado as estratégias utilizadas para a otimização da performance energética de uma aplicação que gere uma agência noticiosa, desenvolvida com a utilização da linguagem de programação JAVA.

Esta aplicação foi desenvolvida por alunos no seu primeiro curso de programação em JAVA.

Como tal é natural que devido a sua falta de experiência em programação JAVA, as decisões tomadas durante o processo de desenvolvimento nem sempre a tivessem eficiência do programa como objetivo principal.

Isso dá origem a pedaços de código ineficientes e com a possibilidade de serem otimizados.

Foi então utilizada a ferramenta *jRAPL*, para fazer a análise de performance do software fornecido tendo em consideração três fatores, tempo de execução, consumo de memória e o consumo de energia

Usando essa análise como base prossegue-se à localização e otimização de secções de código ineficientes, e a sua respetiva comparação dos resultados de performance entre os vários estados de otimização.

Nesta fase ainda não foi tido em atenção a utilização de nenhuma estratégia de Programação Verde em específico, pois esse tipo de solução é apenas para ser implementada na fase seguinte.

Otimizações

Para testar este projeto foi disponibilizado uma *framework black-box testing*. Esta *framework* consiste em dois inputs diferentes e, para cada um deles um output esperado correspondente.

Desta forma, desde que se invoque a classe *Main* para cada um dos inputs, e se obter o seu output esperado o projeto faz o que é pedido.

Assim para facilitar estes testes foi criado um *bash script* com o nome de "run.sh" que devolve "OK", se o output do programa for igual ao output esperado ou "KO", no caso contrário. Este script também fornece a performance do programa para cada um dos outputs.

De modo a apresentar resultados mais fiéis e descartar ligeiras oscilações de valores, cada um dos testes foi executado 5 vezes para cada um dos inputs, e foi calculada a média desses resultados.

Foram utilizados dois computadores para verificar se os resultados eram semelhantes em máquinas diferentes, deste modo eliminando a influência do estado geral da máquina.

O PC-1 tem um processador Intel® Core™ i7-4750HQ, uma gráfica GeForce950m e 8.00GB de memória RAM.

O PC-2 tem um processador Intel® Core™ i5 Dual Core -3210M 2,5 GHZ, uma gráfica *GeForce GT* 640m 2GB e 6.00GB de memória RAM.

Assim, quando se analisou o programa antes de qualquer otimização, foi obtido o seguinte registo de dados de performance para os inputs *t1.txt* e *t2.txt*:

Tabela 1 - t1.txt

	PC - 1	**PC - 2**
GPU (J)	0.0927124	1.42538075
CPU (J)	27.406506	38.8563805
Package (J)	46.2936158	52.673691
Time (*Seconds*)	1.9608421296	3.20223880575

Tabela 2 - t2.txt

	PC - 1	**PC - 2**
GPU (J)	0.074414	0.39575175
CPU (J)	13.7276366	17.581802
Package (J)	21.7070924	23.218914
Time (*Seconds*)	0.7927938266	1.372134181

Depois de retirar os resultados preliminares, foi analisado o código fonte e feito o seu *profilling,* para descobrir onde otimizar o programa.

Antes de mais foi verificado que o código está repleto de múltiplos casos *if* e *else*. O que se tentou

otimizar com o simples uso do *switch*. Quando o *switch* é implementado para várias hipóteses, é criada uma tabela de *hash* ao usar o *lookup table* esperasse ganhos a nível de tempo de execução.

No entanto isso não se verificou para nenhum caso no PC-1, tendo mesmo uma perda de desempenho, e apenas em pequena quantidade no PC-2 mesmo assim piorado alguns dos factores.

Assim, foi feito o *refactoring* e implementado os *switch* obtendo os seguintes resultados:

Tabela 3 - t1.txt

	PC - 1	PC - 2	Ganho% PC1	Ganho%PC2
GPU (J)	0.236548	0.9743222	-155.142	31,64477632
CPU (J)	33.904004	48.38868725	-23.7079	-24,53215309
Package (J)	55.7367678	51.5919218	-20.3984	2,05371824
Time (Seconds)	2.1529077218	3.1075978418	-9.79506	2,955462403

Tabela 4 - t2.txt

	PC - 1	PC - 2	Ganho% PC1	Ganho%PC2
GPU (J)	0.1036622	0.5877808	-28.5936	-48,5226029
CPU (J)	14.7666872	17.0215942	-6.08486	3,186293419
Package (J)	22.7150998	22.6629304	-2.99707	2,394528874
Time (Seconds)	0.7674884586	1.3127040566	5.16956	4,331218129

Gráfico 1 – Alteração *switch* t1.txt PC-1

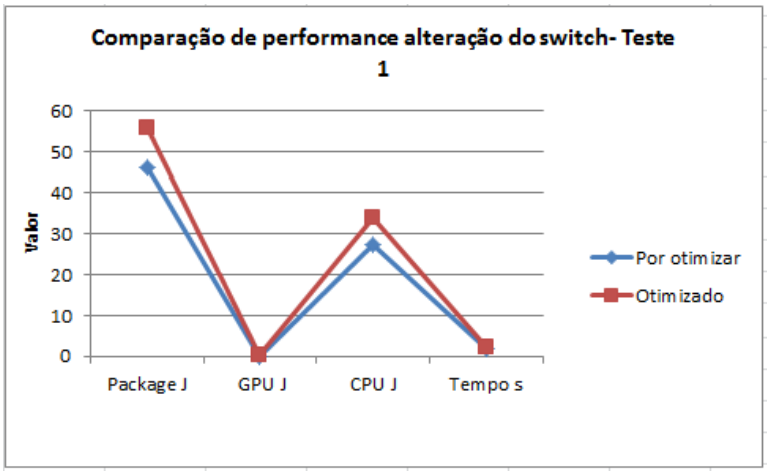


Gráfico 2 – Alteração *switch* t2.txt PC1

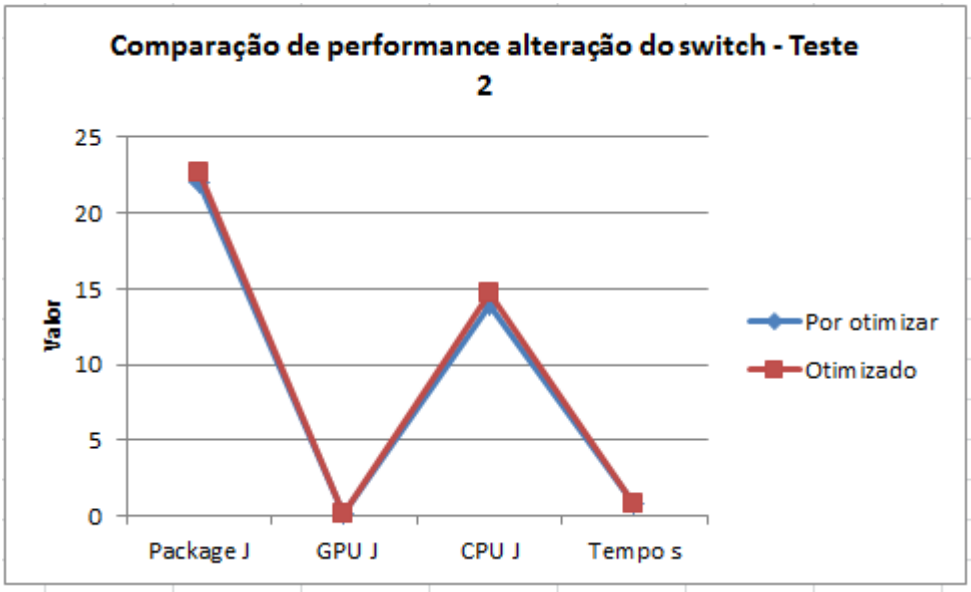


Gráfico 3 – Alteração *switch* t1.txt PC2

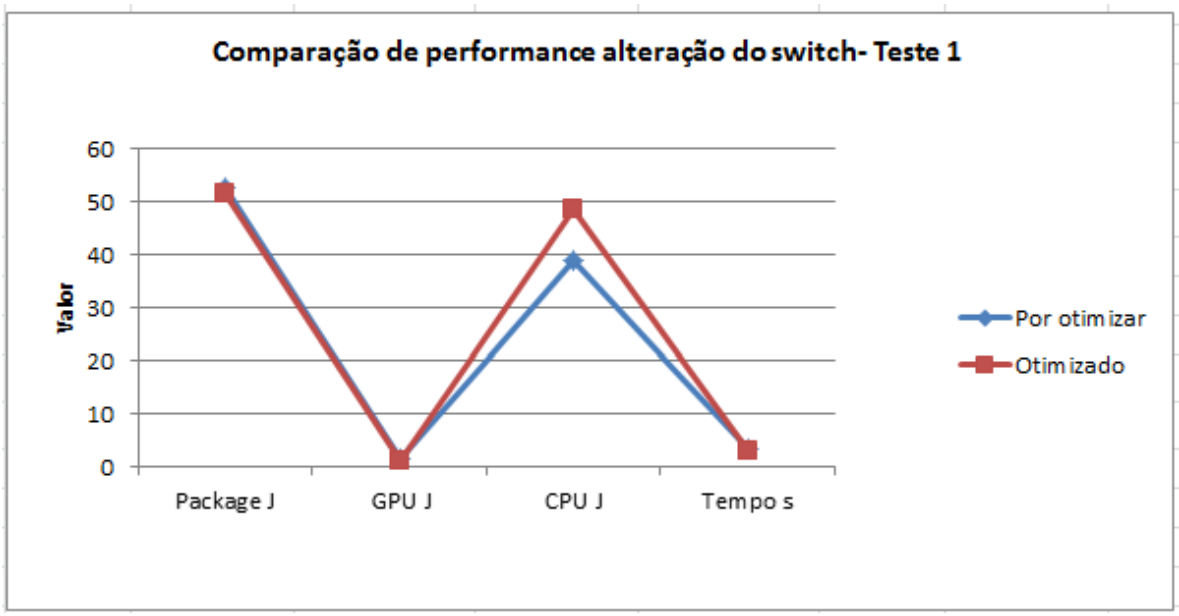
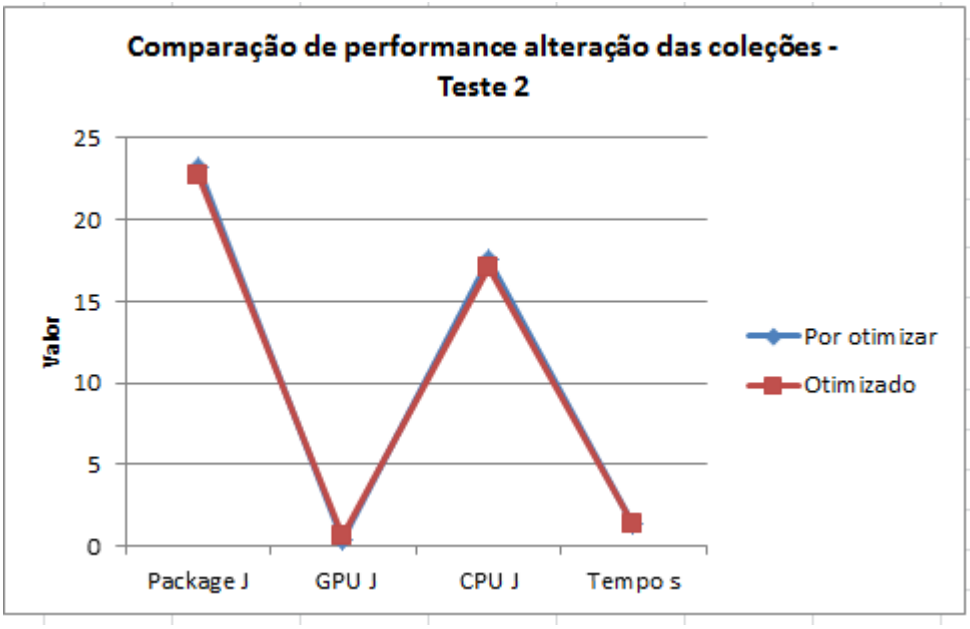


Gráfico 4 – Alteração *switch* t2.txt PC2



Depois verificou-se que estava a ser usada uma lista *List<User>* para os utilizadores e um *List<Article>* para guardar os artigos.

Visto que sempre que os utilizadores são impressos, eles devem estar ordenados, dividiu-se esta lista em quatro coleções diferentes do tipo *TreeSet<User>.* Cada uma destas coleções vai ter um tipo diferente de utilizador: leitores, editores, jornalistas ou colaboradores.

Assim é garantido que estas coleções estão sempre ordenadas, pois são ordenadas na inserção, e não é necessário ordená-las sempre que são impressas como acontecia anteriormente.

Antes quando era para imprimir por tipo de utilizador era necessário percorrer os utilizadores todos e verificar se eram do tipo certo, agora simplesmente imprime-se a coleção que tem os utilizadores daquele tipo.

Foi também criado um *TreeMap<String,User>* em que a chave *String* vai ser o nome do utilizador e o valor *User* vai ser o objeto de cada utilizador.

Esta mudança permite percorrer apenas uma estrutura de dados para verificar se um utilizador se encontra registado em vez de percorrer as 4. Obtendo melhores resultados para um número grande de utilizadores.

Para os artigos foi alterado o *ArrayList<Article>* para duas coleções diferentes para cada um dos tipos de artigo, e armazenados numa coleção *TreeMap<String,Article>* .

A razão é a mesma para o qual a alteração foi feita nos utilizadores, para melhorar os tempos de

procura e reduzir o tempo ao ordenar.

Estas alterações implicam um ligeiro aumento na memória pois vão ser utilizados mais estruturas e inserida alguma redundância para guardar as coleções, no entanto as medidas de performance tiveram ganhos acentuados como podemos ver na seguinte tabela:

Tabela 5 - t1.txt

	PC - 1	PC - 2	Ganho% PC1	Ganho% PC2
GPU (J)	0.07924779999	0.6300908	14.52298	35,33034555
CPU (J)	17.8698362	33.9977356	34.7971	29,74032252
Package (J)	31.091931	45.0657078	32.83754	12,64968191
Time (Seconds)	1.4439319634	2.7017811838	26.36164	13,05885377

Tabela 6 - t2.txt

	PC - 1	PC - 2	Ganho% PC1	Ganho%PC2
GPU (J)	0.057837	1.1025938	28.25284	-87,5858824
CPU (J)	10.225354	22.875485	26.54038	-34,3909667
Package (J)	16.2525632	30.7739626	26.30601	-35,789865
Time (Seconds)	0.6094794914	1.6971776764	24.69306	-29,28867462

Gráfico 5 – Alteração das coleções t1.txt PC-1

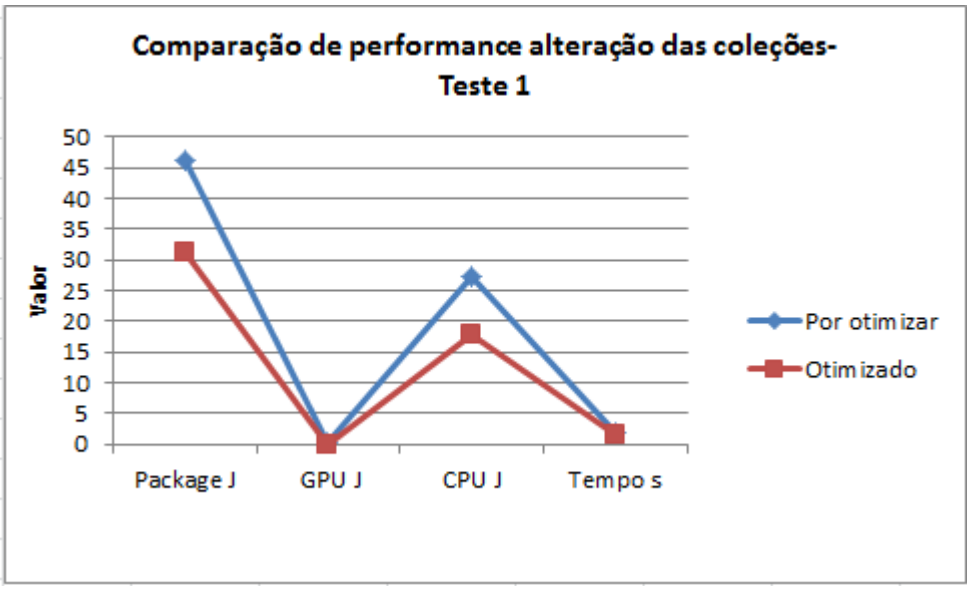


Gráfico 6 – Alteração das coleções t2.txt PC-1

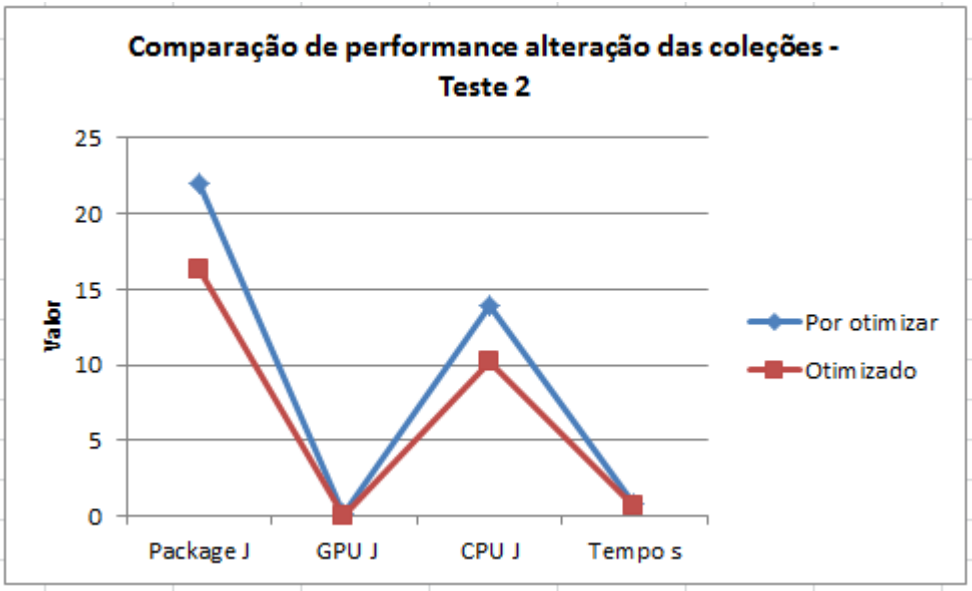


Gráfico 7 – Alteração das coleções t1.txt PC-2

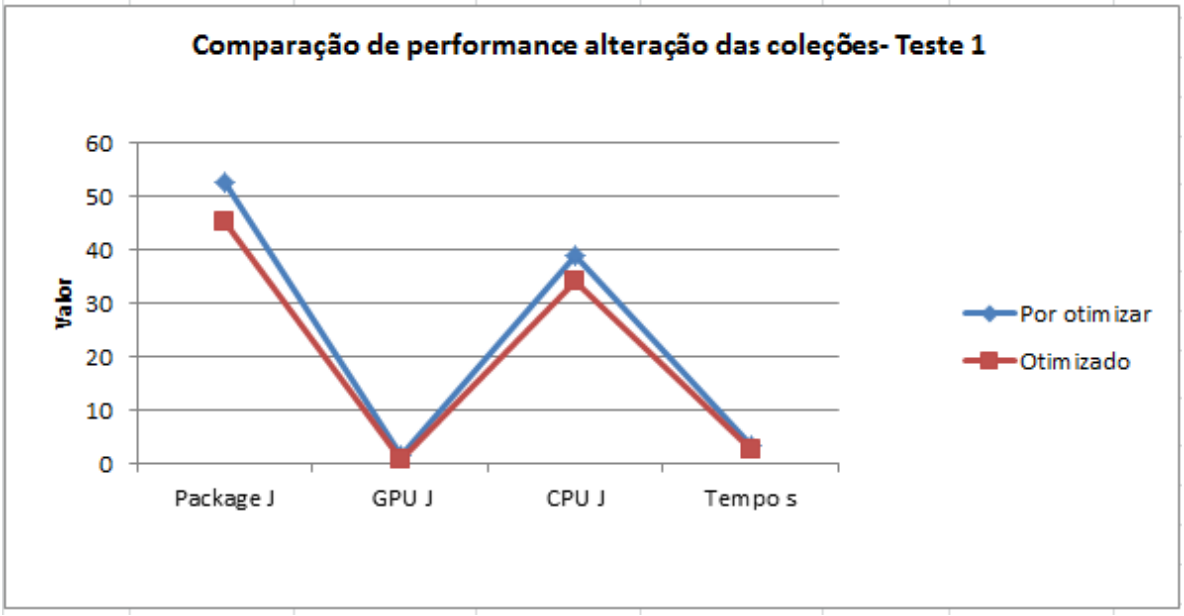
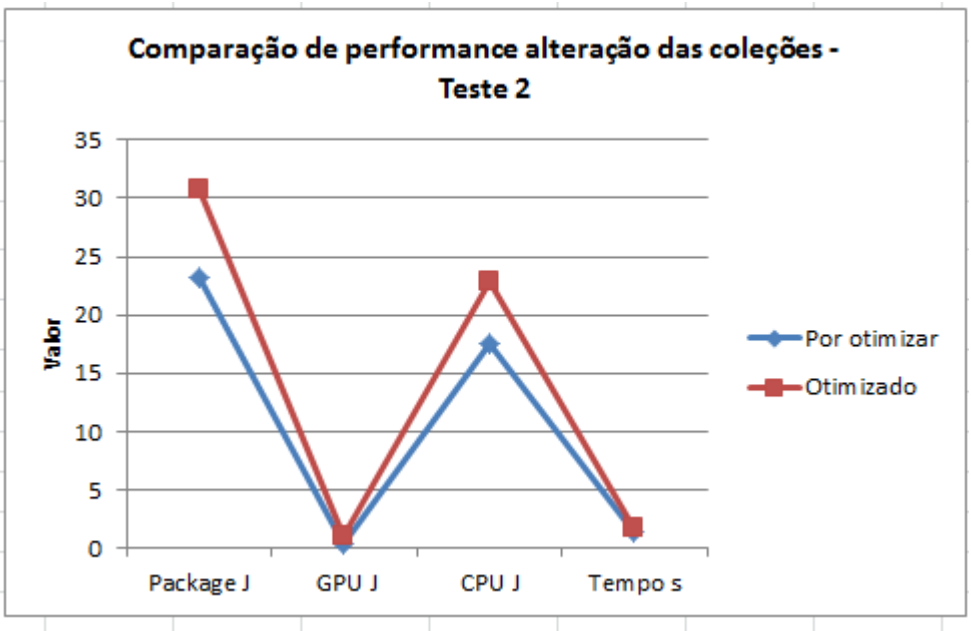


Gráfico 8 – Alteração das coleções t2.txt PC-2



Neste caso o teste teve um comportamento inesperado, pois tendo em conta o teste com t1.txt, também era esperado ter ganhos nesta situação.

Desconfiámos que a máquina teria algum processo pesado a correr na altura dos testes, para além do nosso conhecimento.

Gráfico 9 – Comparação três alternativas t1.txt PC1

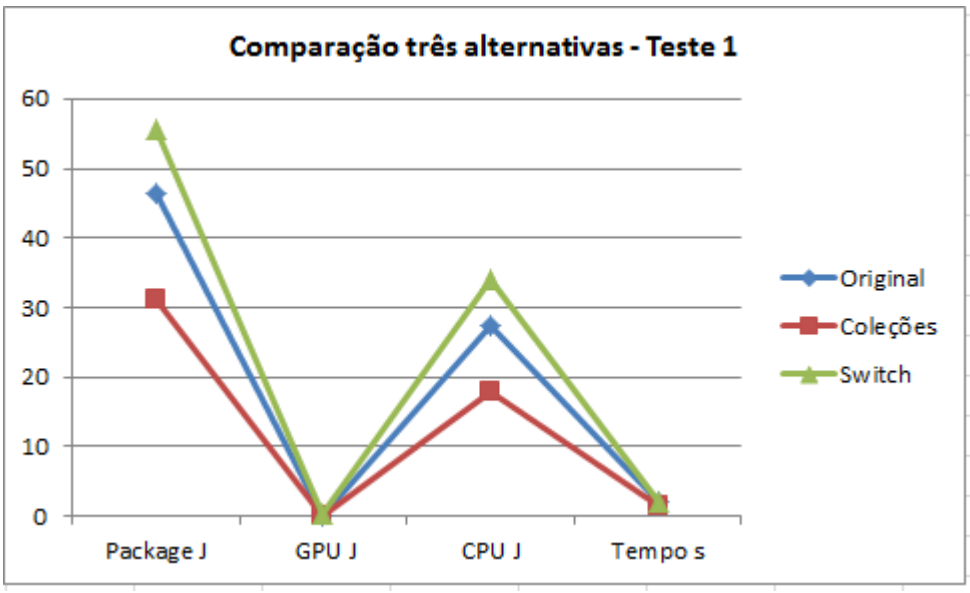


Gráfico 10 – Comparação três alternativas t2.txt PC1

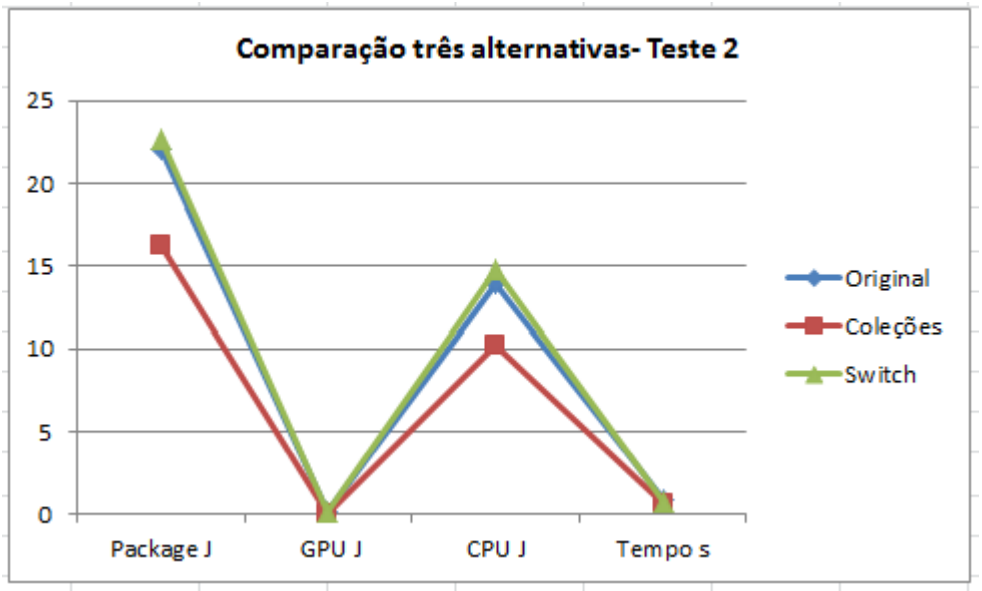


Gráfico 11 – Comparação três alternativas t1.txt PC2

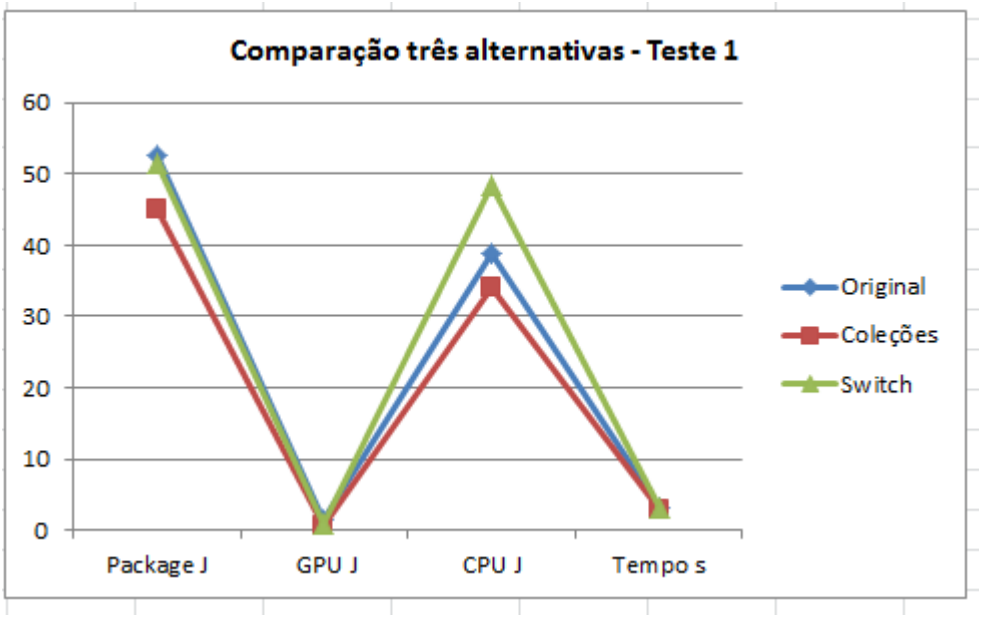
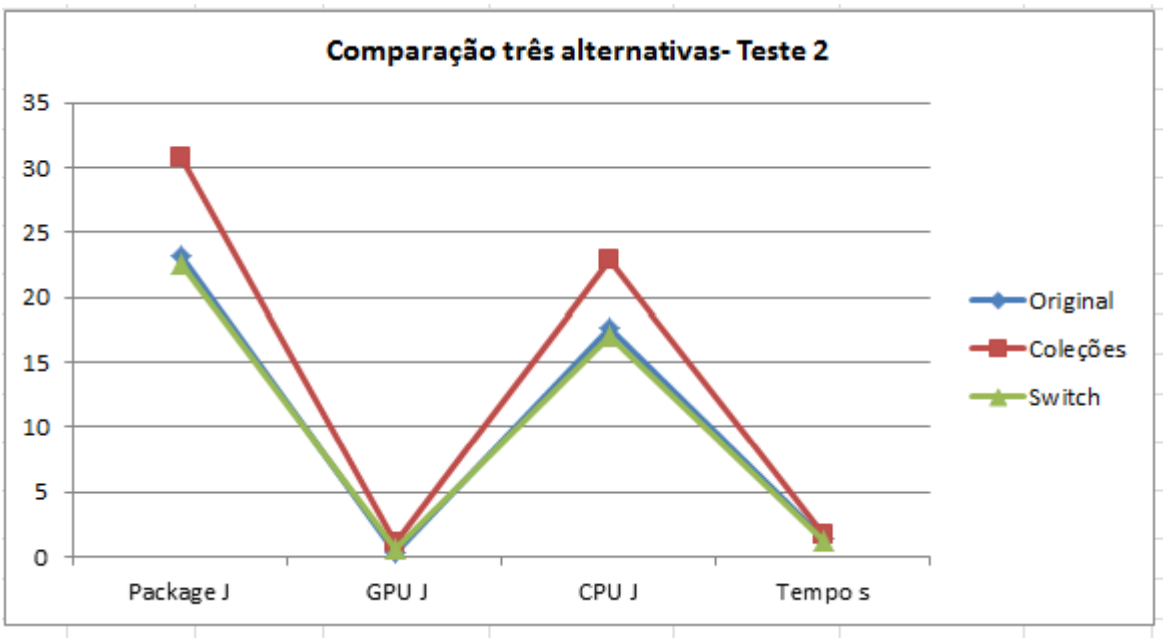


Gráfico 12 – Comparação três alternativas t2.txt PC2



Conclusão

Nesta fase do projeto já se verifica um ganho considerável de performance nos vários aspetos analisados.

No entanto o processo de otimização ainda não está terminado.

Na próxima fase vão ser aplicadas de forma estruturada, técnicas específicas de melhoramento de performance, ao contrário da estratégia de otimização usada nesta fase.

Para esse efeito devem ser investigadas e aplicadas estratégias conhecidas e específicas de Programação Verde, um detalhe que não foi tido em consideração nesta primeira fase de otimização.

Como foi verificado mais tarde na aula verificamos o porque de usar *switch case* trouxe um mau desempenho. Para a segunda fase fica a eliminação do *switch* usando um *hashamap*, até porque a utilização de um *switch* é um *bad smell*.