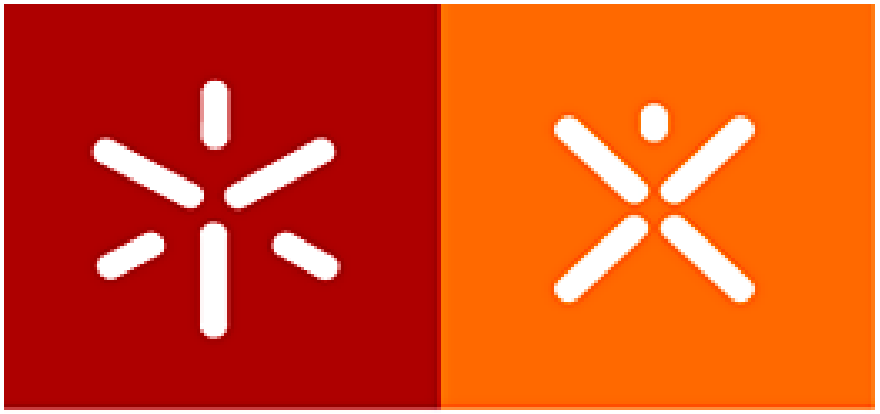


Universidade do Minho

Licenciatura em Engenharia Informática

Laboratórios de Informática III 2023/2024

Grupo 28

André Pereira (a 104275) Marco Gonçalves (a104614) Bruno Miguel (a94662)

Resumo do progresso realizado

Nesta última fase do trabalho focamo-nos ,inicialmente, na realização das queries em falta, nomeadamente, as queries 1,2,7 e 10.

Começamos pela **query 1** , para isto vimos a necessidade de criar uma struct para passageiros bem como novos tipos de **árvores** das quais a maioria foi eventualmente substituída por **hash_tables(glib)** de forma a otimizar a performance.

De seguida resolvemos a **query 2,** que nos trouxe várias dificuldades. Nesta query criamos um grande número de funções auxiliares, fruto da nossa falta de pré-planeamento. As informações necessárias para a realização desta query tinham sido ignoradas até este ponto e só agora demos conta da quantidade vasta de adaptação necessária para usufruir dela.

Continuamos com a **query 7** e com o início da implementação do modo **interativo**.Para a query 7 encontramos uma solução que se manteve intacta até ao final do programa, sendo apenas otimizada a alocação de espaço desnecessária.

Neste ponto do trabalho atingimos uma barreira. As nossas **árvores** podiam e deviam em vários casos ser **substituídas** por **hash_tables**. Até este momento a execução do large dataset era impossível. Esta implementação foi primeiro usada nas queries 5, 6 e 7. Esta remoção afetou também as **árvores obsoletas** que tínhamos noutros módulos.

O maior obstáculo na **execução** do "**dataset large**" era agora o **excesso de memória** alocada pelo programa; nesse sentido seguiu-se uma remodelação de várias structs de modo a conterem **arrays** de caracteres **estáticos** ao em vez de dinâmicos.

À medida que fomos otimizando o programa fomos também **modificando** levemente todas as **queries** já implementadas, permitindo a execução do "dataset large", e fomos também lentamente atualizando o nosso modo interativo, adicionando novos e **melhores menus**.

Antes de prosseguir com a realização da **décima** e última **query**, decidimos **remover os erros do valgrind**, visto que estes **superaram** o limite de **10.000.000**. Para isto **removemos** grande parte das **comparações com NULL**, tendo estas sido trocadas por contadores que são passados quando necessário, por exemplo, ao percorrer uma matriz, garantimos a inicialização de todos os vetores e apontadores utilizados, entre várias outras pequenas mudanças.

Criamos um **módulo** de utilidades que lida com todas as funções que envolvem "**tempo**", de modo a facilitar a sua utilização. Implementamos uma funcionalidade de ler **inputs** do utilizador no menu de queries, **com verificação** dos mesmos.

Finalmente voltamos à **query 10**. Para o seu desenvolvimento vimo-nos forçados a pensar "fora da caixa". Ainda assim acabou por ser a query **impiedosa para a**

performance do trabalho. Enquanto todas as outras queries se limitam a uma impressão glorificada de dados previamente tratados, esta **query obriga** a um **parsing específico**.No que toca ao número de utilizadores, reservas e voos foi relativamente simples criar árvores que guardam no dia correspondente cada registo dessas variáveis. No entanto, o cálculo de passageiros e passageiros únicos envolve **percorrer uma matriz** onde estão guardados todos os utilizadores válidos. Para além disso é necessária a procura prévia dos voos relativos ao período pretendido. O tempo de execução das queries aumentou de, aproximadamente, **1.5 segundos para 11.5** segundos no "large dataset" (computador 1) apenas com a adição desta querie.O que torna o processo tão **pesado** é a quantidade de **inserções em hash_tables** ao percorrer a matriz no modo de print total (input vazio) tudo a fim de contabilizar os passageiros únicos associados a cada ano.

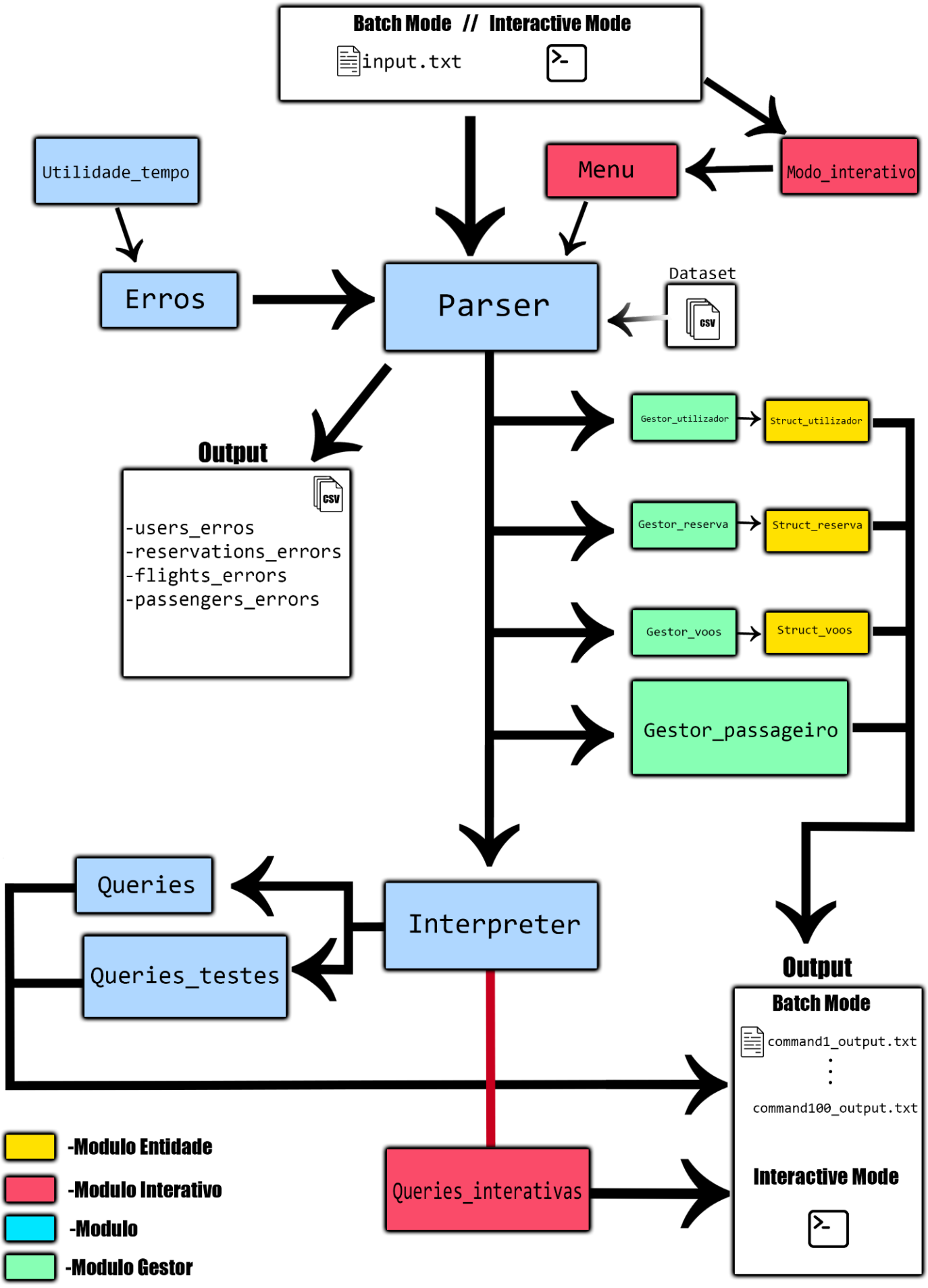
Com as **queries finalizadas**, o trabalho estava finalmente **100% funcional**, mas ainda havia várias **otimizações** que nós sabíamos serem "essenciais", para além da **falta de modularidade**.

Finalizamos o modo interativo, com uma **interface bem construída**, detalhada e intuitiva, da qual nos orgulhamos. Implementamos todas as queries, descrição das mesmas, bem como uma função de paginação e diversos exemplos que facilitam a utilização do programa.

De modo a **acelerar** o processo de **parsing das reservas** (o mais lento) substancialmente, substituímos a nossa árvore de procura genérica por uma **árvore de procura balanceada**. Vimos diminuições consideráveis do **tempo de execução total** do programa (aproximadamente **20%**); o tempo de parsing foi efetivamente dividido por 2.Criamos o módulo de testes e continuamos com a diminuição de alocação de memória através da **remoção de alocações desnecessárias**.

Na criação do trabalho tivemos também em conta o ficheiro dos inputs e os dados que seriam solicitados pelo mesmo, ou seja, **existe** uma espécie de **pré-processamento** onde todas as reservas(**queries 1,2,3,4 e 8**), voos de um utilizador (**query2**) e nomes de certas iniciais (**query 9**) sao incluidos numa hashtable permitindo distinguir quando devemos guardar certas informações ou quando é desnecessário para a execução do programa, **diminuindo** não só o **tempo de processamento** como a **quantidade de memória alocada**.

Na fase **final** do trabalho restou-nos apenas corresponder ao fator da **modularidade** que estava omisso, deste modo, criamos **novos módulos** para o programa de testes e o modo interativo e separamos as entidades e a gestão das mesmas em 2 módulos distintos.


Batch Mode // Interactive Mode
input.txt
Utilidade_tempo
Menu
Modo_interativo
Erros
Parser
Dataset
CSV
Output
CSV
-users_erros
-reservations_errors
-flights_errors
-passengers_errors
Gestor_utilizador
Struct_utilizador
Gestor_reserva
Struct_reserva
Gestor_voos
Struct_voos
Gestor_passageiro
Queries
Queries_testes
Interpreter
Output
Batch Mode
command1_output.txt
command100_output.txt
Interactive Mode
Queries_interativas
-Modulo Entidade
-Modulo Interativo
-Modulo
-Modulo Gestor

Funcionamento da aplicação (Modo Batch)

No início da execução o programa irá fazer um pré -processamento dos dados através do ficheiro de inputs onde regista os dados de reservas, passageiros e utilizadores que necessitam de atenção especial nos parsings futuros.

Depois disto realizamos o parsing genérico de utilizadores,reservas e voos; no parsing de utilizadores guardamos todas as linhas válidas numa matriz e inserimos todos os erros numa hash_table e no ficheiro de erros.

No parsing das reservas guardamos também todas as reservas válidas numa matriz , usufruindo desde já da hash de utilizadores inválidos para identificar erros, guardamos o maior índice entre todas as reservas, contamos o número de reservas válidas e preenchemos o ficheiro de erros de reservas. Para além disto filtramos todas as reservas que não serão necessárias para a execução das queries pré processadas no ficheiro de inputs.

Finalmente no parsing de voos testamos a linha e, caso seja válida, guardamos numa matriz, registramos o número de voos total, número de voos válidos e que voos são voos errados através de uma hash correspondente ao id desse voo. Com o número de voos criamos uma hash que irá registar os passageiros por voo.

Antes do último parser genérico (passageiros) foi-nos útil realizar o parsing específico dos utilizadores. Neste parsing usufruímos de um struct utilizador e guardamos esses novos utilizadores em duas principais estruturas : uma hash_table onde cada inicial de um nome corresponde a um índice e esse índice que aponta para uma árvore binária onde todos os utilizadores com essa inicial estão organizados (**Querie 9**); e também uma outra hash_table onde temos uma lista de todos os utilizadores necessários para as queries inseridos pelo seu id exclusivo(**Queries 1 e 2**) aproveitamos o pré processamento que limita profundamente a quantidade de utilizadores a processar e inserir.

Finalmente temos uma árvore não binária como estrutura auxiliar para a **query 10** onde registramos os utilizadores no dia correspondente à criação das suas contas .Este parsing específico foi pertinente antes do parsing dos passageiros pois é neste parsing(passageiros) que realizamos um registo dos

voos de cada utilizador , ou seja, com a hash_table de id criada previamente podemos aceder á struct utilizador associada a um certo passageiro e , caso seja um passageiro válido, registar num valor dentro da própria struct utilizador o número de voos total associado ao mesmo;também registramos num array a quantidade de passageiros em cada voo recorrendo ao id de voo associado a cada passageiro. Esses processos são realizados na validação de passageiro, enquanto que no restante do processo de parsing registramos os passageiros válidos numa matriz e contamos o número deles.

Agora passamos ao parsing das reservas, este que é o mais longo e ineficiente de todo o projeto pois envolve inserções em árvores AVL de grande dimensão. Aqui realizamos um registo similar ao de utilizadores em que inserimos as reservas numa árvore não binária de forma a identificar o número de reservas realizadas por cada dia (**Query 10**) vale ressaltar que as queries filtradas no parser genérico inicial foram já registadas de modo a calcular os valores corretamente. Todas as reservas guardadas na matriz são guardadas numa hash_table onde é utilizado o endereço de hotel para encontrar o endereço de inserção e onde são inseridas numa árvore AVL (**Queries 3, 4 e 8**) são também todas guardadas numa hash_table pelo seu id de reserva (**Querie 1**).

Finalmente , caso uma reserva tenha sido registada como necessária para a Query 2, ou seja caso seja uma reserva realizada por um utilizador pedido por tal query, aumentamos o número de reservas correspondente ao utilizador.

O penúltimo parser específico é o parser dos voos, começamos por ,mais uma vez, registar o voo, no dia correspondente à sua data esperada de saída, numa árvore não binária de estrutura idêntica àquelas mencionadas previamente (**Query 10**). Agora guardamos numa hash_table correspondente ao id de voo todos os voos válidos(**Query 1**) e inserimos também todos os voos numa árvore binária não balanceada (**Query 10**), por último usufruímos de uma hash_table correspondente ao aeroporto de cada voo onde guardamos o número total de passageiros desse aeroporto (com a ajuda do array onde registramos os passageiros por voo previamente) bem como todos os voos partidos desse mesmo aeroporto numa árvore binária ordenada por data de partida prevista. (**Queries 5,6 e 7**)

No último parser específico olhamos para os passageiros , este é relativamente simples e resume se a uma inserção de todos os ids de voos associados a um certo utilizador numa árvore binária dentro da struct utilizador ,

neste processo são também ignorados os utilizadores não requisitados pelos inputs permitindo um processamento mais rápido. (**Query 2)**

Na fase final de execução o interpreter lê o ficheiro de inputs e chama a função choose_queries onde são selecionadas as funções correspondentes às queries pretendidas com tratamento do input esperado.

Para as queries 1 e 2 temos fácil acesso aos dados necessários através das hash_tables criadas previamente. Para as queries 3 ,4 , 5 ,8 e 9 basta usufruir das árvores de modo que o processo é simples e rápido, resumindo-se a funções de print no caso das queries 4,5,8 e 9 e a um cálculo de média simples na query 3.

Nas queries 6 e 7 recorremos também a árvores criadas previamente que facilitam o processamento mas envolvem maior complexidade de processamento principalmente na query 7 onde tivemos que criar várias funções auxiliares para realizar a organização de arrays e o cálculo das medianas. Finalmente a querie 10 que revelou ser a pior não só em termos de planeamento como de execução e eficiência , nesta query recorremos a processos lentos e dispendiosos de processamento explicados previamente neste relatório.

(Modo Interativo)

No modo interativo temos um processamento muito similar a este na medida em que apenas removemos o pré processamento pois é agora impossível prever que métricas serão necessárias ou não na execução das queries; qualquer query e input é possível então o tempo de processamento reflete essa quantidade extra de informação.

Nota- a única diferença entre o módulo de queries e o de queries para testes são os prints de performance e a comparação com ficheiros de output não existindo qualquer mudança substancial na lógica do código.

Testes de Desempenho

Especificações de hardware:

Computador 1:

Membro do Grupo : André
CPU: 11th Gen Intel(R) Core(™) i5-1135G7 (2,4GHz base/4.2GHz Boost)
GPU: TigerLake-LP GT2 [Iris Xe Graphics]
Memória RAM: 12Gb 3200 MHz

Computador 2:

Membro do Grupo : Marco
CPU: AMD Ryzen 7 4800H with Radeon Graphics (2,9GHz base/4.2GHz Boost)
GPU: AMD Renoir Radeon RX Vega 7 (Up to 1600 MHz)
Memória RAM: 16Gb 3200 MHz

Computador 3:

Membro do Grupo : Bruno
CPU: Intel(R) Core(™) i5-8300H CPU (2,3GHz base/4.0GHz Boost)
GPU: CoffeeLake-H GT2 [UHD Graphics 630]
Memória RAM: 12Gb 2666MHz

Teste 1				
Computador	1	2	3	Média
Parser Utilizadores 1	0,728115	0,798046	0,95743	0,827863667
Parser de Reservas	4,453573	4,812583	5,775531	5,013895667
Parser de Voos	0,185265	0,221833	0,254644	0,220580667
Parser de Utilizadores 2	4,293373	4,110509	5,263163	4,555681667
Parser de Passageiros	7,915993	9,117093	10,122498	9,051861333
Parser de Reservas 2	12,22145	12,22097	14,808224	13,083549
Parser de Voos 2	0,50379	0,496766	0,634633	0,545063
Parser de Passageiros 2	1,24992	1,341758	1,404945	1,332207667
Tempo Total	42,31261	43,40377	50,248655	45,32167933
Tempo Queries	9,006617	8,603651	9,032777	8,881015

Teste 2				
Computador	1	2	3	Média
Parser Utilizadores 1	0,72421	0,800132	1,036732	0,853691333
Parser de Reservas	4,376147	4,76747	6,301381	5,148332667
Parser de Voos	0,187139	0,220635	0,291893	0,233222333
Parser de Utilizadores 2	4,317616	4,106387	6,135134	4,853045667
Parser de Passageiros	8,197253	9,028322	11,443495	9,556356667
Parser de Reservas 2	12,32258	12,15496	16,651426	13,70965667
Parser de Voos 2	0,505727	0,498086	0,744747	0,582853333
Parser de Passageiros 2	1,253636	1,350661	1,532891	1,379062667
Tempo Total	42,65069	43,47036	57,007307	47,70945367
Tempo Queries	9,00451	8,631299	10,756624	9,464144333

Teste 3				
Computador	1	2	3	Média
Parser Utilizadores 1	0,728943	0,789054	0,964494	0,827497
Parser de Reservas	4,430963	4,82688	6,036771	5,098204667
Parser de Voos	0,187892	0,218843	0,260438	0,222391
Parser de Utilizadores 2	4,410154	4,099726	6,058714	4,856198
Parser de Passageiros	8,045183	9,084423	11,542248	9,557284667
Parser de Reservas 2	12,43963	12,09452	16,523212	13,68578667
Parser de Voos 2	0,518633	0,498848	0,727909	0,581796667
Parser de Passageiros 2	1,261495	1,33812	1,441258	1,346957667
Tempo Total	42,69411	43,22911	55,768276	47,230498
Tempo Queries	8,902699	8,599478	9,983955	9,162044

Os testes foram executados com todos os computadores laptop ligados à corrente em modo de performance do ubuntu com o large_dataset providenciado. (tempos em segundos)

O que mais sobressalta nos testes é a discrepância entre os computadores 1 e 2 em relação ao 3.Podemos notar que a maioria do tempo extra na execução está distribuído pelos parsers de dados maiores e não na execução das queries onde a diferença de performance não afeta tanto o tempo final. De facto , principalmente no parser de reservas 2 , essa diferença fica evidente.Isto deve-se ao facto de nesse mesmo parsing usufruirmos de árvores que, ainda que sejam AVL, são grandes e afetadas pela velocidade da memória e da CPU para realizar inserções rápidas. A diferença em velocidade da memória (3200 vs 2666 MHz) afeta todo o processo e resulta no tempo de execução que é cerca de 30% pior no computador 3 em relação ao computador 1 onde conseguimos melhor performance geral.

Entre os computadores 1 e 2 vemos existe também uma vantagem para o computador 1, desta vez não podendo esta ser atribuída à velocidade de memória que é igual entre as duas máquinas , sendo que a 2 possui até uma maior capacidade (16GB > 12GB), para além disso concluímos que a GPU não foi de grande importância na execução do programa pois a Radeon RX Vega 7 da APU do computador 2 supera consideravelmente os gráficos integrados da CPU 1 (TigerLake-LP GT2 [Iris Xe Graphics]).

É difícil dizer o porquê de uma CPU executar um programa mais rápido do que outra quando o ganho é tão ligeiro (cerca de 1,6%) , neste caso a CPU 2 tem o dobro dos núcleos de threads (8 núcleos e 16 threads) mas o boost máximo de clock é de 4,2 GHz para ambas as CPU e o nosso programa foi feito sem qualquer tipo de multithreading em mente e confirmamos que estas frequências são atingidas durante a execução, o que nos deixa com a possível conclusão de que as diferenças de arquitetura simplesmente favorecem mais o processador intel TigerLake ao invés do ryzen Zen2 .

Conclusão

Estamos novamente satisfeitos com o trabalho que realizamos e consideramos que os nossos esforços deram fruto a um programa relativamente bem estruturado e executado. O nosso código não está tão organizado e gerido como poderia mas achamos que aproveitamos bem o tempo para realizar todos os objetivos essenciais deste projeto.Tínhamos ainda várias ideias de possíveis implementações que não chegaram a acontecer como : gerir melhor as alocações do modo interativo, simplificar o parsing de voos para remover dependências, dividir as árvores AVL de reservas por ano/mês para diminuir tempos de inserção,

alterar a ordem de parsing de informação de forma a libertar memória alocada mais cedo e criar modos de estatísticas e de output específicos entre outras.