UNIVERSIDADE FEDERAL DE MINAS GERAIS

Estruturas de Dados

Trabalho Prático II: Otimização dos Servidores

Daniel Neiva da Silva

Belo Horizonte, 2021

1 - Introdução

O trabalho proposto se apresenta num contexto onde a humanidade avançou tecnologicamente de tal modo que é possível que uma pessoa faça o upload de usa mente para corpos sintéticos. Neste cenário, um dos desafios apresentados é a segurança da plataforma. Desta forma, os cientistas da Rellocator CO, empresa responsável pelo serviço, propuseram uma otimização dos servidores, fazendo com que o tempo de envio das consciências seja reduzido, fazendo com que estas fiquem menos tempo vulneráveis a qualquer ataque ou agente malicioso.

Com o propósito da otimização, deseja-se ordenar as consciências, possibilitando a implementação de um algoritmo mais vantajoso no upload das consciências já ordenadas.

Para avaliar a viabilidade de tal melhoria, alguns pontos devem ser observados. As consciências a serem ordenadas são compostas por 2 campos. O primeiro é uma string de texto que representa o nome da pessoa. O segundo é uma string de binários que representa a consciência da pessoa. Desta forma, necessitam ser avaliados 2 aspectos: o tempo de execução e a estabilidade de possíveis métodos. A primeira, visa avaliar o real impacto em termos de custo de execução do upload, e a segunda visa avaliar se os campos de fato são configurados da forma desejada.

Neste contexto, este trabalho prático visa avaliar quatro possíveis combinações de métodos de ordenação.

2 - Método

2 - Estrutura de dados.

O primeiro passo para a elaboração do projeto foi a escolha de uma estrutura de dados. A princípio, foi analisada a possibilidade de implementar uma lista encadeada para os testes e um struct com duas strings para as consciências. No entanto, a implementação de tal estrutura traria pouco benefício. Dado que o número de consciências é passado como argumento na hora da execução, era possível utilizar um array nativo do C++. Quanto à consciência, a utilização de um struct de duas strings como atributos foi utilizado, de modo a tornar o desenvolvimento mais organizado.

É importante ressaltar que, apesar de a utilização de uma lista ser vantajosa num ambiente de testes, no cenário real, o número de consciências na fila para serem transferidas muda em tempo de execução. Esta mudança poderia ser levantada como um desafio de implementação em um cenário real ao se escolher trabalhar com arrays estáticos,

mas uma solução para o problema não seria difícil. Uma solução possível, seria a implementação de uma função que quantifica o tamanho da fila de consciências e gerencia o algoritmo de ordenação. Outra solução possível seria a computação das consciências por pacotes. Sendo assim, ponderou-se que a escolha de um array estático não seria prejudicial para o experimento.

3 - Arquitetura

O projeto conta, ao todo, com seis estruturas. Uma delas, *Conscience*, é um struct com dois atributos (name_person, conscience_person) e uma função *swapCons,* utilizada por alguns dos algoritmos de ordenação para trocar duas consciências de posição entre elas.

Outra estrutura, *Utils*, é um namespace com três funções de uso geral que foram agrupadas neste namespace para melhor organização do código.

As outras quatro estruturas são dos algoritmos de ordenação: HeapSort, MergeSort, QuickSort, RadixSort. Além das operações intrínsecas destes algoritmos, para cada um foi implementada uma função *test[Name]*, que executa o algoritmo e retorna o tempo de execução deste.

Os algoritmos de ordenação foram baseados nas implementações mostradas em sala de aula. Foi usada também como fonte de consulta, a página da web GeeksForGeeks e o livro Algorithms in C, 1946, Robert Sedgewick.

Na função principal foi deixado apenas instruções e lógica para a leitura e interpretação dos argumentos passados em linha de comando ao rodar o programa.

4 - Análise de complexidade

A análise de complexidade será descrita nessa seção em duas partes. A primeira, irá apresentar os métodos de ordenação implementados e suas respectivas complexidades de tempo e espaço. A segunda, irá tratar da complexidade do programa como um todo, e possíveis melhorias que podem ser implementadas.

4.1 - HeapSort

O HeapSort é um algoritmo que, dado um vetor de tamanho n, compara o primeiro termo n-1 vezes, o segundo n-2 e assim por diante. Conforme apresentado em sala de aula, o HeapSort pode ser implementado com diferentes estruturas de dados. Neste trabalho, foi utilizada um Heap, haja vista que este tem uma complexidade menor que os outros algoritmos.

Em termos de tempo, o método que constrói o Heap, chamado neste projeto de buildHeap, tem uma complexidade de O(n) no pior caso. O método que faz a ordenação, tem uma complexidade de O(Logn). Desta forma, o HeapSort terá uma complexidade de tempo geral de O(nLog n).

Em termos de espaço, o HeapSort trabalha comparando os elementos e apenas trocando os elementos de posição. Desta forma, as estruturas de dados e/ou variáveis auxiliares são fixas, portanto a complexidade de espaço é O(1).

4.2 - MergeSort

O MergeSort é um algoritmo recursivo do tipo Divisão e Conquista. Ele funciona dividindo o input em duas metades, fazendo uma chamada recursiva a si mesma, e por fim, combina os sub-arrays de forma ordenada.

Em termos de tempo, o MergeSort é um algoritmo recursivo, onde dado um array de tamanho n, irá dividir o array ao meio, e fazer uma chamada recursiva de em cada metade, até que se tenha apenas um elemento, operação cuja função de recorrência é 2T(n/2). A outra operação será a combinação de todos os sub-arrays, com custo linear n. Desta forma, a função de complexidade geral é expressa pela função de recorrência T(n) = 2T(n/2) + n. Com o caso dois do Teorema Mestre, temos que a complexidade derivada desta função de recorrência é Θ(nLog n).

Em termos de espaço, o MergeSort pode ser feito de duas maneiras, uma tradicional, de complexidade O(n), e outra “in-place”, onde todas as operações são feitas dentro do mesmo array, tornando o custo O(1). Neste trabalho foi implementada a primeira opção, de complexidade O(n), por conta de sua estabilidade que será discutida posterioremente.

4.3 - QuickSort

O QuickSort também é do tipo Divisão e Conquista. Ele funciona de maneira similar ao MergeSort, sendo a diferença que a divisão é feita através de um pivô arbitrário, o qual tem todos os elementos maiores que ele à sua direita e os menores à esquerda ao fim de cada iteração.

A complexidade de tempo, no algoritmo implementado é de O(nLog n). O pivô escolhido para a implementação é fundamental para a definição da complexidade. Neste trabalho, foi escolhido o pivô como sendo o elemento que a posição que resulta da média da adição da primeira mais a ultima posição. Ou seja, o array sempre será dividido em duas

partes iguais, ou de tamanhos muito próximos (em casos de arrays com tamanho par, como os casos teste deste projeto).

A complexidade de espaço, no algorítimo implementado, é de O(Log n). Fazendo uso da recursividade de cauda, pode-se ordenar uma metade de cada vez, fazendo chamadas recursivas até ordenar a menor parte da esquerda, e só após isso, volta-se um nível acima e ordena-se a parte da esquerda. Desta forma, não é necessário memória adicional para a chamada recursiva da esquerda e da direita, uma vez que as chamadas da direita só irá ocorrer quando as da esquerda já tiverem terminado.

4.4 - Radix Exchange Sort

Diferente dos algoritmos anteriores, o Radix Exchange Sort (chamado daqui em diante de Radix), não é um algoritmo de comparação. Em vez de comparar os elementos um com os outros, o Radix compara os bits de cada elemento, do mais significante para o menos significante.

A complexidade de tempo do Radix depende de dois fatores: o número de elementos no array e o número de bits no elemento. Sendo n o número de elementos, e k o número de bits, o Radix irá fazer k comparações em n elementos. No caso deste projeto, o número k de bits é fixo (8), portanto a complexidade de tempo é O(8n) = O(n).

A complexidade de espaço, assim como o quicksort, é de O(Log n), pelo mesmo motivo.

5 - Configuração Experimental

O projeto foi desenvolvido na linguagem C++, e compilado pelo compilador G++ da GNU Compiler Collection, na versão 9.3.0 e standard C++11. Os testes foram realizados em computador pessoal, com as seguintes especificações:

- Sistema Operacional: Ubuntu 20.04.2 LTS x86_64
- CPU: Intel i5-7200U @3.100GHz

  Memória: 12GB DDR4

Para analisar a efetividade do sistema, foram realizados testes para verificar a eficiência dos algoritmos em termos de tempo e memória.

Usando a biblioteca “chrono” do C++, foi medido o tempo de execução de todo teste realizado. Para obter um resultado mais conciso, foram realizados 100 testes consecutivos de cada configuração, e destes retirada a media, com o propósito de simular um ambiente mais próximo do real, onde os servidores da RellocatorCO trabalhariam ininterruptamente. Todos os testes de tempo de execução foram feitos no sistema mencionado acima, com apenas aplicações necessárias para o funcionamento do sistema em execução. A execução consecutiva não foi feita internamente no programa, mas sim com a execução do seguinte comando no terminal bash:

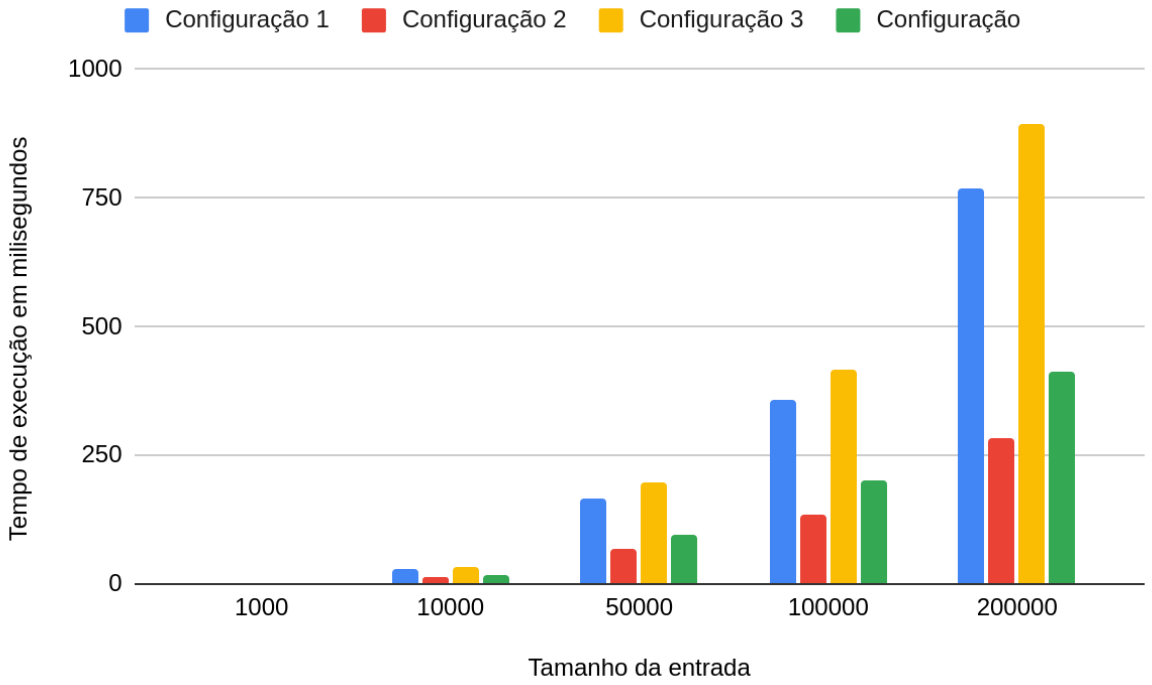
```
for i in {1..10}; do [arquivo_entrada] [tipo_teste] [num_consc]; done
```

Usando o utilitário Valgrind, foi feito um teste de consumo de memória, a fim de testar qual seria a necessidade de memória em uma aplicação real. Foi usada a ferramenta “massif” do utilitário, que cria um reporte sobre o uso de memória de uma aplicação.

Também foi implementado um método “int isStable” que retorna a quantidade de erros de ordenação no campo de consciências, após a ordenação por nomes.

6 - Resultados

Nesta seção, serão apresentados os resultados dos testes mencionados na seção anterior, bem como a análise relativa à estabilidade dos algorítmos. A apresentação será segmentada pelos tipos de teste requeridos na especificação do trabalho.



Na figura acima são mostrados os tempos de execução para cada configuração de algorítmos. É possível observar a grande diferença entre as combinações, sendo a combinação QuickSort + Radix a mais eficiente. Podemos notar também que, para quantidades pequenas de dados, os valores são inexpressivos.

Com relação à memória, as configurações 1 e 2 apresentaram valores parecidos, alocando certa de 12,878 MB no teste com 200.000 consciências. As configurações 3 e 4 também apresentaram valores parecidos entre si, alocando aproximadamente 25,681 MB cada, também no teste com 200.000 consciências.

Na avaliação da estabilidade, o algoritmo quicksort não é estável. De acordo com o método de implementação, o MergeSort se mostrou estável. É necessário ressaltar, que o método implementado para testar a estabilidade acusou um erro para o MergeSort, rodando na configuração 3 com 200.00 consciências. Não foi possível identificar nenhum erro de implementação do método de ordenação, e sendo este o único erro, considerou-se que é uma anomalia ou erro de implementação que passou despercebido.

7 - Conclusões

A avaliação final da implementação deste módulo nos sistemas da RellocatorCO deve levar em conta dois resultados principais:

A configuração que se mostrou mais eficiente foi a configuração 3. Esta, no entanto, apresenta uma anomalia que não foi possível identificar a causa. É preciso avaliar o quão dependente de uma ordenação o sistema da RellocatorCO será. Acredita-se ser mais vantajoso prosseguir com o terceiro método, caso seja avaliado que os resultados obtidos em termos de tempo de execução e gasto de memória sejam vantajosos. As fontes acadêmicas apontam que o MergeSort é estável, portanto é seguro dizer que com mais desenvolvimento, este bug será resolvido.

Referências:

Sedgewick, Robert, 1946 - Algorithms in C, United States of America: Addison-Wesley Publishing Company, Inc.

Wakely, Jonathan. Installing GCC. GCC Wiki, 2017. Disponível em https://gcc.gnu.org/wiki/InstallingGCC. Acesso em 10 de agosto de 2021.

QuckSort. Geeks for Geeks, 2021. Disponível em https://www.geeksforgeeks.org/quick-sort/. Acessado em 10 de agosto de 2021.

Apêndice I: Instruções de compilação

As instruções a seguir levam em consideração um ambiente Linux.

Para a compilação do programa, é necessário ter instalado o compilador G++. Este pode ser instalado pelo pacote “build-essentials”, disponível no gerenciador de pacotes da maioria das distribuições Linux, ou ser baixado diretamente do website da desenvolvedora.

Tendo o G++ instalado, basta digitar o comando “make” na pasta raiz do projeto para compilá-lo. Para executar o programa, basta executar o arquivo “run.out” gerado pelo computador. Este pode ser executado, também da pasta raiz do projeto, utilizando o comando “./bin/run.out [nome_arquivo] [tipo_teste] [qtd_consciencias]”, substituido “nome_arquivo” pelo nome do arquivo de homologação, “tipo_teste” pelo número de uma das 4 configurações do projeto e “qtd_consciencias” por um número entre 1 e 200.000.

Exemplo de compilação:

```
Arquivo  Editar  Ver  Pesquisar  Terminal  Ajuda
danielneiva@ubuntu:~/Projetos/tp2-ed$ make
g++ -std=c++11 -g -Wall -o ./bin/run.out  ./obj/Utils.o  ./obj/QuickSort.o  ./obj/Main.o
 ./obj/MergeSort.o  ./obj/HeapSort.o  ./obj/RadixSort.o
danielneiva@ubuntu:~/Projetos/tp2-ed$
```

Exemplo de execução:

```
Arquivo  Editar  Ver  Pesquisar  Terminal  Ajuda
danielneiva@ubuntu:~/Projetos/tp2-ed$ ./bin/run.out homologacao.txt 1 200000
```

```
Arquivo  Editar  Ver  Pesquisar  Terminal  Ajuda
Zula 00101011 43
Zula 11111010 250
Zula 01101100 108
Zula 00101101 45
Zula 10010110 150
Zula 01011001 89
Zula 00010110 22
Zula 01100100 100
Zula 10010101 149
Zula 00101100 44
danielneiva@ubuntu:~/Projetos/tp2-ed$
```