Análise do impacto no tempo de compilação e execução causado pelo processo de otimização de um compilador em algorítimos de ordenação

Gabriel Eduardo Lima
IFC Campus Blumenal
Blumenau, Santa Catarina
limaedugabriel@gmail.com

Ricardo de la Rocha Ladeira
IFC Campus Blumenal
Blumenau, Santa Catarina
ricardo.ladeira@ifc.edu.br

ABSTRACT

O presente trabalho tem como objetivo apresentar os impactos obtidos no que refere-se ao tempo de execução e compilação, considerando diferentes níveis de otimização para algoritmos os algoritmos de ordenação *Bubble Sort* e *Merge Sort*. Para isso, foram realizados testes onde ambos os algoritmos desenvolvidos em C foram executados para tamanhos de entradas diferentes, considerando os casos de testes com o código compilado usando as principais opções de otimização de tempo ofertadas pelo compilador GCC. Como principais resultados pode-se observar um ganho relativo de performance para ambos algoritmos, acompanhados por um aumento no tempo de compilação. Além disso, foi possível notar que o algoritmo *Bubble Sort*, apesar de apresentar redução de tempo percentual maior em comparação ao *Merge Sort*, ele não pode usufruir de processos de otimização mais complexos, chegando em uma estagnação de otimização. Por fim, o presente trabalho possui diversos tópicos que podem ser evoluídos e tratados no que refere-se a análise do impacto da fase de otimização na execução de algoritmos de ordenação.

KEYWORDS

Compilador, Otimização, Tempo de Execução, *Bubble Sort*, *Merge Sort*

1 INTRODUÇÃO

Inegavelmente os computadores tornarem-se parte fundamental da sociedade moderna nos últimos anos. Diversas áreas beneficiam-se da tecnologia para a produção de resultados de forma prática e concisos – como exemplo, pode-se citar desde aplicações científicas e aplicações empresarias, até mesmo para a programação de sistemas [1]. Esse grande sucesso dos computadores dá-se muito pela capacidade de serem programados para realizarem diferentes atividades.

Através da especificação de algoritmos – sequências de passos bem definidos para realização de uma tarefa – é que pode-se explorar diversos problemas de áreas distintas. Todavia, deve-se lembrar que os dispositivos computacionais eletrônicos comuns são estruturados sobre uma linguagem binária – que representa o estado da energia em seus circuitos. Visando abstrair a complexidade arquitetural dos computadores, linguagens de programação foram criadas como ferramentas para uma comunicação simplificada com a máquina.

O tópico de linguagens de programação não é foco do trabalho atual, porém um dos pontos fundamentais relacionados ao tema é de interesse – o método de tradução. Programas escritos em qualquer linguagem necessitam ser traduzidos para sequências de comandos que a máquina compreenda. De maneira geral, pode-se dividir as linguagens nesse quesito em Interpretadas – que são traduzidas e executadas instrução por instrução por um interpretador - e Compiladas – que são traduzidas por um compilador uma única vez em conjunto podendo ser executada n vezes [1].

Um dos aspectos fundamentais do trabalho em questão encontre-se nos Compiladores. Esse tipo de ferramenta possui um fluxo de operação que pode ser simplificado por: Código Fonte → Compilador → Código Objeto. O processo de tradução de um compilador pode ser dividido em diferentes etapas, onde de forma geral e resumida têm-se: (I) Análise Léxica – fase inicial em que o código fonte é analisado para a produção de *tokens*; (II) Análise Sintática – verifica se os *tokens* gerados atendem a estrutura gramatical da linguagem; (III) Análise Semântica – verifica a consistência semântica do código, isto é, se ele foi estruturado conforme a especificação da linguagem; e o tópico central do trabalho (IV) Otimização – etapa que visa tornar o código mais eficiente [2].

A otimização é um aspecto importante da geração de códigos, uma vez que os programas escritos e passados como entrada para um compilador podem ter aspectos aprimorados [1] para tornar a execução das instruções mais eficiente. Observado isso, destaca-se que é possível separar a otimização de

[1]O aspecto a ser otimizado depende do objetivo e do contexto da aplicação do programa. Pode-se, por exemplo, buscar por tempos de execução melhores, assim como também é possível priorizar a otimização do uso da memória.

código em duas fases distintas. A primeira refere-se a otimização do código intermediário, onde tem-se uma otimização independente da arquitetura da máquina. Já a segunda é a otimização de código objeto, em faz-se uso do conhecimento da arquitetura para usufruir da otimização- por exemplo, pode-se escolher instruções específicas da máquina que são mais rápidas [5].

Tendo sindo apresentado uma ideia básica de um compilador e da etapa de otimização, é possível apresentar o tema do artigo. O trabalho em questão tem como objetivo principal observar, avaliar e comparar o impacto da etapa de otimização no que diz respeito ao tempo de compilação e execução, aplicado a um conjunto restrito de dois algoritmos de ordenação – *Bubble sort* e *Merge sort*. Espera-se que com o presente trabalho seja possível definir a existência clara ou parcial de vantagens quanto ao uso de otimizações nos algoritmos citados anteriormente.

A título de especificação, optou-se por estudar o impacto da otimização em algoritmos de ordenação, pois além do tema ser um problema clássico e considerado por muitos como o mais fundamental no estudo de algoritmos [3], ele é abordado em praticamente todos os cursos de computação, e considera-se fácil de ser estudado e replicado. Como um ponto extra, elenca-se também a existência de inúmeros algoritmos relacionados ao tema, sendo que cada um possui uma análise de complexidade definida em relação ao número de entradas. Dessa forma, é possível não somente estudar o impacto da otimização em uma classe de algoritmos, mas também para algoritmos de um mesmo tema com tempo de execução naturalmente diferentes.

Compreendido os aspectos teóricos e as motivações que fundamentam o trabalho, pode-se dar sequência para a apresentação do experimento realizado e dos resultados obtidos.

2 MATERIAIS E MÉTODOS

Como uma primeira visão, o experimento realizado pode ser resumido como a execução repetida dos algoritmos de ordenação citados na seção anterior para diferentes entradas pré-definidas. Os algoritmos foram compilados e testados com diferentes opções de otimização para medir e avaliar a diferença de tempo de execução entre os casos de testes.

A fim de tornar a apresentação dessa seção mais prática, optou-se por dividir ela em subseções como se segue: A seção 2.1 apresenta os aspectos gerais dos algoritmos implementados e das entradas utilizadas para execução da ordenação; A seção 2.2 detalha aspectos do compilador utilizado e das opções de otimização consideradas para o trabalho; Por fim, a seção 2.3 comenta sobre o ambiente de execução dos testes e medições.

Antes de prosseguir para as subseções, faz-se importante especificar que todos os códigos e demais artefatos utilizados para a realização dos testes estão disponíveis em repositório público do GitHub [2]. Dessa forma, caso exista interesse do leitor, ele pode estar conferindo exatamente os elementos que produziram os resultados discutidos na seção posterior do artigo.

2.1 Algoritmos e Entradas

Para o trabalho em questão, conforme já fora mencionando anteriormente, foram implementados dois algoritmos de ordenação, sendo eles o *Bubble sort* e o *Merge sort*. A escolha dos algoritmos em questão dá-se pelo fato deles terem ordem de complexidade nos piores casos (mas não somente) diferentes – $O(n^2)$ para o primeiro e $O(n \log n)$ para o segundo. No caso em questão, ambos os algoritmos foram implementados usando a linguagem C, percebido que o objetivo é compilar e realizar procedimentos de otimização ofertados pelo compilador.

Além dos próprios códigos dos algoritmos, optou-se por implementar um programa principal para cada. O programa principal é responsável por incluir o algoritmo de ordenação e receber via linha de comando dois parâmetros – o tamanho de um *array* de inteiros, e o endereço para um arquivo texto contendo os elementos (um por linha) do *array* que deve ser ordenado. Utilizando-se dessa abordagem foi possível criar um conjunto de arquivos representando *array* de tamanho e conteúdo diferentes, sendo assim foi possível utilizar o mesmo conjunto de entrada para ambos algoritmos e para os diferentes processos de otimização testados.

Por fim, outra informação relevante sobre as entradas encontre-se na quantidade de elementos dos *arrays* utilizados. Visando um teste mais elaborado, optou-se por criar 10 *arrays* de cada um dos seguintes tamanhos: 1000, 2500, 5000, 10000, 25000, 50000, 100000, 250000, 500000, e 1000000. Sendo assim, foram mensurados os tempos de execução de um conjunto de 10 entradas, 10 vezes (cada vez com um conjunto de números para ordenação diferente para o mesmo código objeto).

2.2 Compilador e Otimizações Consideradas

No que diz respeito ao compilador utilizado, considerando que os algoritmos de ordenação foram implementados em C e que os testes foram executados em um ambiente Linux (conforme especificado na próxima subseção), optou-se por utilizar o compilador GCC – versão 9.4.0. A ferramenta em questão possui diversas *flags* de otimização, cada uma focando em um aspecto do código. Além disso, o compilador disponibiliza um parâmetro (-O) para selecionar o nível de otimização a ser realizada. Baseado em [4], em sequência estão destacados os níveis de otimização utilizados para os testes, bem como uma breve descrição dos processos (e *flags*) realizados em cada nível.

[2] https://github.com/Lima001/BCC-Artigo-Compiladores

A opção **-O0** é a configurada como padrão. Nela pequenas otimizações são feitas para reduzir o tempo de execução e permitir que o processo de *debug* gere os resultados adequados. Já para a opção **-O1**, o compilador tenta reduzir o tamanho do código, bem como o tempo de execução. Nesse nível de otimização opta-se por processos que não impliquem em aumento significativo no tempo de compilação. A título de especificação, algumas das *flags* usadas nessa fase são: *-fdce* – remove código que não produz efeito no programa; *-fipa-pure-const* – descobre quais funções são constantes;

A opção **-O2** encontra-se como um nível maior de otimização. Nesse nível, praticamente todas os processos que não envolvam ceder espaço e velocidade de compilação são realizados. Além das *flags* do nível anterior, o nível atual considera também, por exemplo, *-fno-peephole2* – desativa otimizações *peephole* específicas para a máquina; e *-ffinite-loops* – permite a eliminação de *loops* que não produzem efeitos colaterais. Observado disso, a opção **-O3** é o último nível de otimização, onde todos os processos que não envolvam condições restritivas são efetuados. Além das otimizações de presentes em -O2, o nível atual considera *flags* como *–floop-interchange* - melhora o desempenho de cache para *loops* aninhados, e permiti que outras otimizações de *loop*, como vetorização, ocorram.

Por fim, tem-se a opção **-Ofast**, que ignora padrões de computação estabelecidos – como a IEEE 754, que refere-se a aritmética de ponto flutuante – para obter o maior nível de otimização possível. Além das *flags* presentes em -O3, a opção atual considera, como exemplo, *-ffast-math* que implica em eventuais quebras dos padrões IEEE para funções matemáticas. Deve-se estar atento a esse tipo de otimização, pois dependendo do contexto do programa, ela pode acabar calculando resultados incorretos.

Apesar de não terem sido abordadas no trabalho, existem outras opções de otimizações que podem ser efetuadas pelo GCC. Como exemplo cita-se: -Os – realiza otimizações relacionadas ao tamanho do código, dando certa preferência inclusive sobre otimizações de velocidade; **-Og** – realiza otimizações para melhorar o processo de *debug* do código; e **-Oz** – similar a otimização **-Os**, porém é capaz de focar agressivamente em reduzir o tamanho de código sem se preocupar com aspectos de velocidade.

2.3 Ambiente de Execução e Medição

Visando especificar o ambiente em que os resultados foram obtidos, a subseção em questão apresenta detalhes da arquitetura e da configuração da máquina utilizada para executar os testes e realizar as medições. Logo em sequência é possível observar os detalhes julgados mais importantes. A repetição da mesma pesquisa em um ambiente diferente do apresentado, produzirá possivelmente uma perspectiva de resultados numéricos diferentes, porém de mesma tendência. Além disso, considerando que certas otimizações são dependentes da arquitetura da máquina que está compilando o código, conhecer detalhes do ambiente faz-se importante.

- Processador - Intel Core i3-4005U; Frequência: 1.70GHz; Cores: 2 *Cores* e 4 *Threads*; Arquitetura: 64 bits; Cache de L1d: 64 KB; Cache de L1i: 64 KB; Cache de L2: 512 KB; Cache de L3: 3 MB;
- Barramento - Frequência: 99,76 MHz;
- Memória RAM - Modelo: HMT451S6AFR8A-PB – SODIMM; Tecnologia: DDR3; Capacidade: 4GB; Frequência: 1600 MHz;
- Armazenamento - HD SAMSUNG HM160HI

Ainda no que diz respeito ao ambiente utilizado no presente trabalho, cita-se que *scripts shell* e *bash* foram utilizados para automatizar etapas. Por serem *scripts* simples pode-se julgar que sua interferência foi mínima. Considerado isso, aproveita-se para elicitar que a ferramenta utilizada para medir os tempos de compilação e execução dos testes foi o comando *time (bash)* com os parâmetros *default*. Os tempos obtidos foram filtrados para que fosse somente considerado o tempo total do teste. Além disso, cita-se que valores altos (maiores que 1 minuto) foram arredondados para um melhor visualização.

Por fim, destaca-se que os testes foram realizados e os resultados mensurados no ambiente operacional Linux. O sistema em questão foi configurado com a distribuição Ubuntu em sua versão 20.04 LTS. Visando reduzir as interferências de processos que não estavam correlacionado com os testes, optou-se por realizar toda a execução no modo Console.

3 RESULTADOS E DISCUSSÃO

Como resultados da execução dos testes especificados anteriormente, obteve-se: o tempo de compilação para os dois algoritmos de ordenação para os diferentes níveis de otimização; e o tempo de execução para os dois algoritmos considerando as diferentes entradas para cada nível de otimização usado. Visando tornar a apresentação direta e simples, a seção atual foi dividade em subseções para analisar resultados específicos. As subseções são como segue: A seção 3.1 apresenta os tempo de compilação; A seção 3.2 apresenta os tempos de execução do algoritmo *merge sort* e a seção 3.3 para o *bubble sort*.

3.1 Tempos de Compilação x Níveis de Otimização

A subseção atual apresenta os tempos de compilação observados para ambos os algoritmos abordados no trabalho considerando os diferentes níveis de otimização abordados. Os tempos foram medidos 10 vezes para que assim fosse possível calcular uma média simples para a comparação de resultados. Conforme a Figura 1, é possível observar os tempos para o algoritmo *merge sort*.

Merge O0	Merge O1	Merge O2	Merge O3	Merge Ofast
0,080s	0,110s	0,125s	0,603s	0,595s
0,078s	0,104s	0,121s	0,548s	0,572s
0,082s	0,104s	0,117s	0,534s	0,518s
0,096s	0,106s	0,151s	0,518s	0,561s
0,104s	0,105s	0,117s	0,560s	0,541s
0,090s	0,101s	0,119s	0,539s	0,604s
0,082s	0,106s	0,117s	0,608s	0,545s
0,080s	0,110s	0,118s	0,533s	0,523s
0,081s	0,105s	0,118s	0,518s	0,538s
0,081s	0,106s	0,117s	0,519s	0,520s
Média	**Média**	**Média**	**Média**	**Média**
0,085s	0,106s	0,122s	0,548s	0,552s

Figura 1: Tempo compilação *Merge Sort* x Níveis de Otimização

Observando as médias de de tempo, é possível notar que para quanto mais otimizações foram realizadas (considerando os níveis), maior foi o tempo para o programa ser compilado – é possível notar que a média para um código com otimização padrão é cerca de 6,5 menor do que para um código com o maior nível de otimização explorado. Todavia, apesar da diferença observada ser relativamente grande, os tempos obtidos foram bons – considerando que a compilação com a otimização mais rigorosa levou cerca de meio segundo para ser efetuada.

Já na Figura 2 é possível perceber os resultados para o algoritmo *bubble sort*. Note que diferentemente do que ocorreu com o algoritmo anterior, a divergência do tempo para a compilação com a opção de otimização padrão para com a otimização mais rigorosa, não foi relativamente grande – não chega nem a ser duas vezes maior. Porém, assim como no caso anterior, ainda assim é possível perceber um acréscimo de tempo ao aplicar-se níveis maiores de otimização

Merge O0	Merge O1	Merge O2	Merge O3	Merge Ofast
0,080s	0,110s	0,125s	0,603s	0,595s
0,078s	0,104s	0,121s	0,548s	0,572s
0,082s	0,104s	0,117s	0,534s	0,518s
0,096s	0,106s	0,151s	0,518s	0,561s
0,104s	0,105s	0,117s	0,560s	0,541s
0,090s	0,101s	0,119s	0,539s	0,604s
0,082s	0,106s	0,117s	0,608s	0,545s
0,080s	0,110s	0,118s	0,533s	0,523s
0,081s	0,105s	0,118s	0,518s	0,538s
0,081s	0,106s	0,117s	0,519s	0,520s
Média	**Média**	**Média**	**Média**	**Média**
0,085s	0,106s	0,122s	0,548s	0,552s

Figura 2: Tempo compilação *Bubble Sort* x Níveis de Otimização

Com os resultados em questão, é possível inicialmente apontar alguns pontos como: Quanto mais otimizações são realizadas, maior é o tempo de compilação (em conformidade com expresso por [2]); Pelo fato do algoritmo *bubble sort* ser mais simples que o *merge sort* não é possível realizar diversas otimizações – o que ajuda a compreender o porque do algoritmo em questão não ter sofrido grandes incrementos de tempo de compilação com opções avançadas de otimização.

3.2 Tempo de Execução *Merge Sort* x Níveis de Otimização

A subseção atual apresenta os resultados obtidos no que tange ao tempo de execução do algoritmo *merge sort* considerando a ordenação de *arrays* de diferentes tamanhos com diferentes níveis de otimização. As figuras 3, 4, 5, 6 e 7 apresentam os resultados para a execução considerando as otimizações -O0, -O1, -O2, -O3 e -Ofast.

Merge O0									
1.000	**2.500**	**5.000**	**10.000**	**25.000**	**50.000**	**100.000**	**250.000**	**500.000**	**1.000.000**
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,006s	0m 0,013s	0m 0,027s	0m 0,066s	0m 0,152s	0m 0,311s	0m 0,609s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,006s	0m 0,013s	0m 0,027s	0m 0,056s	0m 0,219s	0m 0,300s	0m 0,652s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,006s	0m 0,014s	0m 0,041s	0m 0,056s	0m 0,172s	0m 0,297s	0m 0,608s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,006s	0m 0,016s	0m 0,040s	0m 0,056s	0m 0,172s	0m 0,302s	0m 0,608s
0m 0,002s	0m 0,002s	0m 0,003s	0m 0,007s	0m 0,014s	0m 0,027s	0m 0,056s	0m 0,143s	0m 0,295s	0m 0,608s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,011s	0m 0,014s	0m 0,029s	0m 0,055s	0m 0,142s	0m 0,294s	0m 0,607s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,010s	0m 0,014s	0m 0,029s	0m 0,055s	0m 0,163s	0m 0,294s	0m 0,607s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,010s	0m 0,013s	0m 0,027s	0m 0,055s	0m 0,142s	0m 0,294s	0m 0,606s
0m 0,001s	0m 0,002s	0m 0,004s	0m 0,010s	0m 0,014s	0m 0,027s	0m 0,055s	0m 0,148s	0m 0,294s	0m 0,610s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,010s	0m 0,013s	0m 0,027s	0m 0,055s	0m 0,145s	0m 0,294s	0m 0,607s
Média	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**
0.001s	0.002s	0.003s	0.008s	0.014s	0.030s	0.057s	0.160s	0.298s	0.612s

Figura 3: Tempo execução *Merge Sort* Otimização -O0

Merge O1									
1.000	**2.500**	**5.000**	**10.000**	**25.000**	**50.000**	**100.000**	**250.000**	**500.000**	**1.000.000**
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,005s	0m 0,011s	0m 0,020s	0m 0,040s	0m 0,114s	0m 0,223s	0m 0,459s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,005s	0m 0,010s	0m 0,020s	0m 0,040s	0m 0,125s	0m 0,224s	0m 0,433s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,005s	0m 0,010s	0m 0,020s	0m 0,046s	0m 0,104s	0m 0,213s	0m 0,455s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,005s	0m 0,010s	0m 0,020s	0m 0,052s	0m 0,105s	0m 0,211s	0m 0,430s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,005s	0m 0,010s	0m 0,020s	0m 0,040s	0m 0,104s	0m 0,210s	0m 0,434s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,005s	0m 0,010s	0m 0,020s	0m 0,041s	0m 0,107s	0m 0,210s	0m 0,443s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,005s	0m 0,010s	0m 0,020s	0m 0,040s	0m 0,104s	0m 0,210s	0m 0,431s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,005s	0m 0,010s	0m 0,020s	0m 0,041s	0m 0,103s	0m 0,210s	0m 0,432s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,005s	0m 0,010s	0m 0,020s	0m 0,044s	0m 0,102s	0m 0,211s	0m 0,435s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,005s	0m 0,010s	0m 0,020s	0m 0,042s	0m 0,103s	0m 0,210s	0m 0,433s
Média	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**
0,001s	0,002s	0,003s	0,005s	0,010s	0,020s	0,043s	0,107s	0,213s	0,439s

Figura 4: Tempo execução *Merge Sort* Otimização -O1

Merge O2									
1.000	**2.500**	**5.000**	**10.000**	**25.000**	**50.000**	**100.000**	**250.000**	**500.000**	**1.000.000**
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,004s	0m 0,010s	0m 0,018s	0m 0,039s	0m 0,103s	0m 0,193s	0m 0,453s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,005s	0m 0,010s	0m 0,018s	0m 0,037s	0m 0,094s	0m 0,194s	0m 0,394s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,005s	0m 0,010s	0m 0,018s	0m 0,037s	0m 0,117s	0m 0,192s	0m 0,413s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,005s	0m 0,010s	0m 0,019s	0m 0,037s	0m 0,093s	0m 0,194s	0m 0,395s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,005s	0m 0,009s	0m 0,024s	0m 0,037s	0m 0,096s	0m 0,194s	0m 0,398s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,005s	0m 0,009s	0m 0,018s	0m 0,037s	0m 0,094s	0m 0,197s	0m 0,391s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,004s	0m 0,010s	0m 0,020s	0m 0,037s	0m 0,093s	0m 0,196s	0m 0,390s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,004s	0m 0,009s	0m 0,020s	0m 0,038s	0m 0,093s	0m 0,191s	0m 0,391s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,004s	0m 0,009s	0m 0,019s	0m 0,037s	0m 0,095s	0m 0,191s	0m 0,390s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,004s	0m 0,013s	0m 0,018s	0m 0,038s	0m 0,093s	0m 0,198s	0m 0,391s
Média	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**	**Média**
0,001s	0,002s	0,003s	0,005s	0,010s	0,019s	0,037s	0,097s	0,194s	0,401s

Figura 5: Tempo execução *Merge Sort* Otimização -O2

Analisando e comparando as médias para um mesmo tamanho de entrada, porém considerando níveis de otimização diferentes, observa-se que houve um ganho relativo de desempenho. Apesar de para entradas pequenas (menores que 100000) a diferença ter sido pequena, ou até mesmo nula – no caso das entradas de tamanhos inciais -, é possível notar que para entradas maiores existe um bom ganho.

Merge O3									
1.000	2.500	5.000	10.000	25.000	50.000	100.000	250.000	500.000	1.000.000
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,004s	0m 0,009s	0m 0,018s	0m 0,036s	0m 0,102s	0m 0,220s	0m 0,446s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,004s	0m 0,011s	0m 0,018s	0m 0,040s	0m 0,152s	0m 0,221s	0m 0,383s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,004s	0m 0,009s	0m 0,018s	0m 0,041s	0m 0,105s	0m 0,192s	0m 0,389s
0m 0,002s	0m 0,002s	0m 0,003s	0m 0,004s	0m 0,009s	0m 0,018s	0m 0,040s	0m 0,092s	0m 0,189s	0m 0,384s
0m 0,002s	0m 0,002s	0m 0,003s	0m 0,004s	0m 0,010s	0m 0,018s	0m 0,065s	0m 0,093s	0m 0,187s	0m 0,386s
0m 0,002s	0m 0,002s	0m 0,003s	0m 0,004s	0m 0,009s	0m 0,019s	0m 0,044s	0m 0,096s	0m 0,186s	0m 0,383s
0m 0,002s	0m 0,002s	0m 0,003s	0m 0,006s	0m 0,009s	0m 0,018s	0m 0,038s	0m 0,091s	0m 0,207s	0m 0,383s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,004s	0m 0,014s	0m 0,018s	0m 0,036s	0m 0,092s	0m 0,187s	0m 0,382s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,004s	0m 0,009s	0m 0,019s	0m 0,036s	0m 0,093s	0m 0,187s	0m 0,383s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,004s	0m 0,009s	0m 0,018s	0m 0,036s	0m 0,094s	0m 0,186s	0m 0,385s
Média	Média	Média	Média	Média	Média	Média	Média	Média	Média
0,001s	0,002s	0,003s	0,004s	0,010s	0,018s	0,041s	0,101s	0,196s	0,390s

Figura 6: Tempo execução *Merge Sort* Otimização -O3

Merge Ofast									
1.000	2.500	5.000	10.000	25.000	50.000	100.000	250.000	500.000	1.000.000
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,004s	0m 0,010s	0m 0,018s	0m 0,037s	0m 0,107s	0m 0,188s	0m 0,387s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,004s	0m 0,015s	0m 0,018s	0m 0,036s	0m 0,142s	0m 0,225s	0m 0,384s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,004s	0m 0,009s	0m 0,018s	0m 0,037s	0m 0,095s	0m 0,189s	0m 0,386s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,004s	0m 0,010s	0m 0,022s	0m 0,050s	0m 0,095s	0m 0,187s	0m 0,384s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,004s	0m 0,011s	0m 0,019s	0m 0,036s	0m 0,092s	0m 0,187s	0m 0,388s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,004s	0m 0,011s	0m 0,020s	0m 0,036s	0m 0,092s	0m 0,189s	0m 0,387s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,006s	0m 0,011s	0m 0,020s	0m 0,036s	0m 0,091s	0m 0,188s	0m 0,391s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,005s	0m 0,009s	0m 0,019s	0m 0,039s	0m 0,092s	0m 0,188s	0m 0,384s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,004s	0m 0,009s	0m 0,030s	0m 0,038s	0m 0,094s	0m 0,189s	0m 0,393s
0m 0,001s	0m 0,002s	0m 0,003s	0m 0,005s	0m 0,009s	0m 0,019s	0m 0,037s	0m 0,092s	0m 0,188s	0m 0,392s
Média	Média	Média	Média	Média	Média	Média	Média	Média	Média
0,001s	0,002s	0,003s	0,004s	0,010s	0,020s	0,038s	0,099s	0,192s	0,388s

Figura 7: Tempo execução *Merge Sort* Otimização -Ofast

Acredita-se que tal fato ocorre, pois o algoritmo em questão já possui um bom tempo de resposta (complexidade $O(n \log n)$) e dessa forma para casos simples chega-se em uma "estagnação" da otimização de tempo. Porém, para entradas maiores – como 1000000 - os efeitos podem ser percebidos.

3.3 Tempo de Execução *Bubble Sort* x Níveis de Otimização

A subseção atual apresenta os resultados obtidos no que tange ao tempo de execução do algoritmo *bubble sort* considerando a ordenação de *arrays* de diferentes tamanhos com diferentes níveis de otimização. As figuras 8, 9, 10, 11 e 12 apresentam os resultados para a execução considerando as otimizações -O0, -O1, -O2, -O3 e -Ofast.

Bubble O0									
1.000	2.500	5.000	10.000	25.000	50.000	100.000	250.000	500.000	1.000.000
0m 0,006s	0m 0,035s	0m 0,164s	0m 0,706s	0m 4,336s	0m 17,929s	1m 12,068s	7m 30,548s	30m 0,436s	120m 13,258s
0m 0,007s	0m 0,035s	0m 0,163s	0m 0,655s	0m 4,406s	0m 17,791s	1m 11,879s	7m 30,499s	30m 1,093s	120m 14,062s
0m 0,007s	0m 0,034s	0m 0,140s	0m 0,632s	0m 4,297s	0m 17,757s	1m 12,045s	7m 30,619s	29m 59,675s	120m 14,128s
0m 0,007s	0m 0,036s	0m 0,148s	0m 0,636s	0m 4,374s	0m 17,795s	1m 12,121s	7m 30,047s	30m 0,273s	120m 11,450s
0m 0,007s	0m 0,034s	0m 0,178s	0m 0,636s	0m 4,341s	0m 17,888s	1m 11,920s	7m 30,113s	29m 59,965s	120m 14,684s
0m 0,007s	0m 0,034s	0m 0,144s	0m 0,632s	0m 4,353s	0m 17,864s	1m 11,944s	7m 30,266s	30m 0,802s	120m 15,368s
0m 0,007s	0m 0,034s	0m 0,141s	0m 0,633s	0m 4,385s	0m 17,892s	1m 12,051s	7m 30,352s	30m 3,153s	120m 15,171s
0m 0,007s	0m 0,034s	0m 0,140s	0m 0,633s	0m 4,343s	0m 17,798s	1m 11,854s	7m 30,736s	30m 2,174s	120m 17,421s
0m 0,007s	0m 0,036s	0m 0,140s	0m 0,631s	0m 4,319s	0m 17,772s	1m 11,893s	7m 31,738s	30m 1,636s	120m 20,785s
0m 0,007s	0m 0,035s	0m 0,141s	0m 0,633s	0m 4,361s	0m 17,882s	1m 12,049s	7m 30,667s	30m 0,188s	120m 16,465s
Média	Média	Média	Média	Média	Média	Média	Média	Média	Média
0m 0,007s	0m 0,038s	0m 0,150s	0m 0,643s	0m 4,352s	0m 17,837s	1m 12s	7 m 30s	30m 1,2s	120m 15s

Figura 8: Tempo execução *Bubble Sort* Otimização -O0

Analisando e comparando as médias para um mesmo tamanho de entrada, porém considerando níveis de otimização diferentes, observa-se que houve um ganho relativamente alto de desempenho. Analisando a média de tempo para a maior entrada considerando a otimização padrão, com a média de tempo para a maior entrada com nível de otimização -O3 é possível notar uma redução maior que 50% do tempo de execução.

Bubble O1									
1.000	2.500	5.000	10.000	25.000	50.000	100.000	250.000	500.000	1.000.000
0m 0,002s	0m 0,010s	0m 0,042s	0m 0,245s	0m 1,567s	0m 6,640s	0m 27,122s	2m 51,229s	11m 25,371s	45m 55,818s
0m 0,002s	0m 0,010s	0m 0,054s	0m 0,217s	0m 1,556s	0m 6,650s	0m 27,157s	2m 51,229s	11m 25,404s	45m 55,685s
0m 0,003s	0m 0,010s	0m 0,044s	0m 0,207s	0m 1,547s	0m 6,622s	0m 27,216s	2m 51,180s	11m 25,427s	45m 58,011s
0m 0,002s	0m 0,010s	0m 0,043s	0m 0,206s	0m 1,547s	0m 6,670s	0m 27,201s	2m 51,150s	11m 25,165s	45m 56,922s
0m 0,002s	0m 0,010s	0m 0,042s	0m 0,206s	0m 1,545s	0m 6,649s	0m 27,145s	2m 51,321s	11m 25,188s	45m 56,207s
0m 0,002s	0m 0,010s	0m 0,050s	0m 0,213s	0m 1,544s	0m 6,622s	0m 27,147s	2m 51,093s	11m 25,242s	45m 58,413s
0m 0,002s	0m 0,010s	0m 0,051s	0m 0,212s	0m 1,551s	0m 6,622s	0m 27,182s	2m 51,142s	11m 25,625s	45m 55,504s
0m 0,002s	0m 0,010s	0m 0,042s	0m 0,206s	0m 1,550s	0m 6,619s	0m 27,159s	2m 51,237s	11m 25,496s	45m 58,112s
0m 0,003s	0m 0,010s	0m 0,042s	0m 0,206s	0m 1,567s	0m 6,622s	0m 27,129s	2m 51,192s	11m 25,694s	45m 58,140s
0m 0,002s	0m 0,010s	0m 0,043s	0m 0,209s	0m 1,546s	0m 6,649s	0m 27,170s	2m 51,356s	11m 25,412s	45m 57,652s
Média	Média	Média	Média	Média	Média	Média	Média	Média	Média
0m 0,002s	0m 0,010s	0m 0,045s	0m 0,213s	0m 1,552s	0m 6,637s	0m 27,163s	2m 51s	11m 25,2s	45m 57s

Figura 9: Tempo execução *Bubble Sort* Otimização -O1

Bubble O2									
1.000	2.500	5.000	10.000	25.000	50.000	100.000	250.000	500.000	1.000.000
0m 0,002s	0m 0,010s	0m 0,044s	0m 0,263s	0m 1,635s	0m 6,768s	0m 27,466s	2m 52,683s	11m 31,531s	46m 21,101s
0m 0,002s	0m 0,010s	0m 0,046s	0m 0,221s	0m 1,617s	0m 6,761s	0m 27,408s	2m 52,556s	11m 31,584s	46m 25,912s
0m 0,002s	0m 0,010s	0m 0,046s	0m 0,223s	0m 1,603s	0m 6,754s	0m 27,545s	2m 52,755s	11m 31,357s	46m 19,578s
0m 0,002s	0m 0,010s	0m 0,045s	0m 0,221s	0m 1,601s	0m 6,784s	0m 27,541s	2m 52,782s	11m 31,307s	46m 17,975s
0m 0,002s	0m 0,010s	0m 0,051s	0m 0,220s	0m 1,601s	0m 6,741s	0m 27,418s	2m 52,659s	11m 31,094s	46m 21,540s
0m 0,002s	0m 0,010s	0m 0,056s	0m 0,220s	0m 1,604s	0m 6,776s	0m 27,456s	2m 52,790s	11m 31,529s	46m 19,682s
0m 0,002s	0m 0,010s	0m 0,045s	0m 0,221s	0m 1,601s	0m 6,766s	0m 27,410s	2m 52,777s	11m 31,701s	46m 21,685s
0m 0,002s	0m 0,010s	0m 0,046s	0m 0,223s	0m 1,602s	0m 6,752s	0m 27,528s	2m 52,727s	11m 31,339s	46m 19,625s
0m 0,002s	0m 0,010s	0m 0,046s	0m 0,220s	0m 1,602s	0m 6,762s	0m 27,448s	2m 52,629s	11m 31,372s	46m 20,573s
0m 0,002s	0m 0,010s	0m 0,048s	0m 0,221s	0m 1,605s	0m 6,758s	0m 27,520s	2m 52,888s	11m 31,420s	46m 17,599s
Média	Média	Média	Média	Média	Média	Média	Média	Média	Média
0m 0,002s	0m 0,010s	0m 0,047s	0m 0,225s	0m 1,607s	0m 6,762s	0m 27,474s	2m 52,8s	11m 31,2s	46m 20,4s

Figura 10: Tempo execução *Bubble Sort* Otimização -O2

Bubble O3									
1.000	2.500	5.000	10.000	25.000	50.000	100.000	250.000	500.000	1.000.000
0m 0,002s	0m 0,010s	0m 0,048s	0m 0,238s	0m 1,648s	0m 6,800s	0m 27,469s	2m 52,689s	11m 31,759s	46m 21,618s
0m 0,002s	0m 0,010s	0m 0,045s	0m 0,228s	0m 1,606s	0m 6,770s	0m 27,538s	2m 52,722s	11m 31,220s	46m 17,218s
0m 0,002s	0m 0,010s	0m 0,045s	0m 0,221s	0m 1,602s	0m 6,793s	0m 27,474s	2m 52,729s	11m 31,149s	46m 17,438s
0m 0,002s	0m 0,010s	0m 0,046s	0m 0,221s	0m 1,605s	0m 6,764s	0m 27,423s	2m 52,697s	11m 31,611s	46m 21,800s
0m 0,002s	0m 0,010s	0m 0,046s	0m 0,221s	0m 1,603s	0m 6,798s	0m 27,434s	2m 52,571s	11m 31,768s	46m 18,741s
0m 0,003s	0m 0,010s	0m 0,052s	0m 0,221s	0m 1,606s	0m 6,744s	0m 27,404s	2m 52,804s	11m 31,235s	46m 16,844s
0m 0,003s	0m 0,010s	0m 0,045s	0m 0,220s	0m 1,604s	0m 6,789s	0m 27,473s	2m 52,723s	11m 32,077s	46m 20,802s
0m 0,002s	0m 0,010s	0m 0,044s	0m 0,220s	0m 1,612s	0m 6,782s	0m 27,422s	2m 52,763s	11m 31,705s	46m 18,903s
0m 0,003s	0m 0,010s	0m 0,044s	0m 0,223s	0m 1,603s	0m 6,743s	0m 27,449s	2m 52,700s	11m 31,433s	46m 18,370s
0m 0,002s	0m 0,010s	0m 0,047s	0m 0,220s	0m 1,603s	0m 6,788s	0m 27,413s	2m 52,994s	11m 31,025s	46m 19,060s
Média	Média	Média	Média	Média	Média	Média	Média	Média	Média
0m 0,002s	0m 0,010s	0m 0,046s	0m 0,223s	0m 1,609s	0m 6,777s	0m 27,450s	2m 52,8s	11m 31,2s	46m 19,2s

Figura 11: Tempo execução *Bubble Sort* Otimização -O3

Bubble Ofast									
1.000	2.500	5.000	10.000	25.000	50.000	100.000	250.000	500.000	1.000.000
0m 0,014s	0m 0,015s	0m 0,051s	0m 0,251s	0m 1,662s	0m 6,911s	0m 27,813s	2m 54,209s	11m 48,190s	46m 43,959s
0m 0,003s	0m 0,024s	0m 0,062s	0m 0,236s	0m 1,643s	0m 6,860s	0m 27,798s	2m 54,292s	11m 37,429s	46m 44,476s
0m 0,011s	0m 0,017s	0m 0,057s	0m 0,241s	0m 1,659s	0m 6,903s	0m 27,794s	2m 54,118s	11m 37,397s	46m 43,522s
0m 0,004s	0m 0,014s	0m 0,051s	0m 0,240s	0m 1,651s	0m 6,887s	0m 27,807s	2m 54,044s	11m 37,217s	46m 44,786s
0m 0,003s	0m 0,012s	0m 0,054s	0m 0,237s	0m 1,651s	0m 6,911s	0m 27,827s	2m 54,150s	11m 36,573s	46m 42,914s
0m 0,003s	0m 0,012s	0m 0,056s	0m 0,246s	0m 1,668s	0m 6,895s	0m 27,764s	2m 54,171s	11m 37,539s	46m 43,731s
0m 0,003s	0m 0,012s	0m 0,064s	0m 0,233s	0m 1,658s	0m 6,903s	0m 27,792s	2m 54,140s	11m 37,344s	46m 43,872s
0m 0,003s	0m 0,012s	0m 0,053s	0m 0,240s	0m 1,657s	0m 6,897s	0m 27,798s	2m 54,234s	11m 37,789s	46m 42,930s
0m 0,003s	0m 0,012s	0m 0,053s	0m 0,232s	0m 1,651s	0m 6,878s	0m 27,785s	2m 54,322s	11m 37,947s	46m 43,742s
0m 0,003s	0m 0,013s	0m 0,051s	0m 0,234s	0m 1,647s	0m 6,868s	0m 27,817s	2m 54,156s	11m 37,041s	46m 43,911s
Média	Média	Média	Média	Média	Média	Média	Média	Média	Média
0m 0,005s	0m 0,014s	0m 0,055s	0m 0,239s	0m 1,655s	0m 6,891s	0m 27,800s	2m 54s	11m 38,4s	46m 43,8s

Figura 12: Tempo execução *Bubble Sort* Otimização -Ofast

Outro resultado interessante de observar é o fato dos tempos entre os níveis de otimização -O1, -O2 -O3 e -Ofast não serem relativamente distintos. Ao analisar principalmente as otimizações a partir do nível O1, percebe-se que as médias obtidas são muito similares (se não iguais em alguns casos). Esse acontecimento faz-se sentido se somado ao aspecto percebido na análise dos tempos de compilação – o algoritmo em questão, por ser simples, não é capaz de ser consideravelmente otimizado por níveis de otimização maiores.

Sendo assim, de forma similar ao que ocorre com o algoritmo *merge*, porém para todos os tamanhos de entradas, o algoritmo atual enfrenta um efeito de estagnação. Isso significa dizer que não existe grandes diferenças perceptíveis em utilizar um nível menor ou maior de otimização. Com o

primeiro nível -01 já é possível ter ganhos circunstanciais no que refere-se ao tempo de execução.

4 CONCLUSÃO

Após a realização dos experimentos foi possível notar de maneira geral um ganho de desempenho no que diz respeito a tempo de execução seguido por um incremento no tempo gasto para o processo de compilação. Além disso, outros pontos foram possíveis de serem observados com o artigo em questão. Primeiramente, mesmo não relacionado com as otimizações, faz-se relevante ressaltar que o algoritmo *merge sort* demonstrou ser consideravelmente mais rápido que o *bubble* – como esperado, visto a complexidade de tempo de execução ser diferente.

Além disso, outro aspecto interessante é que mesmo através de otimizações com ganhos consideráveis, o algoritmo *bubble sort* não é capaz de superar o outro algoritmo estudado. Em um contexto prático isso significa dizer que mesmo considerando o maior nível de otimização de tempo de execução para o compilador GCC, ainda é mais vantajoso aplicar o outro algoritmo.

Mais uma conclusão interessante de apontar é que mesmo após as otimizações de ambos os algoritmos, os seus comportamentos (no que diz respeito a complexidade) aparentam se manterem. Isso significa dizer que mesmo com as inúmeras modificações realizadas no código visando a sua otimização, a natureza do algoritmo é preservada.

Outro aspecto de destaque é observar que o algoritmo *bubble sort* não teve grandes variações ao avançar os níveis de otimização. Por ser um código simples, nem todas as otimizações previstas podem ser aplicadas e gerar algum impacto positivo. Já o contrário se aplica ao algoritmo de *merge sort*. Por ser mais complexo e abordar a ordenação de maneira diferente, o compilador foi capaz de aplicar mais otimizações ao longo dos diferentes níveis.

Finalizando as conclusões sobre os resultados observados, é possível afirmar que existe vantagem em realizar processos de otimização nos algoritmos de ordenação abordados. Observe que o tempo de compilação aumenta, porém o código é compilado uma única vez para ser executado diversas – usufruindo do tempo de execução sendo reduzido pelas otimizações. Sendo assim, em situações em que algoritmos melhores não estão disponíveis – seja por complexidade de implementação, ou até mesmo de espaço -, otimizar algoritmos de ordenação mais simples pode ajudar a reduzir os impactos negativos no tempo de execução.

Tendo sido apontadas as principais conclusões do trabalho, aproveita-se para apresentar possíveis melhoras e trabalhos futuros baseados no artigo em questão. Como ideias cita-se: A possibilidade de estudar outros algoritmos de ordenação; A possibilidade de focar em otimizações de espaço ao invés de tempo; Pode-se realizar uma pesquisa de um mesmo algoritmo, porém variando a arquitetura da máquina que o executa – para verificar os impactos das otimizações em código objeto; É possível realizar um estudo comparativo de diferentes compiladores e suas otimizações para verificar se algum apresenta ganhos mais significativos do que outros.

REFERÊNCIAS

[1] Sebesta, Robert. Concepts of Programming Languages. 9th ed., Addison-Wesley, 2010.
[2] Aho, Alfred V., et al. Compilers: Principles, Techniques, and Tools. 2nd ed., Pearson, 2006.
[3] Cormen, Thomas H., et al. Introduction to Algorithms. 3rd ed., MIT Press, 2009.
[4] "Optimize Options (Using the GNU Compiler Collection (GCC))." Gnu.org, https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. Accessed 3 July 2022.
[5] Ladeira, Ricardo. Notas de aula sobre Compiladores. 13 Jun. 2022.