Universidade de São Paulo
Instituto de Ciências Matemáticas e Computação
Desenvolvimento de Código Otimizado - SSC 0951

Relatório Trabalho 2

Ferramenta de *Profiling - gproof*

1. Lucas G. Meneses Número: 13671615
2. Henrique S. Marques Número: 11815722
3. Carlos F. C. Lemos Número: 12542630
4. Eduardo S. Rocha Número: 11218692

15 de outubro de 2023,
São Carlos, SP,
Brasil

Conteúdo

1 Introdução

A importância do *profiling* de códigos é inegável no cenário de desenvolvimento de software. Trata-se de uma prática crucial que permite aos programadores identificar gargalos de desempenho e aprimorar a eficiência de suas aplicações. Com a complexidade cada vez maior dos sistemas de *software* atuais, compreender detalhadamente o tempo de execução de diferentes funções e a alocação de recursos se tornou uma etapa essencial para garantir a entrega de produtos de alta qualidade e desempenho robusto.

Como exemplo, imagine um aplicativo de comércio eletrônico que está sofrendo com lentidão e baixo desempenho. Sem a prática do *profiling* de código, os desenvolvedores podem ficar às cegas na tentativa de resolver o problema. No entanto, ao realizar o *profiling*, eles podem identificar que uma função de pesquisa está consumindo uma quantidade desproporcional de recursos de CPU e que a alocação de memória está fora de controle durante as consultas. Com essa análise precisa, os desenvolvedores podem otimizar essa função crítica, reduzindo o tempo de resposta e garantindo que o aplicativo funcione de maneira eficiente, proporcionando uma melhor experiência ao cliente e evitando a perda de vendas devido à lentidão.

Nesse contexto, uma ferramenta conhecida no ambiente de desenvolvimento é o *gproof*, uma ferramenta de *profiling* de código aberto que oferece uma variedade de recursos poderosos para a análise de desempenho de *software*. Por meio do rastreamento do tempo de execução de funções, identificação de chamadas recursivas e análise abrangente de alocação de memória, o *gproof* permite que

os desenvolvedores compreendam a fundo o desempenho de seus códigos. Além disso, com a geração de relatórios detalhados e visualizações gráficas intuitivas, o *gproof* capacita os desenvolvedores a tomar decisões embasadas em dados concretos, priorizando esforços de otimização e melhorando o desempenho geral das aplicações.

2 Metodologia

Para a realização de todos os experimentos, foi utilizado um computador rodando o Sistema Operacional Ubuntu, cuja CPU é um Intel Core i5 6600k com frequência de *clock* de 3,5 GHz, 4 núcleos e 4 *threads*. Quanto à memória, o computador faz uso de 16 GB de RAM do tipo DDR4, cache L1 de dados e de instruções ambas 8-*way* com 4×32 kB de capacidade, L2 4-*way* 4×256 kB e L3 12-*way* de 6 MB.

Nesse sentido, o perfilador *gproof* foi utilizado 10 vezes para avaliação dos tempos de execução de 3 algoritmos de ordenação com comportamento assintótico $O(n^2)$:

- *Bubble Sort*
- *Selection Sort*
- *Insertion Sort*

Assim, em cada execução, o mesmo vetor aleatório com 100000 inteiros de 8 *bytes* foi ordenado por cada um dos 3 algoritmos citados acima, de modo que, entre uma ordenação e outra, a memória cache tenha sido limpa, evitando que haja interferências na quantidade de cache *misses*, tornando, pois, a comparação entre os algoritmos mais justa. Ademais, as médias aritméticas e intervalos de confiança (95 %) referentes às 10 execuções foram calculados, com intuito de melhor avaliar os resultados obtidos. Por fim, o código, os dados brutos e as figuras geradas estão disponíveis em repositório aberto via GitHub.

3 Resultados

Com objetivo de melhorar a visualização dos resultados obtidos com a utilização do perfilador *gproof*, *plotamos* os grafos das chamadas de funções. Conforme será

discutido posteriormente, os tempos de execução foram bastante consistentes ao longo do 10 execuções, apresentando poucas variações. Dessa forma, apesar de não representar com total fidelidade o resultado final do experimento, a Figura 1, referente ao grafo da primeira execução, indica o comportamento padrão dos algoritmos avaliados neste estudo.[1]

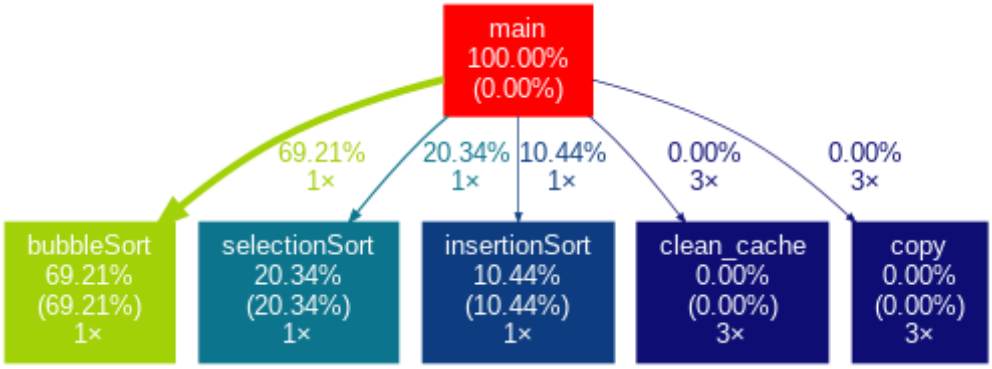

Figura 1: Grafo das chamadas de funções para o primeiro experimento realizado com os algoritmos *Bubble Sort*, *Selection Sort* e *Insertion Sort*. As funções auxiliares *clean_cache* e *copy* foram utilizadas para *reset* da memória cache e cópia dos dados entre vetores (algoritmos *in place*), respectivamente.

Como pode ser visto no grafo, o algoritmo *Bubble Sort* foi o mais custoso em termos de tempo de execução, consumindo, aproximadamente, 70 %, do tempo total gasto para finalizar o programa, seguido do *Selection Sort*, com 20 %, e do *Insertion Sort*, com 10 %.

Nesse sentido, o comportamento observado é totalmente condizente com o esperado ao se avaliar as características particulares de cada heurística: o *Bubble Sort* é um algoritmo de ordenação ineficiente, pois exige muitas trocas e comparações. Isso o torna consideravelmente mais lento do que o *Selection Sort* e o *Insertion Sort*, os quais, apesar de possuir o mesmo comportamento assintótico quadrático, envolvem menos operações de comparação e troca, na maioria dos cenários (*e.g* vetor aleatório). De maneira mais quantitativa, os resultados obtidos também puderam ser analisados através das médias aritméticas e dos intervalos de confiança

[1]Como já citado na Seção 2, todos os 10 grafos podem ser encontrados no repositório do GitHub, acessando $Atividade2 \rightarrow Resultados2 \rightarrow Grafos$.

(95 %). Os valores encontrados podem ser visualizados com a Tabela 1 e com a Figura 2.

Tabela 1: Médias aritméticas e intervalos de confiança (limites inferior e superior, 95 %) das 10 execuções de cada um dos três algoritmos quadráticos avaliados neste estudo.

Algoritmos	Média (s)	Limite Inferior (s)	Limite Superior (s)
Bubble Sort	38,559	37,803	39,315
Selection Sort	11,206	10,951	11,461
Insertion Sort	6,290	6,142	6,438

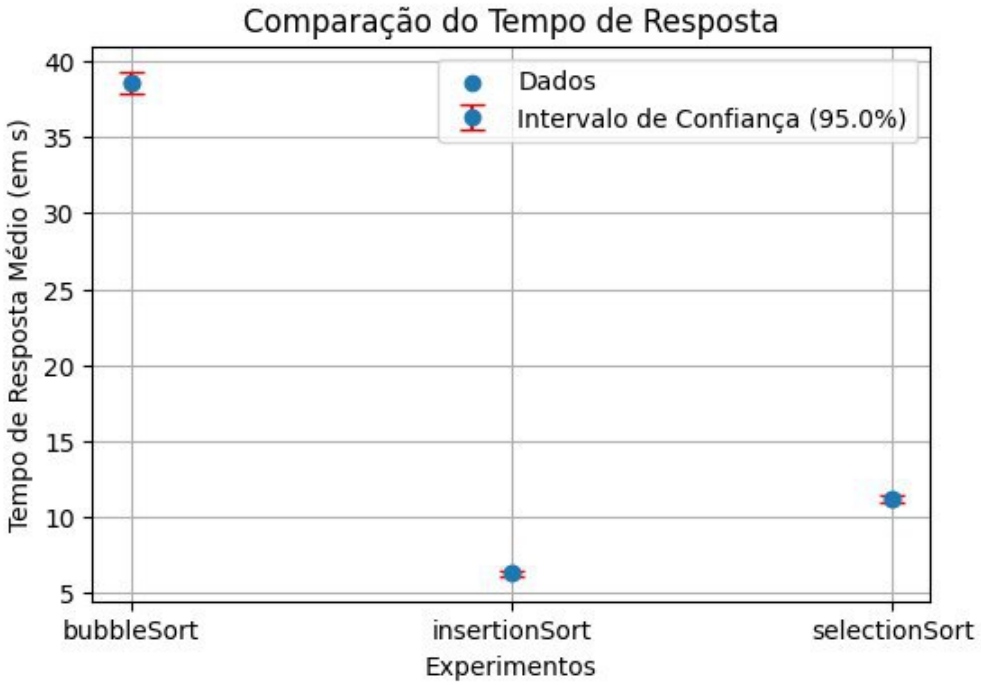

Figura 2: Tempos de resposta (s) em função do experimento realizado. Os círculos azuis indicam a média dos dados obtidos para cada experimento e as barras vermelhas mostram o intervalo de confiança de 95 %.

4 Conclusões

A importância do *profiling* de códigos é fundamental no desenvolvimento de software, permitindo aos programadores identificar gargalos de desempenho e aprimorar a eficiência das aplicações. Nesse contexto, a ferramenta de *profiling* de código aberto *gproof*, testada neste trabalho, se destaca, oferecendo recursos avançados para a análise de desempenho, incluindo o rastreamento de tempo de execução de funções, com relatórios e visualizações gráficas para embasar decisões de otimização.

Os resultados demonstram o uso efetivo do perfilador *gproof* para visualizar o comportamento de algoritmos de ordenação quadráticos, especificamente o *Bubble Sort*, *Selection Sort* e *Insertion Sort*. Notavelmente, o *Bubble Sort* consumiu cerca de 70 % do tempo total de execução, seguido pelo *Selection Sort* com 20 % e o *Insertion Sort* com 10 %. Esses resultados estão alinhados com as expectativas, visto que o *Bubble Sort* é conhecido por ser um algoritmo ineficiente devido às trocas e comparações excessivas, enquanto o *Selection Sort* e o *Insertion Sort* são mais eficientes em média. A análise quantitativa com médias e intervalos de confiança reforça as conclusões, destacando a disparidade nos tempos de execução entre os algoritmos.