Relatório da atividade 03 - Profiling com Gprof

Grupo:

Artur Kenzo - 15652663

Daniel Jorge - 15446861

Gabriel Phelippe - 15453730

Jhonatan Barboza - 15645049

---

1. Introdução

Este relatório detalha a análise de performance de três algoritmos de ordenação, com o objetivo principal de utilizar a ferramenta de profiling `gprof`, que possibilita a comparação do tempo de execução de diferentes funções em um código. Comparou-se algoritmos com a mesma complexidade assintótica e observou as diferenças práticas de performance.

2. Metodologia

A análise foi conduzida seguindo as diretrizes da atividade:

2.1. Código Desenvolvido

Foi implementado um programa em C (`ex2.c`) que contém os seguintes algoritmos de ordenação:

- **Bubble Sort:** Complexidade de tempo $O(n^2)$.
- **Selection Sort:** Complexidade de tempo $O(n^2)$.
- **Insertion Sort:** Complexidade de tempo $O(n^2)$.

Com complexidade de tempo próximas, o gprof pode capturar dados relevantes de todos os algoritmos.

2.2. Ambiente de Teste

O ambiente de teste foi configurado da seguinte forma:

- **Tamanho do Vetor:** Um vetor com `20000` números inteiros (`ARRAY_SIZE`) foi utilizado para os testes.
- **Número de Execuções:** O programa foi executado `10` vezes para garantir que os resultados representassem uma média de performance, minimizando variações pontuais.
- **Geração de Dados:** Um vetor original com números inteiros aleatórios foi gerado uma única vez pela função `generate_random_array`. Para cada uma das 10 execuções uma cópia exata deste vetor original era criada usando a função `copy_array`, garantindo que todos os testes partissem das mesmas condições iniciais.

2.3. Limpeza de Cache

Para evitar que dados em cache de uma execução influenciassem o desempenho da próxima, uma função `clean_cache` foi chamada antes de cada execução dos algoritmos de ordenação. Essa função itera sobre um vetor estático que corresponde ao maior nível de cache do processador, forçando o sistema a sobrescrever os dados antigos.

2.4. Processo de Profiling

O processo de profiling foi automatizado por um `Makefile` e seguiu estes passos essenciais:

1. **Compilação:** O código C foi compilado com a flag `-pg` para que o executável pudesse gerar dados de performance.
2. **Execução:** Ao rodar o programa, foi criado o arquivo `gmon.out`, que contém os dados brutos da análise.
3. **Análise:** O comando `gprof` foi usado para ler o `gmon.out` e gerar um relatório de texto com as métricas de tempo e chamadas de função.
4. **Visualização:** O relatório de texto foi convertido em um grafo de chamadas visual (`.png`) com o auxílio das ferramentas `gprof2dot.py` e `dot`.

3. Resultados e Análise

A análise dos dados coletados pelo `gprof` está dividida em duas partes: o perfil de execução (flat profile) e o grafo de chamadas (call graph).

3.1. Análise do Flat Profile e do Grafo de chamadas

O flat profile sumariza o tempo gasto em cada função do programa. A tabela abaixo extrai os dados mais relevantes do arquivo `saida.txt`.

Nome da Função	% do Tempo Total	Tempo Total (s)	Chamadas
`bubble_sort`	64.42%	1.06	1
`selection_sort`	17.79%	0.29	1
`insertion_sort`	13.50%	0.22	1
`clean_cache`	4.29%	0.07	3
`copy_array`	0.00%	0.00	3
`generate_random_array`	0.00%	0.00	1

Da análise da tabela, observamos que:

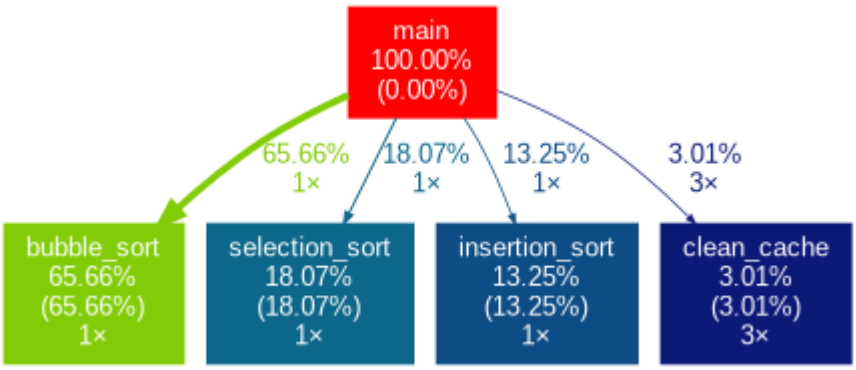
1. **`bubble_sort`**: Foi o algoritmo de ordenação mais lento, correspondendo a **64.42%** do tempo total. O que se justifica pela sua natureza ineficiente, que envolve muitas trocas de elementos.

2. **`selection_sort`** e **`insertion_sort`**: Apresentaram desempenho significativamente melhor que o Bubble Sort, com o `insertion_sort` sendo o mais rápido dos três, com **13,50%** do tempo total.

3. **`clean_cache`**: Embora não seja um algoritmo de ordenação, a função de limpeza de cache consumiu uma parte notável do tempo total (**4.29%**), o que indica que a estratégia de limpeza de cache teve um

impacto mensurável no desempenho geral.

O grafo de chamadas só confirma o vizualizado no `.txt`: o grafo demonstra a relação entre as funções e o tempo gasto em cada uma delas:



A função `main` é o ponto de partida, correspondendo a 100% do tempo de execução do programa e de seus filhos.As ramificações mais "quentes" (que consomem mais tempo) são as chamadas para `bubble_sort` (64.42%), destacadas pela cor verde e pelas linhas mais espessas no grafo. As chamadas para `selection_sort` (17.79%) e `insertion_sort` (13.50%) representam uma porção menor do tempo de execução.

Assim, com base nos dados empíricos, a ordem de eficiência dos algoritmos para o cenário testado é: 1º Insertion Sort (mais rápido) > 2º Selection Sort > 3º Bubble Sort (mais lento)

3.2. Comparação com a Análise Assintótica

Os três algoritmos testados (`Bubble Sort`, `Selection Sort` e `Insertion Sort`) possuem uma complexidade de tempo de pior caso e caso médio de $O(n^2)$. A análise assintótica nos diz que, para entradas grandes, o tempo de execução cresce de forma quadrática.

No entanto, a análise de profiling revelou que, embora pertençam à mesma classe de complexidade, seu desempenho no mundo real é diferente.

- O **Bubble Sort** é conhecido por ser ineficiente na prática devido ao grande número de operações de troca que realiza.
- O **Selection Sort** realiza menos trocas que o Bubble Sort, mas um número fixo de comparações.
- O **Insertion Sort** é geralmente o mais rápido dos três para vetores pequenos ou parcialmente ordenados, pois seu número de comparações e trocas pode ser significativamente menor no melhor caso.

Os resultados obtidos estão de acordo com o esperado pela teoria da computação: a análise assintótica descreve o crescimento do tempo de execução, mas não revela as constantes e os fatores de baixo nível que diferenciam algoritmos da mesma classe de complexidade.

4. Conclusão

Este trabalho demonstrou com sucesso o uso da ferramenta `gprof` para realizar o profiling de um código em C e analisar a performance de diferentes funções. A metodologia de limpar o cache e executar cada

algoritmo múltiplas vezes sobre o mesmo conjunto de dados permitiu uma comparação justa e precisa.

A análise revelou que, para um vetor de 20000 inteiros aleatórios, o `Insertion Sort` foi o algoritmo mais eficiente, seguido pelo `Selection Sort` e, por último, o `Bubble Sort`. Este resultado prático, embora não seja evidente apenas pela análise assintótica de $O(n^2)$, está alinhado com o comportamento conhecido desses algoritmos. A ferramenta `gprof` provou ser eficaz para identificar os gargalos de performance e fornecer insights quantitativos sobre o tempo de execução do código.