Universidade de São Paulo

Instituto de Ciências Matemáticas e de Computação

Desenvolvimento de Código Otimizado 2023.2

Atividade 2: Profiling usando gprof

Gabriel da Cunha Dertoni	11795717
Matheus Ventura de Sousa	11345541
Pedro Lucas de Moliner de Castro	11795784
Vitor Caetano Brustolin	11795589

Profa. Dr. Sarita Mazzini Bruschi

São Carlos, Outubro de 2023

1 Introdução

O objetivo da presente atividade é aprender e aplicar na prática a ferramenta *GNU gprof* para analisar o tempo de execução de cada uma das funções de um programa. Para ilustração, foi construído um código em C com três algoritmos de ordenação baseados em comparação: `heap_sort`, `quick_sort` e `merge_sort`, todos com complexidade assintótica média $O(n \log(n))$. Para solidificação dos resultados do experimento, o código foi executado 10 vezes para obter a média e a variância dos resultados coletados. Um nível de significância $\alpha = 0.05$ foi usado para determinar o intervalo de confiança 95%.

2 Ambiente de execução

O experimento foi compilado usando `gcc` e executado em uma máquina com as especificações descritas na tabela a seguir:

Nome do modelo	11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz
Arquitetura	x86_64
Modos operacionais da CPU	32-bit, 64-bit
Ordem dos bytes	Little Endian
CPUs	12
Threads por núcleo	2
Núcleos por soquete	6
Soquetes	1
Núcleos de NUMA	1
Frequência máxima	4500 MHz
Frequência mínima	800 MHz
Cache de L1d	288 KiB (6 instâncias)
Cache de L1i	192 KiB (6 instâncias)
Cache de L2	7.5 MiB (6 instâncias)
Cache de L3	12 MiB (1 instância)

3 Códigos e execução

O fonte principal da execução é o arquivo `src/main.c` e os cabeçalhos auxiliares estão dentro do diretório `include/`. Dentre os cabeçalhos, o principal é o `sort.h` que contém os algoritmos de ordenação. Vale ressaltar que o `quick_sort` foi reimplementado ao invés de utilizar o `qsort` da biblioteca padrão, pois esse último apresenta múltiplas otimizações e, portanto, não apresentaria claramente suas etapas no grafo final. Para verificar a corretude do programa, foram inseridos *asserts* que checam que vetor está ordenado após cada função, para remover essa parcela da análise para recompilar o código com o a flag `-DNDEBUG`. Há, também, um diretório `scripts/` com o arquivo `profile.py` responsável por gerenciar o processo de análise algorítmica feita.

Para compilação do programa, pode-se usar o comando `make` no terminal, em seguida, para executá-lo de forma independente, utiliza-se o comando `make run`. A diretiva `make requirements` deve ser usada para instalar as dependências dos *scripts*, `make profile` pode ser usada para executar o gprof no código fonte e, finalmente, para criar os grafos, aplica-se o comando `make graph`.

4 Análise de Desempenho

Após 10 repetições do código, obteve-se a média e variância dos tempos de execução resumidos na tabela a seguir:

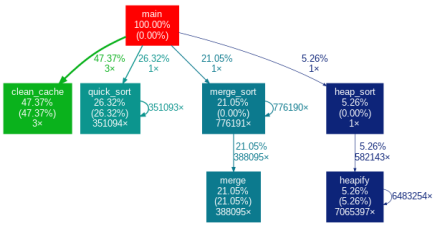
	% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call
clean_cache	38.95 ± 6.15	0.08 ± 0	0.08 ± 0	3	26.20 ± 2.73	26.20 ± 2.73
merge	17.70 ± 6.35	0.15 ± 0.02	0.04 ± 0.01	388095	0 ± 0	0 ± 0
swap_scalars	4.87 ± 3.38	0.20 ± 0.02	0.01 ± 0	8708661	0 ± 0	0 ± 0
heapify	13.37 ± 8	0.17 ± 0.04	0.03 ± 0.02	582143	0 ± 0	0 ± 0
copy_array	0 ± 0	0.20 ± 0.02	0 ± 0	776193	0 ± 0	0 ± 0
random_scalar	0.42 ± 1.32	0.20 ± 0.02	0 ± 0	388096	0 ± 0	0 ± 0
median_pivot	0.44 ± 1.38	0.20 ± 0.02	0 ± 0	351094	0 ± 0	0 ± 0
is_sorted	1.20 ± 1.90	0.20 ± 0.02	0 ± 0	3	0.84 ± 1.17	0.84 ± 1.17
heap_sort	3.09 ± 3.70	0.20 ± 0.02	0 ± 0.01	1	6.51 ± 8.20	41.94 ± 18.93
merge_sort	0.56 ± 1.76	0.20 ± 0.02	0 ± 0	1	1 ± 3.17	37.54 ± 12.76
quick_sort	19.54 ± 5.10	0.15 ± 0.03	0.04 ± 0.01	1	39.54 ± 9.57	42.66 ± 9.04
randomize_array	0 ± 0	0.20 ± 0.02	0 ± 0	1	1 ± 3.17	0 ± 0

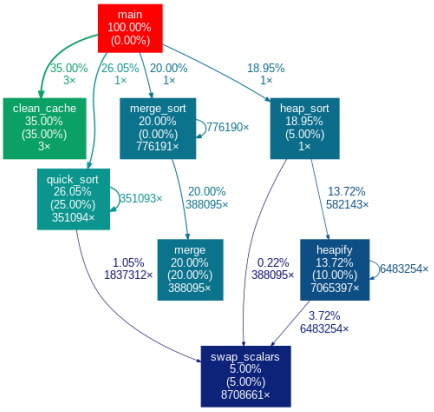
De acordo com a tabela acima, desconsiderando a limpeza da cache, as principais rotinas que usam a CPU de forma não cumulativa são `quick_sort`, `merge` e `heapify`. Ou seja, como esperado, a parcela mais quente do algoritmo de `merge_sort` é o `merge` dos *subarrays* ordenados, a etapa mais relevante do `heap_sort` é a construção da estrutura de `heap` dentro do *array* e a porção do `quick_sort` que mais consome CPU é fazer o particionamento depois de já ter definido um pivot, o que nessa implementação é feito diretamente na função chamada externamente.

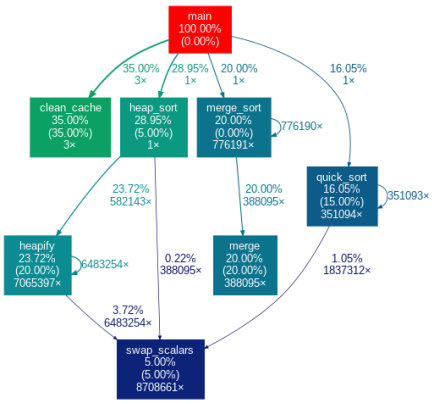
Fazendo uma análise do tempo acumulado por função, vê-se, claramente, como também já era esperado, que o `quick_sort` é o algoritmo mais rápido dentre os três, mesmo usando uma implementação sem tantas otimizações quanto o `qsort` da biblioteca padrão. Esse algoritmo foi assim nomeado justamente por obter tempos reduzidos na prática, mesmo com uma análise assintótica formal ruim no pior caso. O uso do pivot por mediana parece realmente ser uma boa escolha no caso geral, mas é possível que essa diferença de performance fosse ainda maior se a escolha do pivot fosse baseada em alguma informação sobre o vetor desordenado.

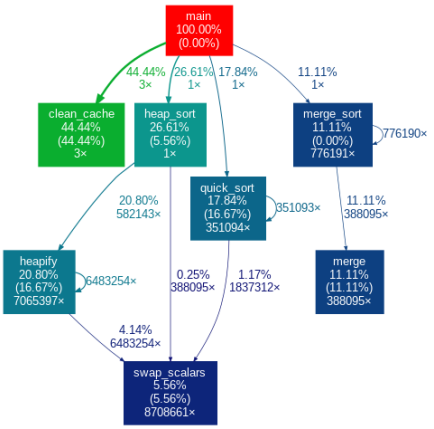
4.1 Representação em grafos

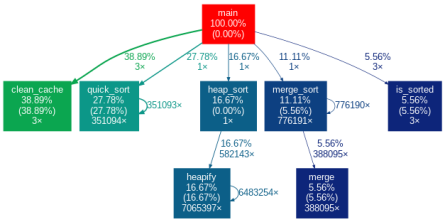
Para renderização dos grafos, foi utilizado um algoritmo na linguagem *Python* chamado gprof2dot criado pelo José Fonseca e o comando `dot` do programa graphviz. Com os resultados do `gprof` em mãos, foram gerados os grafos das 10 execuções do experimento que podem ser vistos abaixo:

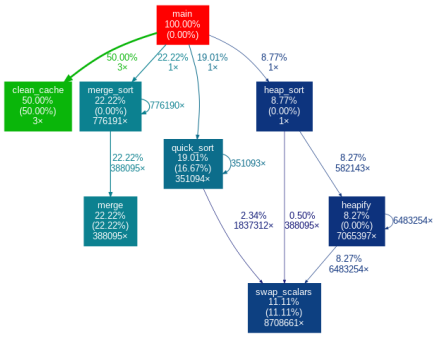


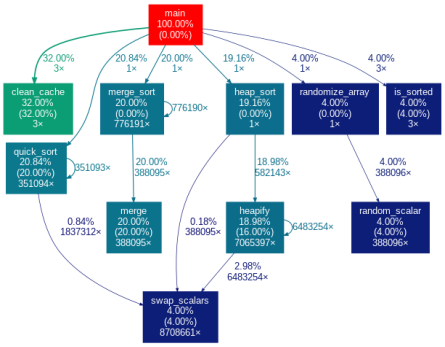


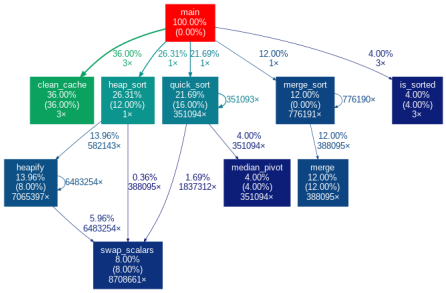


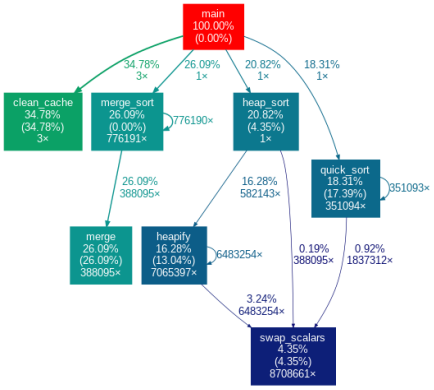


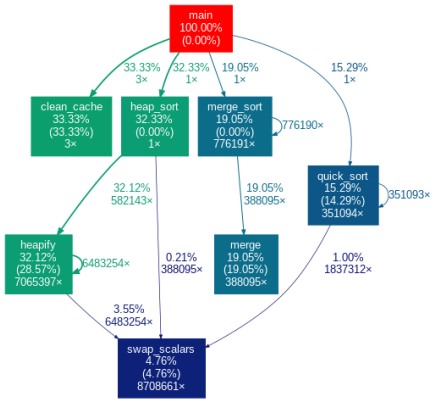












Nota-se que é bastante inconsistente se funções auxiliares como `swap_scalars` ou `median_pivot` tomarão tempo suficiente para aparecerem como funções independentes no grafo. Além disso, é possível ver uma disparidade grande nas porcentagens presentes em cada execução.

5 Conclusão

Conforme apresentado no experimento e seguindo as expectativas do grupo, o algoritmo `quick_sort` se mostrou o mais rápido entre as três formas de ordenação. A utilização da função `clean_cache` foi muito importante para não enviesar o experimento em favor das parcelas executadas posteriormente pois, uma vez que a cache estivesse populada com dados relevantes para o programa, a execução seria mais veloz. Finalmente, apesar de ser uma funcionalidade útil para visualização dos resultados, a geração de grafos com o `gprof` pode ser bastante instável e deve ser acompanhada de uma análise estatística mais rigorosa baseada em múltiplas execuções.