Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação
SSC0951 - Desenvolvimento de Código Otimizado

# Atividade 2

Ferramenta de profiling - *gprof*

Docente: Profa. Dra. Sarita Mazzini Bruschi

Danilo Alves - 10408390
Eduardo Amaral - 11735021
Rafael de Almeida - 11872028

Novembro
2022

# 1 Introdução

Ao analisar programas grandes, pode ser difícil saber em qual lugar do programa deve se concentrar os esforços de otimização. *Profilers* de código, ferramentas de análise que coletam dados sobre a performance de um programa enquanto ele executa, fornecem informações importantes para descobrir quais pontos do código necessitam de mais atenção. O profiling de programas envolve a execução de uma versão de um programa no qual foi incorporado o código de instrumentação para determinar quanto tempo as diferentes partes do programa requerem.

Nesta atividade, será utilizada a ferramenta de profiling *gprof* para análise do tempo de execução de quatro algoritmos de ordenação: *Quicksort*, *Counting sort*, *Insertion sort* e *Merge sort*.

# 2 Descrição do experimento

Para realizar o experimento, foi utilizado um programa auxiliar escrito em *shell script* (`runExperiments.sh`). O programa aceita um valor de *seed* para gerar um vetor de números aleatórios, um tamanho para esse vetor, e um intervalo de restrição para os números aleatórios.

Na presente análise foi fixado um valor de seed para todas as execuções enquanto o tamanho do vetor e o intervalo dos números aleatórios foram variados. A seguir são apresentados os valores utilizados no tamanho e intervalo:

| **Tamanho** | **Intervalo** |
|---:|---|
| 100 | [-100, 100] |
| 10000 | [-10000, 10000] |
| 1000000 | [-1000000, 1000000] |
| 100000000 | [-100000000, 100000000] |

Foram testadas todas as combinações Tamanho × Intervalo com um valor de seed fixado em 42, totalizando 16 execuções. Todos os algoritmos de ordenação foram implementados manualmente, com exceção do Quicksort, em que a implementação da `stdlib.h` foi utilizada.

Dentro da função *main* (`main.c`) os algoritmos de ordenação são executados em sequência, e, para que a cache não tenha influência no tempo de execução, uma função com a finalidade de limpá-la é executada antes de cada algoritmo.

Devido ao tempo de execução extremamente longo, a execução do Insertion sort foi excluída da análise para os vetores maiores que 10000.

O programa que gerou os experimentos, os comandos e os resultados podem ser acessados em: `https://github.com/rolimans/sortingAlgorithmsExperiments`

### 2.1 Software e hardware utilizados

- Software:

| | |
|---|---|
| Sistema operacional: | GNU/Linux |
| Kernel: | 5.15.71-1-MANJARO |
| Compilador: | gcc 12.2.0 |
| Linker: | GNU ld (GNU Binutils) 2.39.0 |
| Profiler: | GNU gprof (GNU Binutils) 2.39.0 |
| Image Generator: | gprof2dot 2022.7.29 |

- CPU Info:

| | |
|---|---|
| Arquitetura: | x86_64 |
| Byte Order: | Little Endian |
| CPU(s): | 4 |
| Nome modelo: | Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz |
| Thread(s) por core: | 2 |
| Core(s) por socket: | 2 |
| Socket(s): | 1 |

## 3 Execuções

### 3.1 Tamanho 100

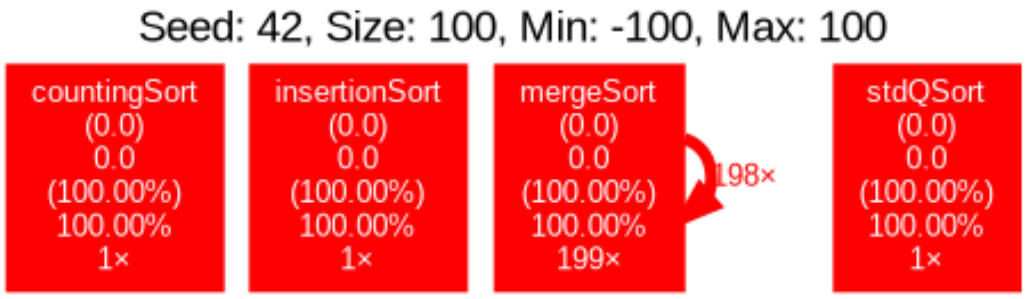


Figura 1: Grafo de chamada da execução com tamanho 100 e intervalo [-100, 100]

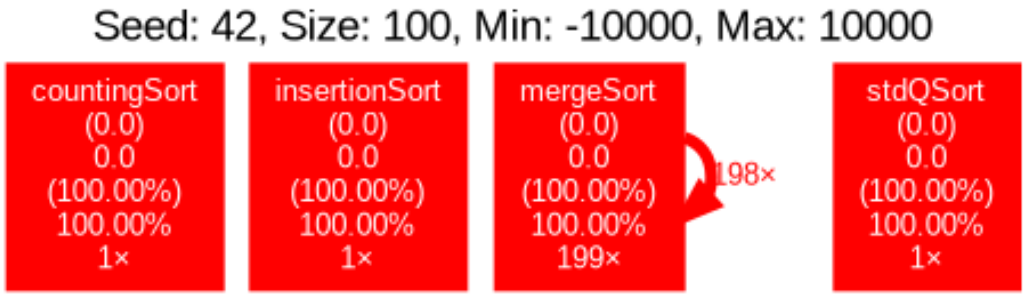


Figura 2: Grafo de chamada da execução com tamanho 100 e intervalo [-10000, 10000]

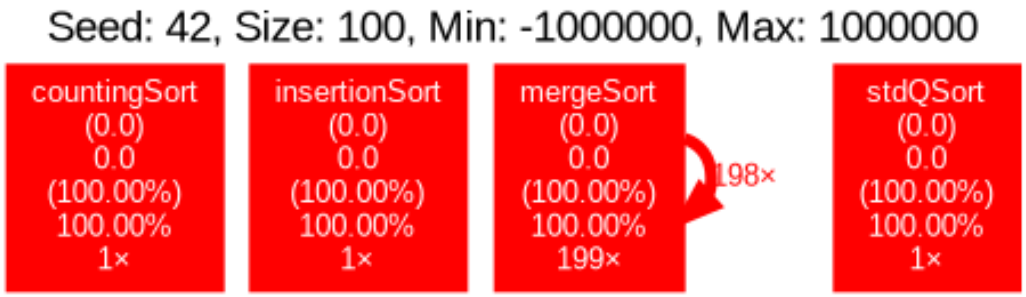


Figura 3: Grafo de chamada da execução com tamanho 100 e intervalo [-1000000, 1000000]

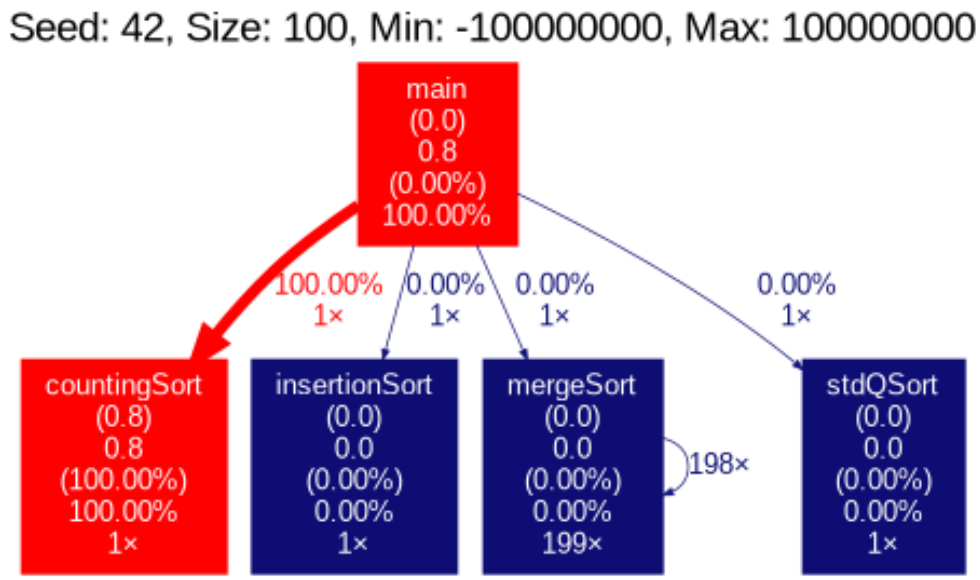


Figura 4: Grafo de chamada da execução com tamanho 100 e intervalo [-100000000, 100000000]

## 3.2 Tamanho 10000

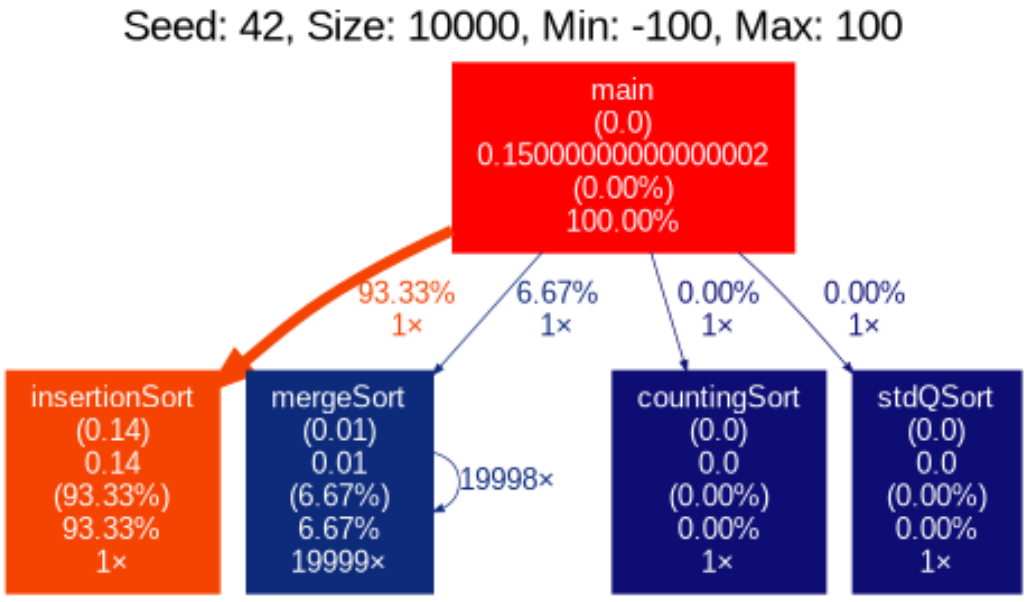


Figura 5: Grafo de chamada da execução com tamanho 10000 e intervalo [-100, 100]

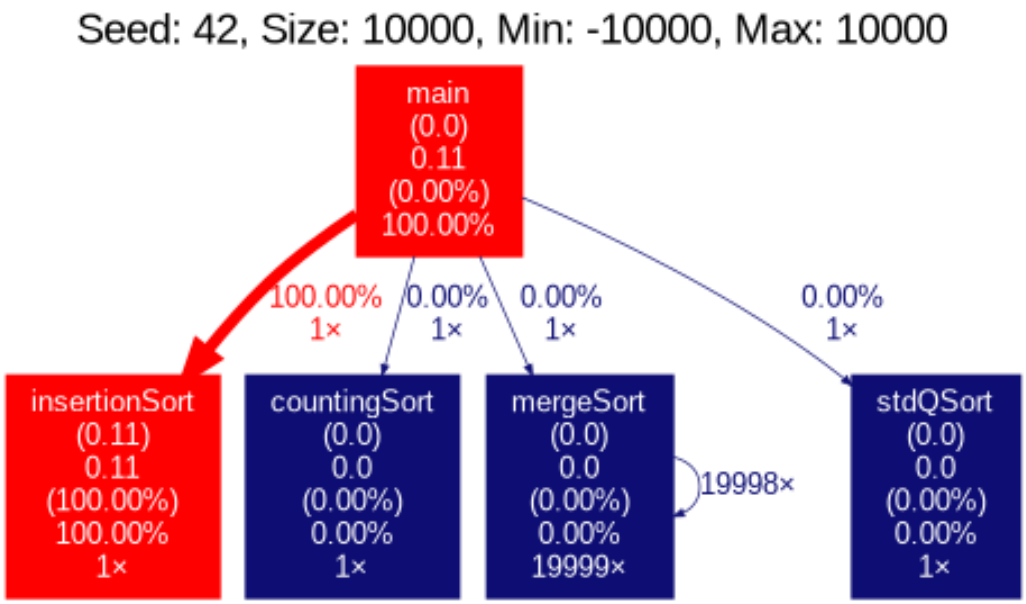


Figura 6: Grafo de chamada da execução com tamanho 10000 e intervalo [-10000, 10000]

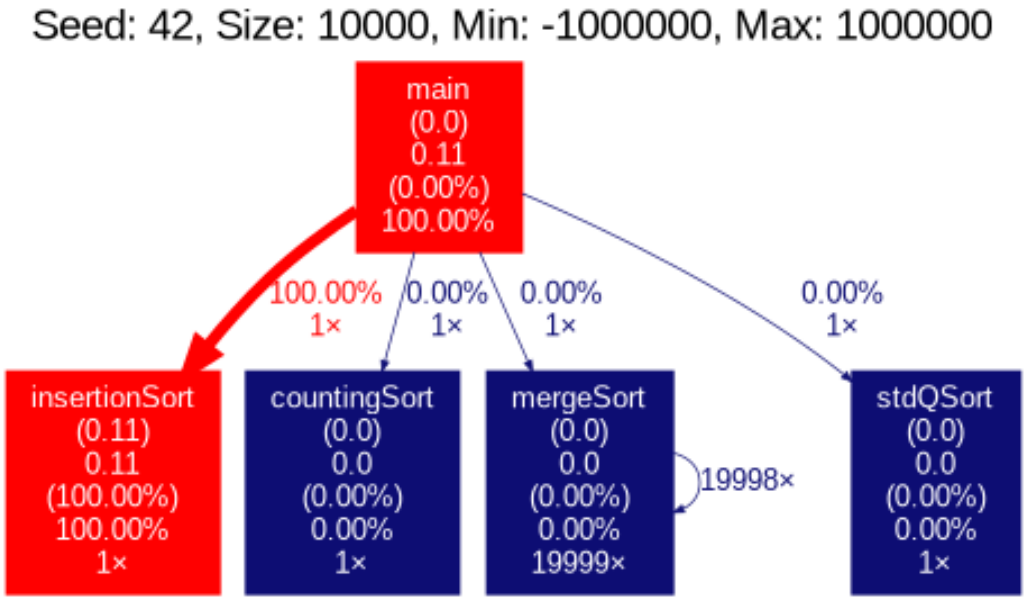


Figura 7: Grafo de chamada da execução com tamanho 10000 e intervalo [-1000000, 1000000]

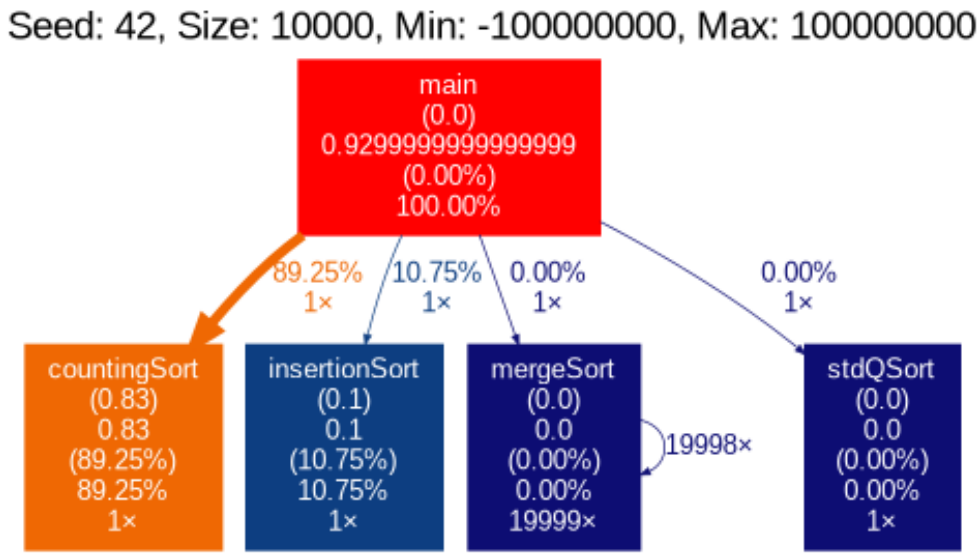


Figura 8: Grafo de chamada da execução com tamanho 10000 e intervalo [-100000000, 100000000]

### 3.3 Tamanho 1000000

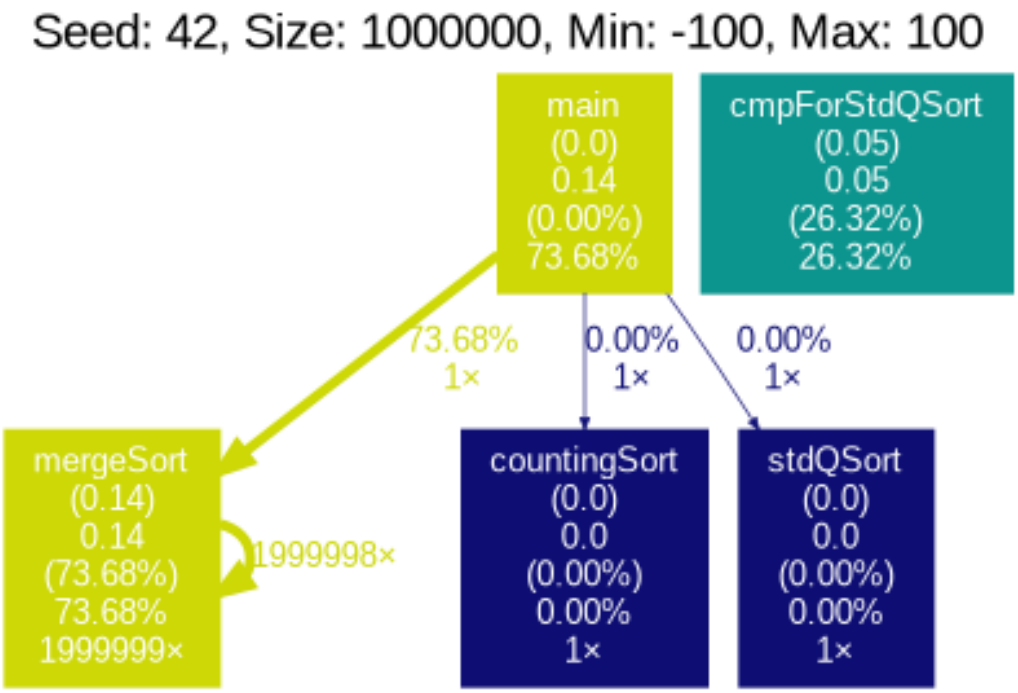


Figura 9: Grafo de chamada da execução com tamanho 1000000 e intervalo [-100, 100]

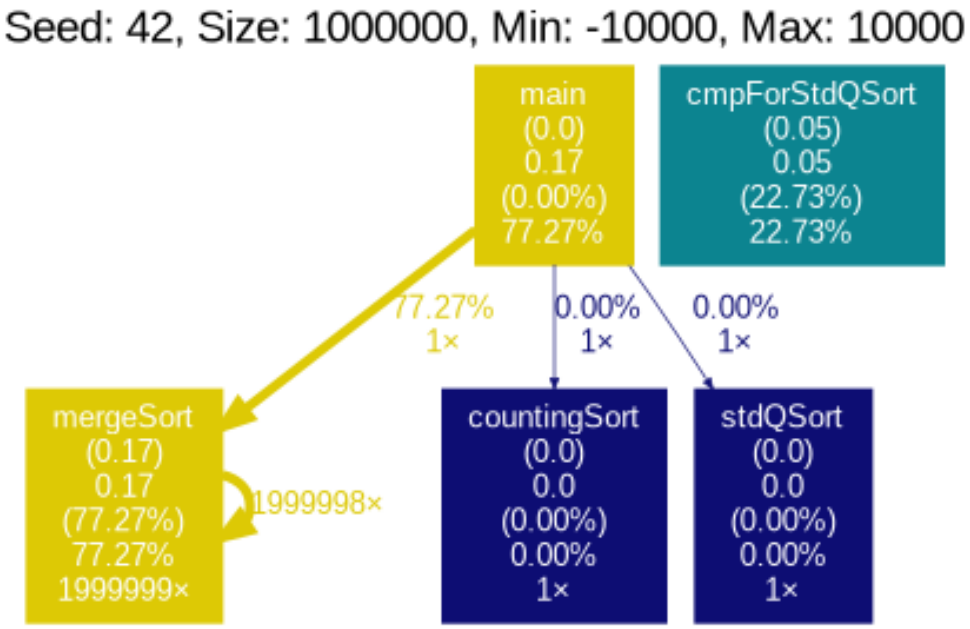


Figura 10: Grafo de chamada da execução com tamanho 1000000 e intervalo [-10000, 10000]

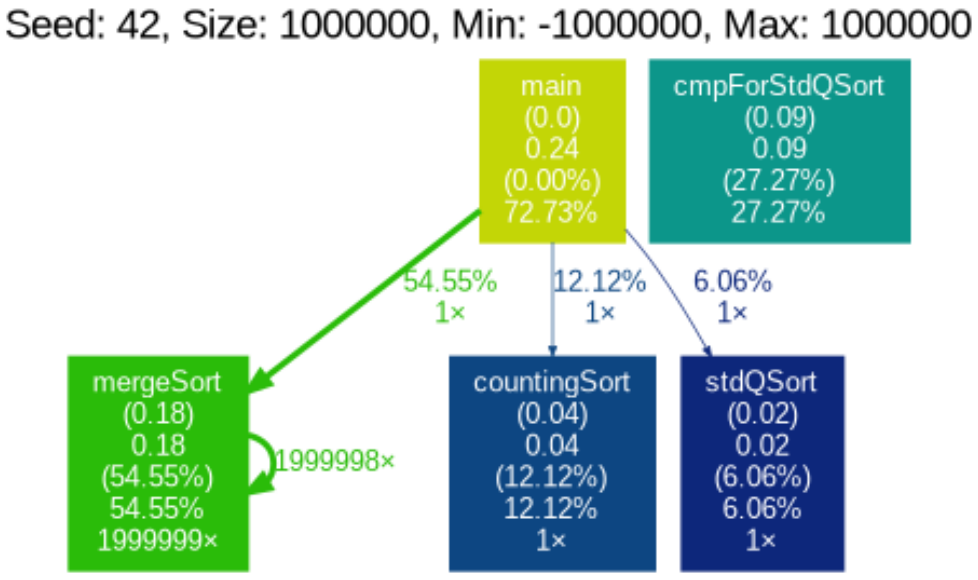


Figura 11: Grafo de chamada da execução com tamanho 1000000 e intervalo [-1000000, 1000000]

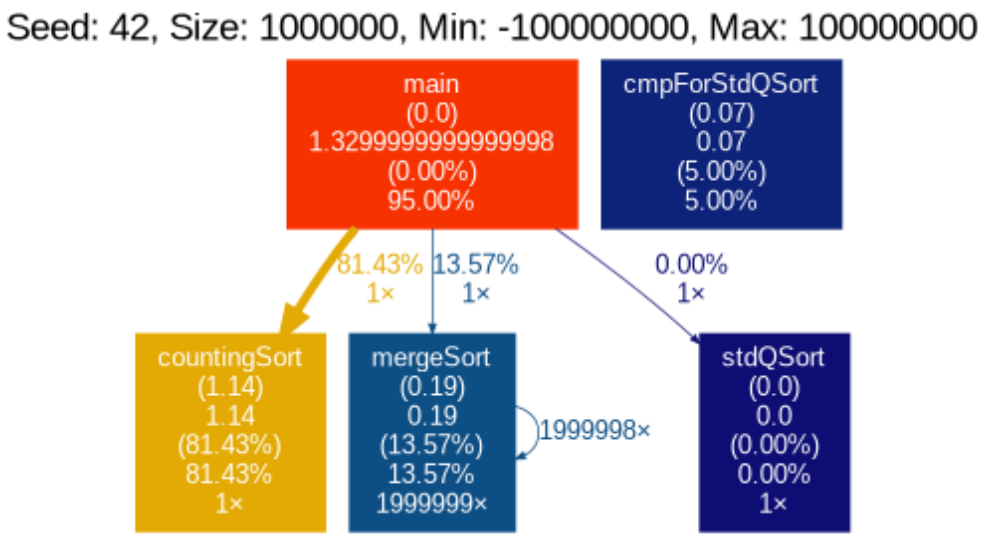


Figura 12: Grafo de chamada da execução com tamanho 1000000 e intervalo [-100000000, 100000000]

## 3.4 Tamanho 100000000

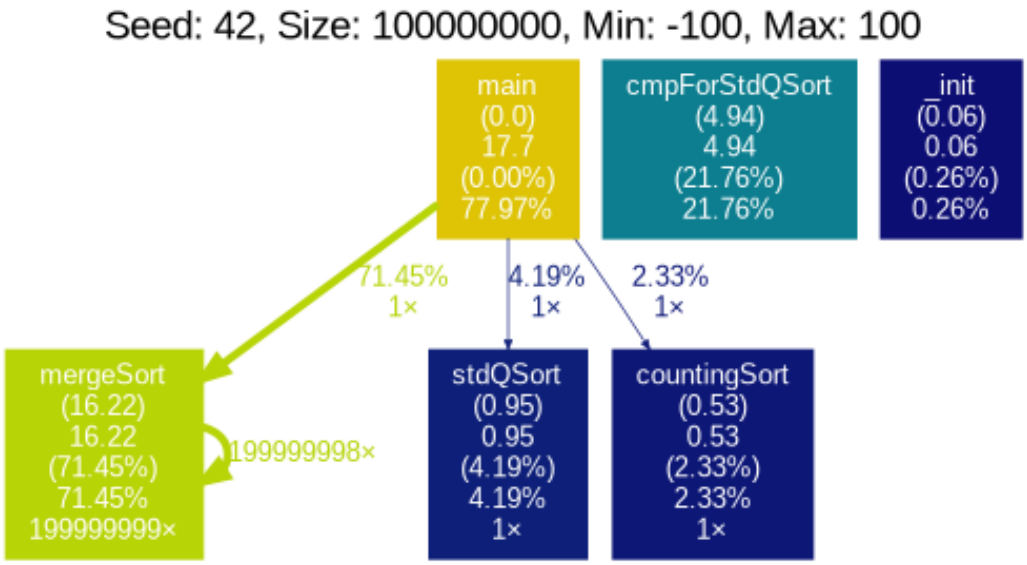


Figura 13: Grafo de chamada da execução com tamanho 100000000 e intervalo [-100, 100]

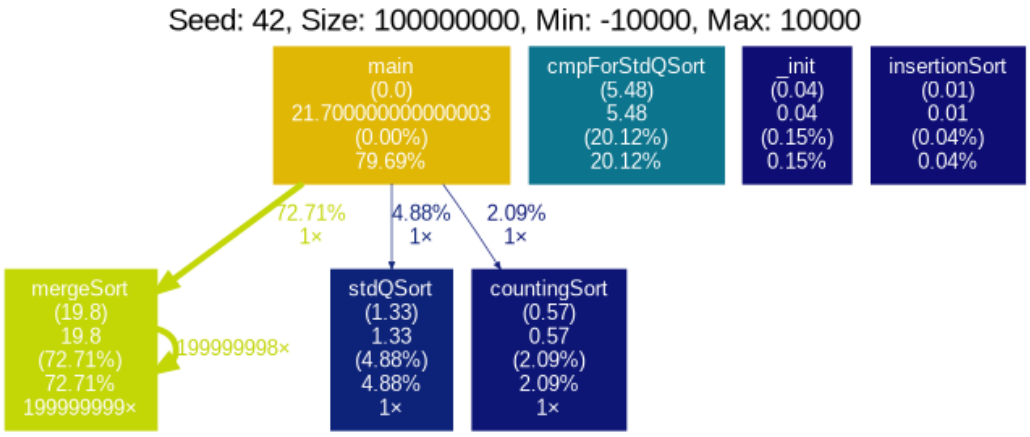


Figura 14: Grafo de chamada da execução com tamanho 100000000 e intervalo [-10000, 10000]

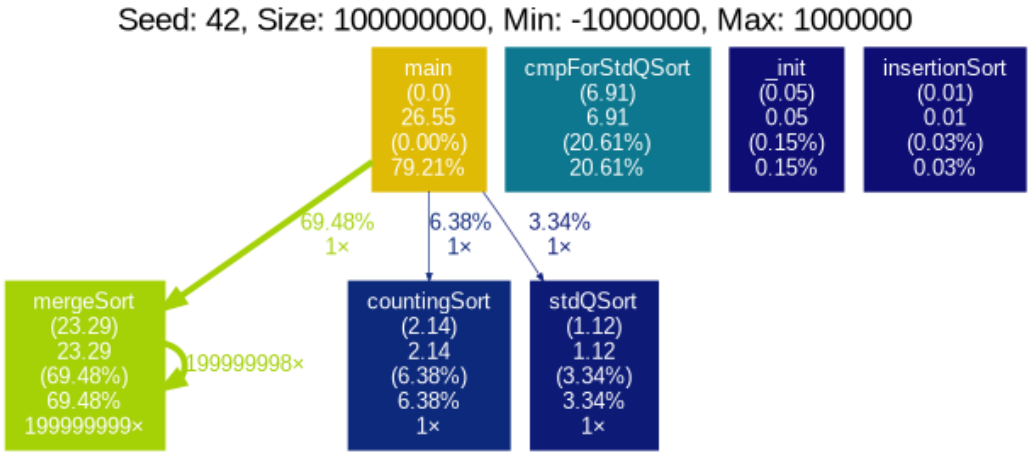


Figura 15: Grafo de chamada da execução com tamanho 100000000 e intervalo [-1000000, 1000000]

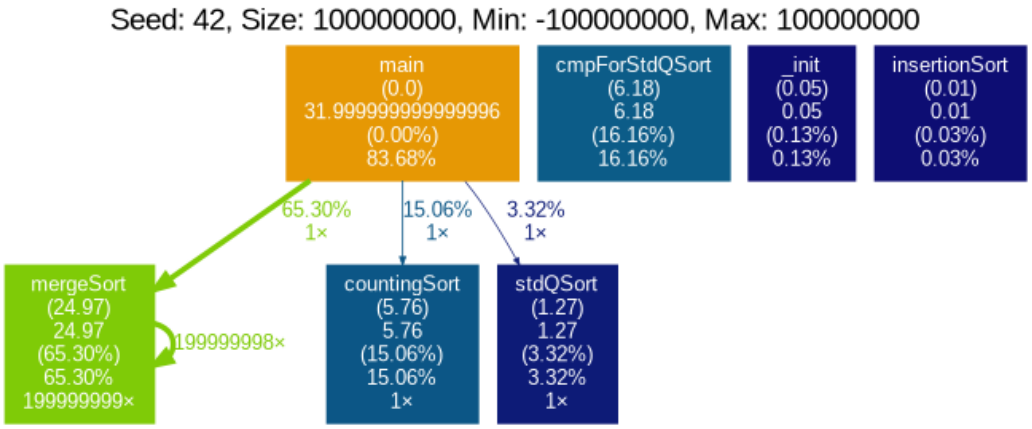


Figura 16: Grafo de chamada da execução com tamanho 100000000 e intervalo [-100000000, 100000000]

# 4 Conclusão

Após analisar os resultados gerados, é possível tirar as seguintes conclusões: O counting sort acaba se saindo pior para casos com ranges muito altos pois o algoritmo necessita de um intervalo menor para ter uma melhor complexidade. Em contrapartida em arrays de tamanho 100000000 é possível observar que o merge sort tem um desempenho um pouco pior que os outros algoritmos de ordenação(o quick e o counting). Além disso é possível observar pelos grafos gerados que a função "cmpForStdQSort"que é uma função de comparação chamada pelo quick sort leva um tempo um pouco maior para arrays grandes do que o próprio algoritmo. Também foi possível notar o rápido crescimento do tempo de execução do inserction sort, motivo pelo qual ele foi excluido de tamanhos maiores dos experimentos. Por fim, é interessante notar que a função que foi mais chamada sempre foi o mergeSort, visto a natureza recursiva do algoritmo.