Trabalho sobre o QuickSort

Henrique Oliveira da Cunha Franco
Professor: Alexei Manso Correa Machado
Disciplina: Projeto e Análise de Algoritmos

12 de junho de 2025

Introdução

Este artigo apresenta uma implementação paralela do algoritmo QuickSort, baseada na técnica de amostragem conhecida como Sample Sort. O método utiliza 4 processadores, particiona os dados usando pivôs extraídos de uma amostra, e ordena cada partição individualmente, explorando paralelismo.

1 Pseudocodigo - QuickSampleSort

```
Procedimento QuickSampleSort(A, n)
1. p = 4; s = 10; tam_amostra = p * s
2. Amostras[0..tam_amostra-1] <- valores aleatorios de A
3. QuickSort(Amostras, 0, tam_amostra - 1)
4. Para i = 0 ate p-2: Pivos[i] = Amostras[(i+1)*s]
5. Inicializar SubVetores sub1 a sub4
6. Para i = 0 ate n-1:
     Se A[i] < Pivos[0] entao sub1 <- A[i]
     Senao se A[i] < Pivos[1] entao sub2 <- A[i]
     Senao se A[i] < Pivos[2] entao sub3 <- A[i]
     Senao sub4 <- A[i]
7. Concatenar sub1, sub2, sub3, sub4 em A
8. QuickSort(A, 0, fim_sub1 - 1)
   QuickSort(A, fim_sub1, fim_sub2 - 1)
   QuickSort(A, fim_sub2, fim_sub3 - 1)
   QuickSort(A, fim_sub3, n - 1)
```

Listing 1: QuickSampleSort

Pseudocódigo QuickSampleSort

O pseudocódigo apresentado segue a ideia central do *Sample Sort* paralelizado, com as seguintes etapas principais:

- **Definição de parâmetros**:

  - Define-se o número de processadores `p = 4` e o número de amostras por processador `s = 10`.
  - O tamanho total da amostra é `p * s = 40`, permitindo uma divisão mais uniforme dos dados.

- **Amostragem aleatória**:
  - São escolhidos 40 elementos aleatórios do vetor original para formar a amostra.
  - Isso garante que os pivôs escolhidos representem bem a distribuição dos dados.

- **Ordenação da amostra**:
  - A amostra é ordenada usando QuickSort para que os pivôs possam ser escolhidos de forma regular.

- **Seleção dos pivôs**:
  - São selecionados `p - 1` pivôs nos índices `s`, `2s`, `3s` da amostra ordenada, ou seja, em posições igualmente espaçadas.

- **Particionamento dos dados**:
  - Cada elemento de `A` é comparado com os pivôs e alocado em um dos quatro subvetores correspondentes.

- **Concatenação e ordenação local**:
  - Os subvetores são concatenados de volta em `A` na ordem correta.
  - Cada subvetor é ordenado independentemente usando QuickSort, simulando a execução paralela em 4 processadores.

2 Codigo Java - QuickSampleSort

```java
import java.util.*;

public class QuickSampleSort {
    public static void main(String[] args) {
        int[] A = new int[40];
        Random rand = new Random();
        for (int i = 0; i < A.length; i++) A[i] = rand.nextInt(100);
        System.out.println("Antes: " + Arrays.toString(A));
        quickSampleSort(A);
        System.out.println("Depois: " + Arrays.toString(A));
    }

    public static void quickSampleSort(int[] A) {
        int p = 4, s = 10;
        int[] amostras = new int[p * s];
        Random rand = new Random();
        for (int i = 0; i < amostras.length; i++)
            amostras[i] = A[rand.nextInt(A.length)];
        Arrays.sort(amostras);
        int[] pivos = {amostras[s], amostras[2*s], amostras[3*s]};
        List<Integer>[] sub = new List[4];
        for (int i = 0; i < 4; i++) sub[i] = new ArrayList<>();
        for (int x : A) {
            if (x < pivos[0]) sub[0].add(x);
            else if (x < pivos[1]) sub[1].add(x);
            else if (x < pivos[2]) sub[2].add(x);
            else sub[3].add(x);
        }
        int idx = 0;
        for (List<Integer> part : sub) {
            int[] temp = part.stream().mapToInt(i -> i).toArray();
            quickSort(temp, 0, temp.length - 1);
            for (int val : temp) A[idx++] = val;
        }
    }

    public static void quickSort(int[] A, int l, int r) {
        int i = l, j = r, pivot = A[(l + r) / 2];
        while (i <= j) {
            while (A[i] < pivot) i++;
            while (A[j] > pivot) j--;
            if (i <= j) {
                int tmp = A[i]; A[i] = A[j]; A[j] = tmp;
                i++; j--;
            }
        }
        if (l < j) quickSort(A, l, j);
        if (i < r) quickSort(A, i, r);
    }
}
```

Listing 2: QuickSampleSort em Java

Implementação Java

A implementação em Java segue o mesmo raciocínio do pseudocódigo, com as seguintes decisões de projeto:

- **Geração de entrada**:
  - Um vetor `A[]` com 40 inteiros aleatórios entre 0 e 99 é criado para teste.
- **Amostragem e ordenação**:
  - A amostra é gerada selecionando `p * s` elementos aleatórios de `A`.

  - A ordenação é feita com `Arrays.sort()`.

- **Seleção dos pivôs**:
  - São escolhidos os pivôs `amostras[s]`, `amostras[2s]`, `amostras[3s]` da amostra ordenada.

- **Distribuição dos elementos**:
  - Os elementos de `A` são distribuídos entre quatro listas (`ArrayList`) com base nos pivôs.

- **Ordenação de sublistas e cópia final**:
  - Cada sublista é ordenada usando QuickSort tradicional.
  - Os valores são reagrupados sequencialmente em `A`, simulando a mesclagem final.

3 Análise de Complexidade dos Algoritmos

A análise de complexidade do *QuickSampleSort* considera os principais passos do algoritmo:

- **Amostragem Aleatória**:
  - Selecionar $p \cdot s$ elementos aleatórios leva tempo $O(p \cdot s)$, que é constante para $p = 4$.

- **Ordenação das Amostras**:
  - A ordenação de $p \cdot s$ elementos é feita com QuickSort: $O((p \cdot s)\log(p \cdot s))$.
  - Como p e s são constantes, este tempo também é considerado $O(1)$.

- **Particionamento do Vetor A**:
  - Cada elemento de A é comparado com no máximo 3 pivôs.
  - Custo total do particionamento: $O(n)$.

- **Ordenação das Partições**:
  - Cada subvetor tem, em média, $\frac{n}{p}$ elementos.
  - Cada um é ordenado via QuickSort, resultando em tempo $O\left(\frac{n}{p}\log\frac{n}{p}\right)$.
  - Como há p partições, o tempo total dessa etapa é:

$$p \cdot O\left(\frac{n}{p}\log\frac{n}{p}\right) = O(n \log \frac{n}{p})$$

Complexidade Total:

$$O(n) + O(n \log \frac{n}{p}) = O(n \log \frac{n}{p})$$

Portanto, com paralelismo ideal, a complexidade do algoritmo é próxima de $O(n \log n)$, mas com vantagem de divisão de carga entre os processadores. Em termos de tempo por processador, temos:

$$T_p(n) = O\left(\frac{n}{p} \log \frac{n}{p}\right)$$

Ou seja, o algoritmo é escalável e eficiente para ambientes com múltiplos núcleos ou máquinas paralelas.

Referencias

- Joseph Jaja. "A Perspective on QuickSort". Computing in Science and Engineering, vol. 2, no. 1, pp. 43–49, 2000.