Análise do Código de Benchmark para Merge Sort Sequencial e Paralelo

Prof. Titular: Gabriel Soares Baptista

Aluno: Davi Rodrigues do Nascimento

Disciplina: Sistemas Operacionais - D009527

Instituição de Ensino: FAESA

Introdução

Este relatório tem como objetivo descrever detalhadamente o funcionamento do código utilizado para benchmark do algoritmo de ordenação Merge Sort, tanto na versão sequencial quanto na versão paralela. O código realiza medições de tempo de execução, gera um gráfico comparativo e salva os resultados em arquivos CSV.

Objetivo do Código

O código busca comparar a eficiência da implementação tradicional do Merge Sort com uma versão otimizada que utiliza processamento paralelo. Os tempos de execução são coletados para diferentes tamanhos de entrada, permitindo visualizar o ganho de desempenho ao utilizar paralelismo ou não.

Estrutura do Código

O código está estruturado nas seguintes etapas principais:

1. **Leitura de Arquivos**: Carrega números a partir de arquivos de texto.
2. **Execução dos Algoritmos**: Mede o tempo de execução do Merge Sort sequencial e paralelo.
3. **Armazenamento dos Resultados**: Salva os tempos em um arquivo CSV.
4. **Geração de Gráfico**: Cria um boxplot comparativo para visualizar os tempos de execução.

Implementação dos Algoritmos

Merge Sort Sequencial

O Merge Sort é um algoritmo de ordenação baseado na estratégia **Dividir para Conquistar**. Ele segue os seguintes passos:

1. Divide o array em duas metades.
2. Ordena cada metade recursivamente.
3. Mescla as duas metades ordenadas.

```
# MERGE SORT SEQUENCIAL
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        left_half = merge_sort(left_half)
        right_half = merge_sort(right_half)

        return merge(left_half, right_half)
    return arr
```

Merge Sort Paralelo

A versão paralela utiliza **multiprocessamento** para dividir o trabalho entre 2 núcleos da CPU. Isso melhora a eficiência para entradas grandes.

```
# MERGE SORT PARALELO
def parallel_merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    with multiprocessing.Pool(processes=2) as pool:
```

```
        left_sorted, right_sorted = pool.map(merge_sort, [left_half, right_half])

    return merge(left_sorted, right_sorted)
```

Medindo o Tempo de Execução

A função `benchmark_sorting` é usada para medir o tempo de execução de cada algoritmo.

```
import time

# MEDIR TEMPO DE EXECUÇÃO
def benchmark_sorting(sorting_function, numbers, iterations=10):
    times = []
    for _ in range(iterations):
        numbers_copy = numbers[:]
        start_time = time.perf_counter()
        sorting_function(numbers_copy)
        end_time = time.perf_counter()
        times.append((end_time - start_time) * 1000)  # Convertendo para mili
ssegundos
    return times
```

Geração dos Gráficos

O gráfico compara os tempos de execução dos dois algoritmos para diferentes tamanhos de entrada.

```
import matplotlib.pyplot as plt

# GERAR GRAFICO COMPARATIVO
def plot_results(seq_times_dict, par_times_dict, title, save_path):
    plt.figure(figsize=(12, 6))

    # Extraindo os dados para cada tamanho de arquivo
    seq_data = [seq_times_dict[key] for key in sorted(seq_times_dict.keys())]
    par_data = [par_times_dict[key] for key in sorted(par_times_dict.keys())]
```

```
    # Número de conjuntos de dados para cada método
    num_seq_data = len(seq_data)
    num_par_data = len(par_data)

    # Adicionando os boxplots com os dados
    plt.boxplot(seq_data, positions=range(1, num_seq_data + 1), widths=0.4,
patch_artist=True, labels=[f"Sequencial {key}" for key in sorted(seq_times_
dict.keys())])
    plt.boxplot(par_data, positions=range(num_seq_data + 1, num_seq_data
+ num_par_data + 1), widths=0.4, patch_artist=True, labels=[f"Paralelo {ke
y}" for key in sorted(par_times_dict.keys())])

    plt.ylabel("Tempo (ms)")
    plt.title(title)
    plt.grid(True)

    # Ajustando os rótulos do eixo X para ficarem na vertical
    plt.xticks(rotation=90)

    # Salvando o gráfico no diretório especificado
    plt.savefig(save_path, bbox_inches='tight')
    print(f"Gráfico salvo em: {save_path}")
    plt.show()
```

Por que a implementação sequencial tem o tempo menor do que o paralelo para listas menores, mas não para as maiores?

A implementação paralela do Merge Sort pode ser mais lenta para listas pequenas devido ao alto custo de criação e gerenciamento de processos, além da comunicação necessária entre eles. Para listas menores, o tempo gasto nessas operações de sincronização e troca de contexto pode superar os benefícios do paralelismo, tornando a abordagem sequencial mais eficiente.

No entanto, para listas maiores, o tempo de ordenação se torna significativamente maior, e o paralelismo começa a trazer vantagens. A divisão do trabalho entre múltiplos núcleos permite uma redução considerável no tempo total de execução, já que o tempo de processamento domina o overhead de gerenciamento de processos. Assim, o paralelismo se torna mais eficaz à medida que o tamanho dos dados aumenta.

Conclusão

Este código fornece uma análise detalhada do desempenho do Merge Sort na versão sequencial e paralela. Com a geração de gráficos e relatórios PDF, é possível visualizar o impacto do paralelismo e identificar cenários onde ele é vantajoso.