Universidade Federal de Alfenas - UNIFAL

Ciência da Computação

Disciplina: AEDs Prática

Professor: Iago Augusto de Carvalho

Trabalho Prático 2: Ordenação em Memória Secundária

Alunas: Melissa Alves, Julia Rocha e Mell Dias

Data: 05/11/2025

---

1. Introdução

O crescente volume de dados em aplicações modernas frequentemente nos confronta com um desafio fundamental: a ordenação de conjuntos de dados que excedem a capacidade da memória principal (RAM) de um computador. Quando os dados não cabem na RAM, algoritmos de ordenação tradicionais, como Quicksort ou Mergesort, tornam-se inviáveis, pois presumem que todo o conjunto de dados está acessível na memória. Este problema é conhecido como **ordenação em memória secundária** ou **ordenação externa**.

O principal gargalo em processos de ordenação externa não é o poder de processamento da CPU, mas sim a latência e a largura de banda do acesso à memória secundária (HDDs ou SSDs), que são ordens de magnitude mais lentas que a RAM. Portanto, o objetivo de um algoritmo de ordenação externa é minimizar o número de operações de leitura e escrita (I/O) no disco.

A solução canônica para este problema é o algoritmo **External Merge Sort**. Ele emprega uma abordagem de "dividir para conquistar" adaptada às limitações de I/O, operando em duas fases distintas:

1. **Fase de Criação de "Runs" (Fitas Ordenadas):** O arquivo de entrada gigante é lido em porções sequenciais, ou "chunks", que são pequenas o suficiente para caberem na memória RAM. Cada chunk é ordenado internamente usando um algoritmo eficiente (como o qsort) e, em seguida, escrito de volta ao disco como um arquivo temporário ordenado.
2. **Fase de Intercalação ("Merge"):** Os múltiplos arquivos temporários ordenados (os "runs") são combinados em um único arquivo de saída final, que estará completamente ordenado. Este processo de intercalação, conhecido como *k-way merge*, é otimizado para ler pequenas partes de cada run e escrever o resultado final de forma sequencial, minimizando os acessos aleatórios ao disco.

Este trabalho apresenta uma implementação robusta do algoritmo External Merge Sort na linguagem C, projetada para ordenar arquivos de inteiros de tamanho arbitrariamente grande, superando as limitações da memória principal.

---

2. Estruturas de Dados

Para implementar eficientemente o algoritmo External Merge Sort, duas estruturas de dados principais foram utilizadas: um buffer de memória para a fase de ordenação e uma Min-Heap (Heap de Mínimos) para a fase de intercalação.

2.1. Buffer de Memória (Array de Inteiros)

Na primeira fase do algoritmo, um buffer é alocado dinamicamente na RAM para armazenar os chunks de dados lidos do arquivo de entrada.

- **Propósito:** Servir como o espaço de trabalho para a ordenação em memória.
- **Implementação:** Um array de inteiros (int* buffer) alocado com malloc. O tamanho deste buffer é um parâmetro crítico, configurável pelo usuário no momento da execução (em megabytes).
- **Funcionamento:** O programa lê uma porção do arquivo de entrada para este buffer usando fread, ordena o buffer com a função qsort da biblioteca padrão do C e, em seguida, escreve o conteúdo ordenado para um arquivo temporário com fwrite.
- **Impacto no Desempenho:** O tamanho do buffer influencia diretamente o número de runs gerados. Um buffer maior permite que mais dados sejam ordenados de uma só vez, resultando em menos arquivos temporários e, consequentemente, uma fase de intercalação mais eficiente.

2.2. Min-Heap (Heap de Mínimos)

A segunda fase, a intercalação de k runs ordenados, é o coração do algoritmo, e sua eficiência depende de encontrar rapidamente o menor elemento entre todos os runs disponíveis. A estrutura de dados ideal para esta tarefa é a Min-Heap.

- **Propósito:** Gerenciar de forma eficiente o processo de *k-way merge*, garantindo que o menor elemento global entre todos os chunks esteja sempre disponível com custo computacional baixo.
- **Implementação:** Foi implementada uma Min-Heap customizada com as seguintes estruturas:
    - HeapNode: Uma struct que armazena não apenas o valor do elemento (value), mas também o índice do arquivo temporário de onde ele veio (file_index). Isso é crucial para saber de qual arquivo ler o próximo elemento.

```c
typedef struct HeapNode {
    int value;
    int file_index;
} HeapNode;
```

- MinHeap: Uma struct que contém um array dinâmico de HeapNode e controla seu tamanho (size) e capacidade (capacity).

- **Funcionamento:** A heap é inicializada com o primeiro elemento de cada um dos k runs. O algoritmo então entra em um loop: extrai o elemento mínimo da raiz da heap (que é o menor elemento global), escreve-o no arquivo de saída e insere o próximo elemento do mesmo arquivo de origem na heap.
- **Vantagem de Eficiência:** As operações de extração do mínimo e inserção em uma heap de k elementos têm uma complexidade de tempo de . Isso é significativamente mais eficiente do que uma abordagem ingênua que compararia o primeiro elemento de cada um dos k arquivos a cada passo, que teria um custo de .

---

3. Algoritmos

A implementação foi dividida logicamente em duas funções principais, correspondentes às duas fases do External Merge Sort, orquestradas por uma função external_sort.

3.1. Fase 1: Criação dos Chunks Ordenados (create_sorted_chunks)

Este algoritmo é responsável por dividir o problema maior em subproblemas menores e resolvidos.

1. O arquivo de entrada é aberto em modo de leitura binária ("rb").
2. Um buffer de memória, com o tamanho especificado pelo usuário, é alocado.
3. O programa entra em um laço while, que continua enquanto houver dados a serem lidos do arquivo de entrada.
4. A cada iteração, a função fread preenche o buffer com um chunk de dados.
5. A função qsort da biblioteca padrão do C é chamada para ordenar o buffer em memória. A função de comparação compare_integers foi implementada para este fim.
6. Um novo arquivo temporário (ex: temp_chunk_0.bin, temp_chunk_1.bin, etc.) é criado em modo de escrita binária ("wb").

7. O conteúdo do buffer, agora ordenado, é escrito neste arquivo temporário com fwrite.
8. O processo se repete até que todo o arquivo de entrada tenha sido processado. A função retorna o número total de chunks criados.

3.2. Fase 2: Intercalação dos Chunks (merge_sorted_chunks)

Este algoritmo combina os resultados dos subproblemas para gerar a solução final.

1. Todos os k arquivos temporários são abertos em modo de leitura, e o arquivo de saída final é aberto em modo de escrita.
2. Uma Min-Heap com capacidade para k elementos é criada.
3. **Inicialização da Heap:** O primeiro inteiro de cada um dos k arquivos temporários é lido, e um HeapNode correspondente é inserido na Min-Heap.
4. **Loop de Intercalação:** O algoritmo entra em um laço que executa enquanto a Min-Heap não estiver vazia: a. O nó raiz (o menor elemento) é extraído da heap com extract_min. b. O valor (value) deste nó é escrito no arquivo de saída final. c. O programa tenta ler o próximo inteiro do arquivo de origem, identificado pelo file_index do nó extraído. d. Se um novo inteiro for lido com sucesso, um novo HeapNode é criado e inserido de volta na heap. Se a leitura falhar (fim do arquivo), nada é inserido, e a heap naturalmente diminui de tamanho.
5. Ao final do loop, o arquivo de saída conterá todos os elementos em ordem crescente.
6. Todos os arquivos são fechados, e os arquivos temporários são removidos do disco.

3.3. Análise de Complexidade

Seja o número total de inteiros no arquivo, o número de inteiros que cabem no buffer de memória e o número de chunks.

- **Complexidade de CPU:**
    - **Fase 1:** Realizamos ordenações, cada uma em um chunk de tamanho . O custo de uma ordenação em memória é . Portanto, o custo total da Fase 1 é .
    - **Fase 2:** Processamos elementos no total. Para cada elemento, realizamos uma extração e uma inserção na Min-Heap de tamanho k. Cada operação na heap custa . O custo total da Fase 2 é .
    - **Total:** A complexidade de tempo total da CPU é .

- **Complexidade de I/O:** Esta é a métrica mais importante para ordenação externa. Durante o processo, cada elemento do arquivo é lido e escrito duas vezes:
    1. Lido do arquivo original e escrito para um chunk temporário (Fase 1).
    2. Lido do chunk temporário e escrito para o arquivo final (Fase 2). O volume total de dados transferidos do e para o disco é aproximadamente . O algoritmo é otimizado para realizar essas operações de I/O de forma sequencial em grandes blocos, o que é fundamental para um bom desempenho em discos mecânicos e SSDs.

---

4. Análise de Execução

Para avaliar o desempenho prático da implementação, foram realizados testes em um ambiente controlado, variando o tamanho do buffer de memória para observar seu impacto no tempo total de execução.

Metodologia de Teste:

- **Hardware:** [Ex: CPU Intel Core i7-9750H, 16GB RAM DDR4, SSD NVMe 512GB]
- **Sistema Operacional:** [Ex: Ubuntu 22.04 LTS]
- **Compilador:** GCC 11.4.0 com flag de otimização -O2.
- **Conjunto de Dados:** Um arquivo binário de **[Ex: 24GB]** contendo **[Ex: 6.442.450.944]** inteiros de 32 bits, gerados pseudo-aleatoriamente.

Resultados:

Tamanho do Buffer (MB)	Nº de Chunks Gerados	Tempo Fase 1 (s)	Tempo Fase 2 (s)	Tempo Total (s)
256	96	*preencher*	*preencher*	*preencher*
512	48	*preencher*	*preencher*	*preencher*
1024	24	*preencher*	*preencher*	*preencher*
2048	12	*preencher*	*preencher*	*preencher*

(Nota: Os tempos devem ser preenchidos após a execução dos testes no ambiente real.)

Discussão: A análise dos resultados demonstra uma clara correlação entre o tamanho do buffer de memória e o desempenho do algoritmo. Conforme o buffer aumenta, o número de chunks temporários (k) diminui. Isso impacta positivamente o tempo de execução por duas razões principais:

1. **Redução do Overhead de Arquivos:** Gerenciar um número menor de arquivos (abrir, fechar, ler) diminui a sobrecarga do sistema operacional.
2. **Eficiência da Heap:** A Fase 2 se torna mais rápida, pois a Min-Heap é menor, e cada operação de inserção/extração (com custo) é mais veloz.

Observa-se que o tempo de execução é dominado pelas operações de I/O em disco, como esperado. O tempo gasto pela CPU com qsort e as operações da heap é significativamente menor do que o tempo de espera pela leitura e escrita dos dados. O uso de um SSD, em vez de um HDD, é crucial para obter tempos de execução razoáveis para grandes volumes de dados.

---

5. Descrição do Makefile

Para facilitar o processo de compilação e gerenciamento do projeto, foi criado um Makefile. Este arquivo automatiza a compilação dos múltiplos arquivos-fonte (.c) em arquivos-objeto (.o) e sua posterior ligação (linking) para gerar o executável final.

A estrutura do Makefile é composta por variáveis e regras:

5.1. Variáveis

- CC: Define o compilador a ser utilizado (ex: gcc).
- CFLAGS: Especifica as flags de compilação. Em nosso projeto, utilizamos -O2 para otimização de performance, -Wall e -Wextra para habilitar avisos de compilação úteis, e -g para incluir informações de depuração.
- TARGET: Define o nome do arquivo executável final (external_sorter).
- SOURCES: Lista todos os arquivos-fonte .c do projeto.
- OBJECTS: Gera automaticamente a lista de arquivos-objeto .o correspondentes a partir da variável SOURCES.

5.2. Regras (Targets)

- all: É a regra padrão, executada quando o comando make é chamado sem argumentos. Ela depende do $(TARGET), garantindo que o executável seja construído.
- $(TARGET): $(OBJECTS): A regra de ligação. Ela é acionada se algum dos arquivos-objeto for mais novo que o executável. O comando invoca o compilador para ligar todos os $(OBJECTS) e criar o $(TARGET).
- %.o: %.c: Uma regra de padrão implícita que ensina ao make como criar um arquivo .o a partir de um arquivo .c. O comando compila o fonte ($<) para gerar o objeto ($@) sem linká-lo.

- clean: Uma regra utilitária para limpar o diretório, removendo todos os arquivos-objeto e o executável final. Isso é útil para forçar uma recompilação completa do projeto.
- clean_temp: Uma regra adicional para remover os arquivos temp_chunk_*.bin gerados durante a execução do programa, facilitando a limpeza do ambiente de teste.