Ordenação Externa e Reconstrução de Arquivo: Uma Abordagem com K-Way Merge

Luiz Guilherme Monteiro Padilha, RA: 1924745

Abstract—Este trabalho apresenta uma implementação e avaliação de desempenho de um algoritmo de ordenação externa para um arquivo.vet de 6.4 GB, com o objetivo de reconstruir um arquivo .TAR. A metodologia baseia-se na criação de runs, sequências ordenadas em memória, cujo tamanho é controlado pelo parâmetro max_blocos,número máximo de registos que o buffer em RAM pode conter para a ordenação interna, seguido pela intercalação destas runs através de um k-way merge com múltiplas passagens, otimizado por um heap de mínimo. Os experimentos variaram o max_blocos para quantificar o seu impacto no tempo de execução, no volume de I/O lógico e no uso de recursos do sistema. Os resultados demonstram que o aumento do max_blocos reduz drasticamente o número de runs e de passagens de intercalação, diminuindo o tempo total de execução de mais de 4 minutos para aproximadamente 1 minuto. A análise identifica um ponto de retornos decrescentes para o tamanho do buffer e valida a teoria de I/O lógico.

Index Terms—Ordenação Externa, K-Way Merge, Heap de Mínimo, Estruturas de Dados

1 Introdução

O Problema da ordenação externa surge naturalmente quando o volume de informações a serem processadas ultrapassa a capacidade de armazenamento da memória principal disponível, diferentemente da ordenação interna, em que pressupõe acesso rápido e irrestrito aos dados, a ordenação externa envolve lidar com grandes arquivos por meio de acessos sequenciais, armazenando apenas partes dos dados por vez na memória principal. Nesse contexto, os algoritmos precisam ser projetados para minimizar operações aleatórias no armazenamento secundário, que é significativamente mais lento, e explorar a leitura e escrita sequencial dos dados, visando eficiência na manipulação de grandes volumes de informações [1].

Historicamente, estratégias de ordenação externa surgiram com o intuito de contornar as limitações técnicas das máquinas disponíveis em cada época, como a restrição na capacidade de memória principal e o número limitado de arquivos abertos simultaneamente pelo sistema operacional. Entre as estratégias clássicas mais consolidadas para tratar esse problema, Ziviani (2010)[1] destaca a intercalação balanceada de múltiplos caminhos, a seleção por substituição e a intercalação polifásica, cada uma dessas abordagens possui características próprias, mas todas têm em comum o princípio de subdividir inicialmente o arquivo original em pequenas sequências ordenadas denominadas *runs*, que são posteriormente intercaladas de maneira iterativa, até que se obtenha um único arquivo ordenado ao final do processo.

Neste trabalho, foi adotado uma abordagem semelhante à proposta por Ziviani (2010)[1], construindo inicialmente diversas runs ordenadas com um tamanho determinado pela memória disponível, utilizando uma estrutura de dados conhecida como *heap* de mínimo, essas *runs* são intercaladas em múltiplos passes até que todos os registros estejam ordenados. Além disso, para a validação experimental, foi desenvolvido um conjunto de testes utilizando arquivos no formato *.TAR*, originalmente com-

• *Programa de Pós-Graduação em Ciência da Computação (PPGCC), Universidade Tecnológica Federal do Paraná - UTFPR. Área de Concentração: Sistemas e Métodos de Computação; Linha de Pesquisa: Teoria da Computação.*
E-mail: luipad@alunos.utfpr.edu.br

Manuscript received Jun 05, 2025.

posto por diferentes tipos de dados, que foram fragmentados em blocos fixos de tamanho inicial de 262 bytes, contendo um índice sequencial, o tamanho útil do dado armazenado até 250 bytes.

Durante os experimentos iniciais, identificamos que os arquivos TAR maiores apresentavam problemas decorrentes do desalinhamento dos blocos em relação ao padrão de 512 bytes exigido pelas especificações do formato *.TAR*, esse desalinhamento ocorreu porque os registros iniciais tinham exatamente 262 bytes e, ao final da reconstrução, não resultavam em um tamanho múltiplo de 512, condição obrigatória para que um arquivo *.TAR* seja reconhecido corretamente por ferramentas padrão.

Por fim, uma preocupação adicional neste estudo foi garantir a integridade completa do arquivo reconstruído, isso implicou não apenas ordenar corretamente os registros pelos índices originais, mas também restaurar a estrutura original do arquivo *.TAR*, cumprindo rigorosamente seus padrões de alinhamento e inserindo o preenchimento adicional necessário ao final do arquivo. Todos os algoritmos foram desenvolvidos e testados levando em conta essas particularidades, e seus resultados experimentais serão apresentados ao longo deste trabalho, destacando especialmente a influência do tipo de armazenamento secundário e das restrições do sistema operacional sobre a eficiência final da solução proposta.

2 METODOLOGIA

A abordagem utilizada neste trabalho foi estruturada em etapas bem definidas, partindo de uma análise cuidadosa do problema proposto, passando pelo desenvolvimento do algoritmo propriamente dito e pela resolução dos desafios encontrados nos arquivos originais de teste fornecidos. Inicialmente, compreendemos que o principal objetivo era ordenar externamente grandes quantidades de registros fixos de 264 bytes, reconstruindo ao final um arquivo válido no formato TAR.

Para compreender claramente a estrutura interna dos arquivos utilizados, observamos que cada registro fornecido possui quatro campos:

- **Chave (idx)**: 8 bytes, representando a posição original do pacote dentro do arquivo original;
- **Tamanho válido (tam)**: 4 bytes, indicando a quantidade real de dados válidos armazenados no campo seguinte;
- **Dados úteis (pacote)**: 250 bytes, fragmentos do conteúdo original;
- **Padding (preenchimento)**: 2 bytes, adicionados para garantir alinhamento interno dos registros.

No entanto, um problema significativo foi identificado logo nas análises iniciais dos arquivos fornecidos (misturado-1234.vet, misturado-actg.vet, e misturado-grande.vet). Os testes revelaram que, dos três arquivos, apenas o arquivo misturado-1234.vet permitia uma reconstrução correta do arquivo original em formato *.TAR*, enquanto os demais apresentavam problemas graves, como a incapacidade de serem reconhecidos por ferramentas padrão de extração *.TAR* (*tar -tf* reportava erros inesperado no final do arquivo).

Uma investigação detalhada utilizando ferramentas de análise de dados binários como *hexdump* revelou que o problema não estava especificamente relacionado à ausência ou presença dos dois *bytes* adicionais de padding, que foi a primeira hipótese levantada, mas sim à maneira como os pacotes de 250 *bytes* haviam sido inicialmente extraídos (fatiados) do arquivo original para formar os registros. Em particular, detectamos que os dados úteis estavam desalinhados em relação aos blocos originais do arquivo *.TAR*, resultando em pacotes incompletos ou com dados que não correspondiam ao esperado por ferramentas de manipulação *TAR*.

A consequência direta desse desalinhamento inicial foi a impossibilidade de reconstruir corretamente arquivos *.TAR* válidos a partir dos arquivos maiores (misturado-actg.vet e misturado-grande.vet). A descoberta desse problema foi confirmada por meio da inspeção manual dos cabeçalhos dos arquivos reconstruídos, que não possuíam as informações corretas nem os blocos finais obrigatórios de 512 *bytes* preenchidos com zeros, exigidos pelo formato *..TAR*.

Essa análise cuidadosa trouxe à necessi-

dade de implementar uma nova estratégia de geração dos arquivos de teste, garantindo desde o princípio que os pacotes fossem fatiados adequadamente do arquivo original, respeitando exatamente a estrutura interna do *.TAR*. Essa medida foi crucial para validar efetivamente o método de ordenação externa.

A figura 1 apresenta forma de extração dos pacotes de 250 *bytes* de um arquivo *.TAR* original e montar os registros com alinhamento adequado:

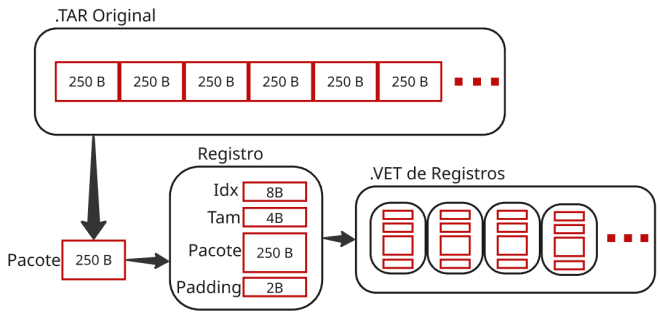

Fig. 1. Extração dos pacotes de 250 *bytes* de um arquivo *.TAR* original.

A figura 2 detalha a estrutura do registro utilizada (RegistroDisco), com destaque para a composição dos campos internos necessários ao correto alinhamento dos dados em memória:

```
typedef struct {
    uint64_t       chave;           // → chave de ordenação
    uint32_t       tamanho;         // → bytes válidos em 'pacote'
    unsigned char  pacote[250];     // → dados úteis
    unsigned char  preenchimento[2]; // → completar 264 bytes
} RegistroDisco;
// sizeof(RegistroDisco) == 264
```

Fig. 2. estrutura do registro utilizada.

Uma vez resolvida essa questão inicial, desenvolvemos o algoritmo de ordenação externa em três fases principais descritas a seguir:

2.1 Criação das Runs Ordenadas com QuickSort (Fase 1)

A etapa inicial da ordenação externa consiste na leitura sequencial do arquivo *..vet*, que armazenam registros de tamanho fixo representando blocos do arquivo original embaralhado. A cada leitura de um bloco que cabe na memória principal, limitada durante a execução, escolhendo quantos registros podem ser lidos por vez, para fins de estudos, a final o espaço em memória principal acessivel nos dias de hoje é bem generoso, os registros são ordenados localmente em RAM e escritos em disco como arquivos temporários, conhecidos como runs ordenadas.

Para realizar a ordenação interna, foi implementadado um algoritmo QuickSort, a fim de permitir maior controle sobre a lógica de comparação dos registros e também para fins didáticos e de análise algorítmica.

A versão implementada utiliza o esquema de partição de Lomuto, conforme descrito por Cormen et al. (2009)[2]. Nesse esquema, o pivô é escolhido como o último elemento da sublista considerada, e a partição é realizada reorganizando os elementos de modo que todos os menores ou iguais ao pivô sejam colocados à sua esquerda, enquanto os maiores ficam à sua direita.

O processo se dá em três passos principais:

1) **Escolha do pivô:** O elemento A[r] (último da sublista A[p..r]) é o pivô;
2) **Partição:** Percorremos a sublista com um índice auxiliar, trocando os elementos para garantir a ordenação relativa ao pivô;
3) **Recursão:** O algoritmo é chamado recursivamente nas duas sublistas A[p..q-1] e A[q+1..r], sendo q a posição final do pivô após a partição.

A complexidade do algoritmo é, em média, O(n log n), sendo eficiente para a ordenação de blocos que cabem em memória. Entretanto, sua eficiência pode degradar para $O(n^2)$ em casos degenerados, como listas já quase ordenadas. Para evitar esse comportamento, seriam possíveis técnicas como mediana de três ou QuickSort híbrido podereiam ser aplicadas, mas foi optado por manter a versão simples para fins de compreensão e controle total da ordenação.

2.2 Intercalação das Runs com *K-Way Merge* e Heap de Mínimos (Fase 2)

Ao término da Fase 1 do processo de ordenação externa, obtemos R arquivos temporários, denominados *runs* (ex: run_000.bin, run_001.bin,

..., run_R-1.bin), cada um contendo registros internamente ordenados. A Fase 2 consiste em intercalar ou mesclar" estas R *runs* em um único arquivo final ordenado. Esta etapa é uma implementação da clássica técnica de "intercalação balanceada de m caminhos" (ou *k-way merge*), como descrito por Ziviani (2010)[1].

Nesta implementação, o parâmetro m (número de caminhos, ou *runs*, a serem intercalados simultaneamente) é denotado por k. Um limite prático MAX_RUNS_ABERTAS (definido como 500 no código) é imposto a k para respeitar o número máximo de descritores de arquivos que um processo pode manter abertos, conforme as limitações do sistema operacional, esse valor pode ser auterado.

Para gerenciar eficientemente a intercalação, selecionando repetidamente o menor registro entre os k registros atualmente na "cabeça" de cada run ativa, utilizamos uma estrutura de dados heap de mínimos.

2.2.1 Estrutura do Nó do Heap

Cada nó no heap de mínimos armazena não apenas o registro em si, mas também a identificação da run de origem. Isso é crucial para que, após um registro ser selecionado como o menor e escrito na saída, o sistema saiba de qual run deve ler o próximo registro para realimentar o heap. A estrutura NoHeap, definida em heap_minimo.h, é apresentada resumidamente na figura 3 abaixo:

```
typedef struct {
    RegistroDisco registro;
    size_t        id_run;
} NoHeap;
```

Fig. 3. estrutura do heap de mínimos.

O campo registro.chave é a chave de ordenação primária, e id_run permite rastrear a origem do dado.

2.2.2 Operações Fundamentais do Heap

A biblioteca heap_minimo.c provê as três operações canônicas para a manipulação do heap, implementado sobre um vetor de NoHeap:

- **void inicializa_heap(NoHeap heap[ ], size_t *t)**: Inicializa o heap, efetivamente esvaziando-o ao configurar seu tamanho (*t) para zero;
- **void inserir_heap(NoHeap heap[ ], size_t *t, NoHeap x)**: Insere um novo nó x no heap e reorganiza os elementos para manter a propriedade de min-heap. Esta operação tem complexidade de tempo O(log k);
- **NoHeap remover_raiz_heap(NoHeap heap[ ], size_t *t)**:Remove e retorna o nó raiz (o menor elemento) do heap, reorganizando os demais para manter a propriedade de min-heap. Esta operação também possui complexidade O(log k).

Internamente, as funções subir_heap(NoHeap h[], size_t i) e descer_heap(NoHeap h[], size_t n, size_t i) são responsáveis pela percolação dos nós, garantindo que a propriedade de min-heap (onde o pai é sempre menor ou igual aos seus filhos) seja mantida após inserções e remoções.

2.2.3 Algoritmo de Intercalação K-Way

O coração da Fase 2 reside na lógica de intercalação, encapsulada em uma função como mesclar_runs_bloco. O processo pode ser decomposto em três etapas principais: inicialização, o loop principal de intercalação, e a finalização. A seguir, detalhamos cada uma dessas etapas com os trechos de código correspondentes.

Primeiramente, o algoritmo aloca memória para as estruturas de dados auxiliares. Um vetor de LeitorRun é criado para gerenciar o acesso de leitura a cada uma das k runs de entrada, e um vetor NoHeap serve como base para o heap de mínimos. O heap é então inicializado, zerando seu contador de tamanho, demonstrado na figura 4

Em seguida, demonstrado na figura 5 o heap é populado com o primeiro registro de cada uma das k runs. Um laço itera sobre os arquivos de entrada, abrindo-os através da estrutura LeitorRun. Se a run não estiver vazia, seu primeiro registro é lido e inserido no heap

```c
LeitorRun *leitores = malloc(k_runs * sizeof *leitores);
NoHeap    *heap_min = malloc(k_runs * sizeof *heap_min);
size_t    heap_size = 0;

inicializa_heap(heap_min, &heap_size);
```

Fig. 4. Alocação de memória e inicialização das estruturas de intercalação.

de mínimos. Cada nó inserido no heap contém o registro e o índice da sua run de origem, informação crucial para a etapa de reabastecimento.

```c
for (size_t i = 0; i < k_runs; i++) {
    inicializa_leitor(&leitores[i], paths_runs_entrada[i]);
    if (leitores[i].tem_reg) {
        inserir_heap(heap_min, &heap_size,
            (NoHeap){ .registro = leitores[i].registro, .id_run = i });
    }
}
```

Fig. 5. Populando o heap de mínimo com os candidatos iniciais.

Após a inicialização, o arquivo de saída é aberto para escrita. O algoritmo então entra em seu loop principal, que executa enquanto houver elementos no heap. Este ciclo "extrai-grava-reabastece" é a essência do k-way merge, sendo demonstrado na figura 6.

1) **Extrair:** A raiz do heap, que por definição é o registro com a menor chave entre todos os candidatos, é removida;
2) **Gravar:** O registro extraído é escrito no arquivo de saída, construindo a run final ordenada;
3) **Reabastecer:** O algoritmo utiliza o id_run armazenado no nó recém-removido para identificar de qual run de entrada ele veio. O próximo registro dessa mesma run é lido e inserido no heap, mantendo o número de elementos no heap constante (até que uma run se esgote).

Quando o loop termina, significa que todas as *runs* de entrada foram completamente processadas e o arquivo de saída está ordenado. A etapa final consiste em um gerenciamento dos recursos, o arquivo de saída é fechado, assim como todos os arquivos de runs de entrada (através da função finaliza_leitor), e a memória alocada dinamicamente para os leitores e para o heap é liberada .

```c
FILE *stream_saida = fopen(path_run_saida, "wb");

while (heap_size > 0) {
    NoHeap menor_no = remover_raiz_heap(heap_min, &heap_size);

    fwrite(&menor_no.registro, sizeof(RegistroDisco), 1, stream_saida);

    size_t id_run_origem = menor_no.id_run;
    avancar_leitor(&leitores[id_run_origem]);

    if (leitores[id_run_origem].tem_reg) {
        inserir_heap(heap_min, &heap_size,
            (NoHeap){ .registro = leitores[id_run_origem].registro,
                      .id_run = id_run_origem });
    }
}
```

Fig. 6. Loop principal da intercalação de runs.

2.2.4 *Análise do Fluxo*

Na inicialização o primeiro registro de cada uma das k runs de entrada é lido e inserido no heap de mínimos. Cada inserção custa O(log k), totalizando O(k log k) para esta etapa. Os LeitorRun são estruturas auxiliares que encapsulam a leitura bufferizada dos arquivos de run.

No Loop Principal o algoritmo entra em um laço que se repete enquanto o heap não estiver vazio. Em cada iteração:

- O menor registro global (a raiz do heap) é extraído (O(log k));
- Este registro é gravado no arquivo de saída;
- O próximo registro da run que originou o elemento extraído é lido;
- Se esta run não estiver exaurida, seu novo registro é inserido no heap (O(log k)). Se N é o número total de registros em todas as runs, cada registro passará pelo heap uma vez (uma extração e uma inserção). Portanto, esta fase tem complexidade O(N log k).

Na finalização, todos os descritores de arquivo são fechados e a memória alocada para as estruturas auxiliares é liberada.

2.2.5 *Intercalação em Múltiplos Passos*

Se o número total de runs geradas na Fase 1, R, excede MAX_RUNS_ABERTAS (o valor de k máximo), o procedimento de intercalação é aplicado em múltiplos passes. Por exemplo, se R = 501 e MAX_RUNS_ABERTAS = 500:

1) **Passo 0:** As primeiras 500 *runs* (run_000.bin a run_499.bin)

são intercaladas, gerando uma run_intermediaria_A.bin, a run restante (run_500.bin) permanece, temos agora 2 runs, run_intermediaria_A.bin e run_500.bin;
2) **Passo 1:** As 2 *runs* resultantes do passo anterior são intercaladas, gerando o arquivo final ordenado.

Cada passo de intercalação reduz significativamente o número de arquivos a serem processados, até que apenas um arquivo final, completamente ordenado, reste, o número de passos é aproximadamente ceil(log_k R), ou seja o teto de (log_k R).

2.2.6 Análise de Complexidade e Consumo de Recursos

Para um único passo de intercalação de k *runs* contendo um total de N registros:

- **Tempo de CPU:** A complexidade dominante é O(N log k). Dado que k é limitado por MAX_RUNS_ABERTAS (500), o fator log k é relativamente pequeno e constante na prática ($\log_2 500 \approx 8.96$);
- **Memória Principal (RAM):** O consumo de memória é primordialmente devido ao heap, que armazena k registros (NoHeap), mais buffers de I/O para cada LeitorRun e para o arquivo de saída. A ordem de grandeza é O(k). Com k=500 e registros de, por exemplo, 264 bytes (como RegistroDisco poderia ser se sizeof(uint64_t) para chave + 256 bytes de outros dados), o heap em si consumiria 500 * (264 + sizeof(size_t)) bytes, algo em torno de 132 Kb a 136 Kb, mais os buffers;
- **Descritores de Arquivo:** São necessários k descritores para as runs de entrada e 1 para a run de saída, totalizando k+1. Com$k \leq 500$, o uso fica bem abaixo do limite típico de 1024 descritores por processo (ulimit -n);
- **Operações de I/O em Disco:** Durante um passe de k-way merge, cada registro é lido uma vez da sua respectiva run de entrada e escrito uma vez na run de saída. O número total de leituras e escritas em disco é proporcional a 2N (ignorando I/O para metadados). O objetivo principal da ordenação externa é minimizar o número total de passes sobre os dados, pois o I/O em disco é a operação mais custosa..

Em resumo, a técnica de k-way merge utilizando um heap de mínimos, como implementada, permite a fusão eficiente de um grande número de runs ordenadas, gerenciando dezenas ou centenas de gigabytes de dados com um consumo de memória RAM relativamente modesto e respeitando os limites de recursos do sistema operacional. Esta abordagem é fundamental para a viabilidade da ordenação externa de grandes volumes de dados.

2.3 Reconstrução do Arquivo *.TAR* (Fase 3)

Com o arquivo final ordenado obtido na fase anterior, realizamos a reconstrução do arquivo original no formato TAR. Os pacotes de dados dos registros ordenados são concatenados sequencialmente, respeitando rigorosamente os tamanhos indicados no campo tam de cada registro.

Para garantir a conformidade do arquivo reconstruído com as especificações *.TAR*, inserimos os blocos adicionais exigidos pelo formato: blocos preenchidos com zeros para alinhar o conteúdo total a múltiplos de 512 bytes, além de dois blocos finais de 512 bytes preenchidos completamente com zeros que indicam o término do arquivo *.TAR*.

2.4 Problemas Iniciais com os Arquivos Originais de Teste e Correção dos Mesmos

Um desafio importante enfrentado neste trabalho foi a ausência de acesso aos arquivos *.TAR* originais que haviam sido usados para gerar os arquivos *.vet* fornecidos (misturado-1234.vet, misturado-actg.vet e misturado-grande.vet). Isso levou a realizar análises cuidadosas dos arquivos reconstruídos, especialmente com o auxílio de comandos do utilitário *TAR* (tar -tf) e ferramentas de inspeção binária (hexdump).

Essas análises revelaram que, exceto para o arquivo misturado-1234.vet, os pacotes de dados estavam deslocados em relação às posições originais do arquivo *.TAR* original. Tal desalinhamento causava perda de integridade no

arquivo reconstruído e impedia que as ferramentas padrão reconhecessem corretamente esses arquivos. Uma hipótese levantada foi que a maneira como os dados haviam sido inicialmente fatiados e inseridos nos registros, pode não ter respeitado exatamente os blocos originais de 250 bytes do arquivo *.TAR* original.

Diante desse cenário, foi desenvolvida um algoritmo para gerar arquivos de teste corretamente alinhados desde o princípio, garantindo que cada pacote de 250 bytes do arquivo TAR fosse precisamente mantido intacto e sequencial ao ser inserido nos registros. Com isso, garantimos que após a ordenação externa e reconstrução, os arquivos finais fossem válidos e plenamente reconhecidos pelas ferramentas padrão *TAR*.

3 EXPERIMENTOS

Nesta secção detalhamos a configuração do ambiente de teste e o procedimento adotado para a coleta de métricas de desempenho do algoritmo de ordenação externa proposto, os dados brutos são apresentados para análise posterior.

3.1 Plataforma de Execuçãos

Os experimentos foram conduzidos em um ambiente de máquina virtual (VM) para garantir um ambiente de teste isolado e consistente, as especificações estão detalhadas na Tabela 1.

TABLE 1
Especificações do ambiente do experimento.

Componente	Especificação
Hardware	VMware Virtual Platform (guest)
CPU	13th Gen Intel Core i5-13420H (6 núcleos lógicos para a VM)
Memória RAM	3.8 Gb disponíveis para a VM
Armazenamento	SSD M.2 no host (disco virtual para o guest)
SO Convidado	Zorin OS 17.3 Core – 64 bits (base Ubuntu 22.04)
Kernel	Linux 5.15-generic
Compilador	gcc -std=c11 -O2 -Iinclude -Wall -Wextra -pedantic -lrt
Medição	Script coleta_metricas.sh e biblioteca monitor.c

3.2 Arquivos de Teste

Foram utilizados dois arquivos de entrada com características distintas para avaliar o algoritmo:

O arquivo grande.vet teve os seus registos previamente embaralhados para assegurar um cenário de ordenação genérico e desafiador.

TABLE 2
Arquivos de Teste.

Arquivos	Tamanho(Bytes)	Registos(264 B)	Conteúdo Original
misturado-1234.vet	40.960	155	arquivo .txt
grande.vet	6.439.165.656	24.390.779	arquivo .txt, arquivo pdf, 14 arquivos .mp4

3.3 Procedimento de Medição

Para cada combinação de arquivos de teste e valor do parâmetro max_blocos,representando o número máximo de registos que o buffer em RAM pode conter para a ordenação interna na Fase 1, o programa de ordenação externa foi executado 3 vezes, o script coleta_metricas.sh foi utilizado para automatizar as execuções e a coleta das seguintes métricas, cujas médias são:

- Número de *runs* geradas na Fase 1;
- Número de passagens de intercalação na Fase 2;
- Tempo de execução (relógio) de cada uma das três fases (medição interna via monitor.c);
- Tempo total de execução (relógio, medido pelo /usr/bin/time);
- Volume de I/O Lógico (Leitura e Escrita em MB, medição interna via monitor.c);
- Pico de Descritores de Arquivo (DA) abertos simultaneamente (medição interna via monitor.c).

3.4 Apresentação dos Resultados

Os resultados médios das três execuções para cada cenário experimental são apresentados nas tabelas seguintes.

A Tabela 3 foca nos parâmetros que definem a estrutura e os limites operacionais da ordenação, como o número de runs iniciais na fase 1, as passagens de intercalação na fase 2 e o uso de descritores de arquivos (DA).

TABLE 3
Configuração estrutural da ordenação por cenário.

Arquivos	max_blocos	N Runs (F1)	Passes (F2)	Máx. DA
misturado-1234.vet	50	4	1	8
grande.vet	50	487816	3	504
grande.vet	5000	4879	2	504
grande.vet	50000	488	1	492
grande.vet	500000	49	1	53

A Tabela 4 e a Figura 7 apresentam as métricas de desempenho temporal, detalhando

o tempo consumido em cada fase do algoritmo e o tempo total de execução. Estes dados são cruciais para identificar os gargalos de processamento.

TABLE 4
Desempenho temporal do algoritmo por fase e total.

Arquivos	max_blocos	Fase 1 (s)	Fase 2 (s)	Fase 3 (s)	Total (s)
misturado-1234.vet	50	0.0007	0.0006	0.0003	0.002
grande.vet	50	33.3086	193.9529	22.2659	249.53
grande.vet	5000	15.4349	53.4621	18.8441	87.74
grande.vet	50000	18.0822	30.0033	18.3689	66.45
grande.vet	500000	22.9163	24.2929	18.7380	65.95

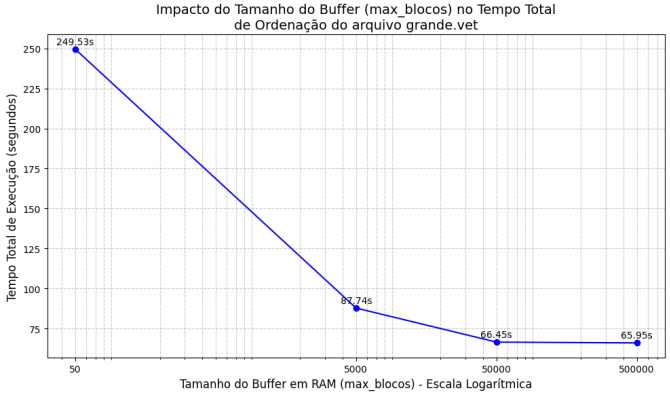

Fig. 7. Impacto do Tamanho do Buffer no Tempo Total de Ordenação.

A Tabela 5 e a Figura 8 detalham o volume de Input/Output (I/O) Lógico, medido em Gigabytes (GB), que o algoritmo processou, esta métrica reflete o total de dados lidos e escritos internamente pelo programa.

TABLE 5
Desempenho temporal do algoritmo por fase e total.

Arquivos	max_blocos	I/O Lido (GB)	I/O Escrito (GB)
misturado-1234.vet	50	0.00011	0.00011
grande.vet	50	29.98	29.67
grande.vet	5000	23.99	23.67
grande.vet	50000	17.99	17.67
grande.vet	500000	17.99	17.67

4 Discussão dos Resultados

A análise dos dados experimentais apresentados permite uma compreensão aprofundada do comportamento e da eficiência do algoritmo de ordenação externa implementado.

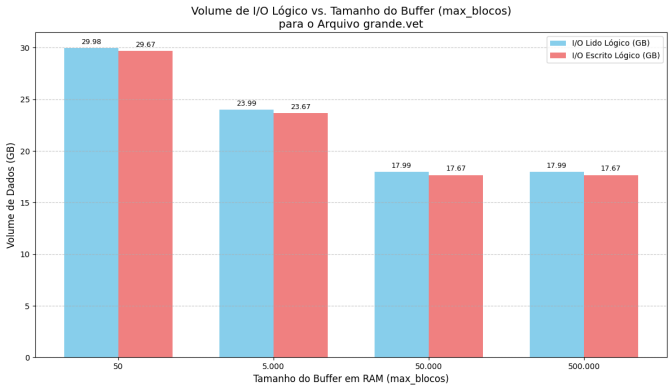

Fig. 8. Volume de I/O Lógico vs. Tamanho do Buffer.

4.1 Validação da Correção e Precisão das Métricas

Os testes iniciais com o arquivo misturado-1234.vet (max_blocos=50) serviram como uma validação fundamental das métricas de I/O Lógico (0.12 MB lido/escrito) e Máximo de Descritores Abertos (8 DAs) corresponderam exatamente aos valores teoricamente esperados para a intercalação de 4 runs numa única passagem. A precisão destas métricas, obtidas através da biblioteca de monitorização interna (monitor.c), contrasta com os valores de I/O reportados pelo /usr/bin/time, que são significativamente subestimados devido ao cache de disco do sistema operacional.

4.2 Impacto do Tamanho do Buffer (max_blocos) no Desempenho Global

A Tabela 4 e a Figura 7 demonstram que o tamanho do buffer de memória alocado na Fase 1 (max_blocos) é o fator predominante que influencia o desempenho global da ordenação externa, ao aumentar max_blocos de 50 para 5.000 (um fator de 100), o tempo total de execução para o Arquivo grande.vet caiu drasticamente de ≈250 segundos (4m 9s) para ≈88 segundos (1m 28s), uma redução de aproximadamente 65%. Um novo aumento para max_blocos=50.000 reduziu o tempo para ≈66 segundos, contudo, o ganho de desempenho tornou-se minimo ao passar de 50.000 para 500.000 max_blocos (de ≈66.45s para ≈ 65.95s), esta observação sugere a existência de um ponto de retornos decrescentes, onde o

benefício de aumentar a memória para as runs iniciais se estabiliza.

4.3 Análise Detalhada por Fase

O tempo da Fase 1 apresentou uma tendência de diminuição ao passar de max_blocos=50 (≈33s) para max_blocos=5000 (≈15s). Isto ocorre porque, embora o volume total de dados lidos e escritos seja o mesmo, realizar menos operações de I/O de blocos maiores é geralmente mais eficiente do que muitas operações de I/O de blocos pequenos, devido ao overhead por operação. Curiosamente, houve um ligeiro aumento no tempo da Fase 1 ao passar de max_blocos=5000 para valores maiores (50k e 500k), isto pode ser atribuído ao overhead de gestão e ordenação (Quicksort) de blocos de memória consideravelmente maiores (ex: 500.000 registos ≈ 126 MB), que podem não se beneficiar tanto da localidade de cache da CPU como blocos de tamanho moderado, no entanto, como o número total de iterações da Fase 1 diminui drasticamente (de 487.816 para 49), este efeito no tempo da Fase 1 é menos pronunciado que o ganho na Fase 2.

A Fase 2 é a fase onde o impacto de max_blocos é mais evidente. Aumentar max_blocos reduz o número de runs (R) geradas na Fase 1, consequentemente, o número de passagens de intercalação PassesMerge = ceil(log_k R), onde k é MAX_RUNS_ABERTAS é igual a 500 diminui.

Com max_blocos=50, geraram-se 487.816 runs, necessitando de 3 passagens de intercalação, resultando num tempo de ≈194 segundos, com max_blocos=5000, geraram-se 4.879 runs, necessitando de 2 passagens, reduzindo o tempo para ≈53 segundos, com max_blocos=50000 e max_blocos=500000, geraram-se 488 e 49 runs, respetivamente. Em ambos os casos, como R ¡= MAX_RUNS_ABERTAS, apenas 1 passagem de intercalação foi necessária. O tempo da Fase 2 caiu para ≈30s e ≈24s, respetivamente. A melhoria de 30s para 24s, mesmo com uma única passagem, deve-se à eficiência de operar um heap de mínimo com menos elementos (49 elementos vs. 488 elementos).

O tempo da Fase 3 permaneceu relativamente estável (entre ≈18s e ≈22s) em todos os cenários para o grande.vet, isto era esperado, pois esta fase envolve uma única leitura sequencial do arquivo ordenado e uma única escrita sequencial do arquivo .tar final, operações cujo desempenho é menos sensível ao max_blocos da Fase 1.

4.4 Análise do Volume de I/O Lógico

O volume de I/O Lógico é o valor que representa a quantidade total de dados que o algoritmo efetivamente processou (lendo e escrevendo) através das chamadas da biblioteca C (fread, fwrite), independentemente de a operação ter sido servida pelo disco físico ou pelo cache do sistema operacional. Esta métrica é fundamental para avaliar a eficiência intrínseca do algoritmo, algo que ferramentas externas como o /usr/bin/time não conseguem capturar com precisão.

O volume de I/O lógico do algoritmo pode ser modelado teoricamente, para cada uma das três fases, o conjunto de dados completo é processado, envolvendo uma leitura completa e uma escrita completa, na Fase 2, este processo repete-se para cada passagem de intercalação. Assim, a quantidade total de dados lidos/escritos pode ser calculada como:

$$\text{I/O Lógico} = \text{TamanhoTotal} \times (1\ [\text{Fase 1}] + \text{PassesMerge}\ [\text{Fase 2}] + 1\ [\text{Fase 3}])$$

Onde TamanhoTotal é o tamanho do Arquivo de entrada e PassesMerge é o número de passagens de intercalação na Fase 2.

Validação do Modelo com os Dados Experimentais: Os dados recolhidos validam este modelo com notável precisão:

4.4.1 Validação do Modelo com os Dados Experimentais

Para max_blocos=50, o algoritmo necessitou de 3 passagens de intercalação:

- I/O Lógico Esperado: ≈6.1 GB × (1+3+1) ≈30.5 GB;
- I/O Lógico Medido: ≈30.7 GB lidos.

Para max_blocos=5000, necessitou de 2 passagens de intercalação:

- I/O Lógico Esperado: ≈6.1 GB × (1+2+1) ≈24.4 GB;
- I/O Lógico Medido: ≈24.5 GB lidos.

Para max_blocos $\geq$ 50000, necessitou de apenas 1 passagem de intercalação:

- I/O Lógico Esperado: ≈6.1 GB × (1+1+1) ≈18.3 GB;
- I/O Lógico Medido: ≈18.4 GB lidos.

A correspondência quase perfeita entre os valores teóricos e os medidos confirma a corretude do entendimento do fluxo de dados e a precisão da biblioteca de monitorização.

A importância de medir o I/O lógico torna-se evidente quando o comparamos com o I/O físico reportado pelo /usr/bin/time, no cenário de max_blocos=50, o algoritmo manipulou logicamente mais de 60 GB de dados, somando leitura e escrita, mas o time reportou apenas ≈135 MB de atividade de I/O combinada.

Esta enorme diferença demonstra a eficácia do cache de disco do sistema operacional, o kernel manteve de forma inteligente as runs mais acedidas na memória RAM, servindo os pedidos de leitura do nosso programa diretamente da RAM e evitando o acesso lento ao disco físico. Embora este cache tenha melhorado drasticamente o tempo de relógio de execução, o volume de trabalho lógico do algoritmo, a quantidade de dados que teve de ser copiada e processada, permaneceu em 60 GB.

4.5 Retornos Decrescentes e Ponto Ótimo

A passagem de max_blocos=50000 para max_blocos=500000 resultou numa melhoria mínima no tempo total de 66.45s para 65.95s, embora o tempo da Fase 2 tenha melhorado ligeiramente, devido a um heap menor, o tempo da Fase 1 aumentou um pouco, e o I/O lógico total permaneceu o mesmo, pois a Fase 2 já estava a ser executada numa única passagem. Isto indica que, para este conjunto de dados e máquina, aumentar o buffer de memória para além do necessário para conseguir uma intercalação de Fase 2 numa única passagem, ou poucas passagens com k elementos no heap, traz pouco benefício e pode até introduzir pequenos overheads. O "ponto ótimo" parece estar em torno de max_blocos que minimiza o número de passagens da Fase 2 sem tornar os blocos da Fase 1 excessivamente grandes para a eficiência do Quicksort e gestão de cache da CPU.

4.6 Conclusões Gerais da Experimentação

Os experimentos demonstraram de forma conclusiva que a implementação do algoritmo de ordenação externa é funcional robusta, o desempenho é fortemente dependente da quantidade de memória alocada para as runs iniciais (max_blocos), o que influencia diretamente o número de passagens de I/O da fase de intercalação.

A utilização de uma biblioteca de monitorização interna forneceu métricas precisas sobre o comportamento do algoritmo, validando a teoria e revelando o impacto do cache do sistema operacional e que existe um equilíbrio no dimensionamento do buffer de memória, onde aumentos para além de um certo ponto, neste caso, o suficiente para reduzir a Fase 2 a uma única passagem de intercalação, oferecem retornos decrescentes em termos de tempo total de execução.

Estes resultados reforçam a validade da abordagem de k-way merge com heap de mínimo para a ordenação eficiente de grandes volumes de dados, mesmo em ambientes com recursos de memória relativamente modestos.

REFERENCES

[1] N. Ziviani, *Projeto de Algoritmos: com implementações em Pascal e C*, 3. ed. Rio de Janeiro: Elsevier, 2010.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, e C. Stein, *Introduction to Algorithms*, 3. ed. Cambridge, MA: MIT Press, 2009.

[3] Free Software Foundation, "Stack Protector — GCC documentation," 2024. [Online]. Available: https://gcc.gnu.org/onlinedocs/ (acesso em: mai. 2025).