



# Ordenação Externa

---

Profa. Graça Nunes



# Ordenação Externa

---

- Ordenar arquivos de tamanho maior que a memória interna disponível
- Algoritmos devem diminuir o número de acessos às unidades de memória externa
  - Custo relacionado à transferência de dados
- Dados armazenados como um arquivo sequencial
- Métodos dependentes do estado atual da tecnologia
  - Tipos de memória externa torna os métodos dependentes de vários parâmetros



# Métodos Gerais

---

- Ordenação por **Intercalação**
- Intercalar significa combinar dois ou mais blocos ordenados em um único bloco, maior, ordenado.



# Exemplo: intercalar (*merge*) 2 listas

<b>Lista1</b>	<b>Lista2</b>
Adams	Adams
Carter	Anderson
Chin	Andrews
Davis	Bech
Foster	Rosewald
Garwich	Schmidt
Rosewald	Thayer
Turner	Walker
	Willis



# Algoritmo de intercalação (*merge*)

---

- Entrada: 2 listas ordenadas
- Lê um nome de cada lista e compara-os
  - Se ambos são iguais, copia o nome para a saída e avança para os próximos nomes das 2 listas
  - Se o nome da Lista1 é menor, ele é copiado para a saída e avança-se na Lista1
  - Se o nome da Lista1 é maior, copia o nome da Lista2 para a saída e avança-se na Lista2

# PROGRAM: merge

```
call initialize()
call input() to get NAME_1 from LIST_1 and NAME_2 from LIST_2
while(MORE_NAMES_EXIST)
    if (NAME_1 < NAME_2)
        write NAME_1 to OUT_FILE
        call input() to get NAME_1 from LIST_1
    else if (NAME_1 > NAME_2)
        write NAME_2 to OUT_FILE
        call input() to get NAME_2 from LIST_2
    else /* match – names are the same */
        write NAME_1 to OUT_FILE
        call input to get NAME_1 from LIST_1
        call input to get NAME_2 from LIST_2
finish_up()
```



# Operações Cosequenciais

---

- *Merging* é exemplo de uma operação cosequencial
  - Envolve o **processamento coordenado** (simultâneo) de **duas ou mais listas de entrada sequenciais**, de modo a produzir uma única lista como saída
- *Matching* (achar valores comuns entre as duas listas) é outro exemplo de operação cosequencial
  - Como funcionaria o *matching*?



# Ordenação Externa

---

- Estratégia geral dos métodos de ordenação externa:
  1. Quebre o arquivo em blocos que caibam na memória disponível
  2. Ordene cada bloco na RAM
  3. Intercale os blocos ordenados, gerando o arquivo ordenado.





# Desempenho

---

- Os algoritmos para ordenação externa devem reduzir o número de passadas sobre o arquivo (*por que?*)
- Uma boa medida de complexidade é o número de vezes que um item é lido ou escrito da/na memória externa
- Bons métodos geralmente envolvem, no total, menos do que 10 passadas sobre o arquivo



# Intercalação como base da ordenação externa

---

## Multiway Merging

- Geração de uma lista ordenada a partir de outras, menores, ordenadas, por intercalação
- **K-way Merge (intercalação em k vias):**  
intercalar k blocos ordenados
- Solução para ordenar arquivos muito grandes
  - Ordenam-se partes menores do arquivo
  - Intercalam-se as partes ordenadas

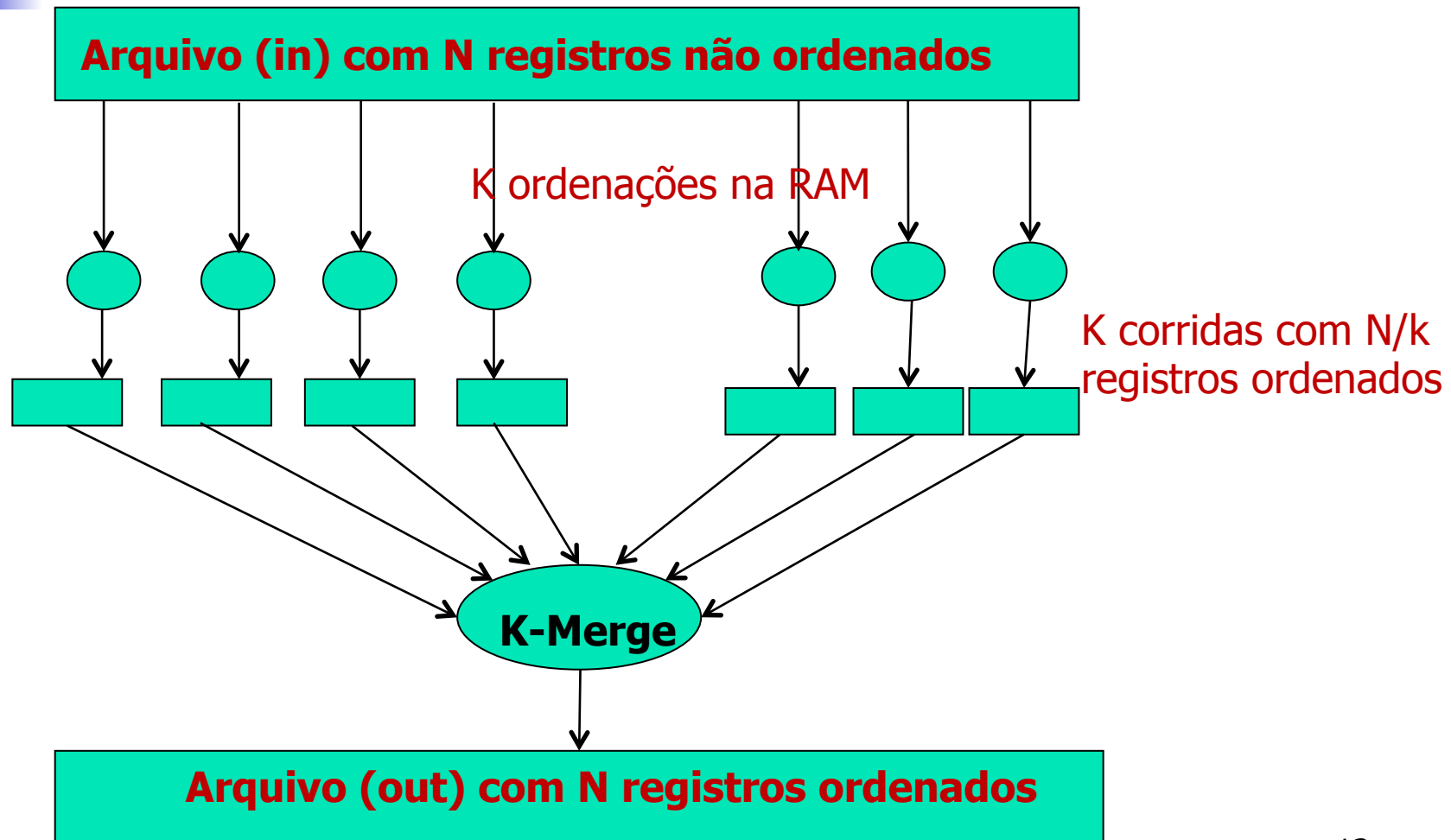


# Ordenação por Intercalação de Arquivos em Disco

---

- k-Way Mergesort
- Ao invés de considerar os registros individualmente, podemos considerar **blocos de registros ordenados** (corridas, ou *runs*)
  - Para minimizar os seeks
- Método envolve 2 fases: geração das corridas (*runs*, blocos ordenados de dados), e intercalação

# Esquema Geral do K-way Mergesort





# Intercalação em k-vias

---

- Esta solução
  - Pode **ordenar arquivos realmente grandes**
  - Geração das corridas envolve apenas acesso sequencial aos arquivos
  - A leitura das corridas e a escrita final também são sequenciais
  - Aplicável também a arquivos mantidos em fita, já que E/S é sequencial



# k-Way Mergesort

---

- **Fase 1: Geração das Corridas**

- Segmentos do arquivo (corridas) são ordenados em memória RAM, usando algum método eficiente de ordenação interna (p.ex., quicksort ou heapsort), e gravados em disco\*
- Corridas vão sendo gravadas a medida em que são geradas

\* Veja na literatura como fazer ordenação e gravação das corridas em paralelo, usando heapsort



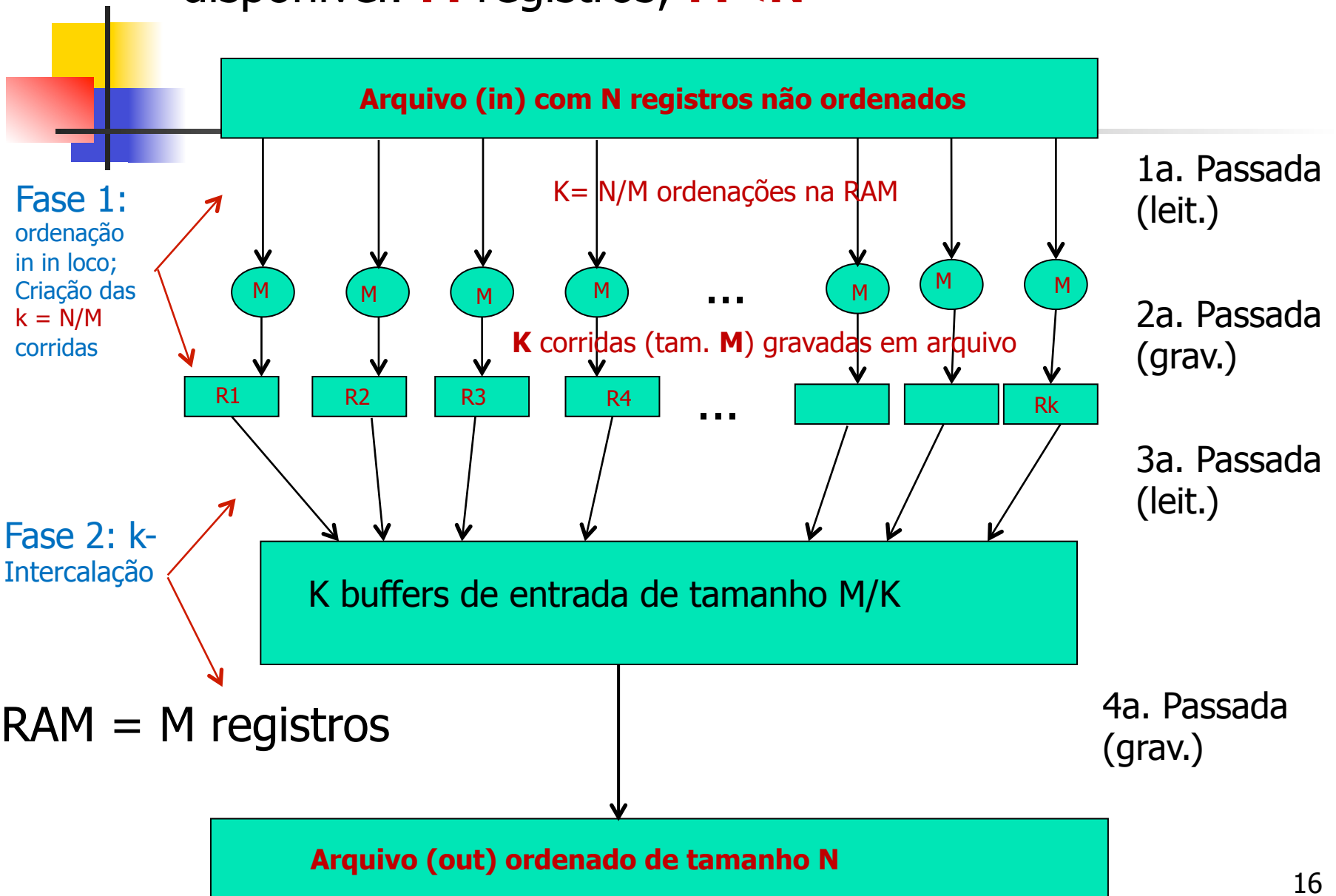
# k-Way Mergesort

---

- **Fase 2: Intercalação**

- As k corridas geradas na fase anterior são intercaladas (k-way), formando o arquivo ordenado, que vai sendo gravado no disco
  - Envolvem buffers de entrada – k chaves das corridas
  - E buffer(s) de saída – gravados no disco assim que completados
    - 1 buffer ou 2 buffers de saída – o que é melhor?

Suponha: Arquivo com **N** registros; Memória disponível: **M** registros; **M < N**



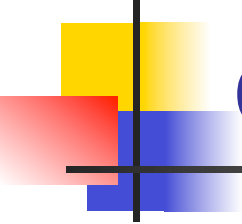




# Complexidade do K-way Mergesort

---

- Fase 1: Criação das Corridas:
  - 1 leitura sequencial do arquivo (N registros)
  - $K=N/M$  gravações sequenciais de M registros
  
- Fase 2: Intercalação:
  - Toda a RAM é usada para buffers de entrada: abrigando K blocos de registros
  - No total,  $K*K$  seeks para ler todos os registros para a intercalação
  - Gravação dos N registros conforme ocorre a k-intercalação.



# Exemplo: cálculo do tempo gasto na ordenação

---

- Apenas para ter um *benchmark*; não levar os números a sério!
- Supondo
  - Arquivo com 80 MB, com 800.000 registros de 100 bytes, e cada corrida com 1 MB
  - 1MB = 10.000 registros
  - Arquivo armazenado em áreas contíguas do disco (*extents*), *extents* alocados em mais de uma trilha, de tal modo que um único *rotational delay* é necessário para cada acesso
  - Características do disco
    - tempo médio para seek: 18 ms
    - atraso rotacional: 8.3 ms
    - taxa de transferência: 1229 bytes/ms
    - tamanho da trilha: 20.000 bytes



# Exemplo

---

- **Quatro passos** a serem considerados

Fase 1:  
Criação  
corridas

- Leitura dos registros, do disco para a memória, para criar as corridas
- Escrita das corridas ordenadas para o disco

Fase 2:  
Intercalação

- Leitura das corridas para intercalação
- Escrita do arquivo final em disco



# Leitura dos registros e criação das corridas

---

- Lê-se 1MB de cada vez, para produzir corridas de 1 MB
- Serão 80 leituras, para formar as 80 corridas iniciais – *80-way Merge*
- O tempo de leitura de cada corrida inclui o tempo de acesso a cada bloco (seek + *rotational delay*) somado ao tempo necessário para transferir cada bloco



# Leitura dos registros e criação das corridas

---

seek = 18ms, rot. delay = 8.3ms, total 26.3ms

Tempo total para a fase de ordenação:

$80 * (\text{tempo de acesso a uma corrida}) + \text{tempo de transferência de 80MB}$

Acesso:  $80 * (\text{seek} + \text{rot. delay} = 26.3\text{ms}) = 2\text{s}$

Transferência: 80 MB a 1.229 bytes/ms = 65s

Total: 67s



# Escrita das corridas ordenadas no disco

---

- Idem à leitura!

Serão necessários outros 67s



# Leitura das corridas do disco para a memória (para intercalação)

---

- 1MB de MEMÓRIA para armazenar 80 buffers de entrada
  - portanto, cada buffer armazena 1/80 de uma corrida (12.500 bytes) → cada corrida deve ser acessada 80 vezes para ser lida por completo
- 80 acessos para cada corrida X 80 corridas
  - 6.400 seeks
- considerando acesso = seek + rot. delay
  - $26.3\text{ms} \times 6.400 = 168\text{s}$
- Tempo para transferir 80 MB = 65s

# Efeito da buferização sobre o número de seeks quando a RAM é usada totalmente como buffer de entrada

1a. Corrida = dividida em 80 buffers (80 seeks)



2a. Corrida = dividida em 80 buffers (80 seeks)



3a. Corrida = dividida em 80 buffers (80 seeks)



...

80a. Corrida = dividida em 80 buffers (80 seeks)



800.000  
registros  
ordenados





# Escrita do arquivo final em disco

---

- Precisamos saber o tamanho dos *buffers* de saída
- Nos passos 1 e 2, a MEMÓRIA funcionou como *buffer de entrada*, mas agora a MEMÓRIA está armazenando os dados a serem intercalados
- Para simplificar, assumimos que seja possível alocar 2 *buffers* de saída de 20.000 bytes, para escrita
  - dois para permitir *double buffering* (enquanto um é gravado no disco, o outro é preenchido com novos registros ordenados); 20.000 porque é o tamanho da trilha no nosso disco hipotético



# Escrita do arquivo final em disco

---

- Com *buffers* de 20.000 bytes, precisaremos de  $80.000.000 \text{ bytes} / 20.000 \text{ bytes} = 4.000 \text{ seeks}$
- Como tempo de seek+rot.delay = 23.6ms por seek, 4.000 seeks usam  $4.000 \times 26.3$ , e o total de 105s.
- Tempo de transferência é 65s



# Tempo total

---

- leitura dos registros para a memória para a criação de corridas: 67s
- escrita das corridas ordenadas para o disco: 67s
- leitura das corridas para intercalação:  $168 + 65 = 233$  s
- escrita do arquivo final em disco:  $105 + 65 = 170$  s
- **tempo total do Mergesort = 537 s (~9 minutos)**
  - Tempo dominado pelo acesso ao disco, e não pelo tempo de intercalação na RAM



# Comparação

---

- Quanto tempo levaria um método que não usa intercalação (como o keysort, p.ex.)?
  - Se for necessário um *seek* separado para cada registro, i.e, 800.000 seeks a 26.3ms cada, o resultado seria um tempo total (só para *seek*) = 21.040s = **5 horas e 40s** !



# Quando usar Mergesort

---

- Quando a ordenação deve ser feita só raramente
- Quando o arquivo não for grande demais
- Quando o Mergesort passa a ser problemático?



# Ordenação de um arquivo com 8.000.000 de registros

---

- Análise - arquivo de 800 MB
- O arquivo aumenta, mas a memória não!
  - Em vez de 80 corridas iniciais, teremos 800
  - Portanto, seria necessária uma intercalação em 800-vias ( $k=800$ ) no mesmo 1 MB de memória, o que implica em que a memória seja dividida em 800 buffers na fase de intercalação



# Ordenação de um arquivo com 8.000.000 de registros

---

- Cada buffer comporta 1/800 de uma corrida, e cada corrida é acessada 800 vezes
- $800 \text{ corridas} \times 800 \text{ seeks/corrída} = 640.000$  seeks no total
- O **tempo** total agora é **superior a 5 horas e 19 minutos**, aproximadamente 36 vezes maior do que o arquivo de 80 MB (que é apenas 10 vezes menor do que este)

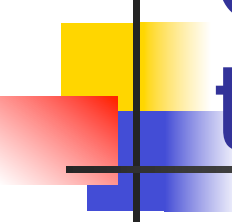


# Ordenação de um arquivo com 8.000.000 de registros

---

- **Definitivamente:** necessário **diminuir o tempo gasto** obtendo dados na fase de intercalação





# O custo de aumentar o tamanho do arquivo

---

- A **grande diferença de tempo** na intercalação dos dois arquivos (de 80 e 800 MB) é consequência da diferença nos **tempos de acesso às corridas** (seek e rotational delay) para intercalá-las
- Em geral, para uma intercalação em K-vias de K corridas, em que cada corrida é do tamanho da MEMÓRIA disponível, o tamanho do buffer para cada uma das corridas é de:  
$$(1/K) \times \text{tamanho da MEMÓRIA} = (1/K) \times \text{tamanho de cada corrida}$$
$$(M/K)$$



# Complexidade do Mergesort

---

- Como temos  $K$  corridas gastando  $K$  seeks cada uma, a operação de intercalação requer  $K^2$  seeks
- Medido em termos de seeks, o Mergesort é  $O(K^2)$
- Como  $K$  é diretamente proporcional à  $N$  ( $K=N/M$ ), o Mergesort é  $O(N^2)$ , em termos de seeks



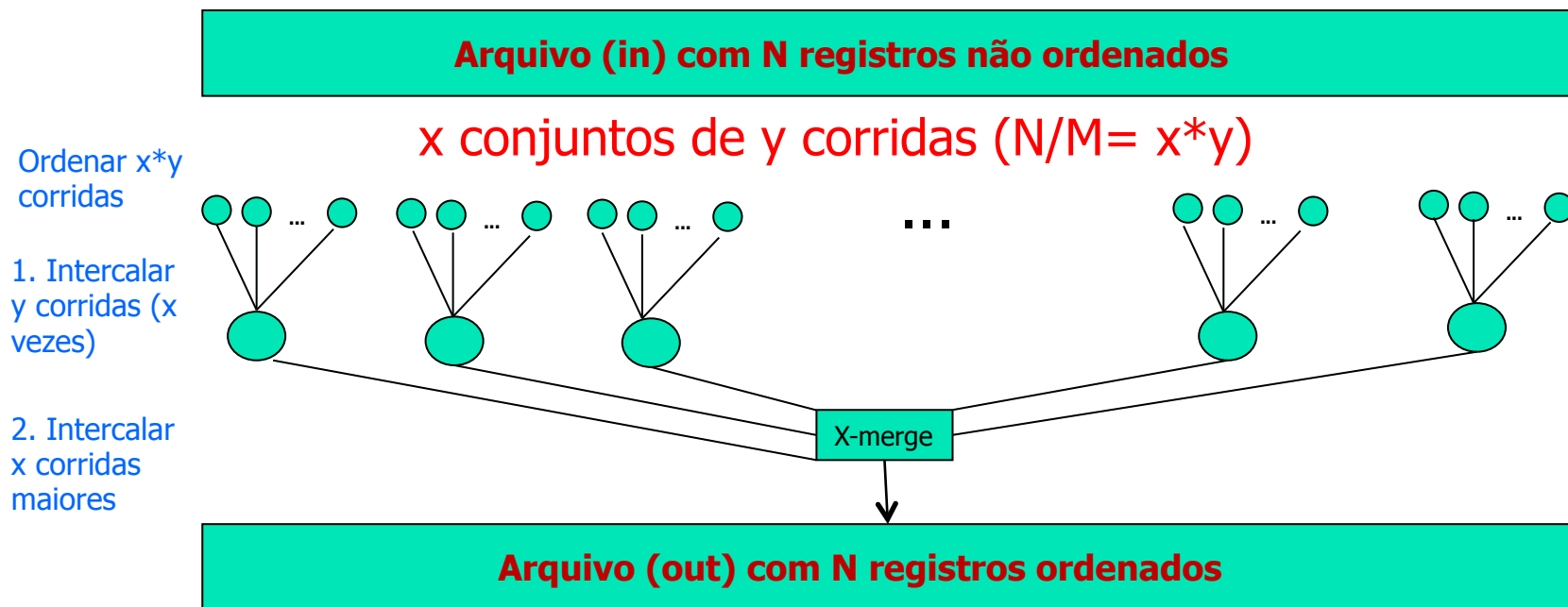
# Maneiras de reduzir esse tempo

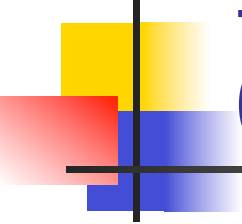
---

1. **usar mais hardware** (disk drives, MEMÓRIA, canais de I/O)
2. realizar a **intercalação em mais de uma etapa**, o que reduz a ordem de cada intercalação, aumentando o tamanho do buffer para cada corrida, portanto, transferindo mais registros com um único seek.
3. **aumentar o tamanho das corridas** iniciais (veja na bibliografia – *Replacement Selection*)
4. **realizar I/O simultâneo** à intercalação

# Redução do número de seeks: Intercalação em Múltiplos Passos (*Multistep Merging*)

- ao invés de intercalar todas as corridas simultaneamente, o grupo original é dividido em subgrupos menores ( $K = x * y$ )





# Redução do número de seeks: Intercalação em Múltiplos Passos (*Multistep Merging*)

---

- intercalação é feita para cada sub-grupo
- para cada sub-grupo, um espaço maior é alocado para cada corrida, portanto um número menor de seeks é necessário
- uma vez completadas todas as intercalações pequenas, o segundo passo completa a intercalação de todas as corridas



# Intercalação em Múltiplos Passos

---

- No exemplo do arquivo com  $N=800$  MB tínhamos 800 corridas com 10.000 registros cada ( $M=1\text{MB}$ ). Para esse arquivo, a intercalação múltipla poderia ser realizada em dois passos:
  - primeiro, a intercalação de  $x=25$  conjuntos de  $y=32$  corridas cada ( $x*y=800$ )
  - depois, uma intercalação em 25-vias



# Intercalação em Múltiplos Passos

---

- Passo único visto anteriormente exige 640.000 seeks
- Para a intercalação em 2 passos, temos, no passo 1:
  - Cada intercalação em 32-vias aloca buffers que podem conter  $1/32$  de uma corrida. Então, serão realizados  $32 \times 32 = 1024$  seeks
  - Então, 25 vezes a intercalação em 32-vias exige  $25 \times 1024 = 25.600$  seeks
  - Cada corrida resultante tem  $32 \times 10.000 = 320.000$  registros = 32 MB



# Intercalação em Múltiplos Passos

---

- No passo 2, cada uma das 25 corridas de 32 MB pode alocar 1/25 do buffer
  - portanto, cada buffer aloca 400 registros, ou seja, 1/800 corrida. Então, esse passo exige 800 seeks por corrida, num total de  $25 \times 800 = 20.000$  seeks
- Total de seeks nos dois passos:  $25.600 + 20.000 = 45.600$  (contra 640.000 anteriores)





# E o tempo total de intercalação?

---

- Nesse caso, cada registro é transmitido 4 vezes, em vez de duas. Portanto, gastamos mais 651s em tempo de transmissão
- Ainda, cada registro é escrito duas vezes: mais 40.000 seeks (assumindo 2 buffers de 20.000 bytes cada)
- Somando tudo isso, o tempo total de intercalação = 5.907s ~ **1 hora 38 min**
  - A intercalação em 800 vias consumia ~5 horas...