1

Técnicas de Concepção de Algoritmos (1ª parte): divisão e conquista

R. Rossetti, A. P. Rocha, L. Ferreira, J. P. Fernandes, F. Ramos, G. Leão
CAL, MIEIC, FEUP

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP

2

Divisão e Conquista (*divide and conquer*)

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP

3

Divisão e conquista

- Dividir o problema em subproblemas que são instâncias mais pequenos do mesmo problema.
- Conquistar os subproblemas resolvendo-os recursivamente; se os subproblemas forem suficientemente pequenos, resolvem-se diretamente.
- Combinar as soluções dos subproblemas para obter a solução do problema original.

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

4

Notas

- Subproblemas devem ser disjuntos (senão, usar programação dinâmica).
- Dividir em subproblemas de dimensão similar para maior eficiência.
- Para existir divisão, devem existir 2 ou mais chamadas recursivas.
- Algoritmos adequados para processamento paralelo.

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

5

Exemplo: ordenação de *arrays*

- Mergesort
  - Ordenar 2 subsequências de igual dimensão e juntá-las
  - T(n) = O(n log n), tanto no pior caso como no caso médio
  - S(n) = n
- Quicksort
  - Ordenar elementos menores e maiores que *pivot*, concatenar
  - T(n) = O(n^2) no pior caso (1 elemento menor, restantes maiores)
  - T(n) = O(n log n) no melhor caso e no caso médio (*)

    (*) com escolha aleatória do pivot!
  - S(n) = 1

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

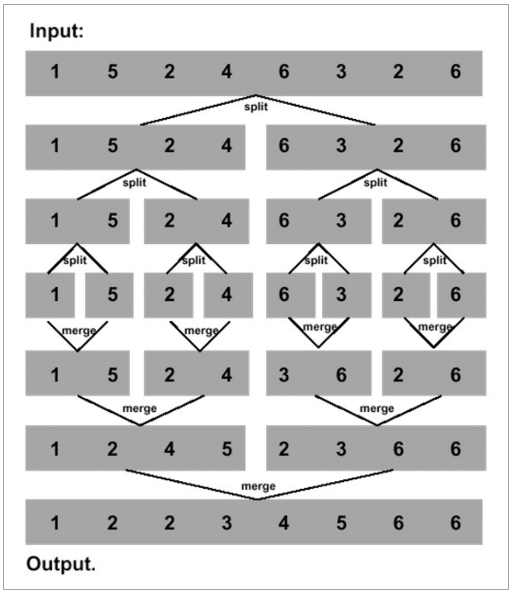
6

Exemplo 1: *Mergesort*

- Seja **S** = (s_1, …, s_n) uma sequência (array ou lista) a ordenar.
- Caso base: Se **S**=() ou **S**=(s_1), então nada é necessário!
- Dividir a sequência S em duas subsequências $\mathbf{S}_1$ e $\mathbf{S}_2$, cada uma com ~n/2 elementos
- Conquistar $\mathbf{S}_1$ e $\mathbf{S}_2$, ordenando-as com mergesort (isto é, aplicando recursivamente o mesmo procedimento)
- Combinar as sequências ordenadas $\mathbf{S}_1$ e $\mathbf{S}_2$ numa sequência ordenada única *S*
- Fazer o mais possível *in-place*.

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

Ilustração



Pseudo-código (1/2)

```
// Sorts array A between indices p and r.
Merge-Sort(A, p, r)
   if p < r then
      q ← ⌊(p + r) / 2⌋
      Merge-Sort(A, p, q) ‖ Merge-Sort(A, q + 1, r)
      Merge(A, p, q, r)
```

possibly in parallel

9

Pseudo-código (2/2)

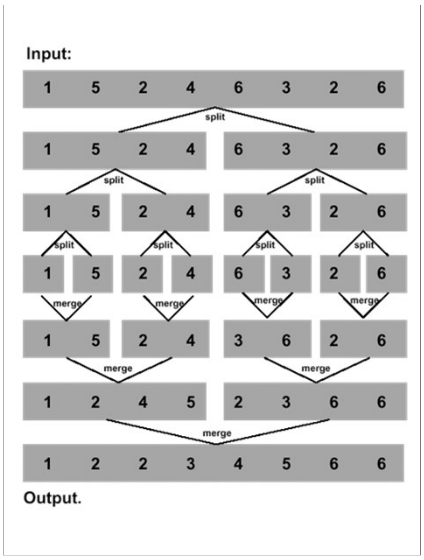
```
//Merges sorted subarrays A[p..q] and A[q+1..r]
//into a single sorted subarray A[p..r].
Merge(A, p, q, r)
  // Copy the subarrays into auxiliary
  // memory with a sentinel
  L ← (A[p],…,A[q], ∞), R ← (A[q+1],…,A[r], ∞)

  // Repeatedly take the smallest leftmost
  // element of L and R
  i ← 1, j ← 1
  for k = p to r do
     if L[i] ≤ R[j] then A[k] ← L[i], i ← i+1
     else A[k] ← R[j], j ← j+1
```

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP

10

Eficiência temporal

- Profundidade de recursão (nº de **níveis**) é $\lceil \log_2 n \rceil$
- Em cada nível, as várias operações de ***split*** podem ser efetuadas em tempo total **O(*n*)**
- Em cada nível, as várias operações de ***merge*** podem ser efetuada em tempo total **Θ(n)**
- Logo, tempo total (em qq caso) é **T(n) = Θ(n log n)**



Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP

Nota sobre notação assintótica

- T(n) – tempo de execução do algoritmo em função do tamanho *n* da entrada (no caso pior/melhor/médio)
- T(n) = O(f(n)) - *f(n)* é um limite superior (*upper bound*) assintótico para *T(n)*, isto é,

  $$\exists\ c>0,\ n_0>0 \bullet \forall n>n_0 \bullet 0 \leq T(n) \leq c\ f(n)$$

- T(n) = Θ(f(n)) - *f(n)* é um limite apertado (*tight bound*) assintótico para *T(n)*, isto é,

  $$\exists\ c_1>0,\ c_2>0,\ n_0>0 \bullet \forall n>n_0 \bullet c_1 f(n) \leq T(n) \leq c_2 f(n)$$

Nota sobre cálculo de T(n) em funções recursivas (1/2)

- **`Merge(A,p,q,r)`**
  - É óbvio que gasta um tempo Θ(n), com *n = r-p+1* (tamanho da sequência a processar e nº de iter. do ciclo **`for`**).
- **`Merge-Sort(A,p,r)`**
  - Vamos assumir que cada instrução tem um tempo de execução constante nas várias execuções
  - Para simplificar, vamos assumir que o tamanho da sequência original é uma potência de 2 (> 0)
  - $$T(n) = \begin{cases} c_1, & if\ n = 1 \\ c_2 + 2T\left(\frac{n}{2}\right) + c_3 n, & if\ n > 1 \end{cases}$$ (com n = r-p+1)

13

Nota sobre cálculo de T(n) em funções recursivas (2/2)

- Tentando inferir expressão geral:
  - $T(1) = c_1$
  - $T(2) = c_2 + 2(c_1) + 2c_3 = 2c_1 + c_2 + 2c_3$
  - $T(4) = c_2 + 2(2c_1 + c_2 + 2c_3) + 4c_3 = 4c_1 + 3c_2 + (4+4)c_3$
  - $T(8) = c_2 + 2(4c_1 + 3c_2 + 8c_3) + 8c_3 = 8c_1 + 7c_2 + (8+8+8)c_3$
  - …
  - $T(n) = n\ c_1 + (n - 1)\ c_2 + n \log_2 n\ c_3$ (provar por indução!)

- Conclui-se então que $T(n) = \Theta(n \log n)$

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

14

Optimizações

Optimizações e ganhos experimentais (*speedup*) conseguidos a ordenar *arrays* aleatórios de tamanho $n=10^7$

- Opção do linker: -Wl,--stack,0xFFFFFFF (para array caber na stack)
- Opção do compilador: -O3 (*optimize most*)

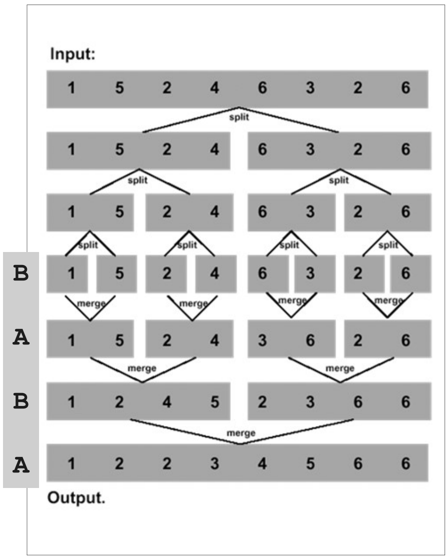
	Tempo (ms)	Ganho (*speedup*)
Merge sort, abordagem base (slides anteriores)	1330	-
Optimização da memória auxiliar	1226	x 1,08
Ordenação por inserção de arrays com n < 20	1078	x 1,14
Percorrer *arrays* com apontadores em vez de índices, usar `register`, usar `memcpy`	977	x 1,10
Processamento paralelo (4 *cores*, 8 *threads*)	**398**	**x 2,45**
Ganho total		**x 3,34**
std::sort (quick sort)	769	

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

15

Optimização da memória auxiliar

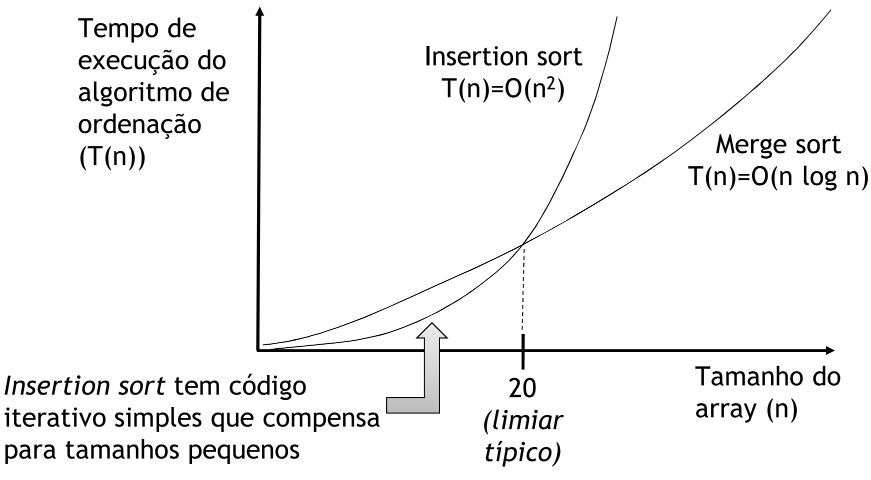
- Em vez de fazer cópias para memória auxiliar em cada **`Merge`** …
- Cria-se inicialmente uma cópia (`B`) de `A`
- As operações de **`Merge`** vão alternadamente colocando os resultados em `A` e `B`
- O tempo gasto nestas cópias é reduzido de Θ(n log) para Θ(n)



Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP

16

Ordenação por inserção de arrays com n < 20



Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP

17

Processamento paralelo

- Com *k* processadores ou núcleos (*cores*), executando as chamadas recursivas em paralelo, pode-se ter um ganho de desempenho de até *k* vezes
  - Em C++, número de núcleos é dado por

    `std::thread::hardware_concurrency()`

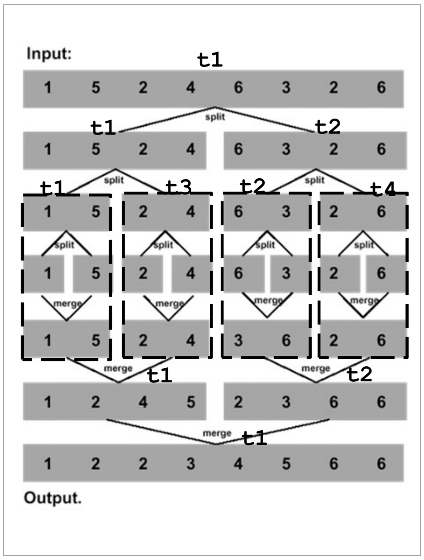
- Execução paralela é conseguida usando múltiplos *threads* (pois estes executam em paralelo, partilhando o mesmo espaço de endereçamento)
  - Normalmente, desempenho ótimo com nº *threads* = nº *cores*
  - Em processadores com *hyper-threading,* nº ótimo é 2 x nº *cores* (https://en.wikipedia.org/wiki/Hyper-threading)

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP

18

Ilustração

- Exemplo com 4 *cores*
- Divisão de trabalho por 4 *threads* concorrentes



Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP

19

Implementação em C++

```
#include <thread>
template <typename T>
void MergeSort(T A[], int p, int r, int threads){
 if (p < r) {
  int q = (p + r) / 2;
  if (threads > 1) {
   std::thread t([=](){MergeSort(A,p,q, threads/2);});
   MergeSort(A, q+1, r, threads /2);
   t.join();
  }
  else {
    MergeSort(A, p, q, 1); MergeSort(A, q + 1, r, 1)
  }
  Merge(A, p, q, r);
 }
}
```

Nº de *threads* a usar (potência de 2) (inicialmente ~ nº de *cores*).

1) Lança 1ª chamada recursiva num novo *thread t* separado.

2) Executa 2ª chamada recursiva neste *thread*.

3) Espera que o outro *thread* termine.

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP

20

Nota sobre funções lambda em C++

Neste caso o argumento do construtor do thread *t* é uma função lambda (função definida *on the fly)*.

std::thread t ([=] () {MergeSort(…);});

"=" significa que o corpo da função pode usar por cópia todas as variáveis locais da função em que se insere

sem argumentos neste caso

corpo da função definida

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP

Exemplo 2: cálculo de x^n

- Resolução iterativa com *n* multiplicações: T(n) = O(n)
- Resolução mais eficiente, com divisão e conquista:

$$x^n = \begin{cases} 1, \text{ se } n = 0 \\ x, \text{ se } n = 1 \\ x^{\frac{n}{2}} \times x^{\frac{n}{2}}, \text{ se } n \text{ par} > 1 \\ x \times x^{\frac{n-1}{2}} \times x^{\frac{n-1}{2}}, \text{ se } n \text{ ímpar} > 1 \end{cases}$$

```
double power(double x, int n) {
  if (n == 0) return 1;
  if (n == 1) return x;
  double p = power(x, n / 2);
  if (n % 2 == 0) return p * p;
  else return x * p * p;
}
```

- Nº de multiplicações reduzido para ~$\log_2 n$
- T(n) = O(log n) …. mas S(n) = O(log n) (espaço)
- Nota: classificação como divisão e conquista não é consensual, por os 2 subproblemas serem idênticos (logo só há 1 chamada recursiva)

Exemplo 3: pesquisa binária

- Seja **S**=(s_1,…,s_n) uma sequência ordenada de *n* elementos, e *x* um elemento que se pretende procurar em ***S***.
- Casos bases:
  - Se S=(), falha!
  - Se $x=s_m$, c/ $m=\lfloor(1+n)/2\rfloor$ (elem. médio), retorna-se a posição!
- Dividir S em duas subsequências, L=(s_1, …., s_{m-1}) e R=(s_{m+1}, …., s_n), à esquerda e à direita do elemento médio.
- Conquistar: se $x < s_m$ ($x > s_m$) continua-se a pesquisa em L (R, resp.)
- T(n) = O(log n)
- Nota: classificação como divisão e conquista não é consensual, por um dos 2 subproblemas ser vazio (logo basta 1 chamada recursiva).

Referências

- T.H. Cormen, C. E. Leiserson, R. L. Rivest , C. Stein. Introduction to Algorithms, 3rd Edition. MIT Press, 2009
  - Capítulo 4 - Divide-and-Conquer
  - Secção 27.3 - Multithreaded Merge Sort
- Mark Allen Weiss. Data Structures & Algorithm Analysis in Java. Addison-Wesley, 1999
- Steven S. Skiena. The Algorithm Design Manual. Springer 1998
- Robert Sedgewick. Algorithms in C++. Addison-Wesley, 1992