1

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

J. Pascoal Faria, R. Rossetti, L. Ferreira
CAL, MIEIC, FEUP

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

2

Divisão e Conquista (*divide and conquer*)

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

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

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

4

Divisão e conquista

- Dada uma instância do problema x, a técnica Divisão-e-Conquista funciona da seguinte maneira:

```
function DAC(x)
   if x is small enough then
      solve x directly
   else
      divide x into smaller instances x1, …, xk
      for i = 1 to k do yi ← DAC(xi) {conquer}
      combine y1,…,yk to obtain solution y
      return y
```

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

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

- 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!

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

6

Exemplo: *Mergesort*

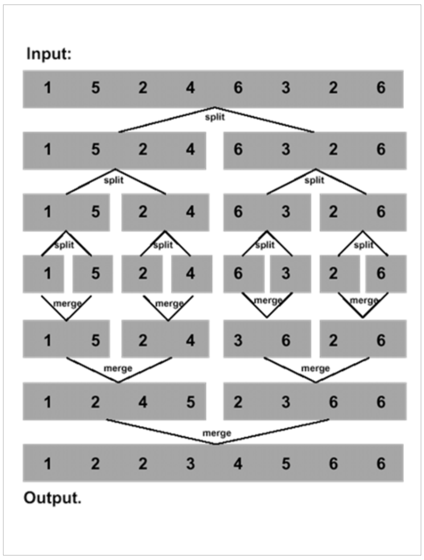
- Seja **S** = (s_1, …, s_n) uma sequência (array ou lista) a ordenar.
- Caso base: Se **S**=() ou **S**=(s_1), então nada é necessário!
- Dividir: partir a sequência S em duas subsequências $\mathbf{S}_1$ e $\mathbf{S}_2$, cada uma com ~n/2 elementos
- Conquistar: ordenar $\mathbf{S}_1$ e $\mathbf{S}_2$, utilizando mergesort (isto é, aplicando recursivamente o mesmo procedimento)
- Combinar: unir 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 (2017-2018)

7

Ilustração



Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

8

Algoritmo em pseudo-código

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

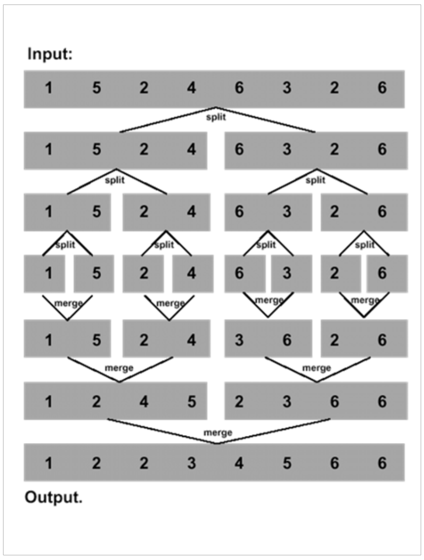
```
// Merges two sorted subsequences A[p..q] and A[q+1..r]
// into a single sorted subsequence A[p..r].
Merge(A, p, q, r)
   //Copy the subsequences into aux. memory with a sentinel
   L ← (A[p], …,A[q], ∞),  R ← (A[q+1], …,A[r], ∞)
  //Repeatedly take the smallest leftmost element of L and R
  i ← 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 (2017-2018)

9

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 **$\Theta(n)$**
- Em cada nível, as várias operações de *merge* podem ser efetuada em tempo total **$\Theta(n)$**
- Logo, tempo total (em qq caso) é **$T(n) = \Theta(n \log n)$**



Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

10

Nota sobre notação assintótica

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

$$\exists\ \mathbf{c>0,\ n_0>0 \bullet \forall n>n_0 \bullet 0 \le T(n) \le c\ f(n)}$$

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

$$\exists\ \mathbf{c_1>0,\ c_2>0,\ n_0>0 \bullet \forall n>n_0 \bullet c_1 f(n) \le T(n) \le c_2 f(n)}$$

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

11

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

- `Merge(A,p,q,r):` é obvio que gasta um tempo Θ(n), em que n é *o* tamanho da sequência a processar (n = r-p+1), e também o nº de iterações do ciclo **`for`**.
- `Merge-Sort(A,p,r:` assumindo que cada instrução tem um tempo de execução constante nas várias execuções, o tempo de execução pode ser definido pela seguinte fórmula de recorrência, em que *n* é o tamanho da sequência a processar (n = r-p+1)

$$\boldsymbol{T(n)} = \begin{cases} \mathbf{c_1}, & \boldsymbol{if\ n = 1} \\ \mathbf{c_2} + \boldsymbol{2T\left(\frac{n}{2}\right)} + \mathbf{c_3}\boldsymbol{n}, & \boldsymbol{if\ n > 1} \end{cases}$$

- Para simplificar, vamos assumir que o tamanho da sequência original é uma potência de 2.

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

12

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

- Tentando inferir expressão geral:
  - $T(1) = c_1$
  - $T(2) = c_2 + 2(c_1) + 2\ c_3 = 2c_1 + c_2 + 2c_3$
  - $T(4) = c_2 + 2(2c_1 + c_2 + 2c_3) + 4c_3 = 4c_1 + 3c_2 + 8c_3$
  - $T(8) = c_2 + 2(4c_1 + 3c_2 + 8c_3) + 8c_3 = 8c_1 + 7c_2 + 24c_3$
  - $T(16) = c_2 + 2(8c_1 + 7c_2 + 24c_3) + 16c_3 = 16c_1 + 15c_2 + 64c_3$
  - …
  - $T(n) = n\ c_1 + (n - 1)\ c_2 + n \log_2 n\ c_3$ (provar por indução!)
- Conclui-se então que T(n) = Θ(n log n)

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

13

Optimizações

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

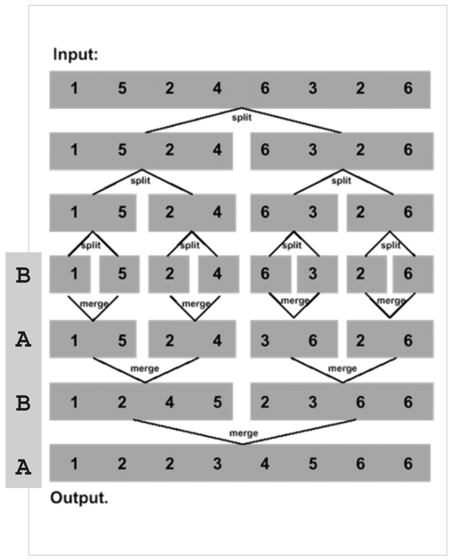
	Tempo (ms)	**Ganho (*speedup*)**
std::sort	7076	-
Merge sort, abordagem base (slides anteriores)	3354	**x 2,11**
Optimização da memória auxiliar (ver a seguir)	2448	**x 1,37**
Ordenação direta de *arrays* pequenos (ordenação por inserção quando n < 20)	1985	**x 1,23**
Percorrer *arrays* com apontadores alocados com *register*, em vez de usar índices	1089	**x 1,82**
Processamento paralelo (ver a seguir)	484	**x 2.25**
Ganho total		**x 14.62 !!**

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

14

Optimização da memória auxiliar

- Em vez de fazer cópias para memória auxiliar em cada `Merge`, cria-se inicialmente uma cópia (B) de A, e 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)
- Tempo total pode ser reduzido até metade



Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

15

Optimização da memória auxiliar

```
// Sorts array A of size n.
Merge-Sort(A, n)
   Merge-Sort(A, copy(A), 1, n)
```

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

```
// Merges two sorted subsequences B[p..q] and B[q+1..r]
// into a single sorted subsequence A[p..r].
Merge(A, B, p, q, r)
  i ← p, j ← q + 1
  for k = p to r do
    if j > r ∨ i ≤ p ∧ B[i]≤B[j] then A[k] ← B[i], i ← i+1
    else A[k] ← B[j], j ← j+1
```

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

16

Processamento paralelo

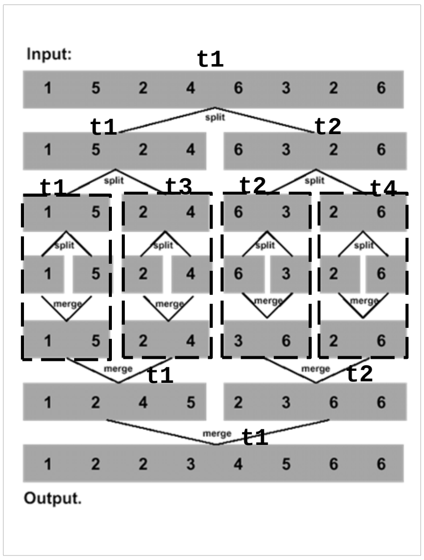
- Com *k* processadores ou núcleos (*cores*), executando as chamadas recursivas em paralelo, pode-se ter um ganho de desempenho de até *k* vezes
  - Em C++, número de núcleos é dado por
    - **`std::thread::hardware_concurrency()`**
- Execução paralela é conseguida usando múltiplos *threads* (pois estes executam em paralelo)
  - P/ desempenho ótimo, usar nº de *threads* = nº de núcleos
- Resultados experimentais:
  - Tempo médio sem paralelização, n = 10^7 : 1639 ms
  - Tempo médio com paralelização, n = 10^7, 4 núcleos: 738 ms
  - Ganho (*speedup*): 2.2 vezes mais rápido

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

17

Processamento paralelo

- Com 4 *cores*, divisão de trabalho por 4 *threads* concorrentes



Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

18

Processamento paralelo em C++

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

Nº de *threads* a usar (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 (2017-2018)

Nota: 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

corpo da função definida

sem argumentos neste caso

Exemplo: 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;
}
```

- Divisão em 2 subproblemas idênticos, junção em tempo O(1)
- Nº de multiplicações reduzido para aprox. $\log_2 n$
- 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)

21

Exemplo: 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!
- Divisão e conquista:
  - Dividir: divide-se 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 $x < s_m$, continua-se a pesquisa em L; se $x > s_m$, continua-se a pesquisa em R.
- 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) e não existir passo de combinação.

22

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)
- 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

23

Em suma...

- Programação dinâmica (*dynamic programming*)
  - Contexto: Problemas de solução recursiva.
  - Objectivo: Minimizar tempo e espaço.
  - Forma: Induzir uma progressão iterativa de transformações sucessivas de um espaço linear de soluções.
- Algoritmos ganânciosos (*greedy algorithms*)
  - Contexto: Problemas de optimização (max. ou min.)
  - Objectivo: Atingir a solução óptima, ou uma boa aproximação.
  - Forma: tomar uma decisão óptima localmente, i.e., que maximiza o ganho (ou minimiza o custo) imediato

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)

24

Em suma...

- Algoritmos de retrocesso (*backtracking*)
  - Contexto: problemas sem algoritmos eficientes (convergentes) para chegar à solução.
  - Objectivo: Convergir para uma solução.
  - Forma: tentativa-erro. Gerar estados possíveis e verificar todos até encontrar solução, retrocedendo sempre que se chegar a um "beco sem saída".
- Divisão e conquista (*divide and conquer*)
  - Contexto: Problemas passíveis de se conseguirem sub-dividir.
  - Objectivo: melhorar eficiencia temporal.
  - Forma: agregação linear da resolução de sub-problemas de dimensão semelhantes até chegar ao caso-base.

Técnicas de Concepção de Algoritmos, CAL – MIEIC/FEUP (2017-2018)