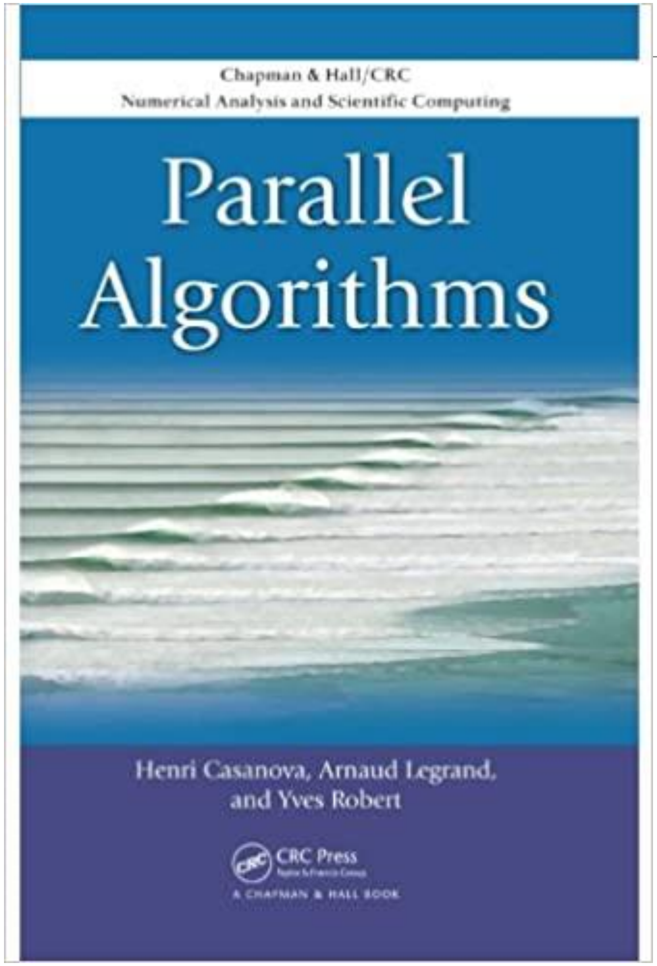
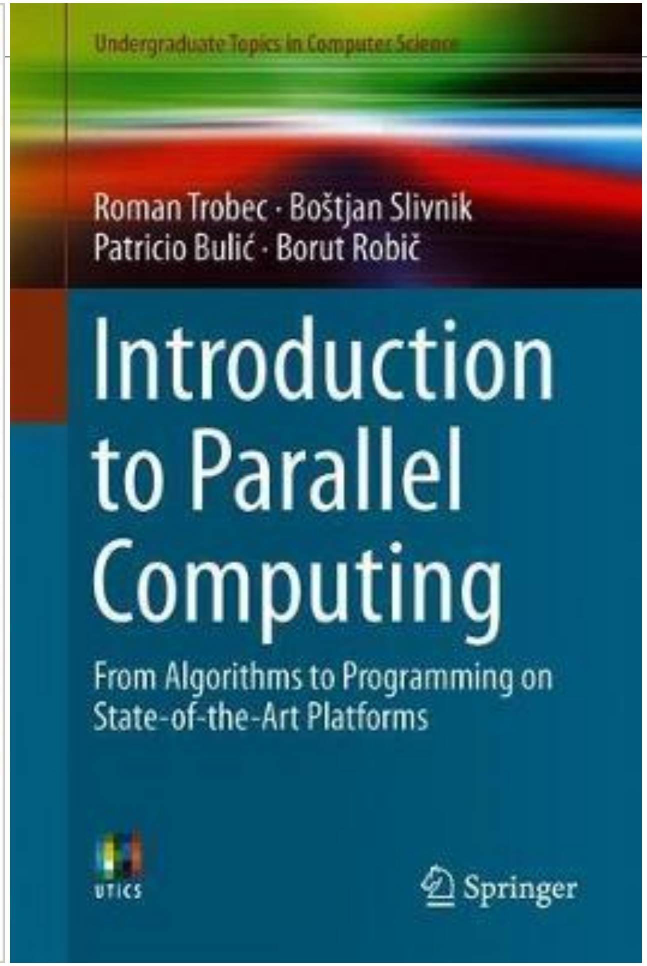
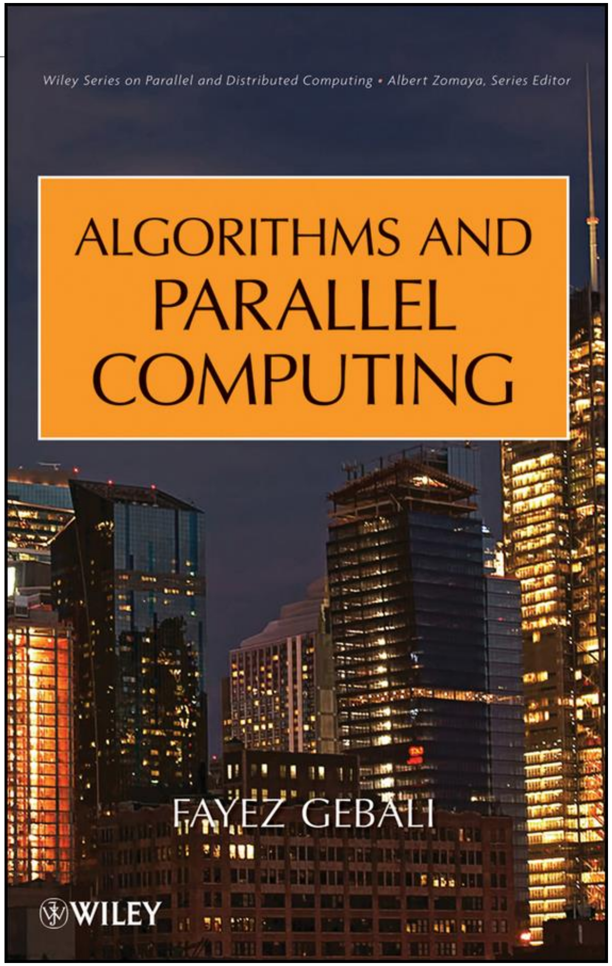
Parallel Programming

Lec 5

Books

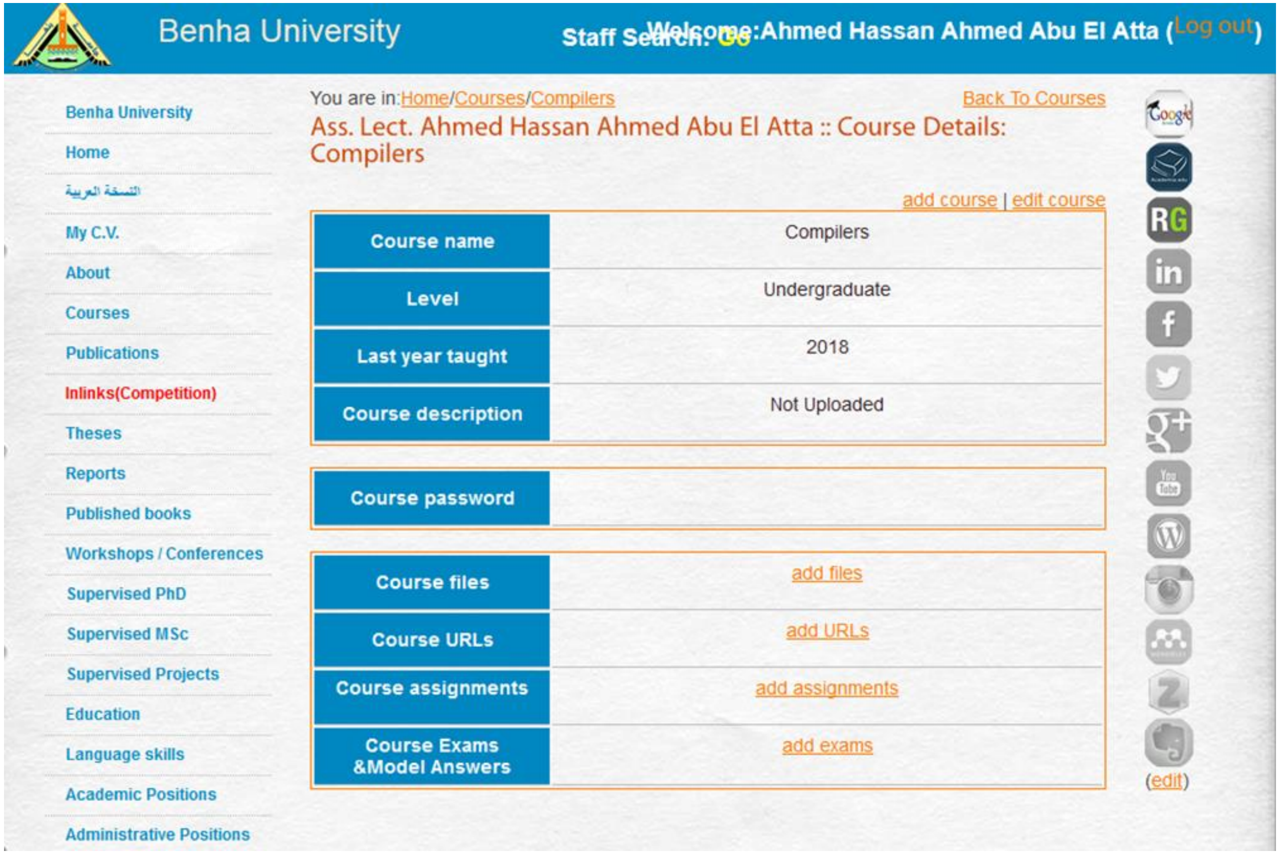

Chapman & Hall/CRC
Numerical Analysis and Scientific Computing
Parallel Algorithms
Henri Casanova, Arnaud Legrand, and Yves Robert
CRC Press
A CHAPMAN & HALL BOOK


Undergraduate Topics in Computer Science
Roman Trobec · Boštjan Slivnik
Patricio Bulić · Borut Robič
Introduction to Parallel Computing
From Algorithms to Programming on State-of-the-Art Platforms
UTICS
Springer


Wiley Series on Parallel and Distributed Computing • Albert Zomaya, Series Editor
ALGORITHMS AND PARALLEL COMPUTING
FAYEZ GEBALI
WILEY

PowerPoint

http://www.bu.edu.eg/staff/ahmedaboalatah14-courses/14779



Bubble Sort

In bubble sort, the largest number is first moved to the very end of the list by a series of compare-and-exchange operations, starting at the opposite end.

The procedure repeats, stopping just before the previously positioned largest number, to get the next-largest number.

In this way, the larger numbers move (like a bubble) toward the end of the list.

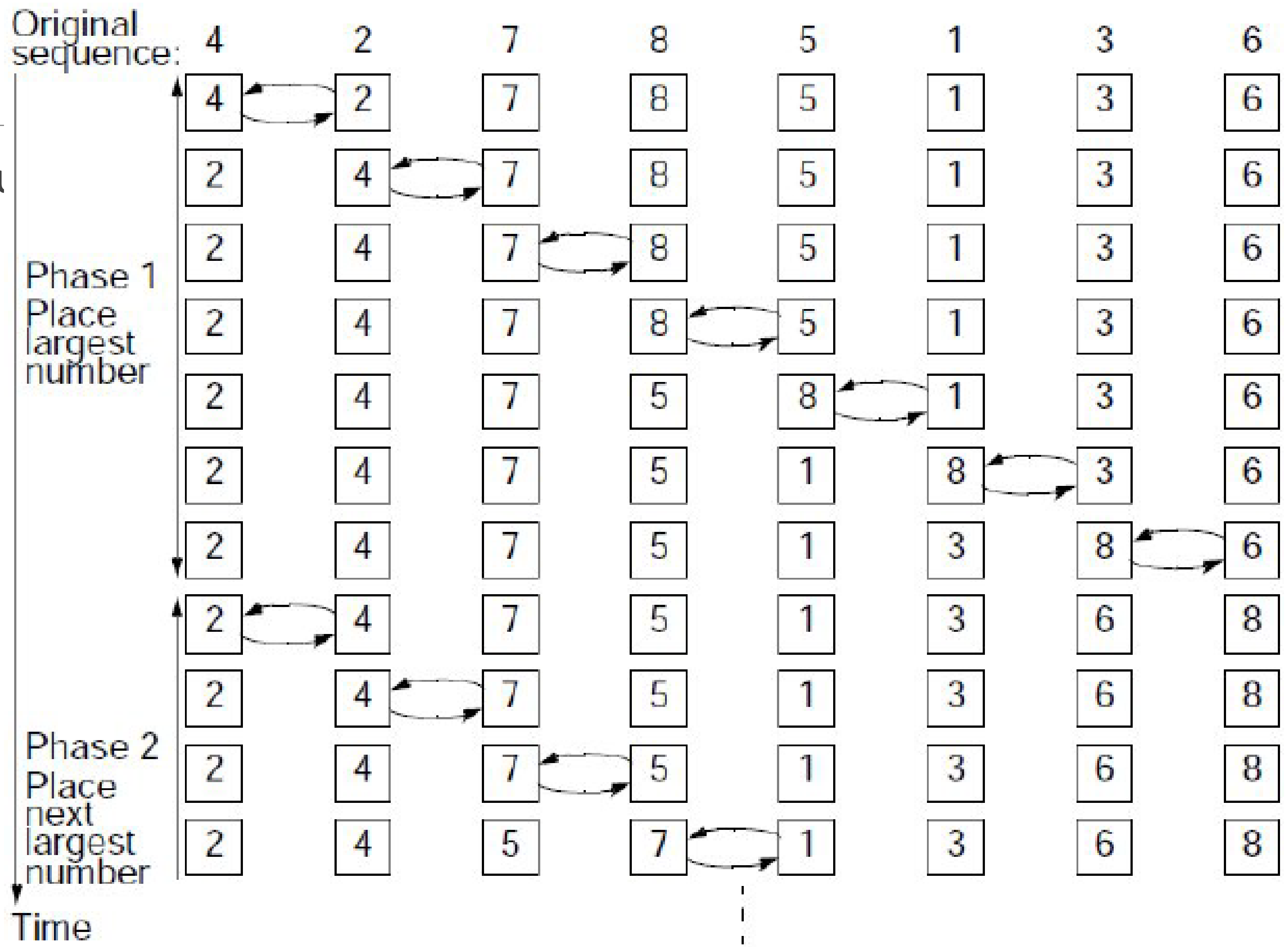
Bubble Sort

First pass

54	26	93	17	77	31	44	55	20	Exchange
26	54	93	17	77	31	44	55	20	No Exchange
26	54	93	17	77	31	44	55	20	Exchange
26	54	17	93	77	31	44	55	20	Exchange
26	54	17	77	93	31	44	55	20	Exchange
26	54	17	77	31	93	44	55	20	Exchange
26	54	17	77	31	44	93	55	20	Exchange
26	54	17	77	31	44	55	93	20	Exchange
26	54	17	77	31	44	55	20	93	93 in place after first pass

Bubble Sort

Bu



Bubble Sort

```
For (i = N - 1; i > 0; i--)
        For (j = 0; j < i; j++)
                If (a[j] > a[j+1])
                        temp = a[j];
                        a[j] = a[j+1];
                        a[j+1] = temp;
                End If
        End For
End For
```

Bubble Sort

The total number of steps in the bubble sort algorithm is

T(n) = (n-1) + (n-2) + (n-3) + … + 3 + 2 + 1

T(n) = $\sum_{i=1}^{n-1} i = \frac{(1+(n-1))}{2}(n-1) = \frac{n(n-1)}{2}$

which corresponds to a time complexity of T(n) = O(n^2).

Parallel Odd Even Transposition Sort

It is based on the Bubble Sort technique, which compares every 2 consecutive numbers in the array and swap them if first is greater than the second to get an ascending order array.

It consists of 2 phases – the odd phase and even phase:

- **Odd phase:** Every odd indexed element is compared with the next even indexed element(considering 1-based indexing).
- **Even phase:** Every even indexed element is compared with the next odd indexed element.

Odd Even Transposition Sort

Unsorted array: 2, 1, 4, 9, 5, 3, 6, 10

Step 1(odd):	2	1	4	9	5	3	6	10
Step 2(even):	1	2	4	9	3	5	6	10
Step 3(odd):	1	2	4	3	9	5	6	10
Step 4(even):	1	2	3	4	5	9	6	10
Step 5(odd):	1	2	3	4	5	6	9	10
Step 6(even):	1	2	3	4	5	6	9	10
Step 7(odd):	1	2	3	4	5	6	9	10
Step 8(even):	1	2	3	4	5	6	9	10

Sorted array: 1, 2, 3, 4, 5, 6, 9, 10

Odd Even Transposition Sort

Step	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
0	4	2	7	8	5	1	3	6
1	2	4	7	8	1	5	3	6
2	2	4	7	1	8	3	5	6
3	2	4	1	7	3	8	5	6
4	2	1	4	3	7	5	8	6
5	1	2	3	4	5	7	6	8
6	1	2	3	4	5	6	7	8
7	1	2	3	4	5	6	7	8

Time

Odd Even Transposition Sort

```
For i = 0 to i < n do
    If( i  % 2 == 0)
        For j = 0 to j < n-1 do in parallel
            If(j % 2 == 0)
              x[j] = min(x[j], x[j+1])
            Else
              x[j] = max(x[j-1], x[j])
    Else
        For j = 0 to j < n-1 do in parallel
            If(j % 2 == 0)
              x[j] = max(x[j-1], x[j])
            Else
              x[j] = min(x[j], x[j+1])
```

Odd Even Transposition Sort

Each iteration cost constant step:

T(n) $= \sum(1 + 1 + 1 + \cdots + 1) = \sum_{0}^{n-1} 1 = n$

It takes n steps to obtain the final sorted list in a parallel implementation, which corresponds to a time complexity of O(n)

Odd-Even Merge

Odd-even mergesort is a parallel sorting algorithm based on the recursive application of the odd-even merge algorithm.

It merges sorted sublists bottom up – starting with sublists of size 2 and merging them into bigger sublists – until the final sorted list is obtained.

Odd-Even Merge

start with a two sorted lists of length n/2:

2	3	4	7	1	5	6	8

consider elements with odd and even index:

2	3	4	7	1	5	6	8

sort odd- and even-indexed elements separately:

1	3	2	5	4	7	6	8

final sequence is nearly sorted (only pairwise exchange required)

OddEvenSplit

OddEvenSplit (Input :: A: Array [0 . . n-1] , Output :: Odd:Array [0 . . (n-1)/2] , Even:Array [0 . . (n-1)/2])

```
For j = 0 to j < n do in parallel
    If(j % 2 == 0)
        Even [ j/2 ] = A[ j ] ;
    Else
        Odd [ (j+1)/2 ] := A[ j ];
```

OddEvenJoin

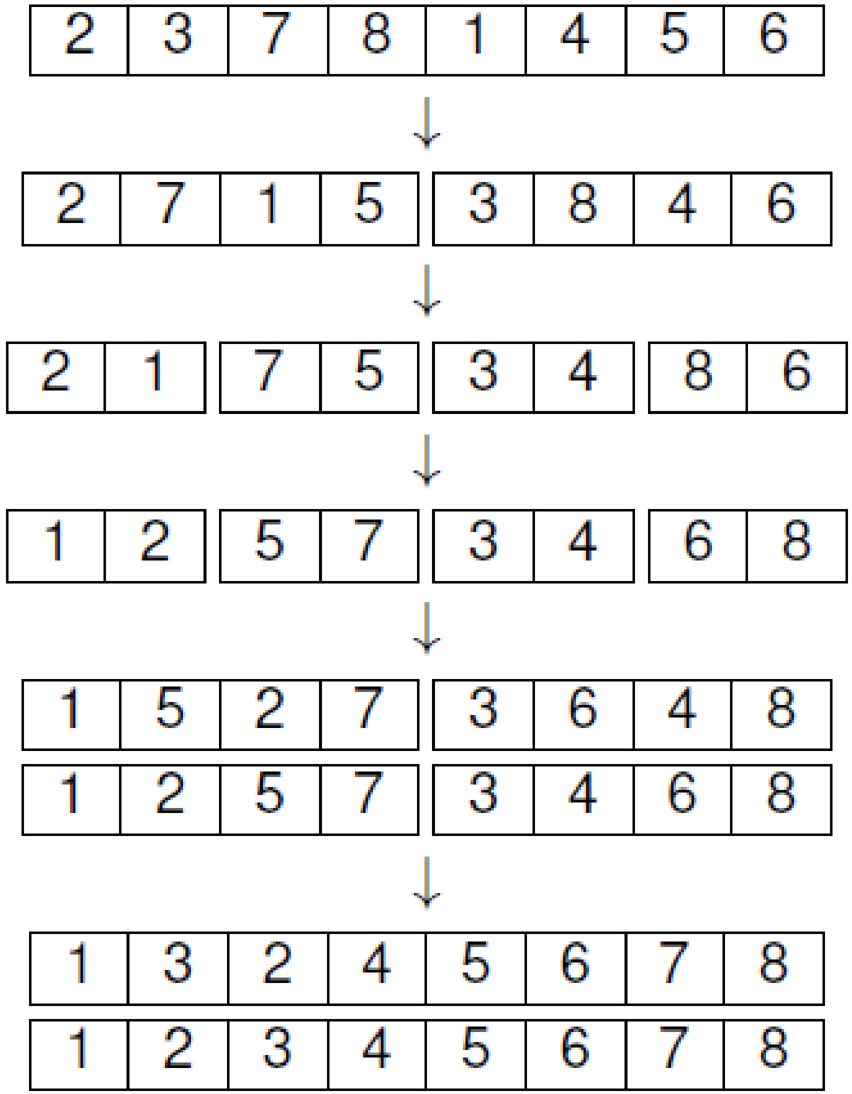
OddEvenJoin (Input :: Odd : Array [0 . . (n-1) / 2] , Even :Array[0 . . (n-1)/2], Output :: A:Array[0 . . n-1])

```
For j = 0 to j < n do in parallel
    If(j % 2 == 0)
        A[ j ] = Even [ j/2 ] ;
    Else
        A[ j ] = Odd[ (j+1)/2 ] ;
```

Parallel OddEvenMerge

```
OddEvenMerge (A: Array [ 0 . . n-1 ])
  If n ≤ 2
        SortTwo (A);
  Else
        OddEvenSplit (A, Odd, Even) ;
        do in parallel
            OddEvenMerge (Odd) ;
            OddEvenMerge (Even) ;
        OddEvenJoin (A, Odd, Even) ;
        for j = 1 to j < (n/2) do in parallel
            If A[2j ] > A[2j +1]
                 exchange A[2j ] and A[2j +1]
```

OddEvenMerge



OddEvenMergeSort

OddEvenMergeSort (A: Array [0 . . . n-1])

// n assumed to be 2^k

If n ≥ 2

    do in parallel

        OddEvenMergeSort (A [0 . . . (n/2)-1]) ;

        OddEvenMergeSort (A[n/2 . . . n-1]) ;

    OddEvenMerge(A) ;

Complexity of Odd-Even MergeSort

Complexity of OddEvenMerge:

- O(log n) subsequent steps
- each step executed on n/2 processors

Complexity of Odd-Even MergeSort:

- requires executions of OddEvenMerge on subarrays of lengths
- k = 2, 4, … ; n
- each OddEvenMerge step requires O(log k) steps
- number of total subsequent steps:
- log 2 + log 4 + … + log n = $\sum_{i=1}^{\log n} i$ = (log n) * ((log n) + 1)/2 = $O(\log n)^2$