# CPSC 4770/6770

# Distributed and Cluster Computing

## Lecture 14: Parallel Sorting

# Sorting Algorithms

- Rearranging a list of numbers into increasing (strictly non-decreasing) order
- Sorting number is important in applications as it can make subsequent operations more efficient

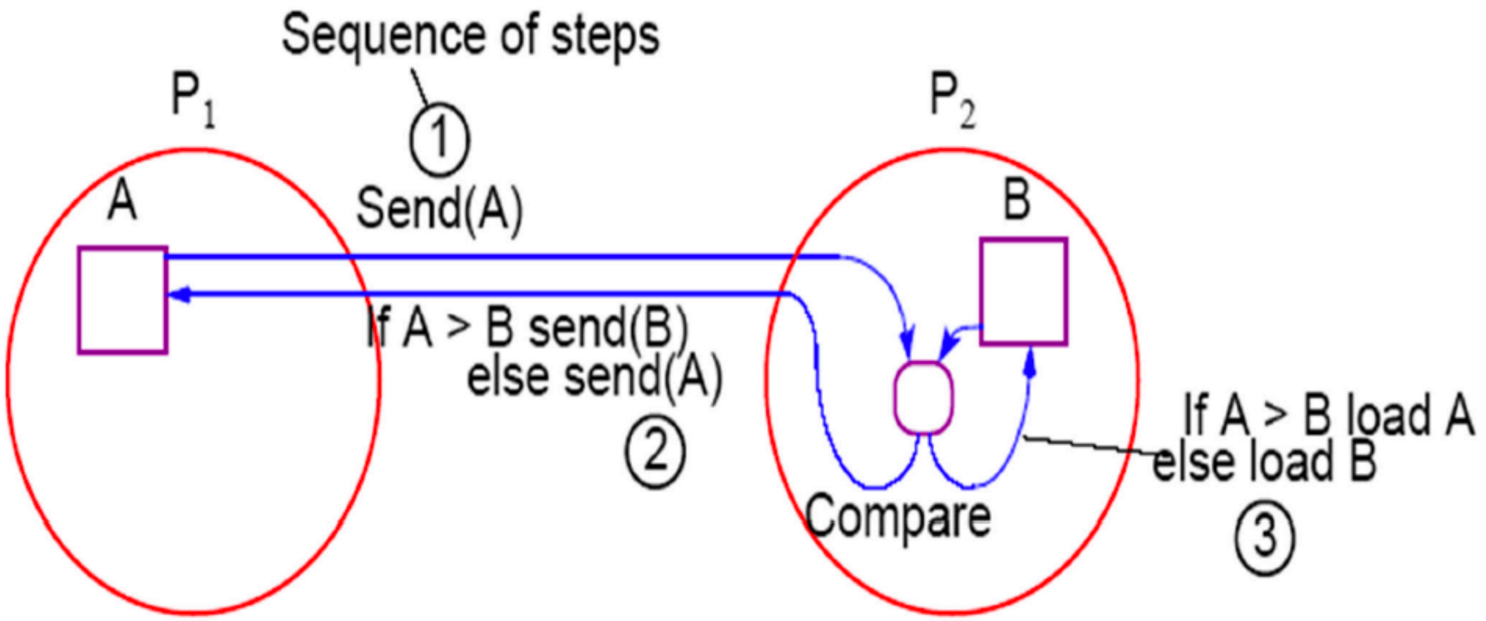
# Compare-and-Exchange Sorting Algorithms

- “Compare and exchange” -- the basis of several, if not most, classical sequential sorting algorithms
- Two numbers, say A and B, are compared
- If A > B, A and B are exchanged, i.e.:

```
if (A > B) {
        temp = A;
        A = B;
        B = temp;
}
```

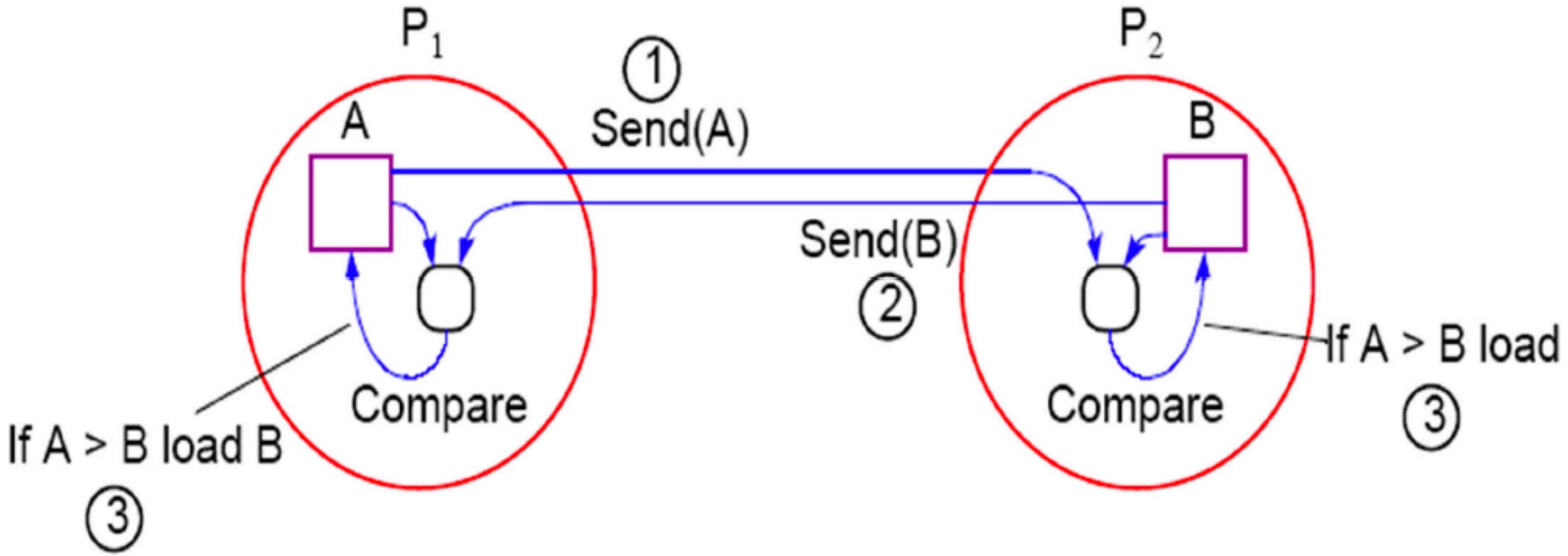
# Message-Passing Compare and Exchange (Version 1)

- $P_1$ sends A to $P_2$, which compares A and B and sends back B to $P_1$ if A is larger than B (otherwise it sends back A to $P_1$):



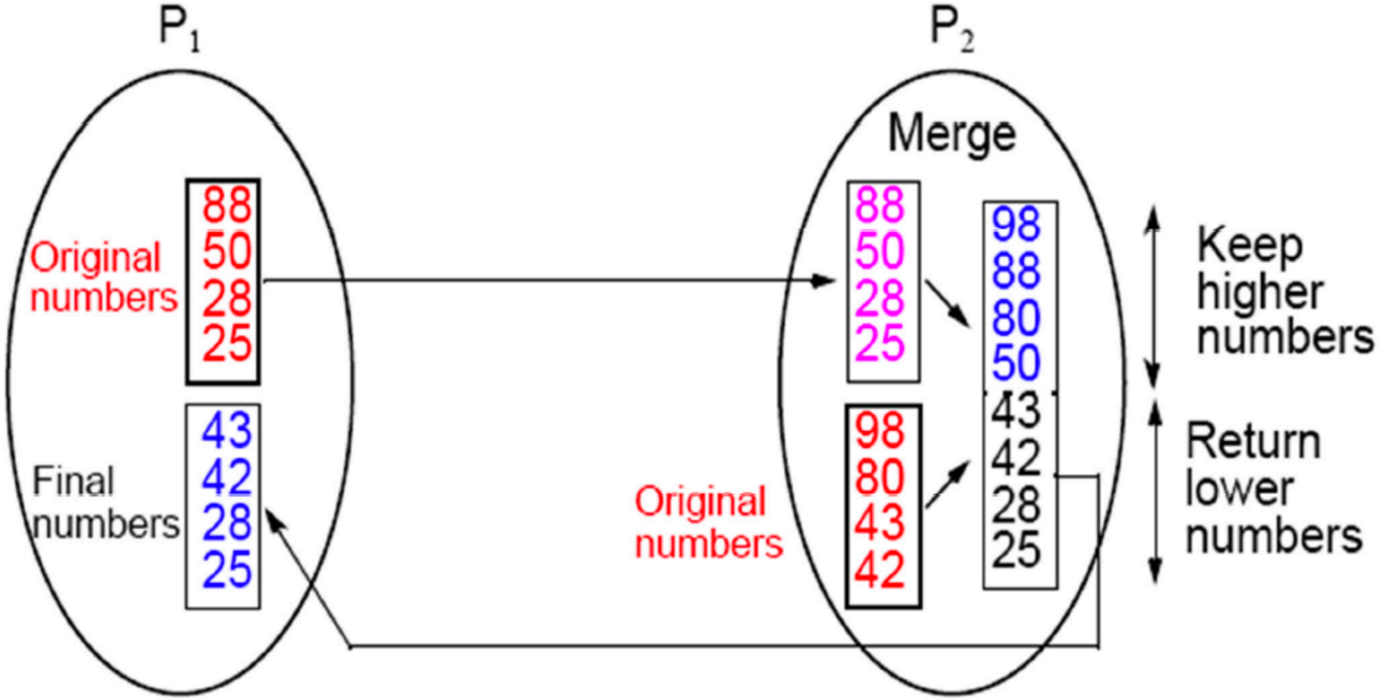
# Alternative Message Passing Method (Version 2)

- $P_1$ sends A to $P_2$ and $P_2$ sends B to $P_1$. Then both processes perform compare operations. $P_1$ keeps the larger of A and B and $P_2$ keeps the smaller of A and B:

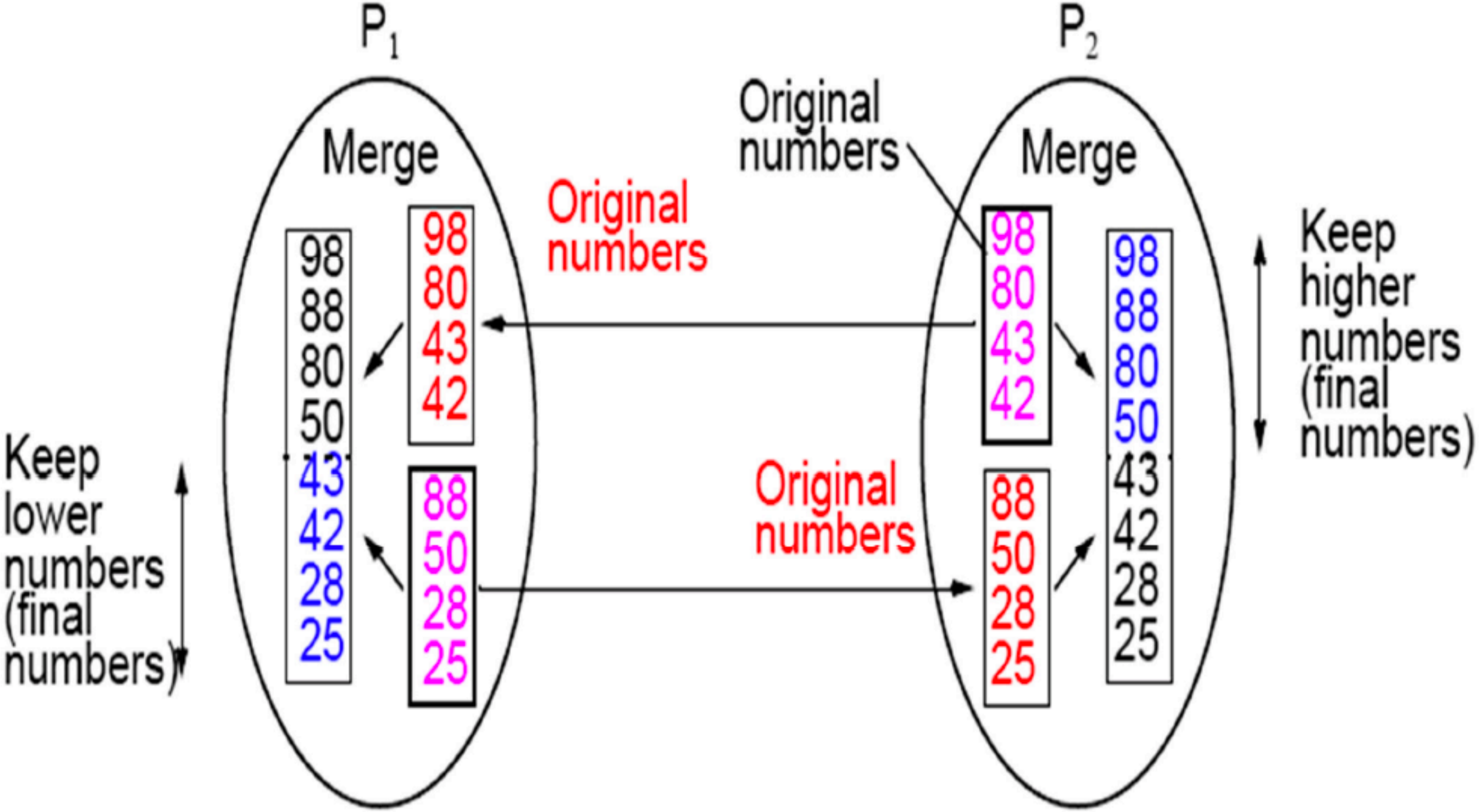


# Data Partitioning (Version 1)

- p processors and n numbers
- n/p numbers assigned to each processor:



# Data Partitioning (Version 2)

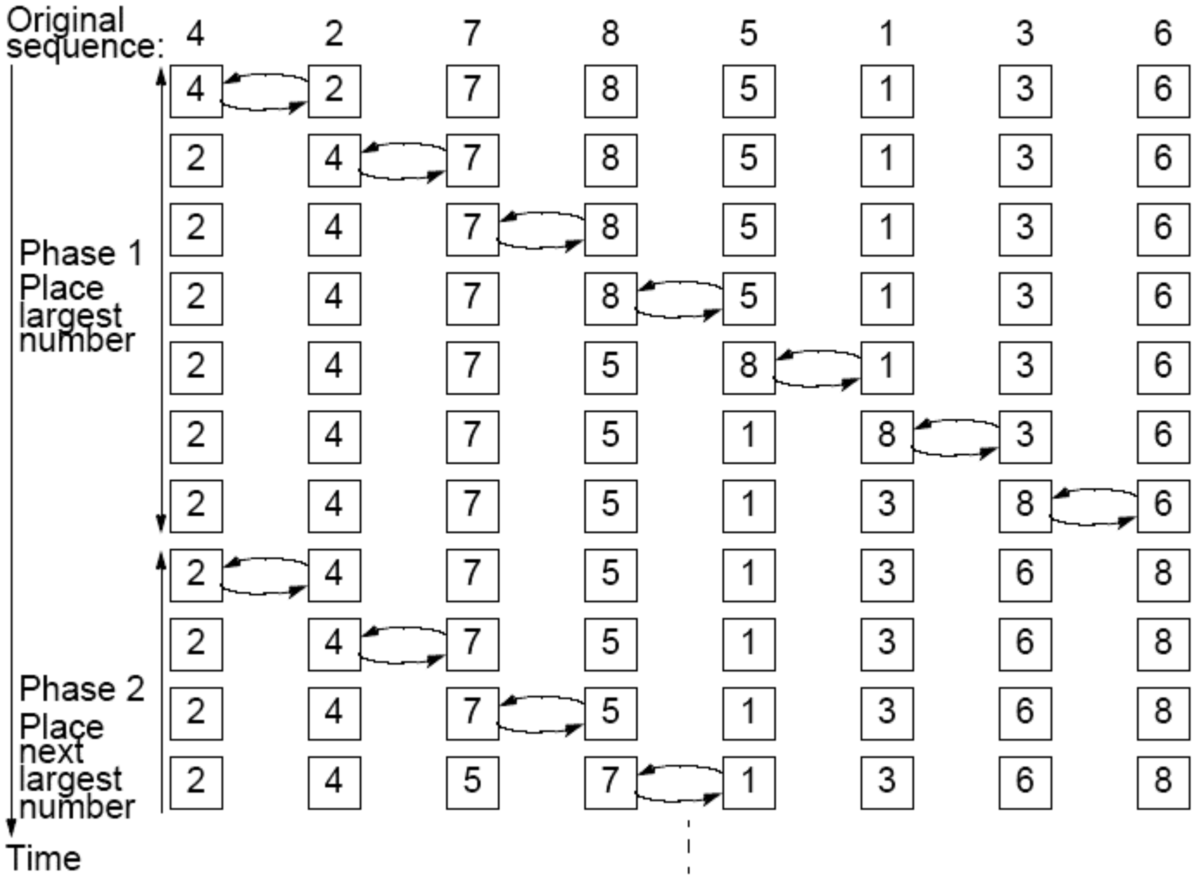


# Partitioning Numbers into Groups

- p processors and n numbers
- n/p numbers assigned to each processor
- applies to all parallel sorting algorithms to be given as number of processors usually much less than the number of numbers

# Bubble Sort

- Largest number moved to the end of the list by a series of compares and exchanges, starting at the opposite end
- Actions repeated with subsequent numbers, stopping just before that previously positioned number
- In this way, large numbers move ("bubble") toward one end




Wilkinson, Barry, and Michael Allen. Parallel programming. 2nd Ed. 2003.

# Bubble Sort – Time Complexity

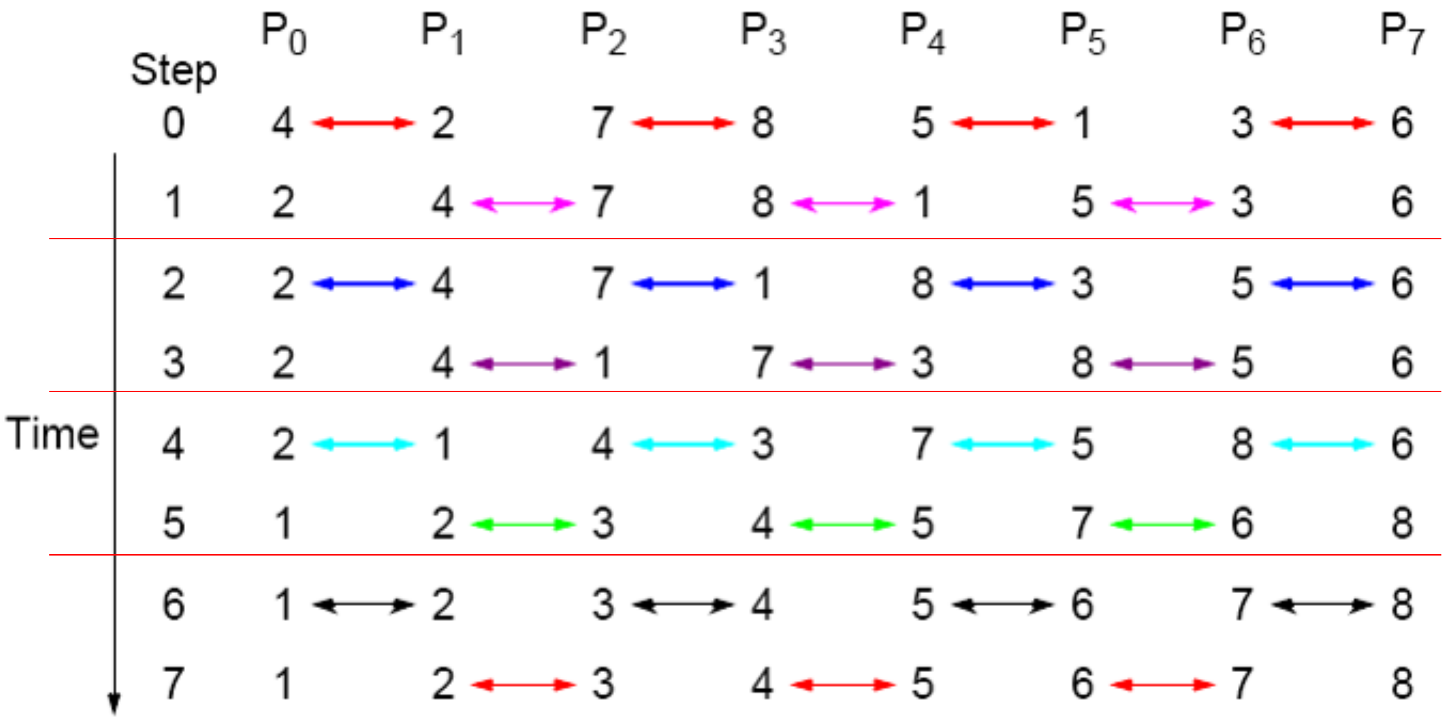
- Number of compare and exchange operations

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

Indicates time complexity of O($n^2$) if a single compare-and-exchange operation has a constant complexity, O(1). Not good but can be parallelized.

# Odd-Even (Transposition) Sort (Parallel)

- Transposition Sort is a variation of Bubble Sort
- Operates in two alternating phases, *even* and *odd*
- **Even phase**: Even-numbered processes exchange numbers with their right neighbor
- **Odd phase**: Odd-numbered processes exchange numbers with their right neighbor



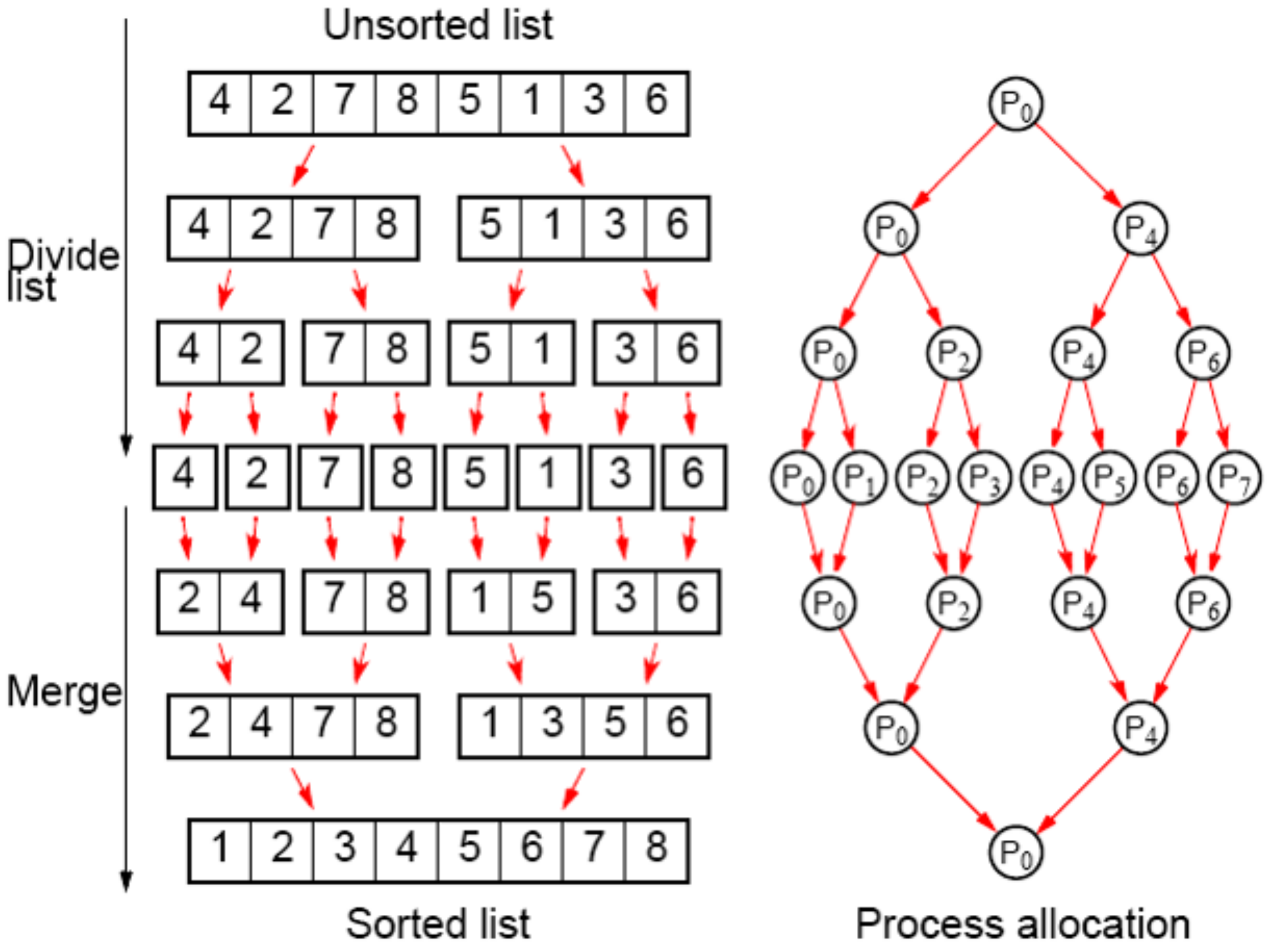
# transposition.py (parallel)

```python
#!/usr/bin/env python
# transposition.py
import numpy as np
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank(); size = comm.Get_size(); status = MPI.Status();
N = 16
unsorted = np.zeros(N, dtype="int")
final_sorted = np.zeros(N, dtype="int")
local_array = np.zeros(int(N / size), dtype="int")
local_tmp = np.zeros(int(N / size), dtype="int")
local_remain = np.zeros(2 * int(N / size), dtype="int")

if rank == 0:
    unsorted = np.random.randint(low=0,high=N,size=N)
    print (unsorted)
comm.Scatter(unsorted, local_array, root = 0)

local_array.sort()
for step in range(0, size):
    print ("Step: ", step)
    if (step % 2 == 0):
        if (rank % 2 == 0):
            des = rank + 1
        else:
            des = rank - 1
    else:
        if (rank % 2 == 0):
            des = rank - 1
        else:
            des = rank + 1

    if (des >= 0 and des < size):
        print ("My rank is ", rank, " and my des is ", des)
        comm.Send(local_array, dest = des, tag = 0)
        comm.Recv(local_tmp, source = des)
        print ("Rank ", rank, " ", step, ": Initial local_array: ", local_array)
        print ("Rank ", rank, " ", step, ": Received local_tmp:", local_tmp)
        local_remain = np.concatenate((local_array, local_tmp), axis=0)
        local_remain.sort()

        if (rank < des):
            local_array = local_remain[0:int(N/size)]
        else:
            local_array = local_remain[int(N/size):2 * int(N/size)]
        print ("Rank ", rank, " ", step, ": Retained portions: ", local_array)
comm.Gather(local_array, final_sorted)

if (rank  == 0):
    print (final_sorted)
```

```
[jin6@node0261 13-parallel-sorting]$ mpirun -np 4 --mca mpi_cuda_support 0 python transposition.py
[11  9  4 12  7 12  0 15  4  7 14 10  1  0 15  2]
Step:  0
My rank is  0  and my des is  1
Rank  0   0 : Initial local_array:  [ 4  9 11 12]
Rank  0   0 : Received local_tmp: [ 0  7 12 15]
Step:  0
My rank is  1  and my des is  0
Rank  1   0 : Initial local_array:  [ 0  7 12 15]
Rank  1   0 : Received local_tmp: [ 4  9 11 12]
Step:  0
My rank is  2  and my des is  3
Step:  0
My rank is  3  and my des is  2
Rank  3   0 : Initial local_array:  [ 0  1  2 15]
Rank  3   0 : Received local_tmp: [ 4  7 10 14]
Rank  2   0 : Initial local_array:  [ 4  7 10 14]
Rank  2   0 : Received local_tmp: [ 0  1  2 15]
Rank  2   0 : Retained portions:  [0 1 2 4]
Step:  1
My rank is  2  and my des is  1
Rank  2   1 : Initial local_array:  [0 1 2 4]
Rank  3   0 : Retained portions:  [ 7 10 14 15]
Step:  1
Step:  2
My rank is  3  and my des is  2
Rank  0   0 : Retained portions:  [0 4 7 9]
Step:  1
Step:  2
My rank is  0  and my des is  1
Rank  0   2 : Initial local_array:  [0 4 7 9]
Rank  0   2 : Received local_tmp: [0 1 2 4]
Rank  0   2 : Retained portions:  [0 0 1 2]
Step:  3
Rank  1   0 : Retained portions:  [11 12 12 15]
Step:  1
My rank is  1  and my des is  2
Rank  1   1 : Initial local_array:  [11 12 12 15]
Rank  1   1 : Received local_tmp: [0 1 2 4]
Rank  1   1 : Retained portions:  [0 1 2 4]
Step:  2
My rank is  1  and my des is  0
Rank  1   2 : Initial local_array:  [0 1 2 4]
Rank  1   2 : Received local_tmp: [0 4 7 9]
Rank  1   2 : Retained portions:  [4 4 7 9]
Step:  3
My rank is  1  and my des is  2
Rank  1   3 : Initial local_array:  [4 4 7 9]
Rank  2   1 : Received local_tmp: [11 12 12 15]
Rank  2   1 : Retained portions:  [11 12 12 15]
Step:  2
My rank is  2  and my des is  3
Rank  2   2 : Initial local_array:  [11 12 12 15]
Rank  2   2 : Received local_tmp: [ 7 10 14 15]
Rank  3   2 : Initial local_array:  [ 7 10 14 15]
Rank  3   2 : Received local_tmp: [11 12 12 15]
Rank  3   2 : Retained portions:  [12 14 15 15]
Step:  3
Rank  2   2 : Retained portions:  [ 7 10 11 12]
Step:  3
My rank is  2  and my des is  1
Rank  2   3 : Initial local_array:  [ 7 10 11 12]
Rank  2   3 : Received local_tmp: [4 4 7 9]
Rank  2   3 : Retained portions:  [ 9 10 11 12]
Rank  1   3 : Received local_tmp: [ 7 10 11 12]
Rank  1   3 : Retained portions:  [4 4 7 7]
[ 0  0  1  2  4  4  7  7  9 10 11 12 12 14 15 15]
```

# Merge Sort

- A classical sequential sorting algorithm using divide-and-conquer approach
- Unsorted list first divided into half. Each half is again divided into two. Continued until individual numbers obtained.
- Then pairs of numbers combined (merged) into sorted list of two numbers.
- Pairs of these lists of four numbers are merged into sorted lists of eight numbers.
- This is continued until the one fully sorted list is obtained.



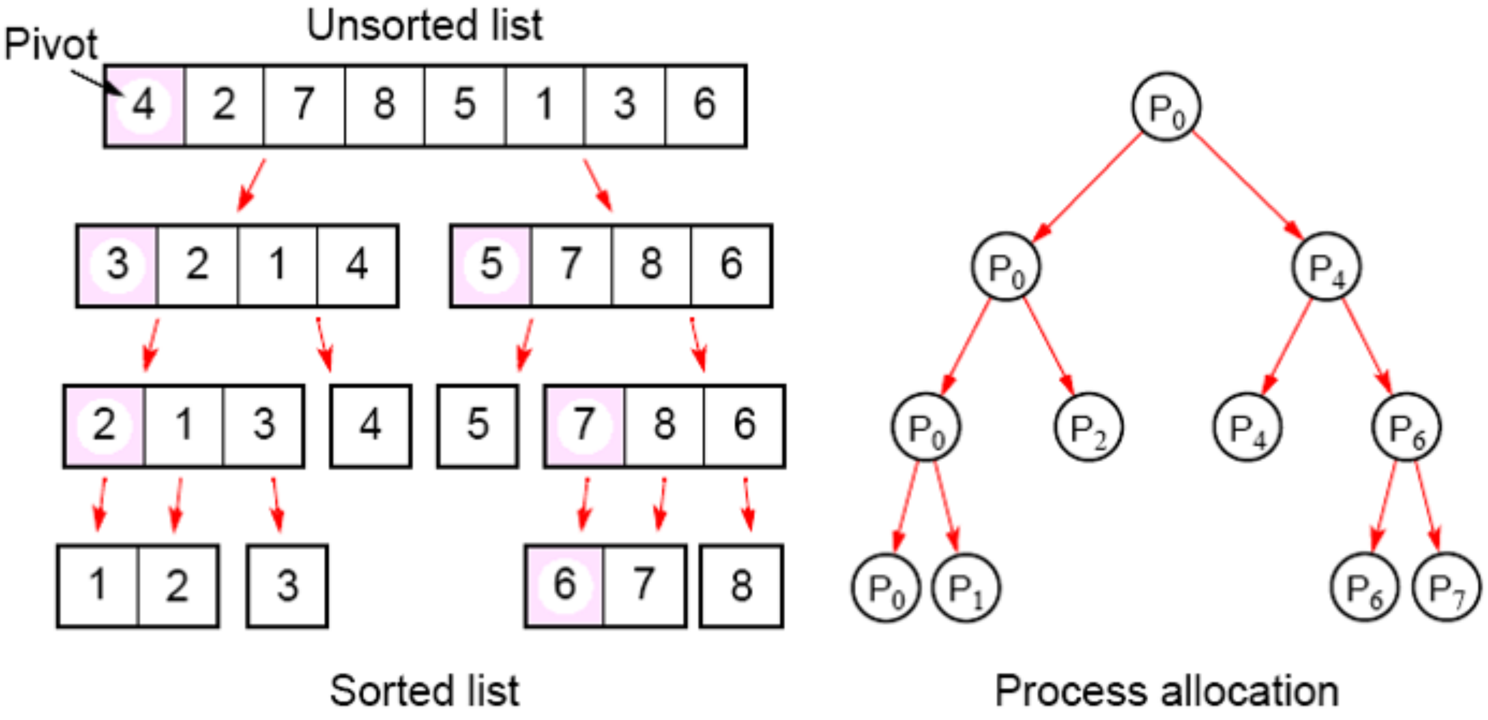
# merge.py (parallel)

```python
#!/usr/bin/env python
# merge.py
import numpy as np
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank(); size = comm.Get_size(); status = MPI.Status();
N = 16
unsorted = np.zeros(N, dtype="int")
final_sorted = np.zeros(N, dtype="int")
local_array = np.zeros(int(N / size), dtype="int")
local_tmp = np.zeros(int(N / size), dtype="int")
local_remain = np.zeros(2 * int(N / size), dtype="int")

if rank == 0:
    unsorted = np.random.randint(low=0,high=N,size=N)
    print (unsorted)
comm.Scatter(unsorted, local_array, root = 0)

local_array.sort()

step = size / 2
print ("Rank: ", rank)
while (step >= 1):
    if (rank >= step and rank < step * 2):
        comm.Send(local_array, rank - step, tag = 0)
    elif (rank < step):
        local_tmp = np.zeros(local_array.size, dtype="int")
        local_remain = np.zeros(2 * local_array.size, dtype="int")
        comm.Recv(local_tmp, rank + step, tag = 0)
        i = 0 #local_array counter
        j = 0 # local_tmp counter
        for k in range (0, 2 * local_array.size):
            if (i >= local_array.size):
                local_remain[k] = local_tmp[j]
                j += 1
            elif (j >= local_array.size):
                local_remain[k] = local_array[i]
                i += 1
            elif (local_array[i] > local_tmp[j]):
                local_remain[k] = local_tmp[j]
                j += 1
            else:
                local_remain[k] = local_array[i]
                i += 1
        print ("Step: ", step)
        print ("local array: ", local_array)
        print ("local tmp: ", local_tmp)
        print ("local remain: ", local_remain)
        local_array = local_remain
    step = step / 2

if (rank  == 0):
    print (local_array)
```

```
[jin6@node0378 13-parallel-sorting]$ mpirun -np 4 --mca mpi_cuda_support 0 python merge.py
[ 1 10 10  2 14  4 12  5  0  8 11  3  5 12 15  4]
Rank:  0
Step:  2.0
local array:  [ 1  2 10 10]
local tmp:  [ 0  3  8 11]
local remain:  [ 0  1  2  3  8 10 10 11]
Rank:  1
Step:  2.0
local array:  [ 4  5 12 14]
Rank:  2
Rank:  3
Step:  1.0
local array:  [ 0  1  2  3  8 10 10 11]
local tmp:  [ 4  4  5  5 12 12 14 15]
local remain:  [ 0  1  2  3  4  4  5  5  8 10 10 11 12 12 14 15]
[ 0  1  2  3  4  4  5  5  8 10 10 11 12 12 14 15]
local tmp:  [ 4  5 12 15]
local remain:  [ 4  4  5  5 12 12 14 15]
```

# Quick Sort

- Very popular sequential sorting algorithm that performs well with average sequential time complexity of O(nlogn).
- First list divided into two sublists.
- All numbers in one sublist arranged to be smaller than all numbers in other sublist.
- Achieved by first selecting one number, called a pivot, against which every other number is compared. If number less than pivot, it is placed in one sublist, otherwise, placed in other sublist.
- Pivot could be any number in list, but often first number chosen. Pivot itself placed in one sublist, or separated and placed in its final position.



# quicksort.py (parallel)

```python
#!/usr/bin/env python
# quicksort.py
import numpy as np
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank(); size = comm.Get_size(); status = MPI.Status();
N = 16
HAS = 1
HASNOT = 0
unsorted = np.zeros(N, dtype="int")
final_sorted = np.zeros(N, dtype="int")
local_array = None
local_tmp = None
local_tmp_size = np.zeros(1,dtype="int")

if rank == 0:
    unsorted = np.random.randint(low=0,high=N,size=N)
    print ("Unsorted array ", unsorted)
    local_array = unsorted

distance = size / 2
print ("Rank: ", rank)
while (distance >= 1):
    if (rank % distance == 0 and (rank / distance) % 2 == 0):
        print ("Rank ", rank, " send to rank ", int(rank + distance))
        if (local_array is not None):
            if local_array.size == 1 or np.unique(local_array).size == 1:
                comm.Send(local_array[0], dest = rank + distance, tag = HASNOT)
            else:
            #    print ("median is ", np.median(local_array))
                local_tmp = local_array[local_array > np.median(local_array)]
                comm.Send(np.full(shape = 1, fill_value = local_tmp.size, dtype="int"), dest = rank + distance, tag = HAS)
                comm.Send(local_tmp, dest = rank + distance, tag = HAS)
                local_array = local_array[local_array <= np.median(local_array)]
        else:
            comm.Send(np.zeros(1,dtype="int"), rank + distance, tag = HASNOT)
    elif (rank % distance == 0 and (rank / distance) % 2 == 1):
        comm.Recv(local_tmp_size, source = rank - distance, tag = MPI.ANY_TAG, status = status)
        if status.Get_tag() == HASNOT:
            continue
        else:
            local_array = np.zeros(local_tmp_size[0], dtype="int")
            comm.Recv(local_array, source = rank - distance, tag = MPI.ANY_TAG, status = status)
    distance /= 2
#    print (local_array)

local_array.sort()
print ("Local array at rank ", rank, ": ", local_array)
```

```
[jin6@node0378 13-parallel-sorting]$ mpirun -np 4 --mca mpi_cuda_support 0 python quicksort.py
Unsorted array  [ 9 12 12  5  7  3  3  3  2 14  2  7  4  8 10  6]
Rank:  0
Rank  0  send to rank  2
median is  6.5
Rank  0  send to rank  1
median is  3.0
Local array at rank  0 :  [2 2 3 3 3]
Rank:  1
Local array at rank  1 :  [4 5 6]
Rank:  2
Rank  2  send to rank  3
median is  9.5
Local array at rank  2 :  [7 7 8 9]
Rank:  3
Local array at rank  3 :  [10 12 12 14]
```

# Sorting Conclusions

- Computational time complexity using n processors
- Odd-even transposition sort - $O(n)$
- Parallel merge sort - $O(n)$ but unbalanced processor load and communication
- Parallel quicksort - $O(n)$ but unbalanced processor load, and communication, can degenerate to $O(n^2)$