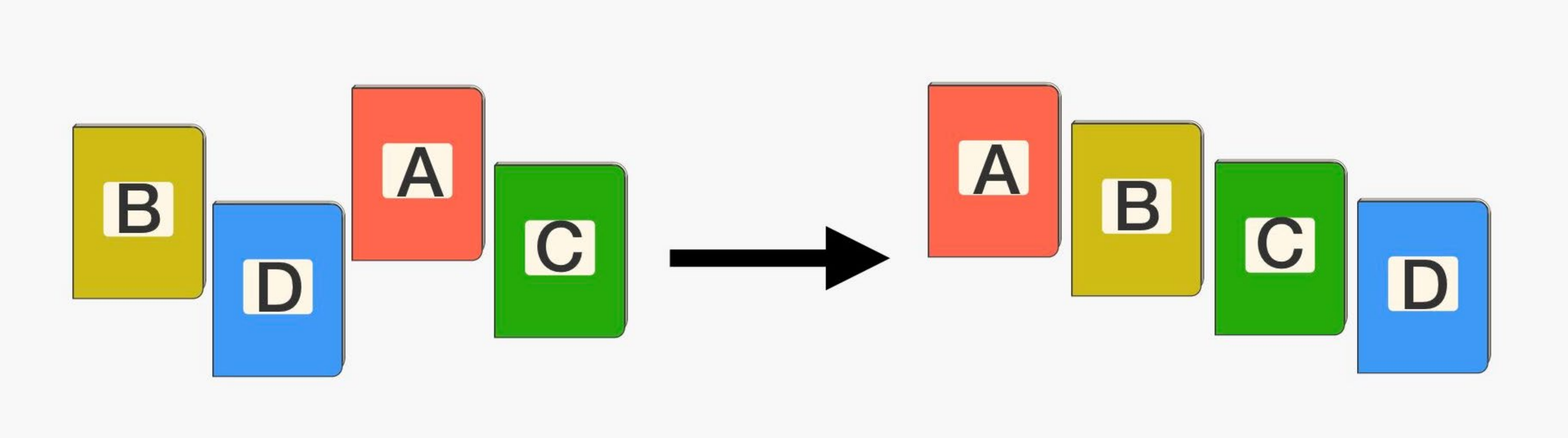
PARALLEL QUICKSORT

Student: Chia Chen Chen
Advisor: Prof. Russ Miller

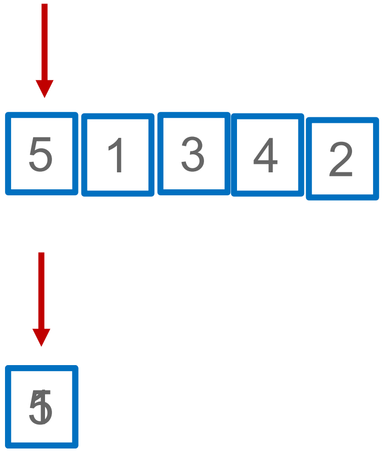
University at Buffalo The State University of New York

Sorting Algorithm

- A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements
- The comparison operator is used to decide the new order of element in the respective data structure.

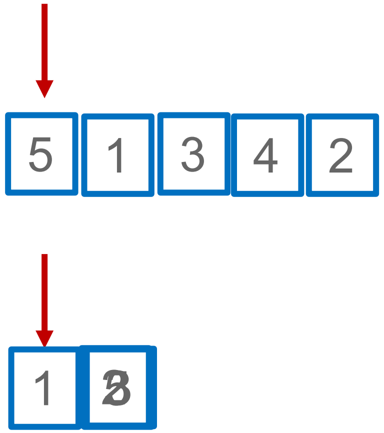


Sorting



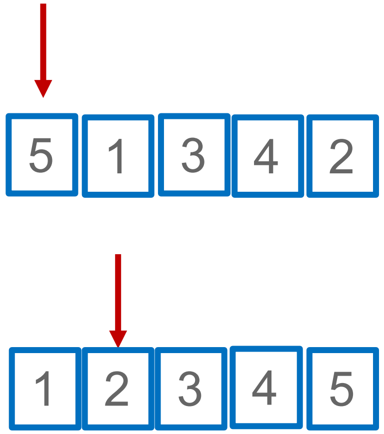
- A brute way is to start from the array beginning and go through every elements every time
- Store the first element in an empty array
- Iterating the array, search for a smaller and hasn’t been stored yet element

Sorting



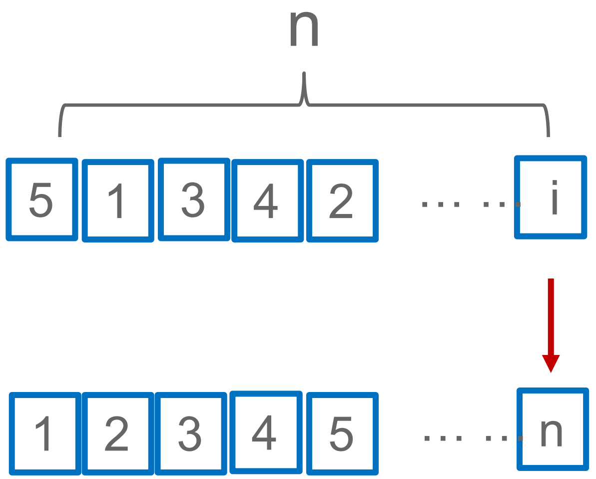
Then store the next element and continue the iteration until all the elements have been added

Sorting



Then store the next element and continue the iteration until all the elements have been added

Sorting



If the array has n elements

=>Running time: $O(n^2)$

Different Sorting Strategy

- Selection Sort
- Bubble Sort
- Recursive Bubble Sort
- Insertion Sort
- Etc.
- Recursive Insertion Sort
- Merge Sort
- Bitonic Sort
- Quick Sort

Quicksort

- QuickSort is a Divide and Conquer algorithm.
- It picks an element as pivot and partitions the given array around the picked pivot. There are different way of quicksort that pick pivot in different ways.

Always pick the first element as pivot.

Always pick the last element as pivot.

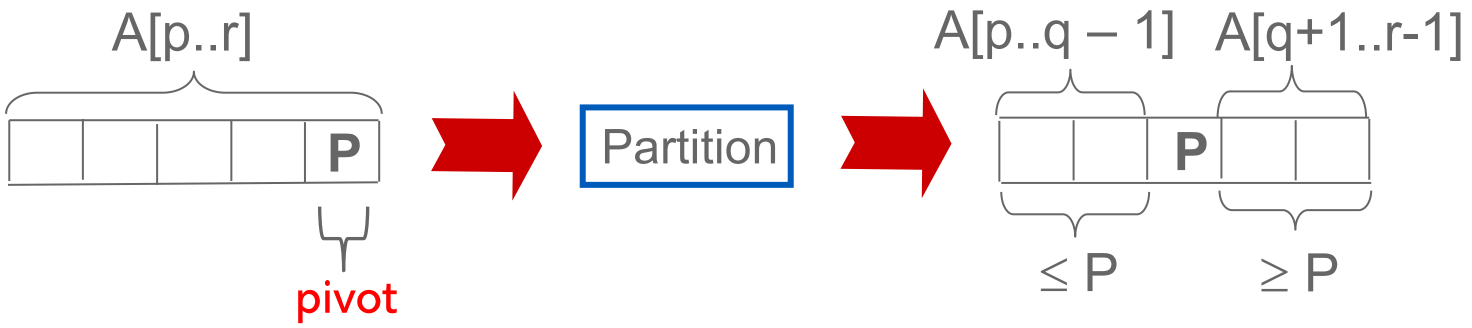
Pick a random element as pivot.

Pick median as pivot.

Quicksort Algorithm Design

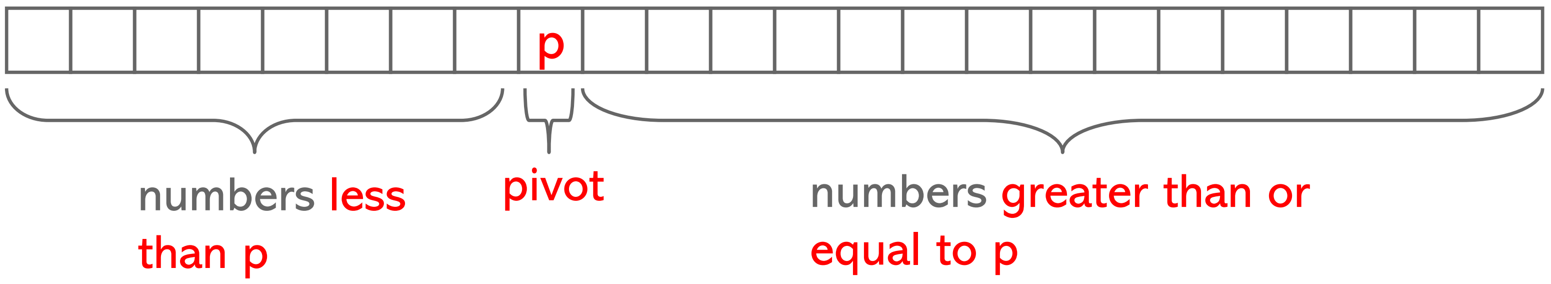
- **Divide:** Partition the array A[p..r] into two subarrays A[p..q–1] and A[q+1..r].
  - Each element in A[p..q-1] < A[q]
  - A[q] ≤ each element in A[q+1..r-1]
  - Index q is computed as part of the partitioning procedure

Partitioning in Quicksort



x: pivot
r: length of array

```
Partition(A, p, r):
    x, i  := A[r], p – 1;
    for j := p to r – 1 do
        if A[j]  <  x then
            i := i + 1;
            A[i] ↔ A[j]
    A[i + 1] ↔ A[r];
    return i + 1
```



QuickSort Algorithm Design

- **Divide:** Partition the array A[p..r] into two subarrays A[p..q–1] and A[q+1..r].
  - Each element in A[p..q-1] < A[q]
  - A[q] ≤ each element in A[q+1..r-1]
  - Index q is computed as part of the partitioning procedure
- **Conquer:** Sort the two subarrays by recursive calls to quicksort

Recursive call Quicksort

```
Quicksort(A, p, r):
    if p < r then
        q := Partition(A, p, r);
        Quicksort(A, p, q – 1);
        Quicksort(A, q + 1, r)
```

QuickSort A[left...right]:

1. if left < right:
    1. Partition A[left...right] such that: //pivot = A[q]
        all A[left...q-1] elements are less than A[q],
        all A[q+1...right] elements are >= A[q]
    2. Quicksort A[left...q-1]
    3. Quicksort A[q+1...right]
2. Terminate

QuickSort Algorithm Design

- **Divide:** Partition the array A[p..r] into two subarrays A[p..q–1] and A[q+1..r].
  - ➢Each element in A[p..q-1] < A[q]
  - ➢A[q] $\leq$ each element in A[q+1..r-1]
  - ➢Index q is computed as part of the partitioning procedure

- **Conquer:** Sort the two subarrays by recursive calls to quicksort

- **Combine:** The subarrays are sorted in place, no work is needed to combine them

Quicksort

- QuickSort is a Divide and Conquer algorithm.
- It picks an element as pivot and partitions the given array around the picked pivot. There are different way of quicksort that pick pivot in different ways.

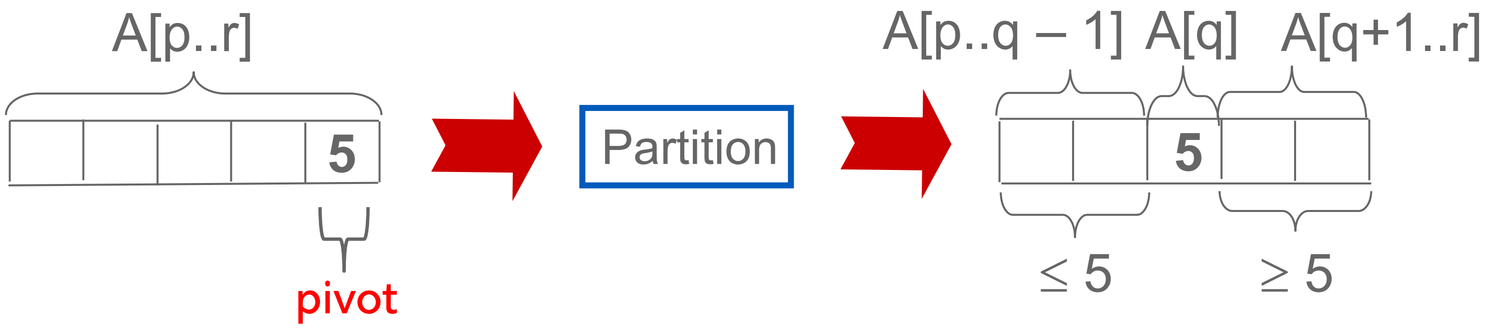
Always pick the first element as pivot.

Always pick the last element as pivot

Pick a random element as pivot.

Pick median as pivot.

Quicksort Pseudocode




```
Quicksort(A, p, r):
    if p < r then
        q := Partition(A, p, r);
        Quicksort(A, p, q – 1);
        Quicksort(A, q + 1, r)
```

```
Partition(A, p, r):
    x, i := A[r], p – 1;
    for j := p to r – 1 do
        if A[j] < x then
            i := i + 1;
            A[i] ↔ A[j]
    A[i + 1] ↔ A[r];
    return i + 1
```

x: pivot
r: length of array

Example

	p									r	
Initially:	2	5	8	3	9	4	1	7	10	**6**	pivot (x) = 6
	ij										
Iteration:	2	5	8	3	9	4	1	7	10	**6**	
	i	j									
	2	5	8	3	9	4	1	7	10	**6**	
		i	j								
	2	5	8	3	9	4	1	7	10	**6**	
		i		j							
	2	5	3	8	9	4	1	7	10	**6**	
			i		j						

```
Partition(A, p, r):
    x, i := A[r], p – 1;
    for j := p to r – 1 do
        if A[j] < x then
            i := i + 1;
            A[i] ↔ A[j]
    A[i + 1] ↔ A[r];
    return i + 1
```

Example

```
                     2  5  3  8  9  4  1  7  10  6
                           i     j
                     2  5  3  8  9  4  1  7  10  6
                           i        j
                     2  5  3  4  9  8  1  7  10  6
                              i        j
                     2  5  3  4  1  8  9  7  10  6
                                 i        j
                     2  5  3  4  1  8  9  7  10  6
                                 i            j
                     2  5  3  4  1  8  9  7  10  6
                                 i                j
After final swap     2  5  3  4  1  6  9  7  10  8
                                 i                j
Next oteratopm:      2  5  3  4  1  6  9  7  10  8
                     ij                ij
Final:               1  2  3  4  5  6  7  8  9  10
```

```
Partition(A, p, r):
    x, i := A[r], p – 1;
    for j := p to r – 1 do
        if A[j] < x then
            i := i + 1;
            A[i] ↔ A[j]
    A[i + 1] ↔ A[r];
    return i + 1
```

Quicksort Running time

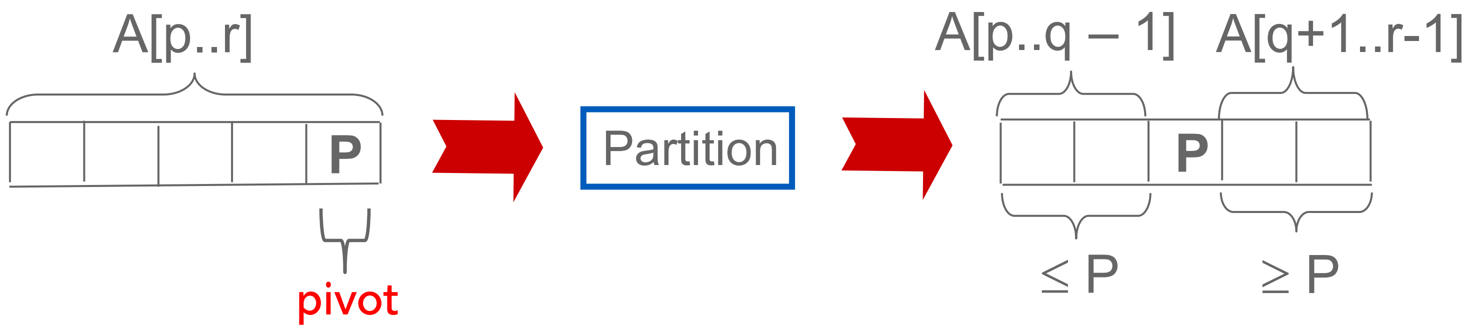
- Quicksort is usually $O(n\lg n)$
- Worst case: if the array is sorted to begin with, the running time will be $O(n^2)$

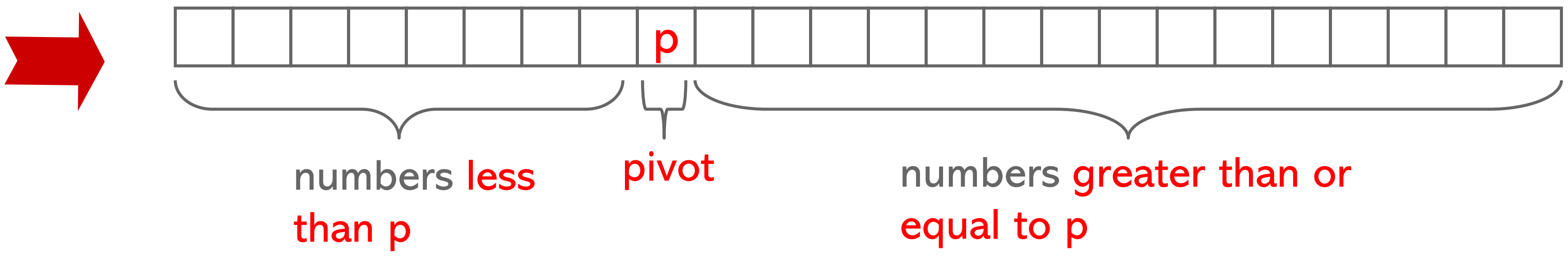
  Array is already sorted in the same order.

  Array is already sorted in reverse order.

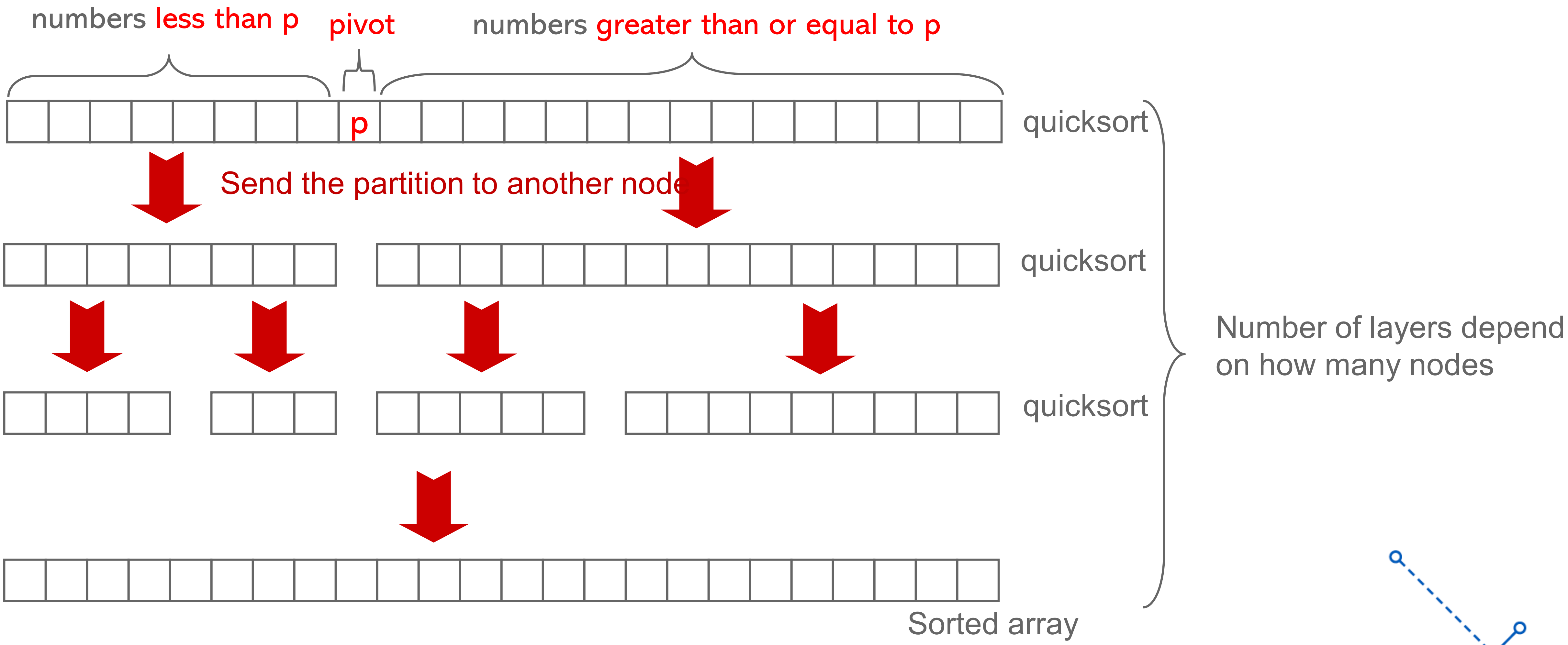
  All elements are the same

Partitioning in Quicksort

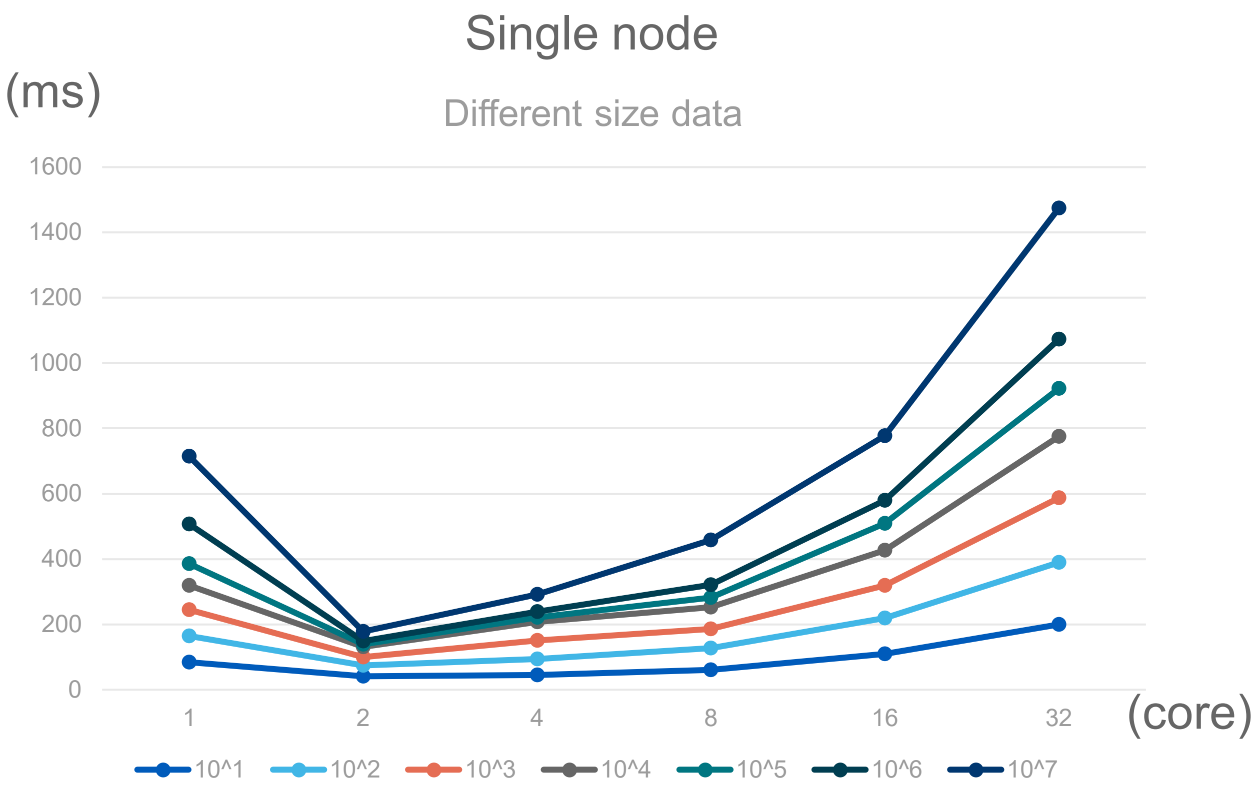




Parallel quicksort

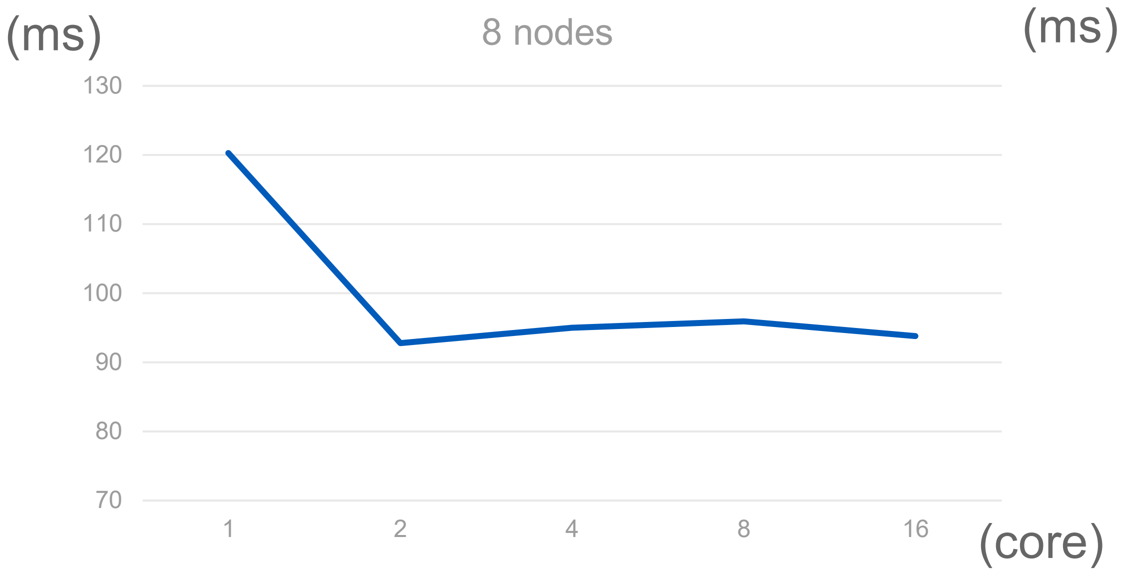


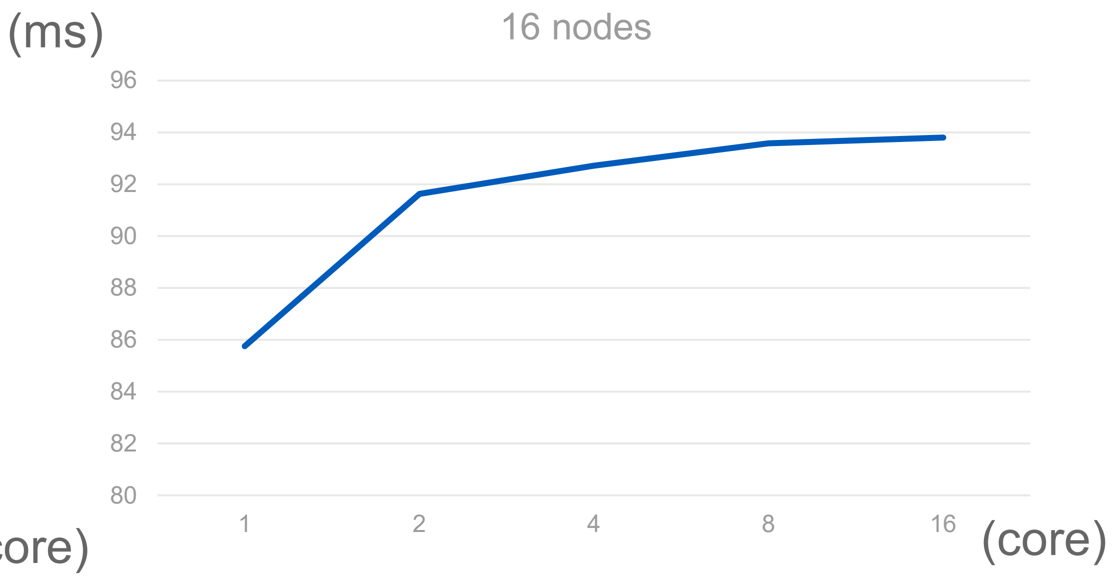
Results


Single node
(ms)
Different size data
1600
1400
1200
1000
800
600
400
200
0
1
2
4
8
16
32
(core)
10^1
10^2
10^3
10^4
10^5
10^6
10^7

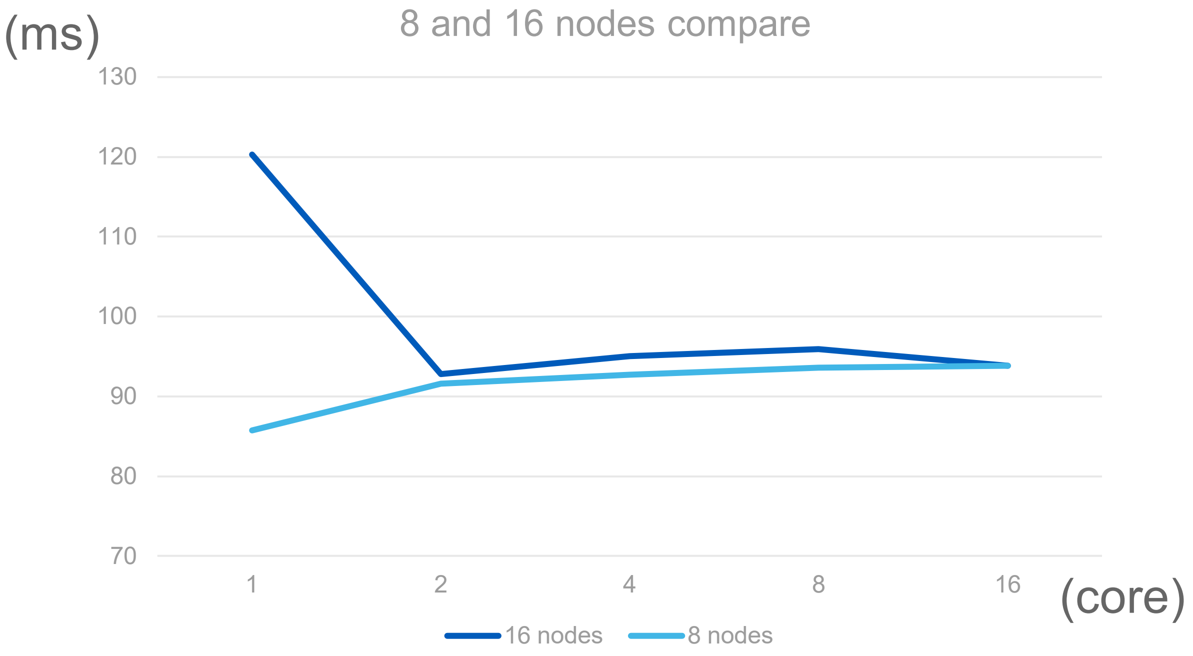
Results

Results


(ms)
8 nodes
130
120
110
100
90
80
70
1
2
4
8
16
(core)


(ms)
16 nodes
96
94
92
90
88
86
84
82
80
1
2
4
8
16
(core)

Results



	1	2	4	8	16
8	120.3148	92.78802	95.00372	95.9435	93.82286
16	85.76758	91.6233	92.7243	93.58568	93.80679

Thank you

Reference

- Algorithms Sequential & Parallel: A Unified Approach (Dr. Russ Miller, Dr. Laurence Boxer)
- Python Program for QuickSort – GeeksforGeeks
- Sorting Algorithms – GeeksforGeeks
- MPI for Python — MPI for Python 3.1.3 documentation (mpi4py.readthedocs.io)