Parallel and Distributed Computing

CS3006

Lecture 16

MPI-III

24th April 2024

Dr. Rana Asif Rehman

Sorting in Parallel Era

Sorting: Overview

- One of the most commonly used and well-studied Algorithms.
- Sorting can be *comparison-based* or *non-comparison-based*.
- The fundamental operation of comparison-based sorting is *compare-exchange*.
- The lower bound on any comparison-based sort of n numbers is $\Theta(n\log n)$.
- Let's explore a comparison-based sorting algorithm.

Sorting: Basics

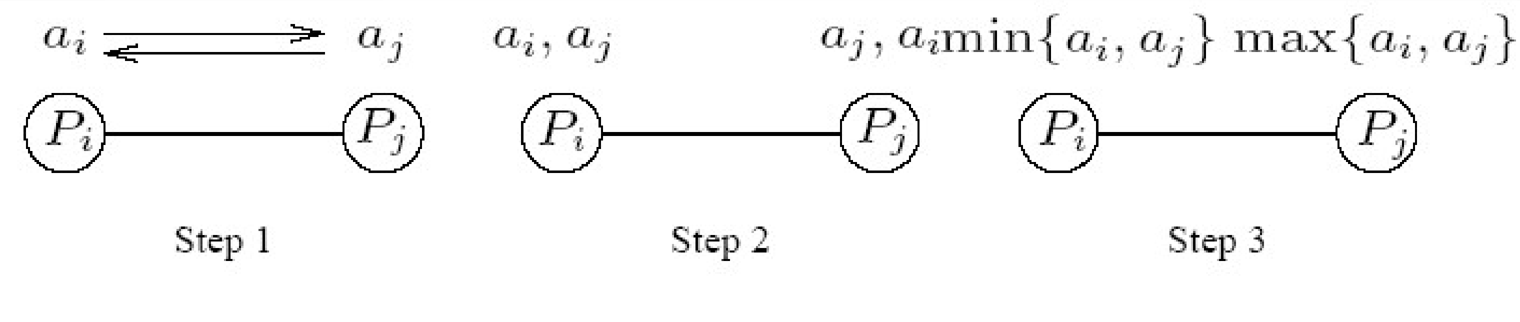
→ What is a parallel sorted sequence?

→ Where are the input and output lists stored?

Answers:

- We assume that the input and output lists are distributed.
- The sorted list is partitioned with the property that each partitioned list is sorted and each element in processor P_i's list is less than that in P_j's list if $i < j$.

Sorting: Parallel Compare Exchange Operation

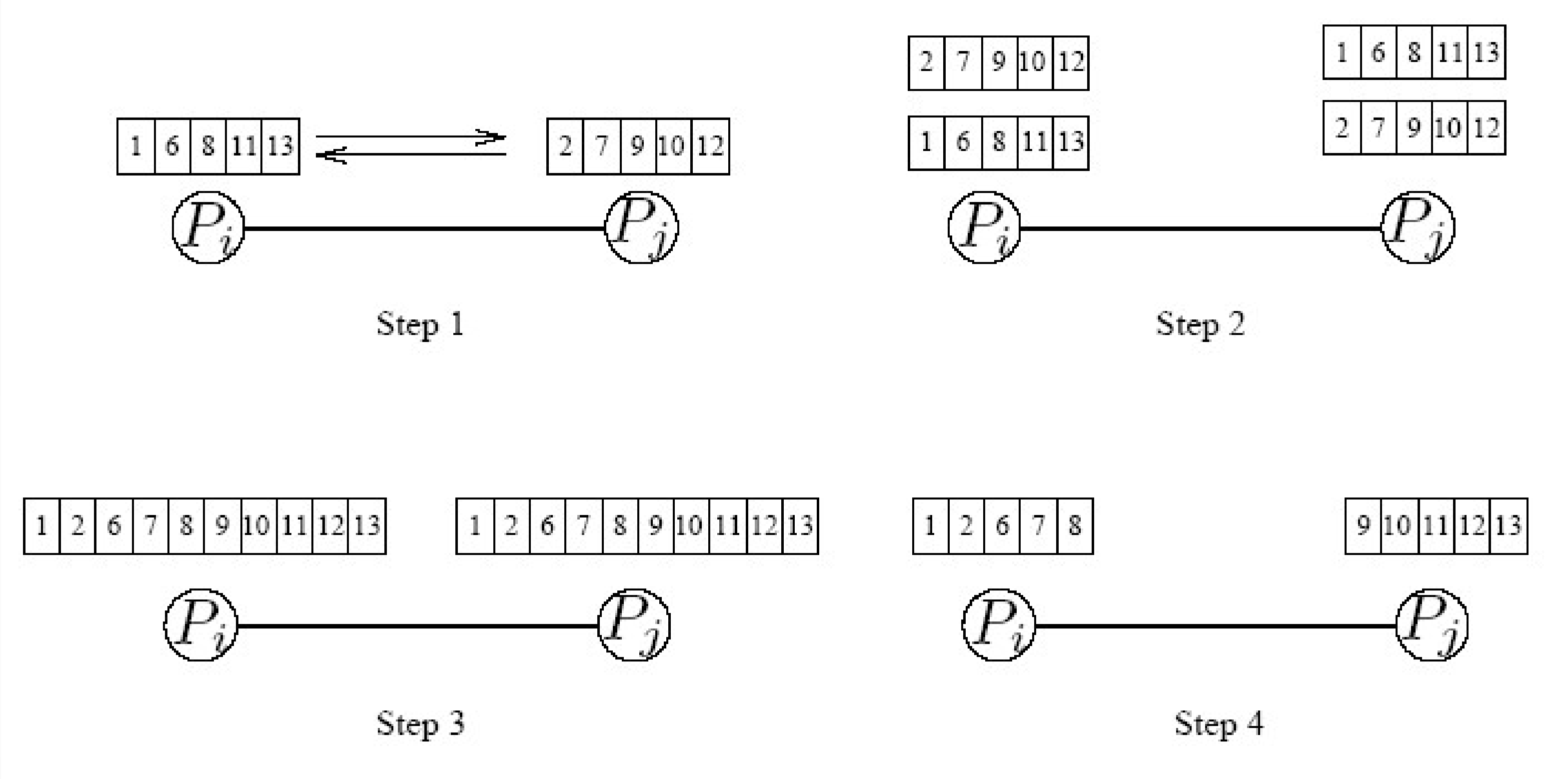


A parallel compare-exchange operation. Processes P_i and P_j send their elements to each other. Process P_i keeps $\min\{a_i, a_j\}$, and P_j keeps $\max\{a_i, a_j\}$.

Sorting: Parallel Compare Exchange Operation [cost estimation]

- If each processor has one element, the compare exchange operation stores the smaller element at the processor with smaller id. This can be done in $t_s + t_w$ time.
- If we have more than one element per processor, we call this operation a compare split. Assume each of two processors have n/p elements.
- After the compare-split operation, the smaller n/p elements are at processor P_i and the larger n/p elements at P_j, where $i < j$.
- The time for a compare-split operation is $(t_s + t_w n/p)$, assuming that the two partial lists were initially sorted.
  - Note that this time is only accounting communication costs. Computation and memory complexities are separate things.

Sorting: Parallel Compare Exchange



A compare-split operation. Each process sends its block of size n/p to the other process. Each process merges the received block with its own block and retains only the appropriate half of the merged block. In this example, process P_i retains the smaller elements and process P_i retains the larger elements.

Bubble Sort and its Variant

The sequential bubble sort algorithm compares and exchanges adjacent elements in the sequence to be sorted:

```
1.    procedure BUBBLE_SORT(n)
2.    begin
3.        for i := n - 1 downto 1 do
4.            for j := 1 to i do
5.                compare-exchange(a_j, a_{j+1});
6.    end BUBBLE_SORT
```

Sequential bubble sort algorithm.

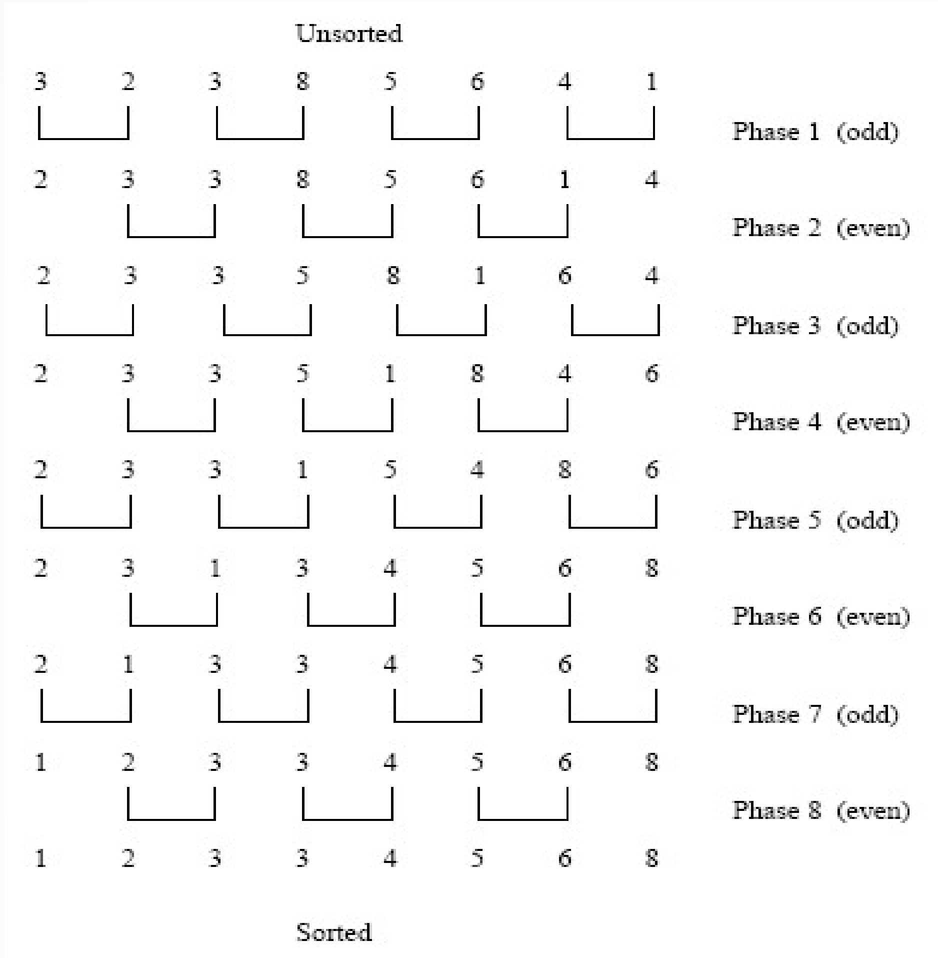
First pass

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 and its Variant

- The complexity of bubble sort is $\Theta(n^2)$.
- Bubble sort is difficult to parallelize since the algorithm has no concurrency.
- A simple variant, though, uncovers the concurrency.

Bubble Sort [Odd-Even Transposition]



Sorting $n = 8$ elements, using the odd-even transposition sort algorithm. During each phase, at most 8 elements are compared. **[This according to sequential algorithm]**

Bubble Sort [Odd-Even Transposition]


```
procedure ODD-EVEN(n)
begin
    for i := 1 to n do
    begin
        if i is odd then
            for j := 0 to n/2 - 1 do
                compare-exchange(a_{2j+1}, a_{2j+2});
        if i is even then
            for j := 1 to n/2 - 1 do
                compare-exchange(a_{2j}, a_{2j+1});
    end for
end ODD-EVEN
```

Sequential odd-even sort algorithm.

Odd-Even Sort (Seq. Complexity)

- After n phases of odd-even exchanges, the sequence is sorted.
- Each phase of the algorithm (either odd or even) requires $\Theta(n)$ comparisons.
- Serial complexity is $\Theta(n^2)$.

Parallel Odd-Even Sort

Time / 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

Parallel time complexity: $T_{par} = O(n)$ (for P=n)

* https://www.slideshare.net/richakumari37266/parallel-sorting-algorithm

Parallel Odd-Even Sort

Algorithm Through Observations:

1. There are total **P** phases/steps. Where P is number of processes
2. **For even phases**
   1. If 'myrank' is even → Communication partner is ('myrank'+1)
   2. If 'myrank' is odd → Communication partner is ('myrank' - 1)
3. **For odd phases:**
   1. If 'myrank' is even → Communication partner is ('myrank' - 1)
   2. If 'myrank' is odd → Communication partner is ('myrank'+1)
4. Communication partners remain constant
5. If 'myrank' is less-than the partner, then keep lower values in compare-split-operation

Parallel Odd-Even Sort

Complexity when *n*==P

- Consider the one item per processor case.
- There are P iterations, in each iteration, each processor does one compare-exchange.
- The parallel run time of this formulation is $\Theta(n)$.
- Parallel run time means computation performed by each of the processors in parallel.

Parallel Odd-Even Sort

Complexity when *n* > P

- Consider a block of n/p elements per processor.
- The first step is a local sort.
- In each subsequent step, the compare exchange operation is replaced by the compare split operation.
- The parallel run time of the formulation is:

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{local sort}} + \overbrace{\Theta(n)}^{\text{comparisons}} + \overbrace{\Theta(n)}^{\text{communication}}.$$

comm. steps for a single process

Parallel Odd-Even Sort Implementation


```
#include <stdlib.h>
#include <mpi.h> /* Include MPI's header file */

main(int argc, char *argv[])
{
 int n; /* The total number of elements to be sorted */
 int npes; /* The total number of processes */
 int myrank; /* The rank of the calling process */
 int nlocal; /* The local number of elements, and the array that stores them */
int *elmnts; /* The array that stores the local elements */
 int *relmnts; /* The array that stores the received elements */
int oddrank; /* The rank of the partner during odd-phase communication */
int evenrank; /* The rank of the partner during even-phase communication */
int *wspace; /* Working space during the compare-split operation */
```

Parallel Odd-Even Sort Implementation


```
/* Initialize MPI and get system information */
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

n = atoi(argv[1]);
nlocal = n/npes; /* Compute the number of elements to be stored locally. */

/* Allocate memory for the various arrays */
elmnts  = (int *)malloc(nlocal*sizeof(int));
relmnts = (int *)malloc(nlocal*sizeof(int));
wspace  = (int *)malloc(nlocal*sizeof(int));

/* Fill-in the elmnts array with random elements */
srandom(myrank);
for (i=0; i<nlocal; i++)
  elmnts[i] = random();

/* Sort the local elements using the built-in quicksort routine */
qsort(elmnts, nlocal, sizeof(int), IncOrder);
```

Parallel Odd-Even Sort Implementation

Determining communication partner during Even and odd steps of the algorithm.

- If my partner is out of bounds, then set it to NULL process.

```
if (myrank%2 == 0) {
  oddrank  = myrank-1;
  evenrank = myrank+1;
}
else {
  oddrank  = myrank+1;
  evenrank = myrank-1;
}

/* Set the ranks of the processors at the end of the linear */
if (oddrank == -1 || oddrank == npes)
  oddrank = MPI_PROC_NULL;
if (evenrank == -1 || evenrank == npes)
  evenrank = MPI_PROC_NULL;
```

Parallel Odd-Even Sort Implementation

P Steps for actual algorithm

```
  /* Get into the main loop of the odd-even sorting algorithm */
  for (i=0; i<npes-1; i++) {
    if (i%2 == 1) /* Odd phase */
      MPI_Sendrecv(elmnts, nlocal, MPI_INT, oddrank, 1, relmnts,
          nlocal, MPI_INT, oddrank, 1, MPI_COMM_WORLD, &status);
    else /* Even phase */
      MPI_Sendrecv(elmnts, nlocal, MPI_INT, evenrank, 1, relmnts,
          nlocal, MPI_INT, evenrank, 1, MPI_COMM_WORLD, &status);

    CompareSplit(nlocal, elmnts, relmnts, wspace,
                 myrank < status.MPI_SOURCE);
  }

  free(elmnts); free(relmnts); free(wspace);
  MPI_Finalize();
}
```

Parallel Odd-Even Sort Implementation

Compare-Split function

```
CompareSplit(int nlocal, int *elmnts, int *relmnts, int *wspace,
             int keepsmall)
{
  int i, j, k;

  for (i=0; i<nlocal; i++)
    wspace[i] = elmnts[i]; /* Copy the elmnts array into the wspace array */

  if (keepsmall) { /* Keep the nlocal smaller elements */
    for (i=j=k=0; k<nlocal; k++) {
      if (j == nlocal || (i < nlocal && wspace[i] < relmnts[j]))
        elmnts[k] = wspace[i++];
      else
        elmnts[k] = relmnts[j++];
    }
  }
  else { /* Keep the nlocal larger elements */
    for (i=k=nlocal-1, j=nlocal-1; k>=0; k--) {
      if (j == 0 || (i >= 0 && wspace[i] >= relmnts[j]))
        elmnts[k] = wspace[i--];
      else
        elmnts[k] = relmnts[j--];
    }
  }
}
```

Parallel Odd-Even Sort Implementation

IncOrder function

```
/* The IncOrder function that is called by qsort is defined as follows */
int IncOrder(const void *e1, const void *e2)
{
  return (*((int *)e1) - *((int *)e2));
}
```

Questions

References

1. Kumar, V., Grama, A., Gupta, A., & Karypis, G. (2017). *Introduction to parallel computing*. Redwood City, CA: Benjamin/Cummings.