Sorting in Parallel

Why?

- Sorting, or rearranging a list of numbers into increasing (decreasing) order, is a **fundamental operation** that appears in many applications

Potential speedup?

- Best sequential sorting algorithms (mergesort and quicksort) have (respectively worst-case and average) time complexity of $\mathcal{O}(n \log(n))$
- The best we can aim with a parallel sorting algorithm using n processing units is thus a time complexity of $\mathcal{O}(n \log(n))/n = \mathcal{O}(\log(n))$
- But, in general, a realistic $\mathcal{O}(\log(n))$ algorithm with n processing units is a goal that is not easy to achieve with comparasion-based sorting algorithms

Compare-and-Exchange

An operation that forms the basis of several classical sequential sorting algorithms is the **compare-and-exchange (or compare-and-swap)** operation.

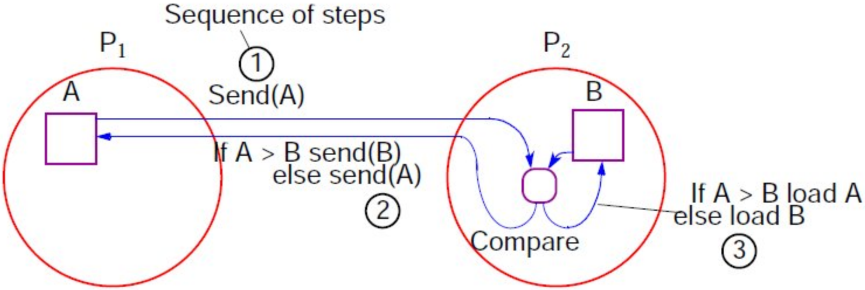
In a compare-and-exchange operation, two numbers, say A and B, are compared and if they are not ordered, they are exchanged. Otherwise, they remain unchanged.

```
if (A > B) { // sorting in increasing order
  temp = A;
  A = B;
  B = temp;
}
```

Question: how can compare-and-exchange be done in parallel?

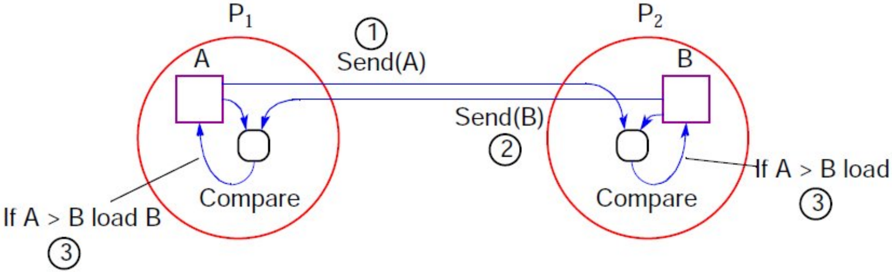
Parallel Compare-and-Exchange

Version 1 – P_1 sends A to P_2, which then compares A and B and sends back to P_1 the $min(A, B)$.



Parallel Compare-and-Exchange

Version 2 – P_1 sends A to P_2 and P_2 sends B to P_1, then both perform comparisons and P_1 keeps the $min(A, B)$ and P_2 keeps the $max(A, B)$.



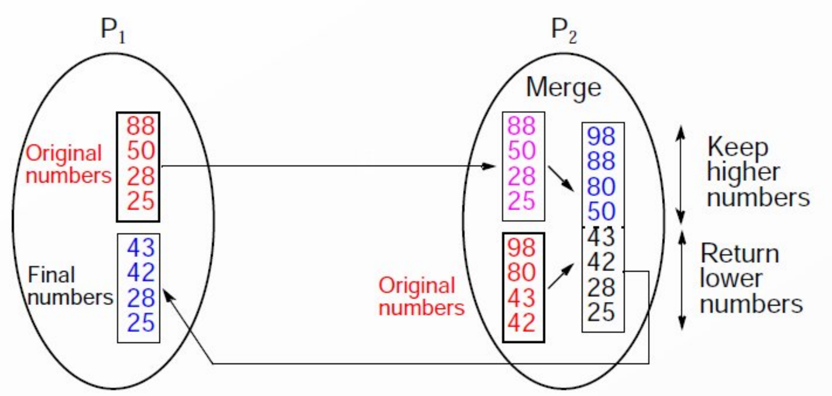
Data Partitioning

So far, we have assumed that there is one processing unit for each number, but normally there would be many more numbers (n) than processing units (p) and, in such cases, a list of n/p numbers would be assigned to each processing unit.

When dealing with lists of numbers, the **operation of merging two sorted lists** is a common operation in sorting algorithms.

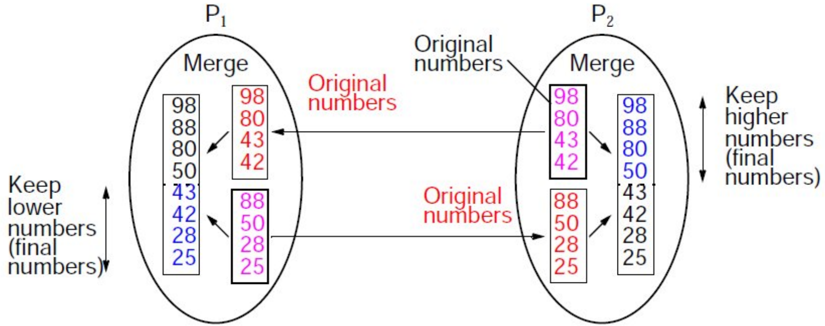
Parallel Merging

Version 1 – P_1 sends its list to P_2, which then performs the merge operation and sends back to P_1 the lower half of the merged list.



Parallel Merging

Version 2 – both processing units exchange their lists, then both perform the merge operation and P_1 keeps the lower half of the merged list and P_2 keeps the higher half of the merged list.



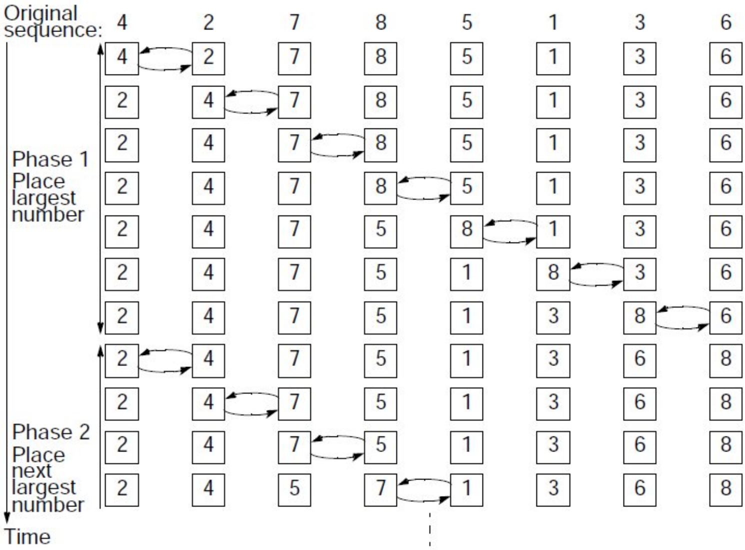
Bubble Sort

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**.

```
for (i = N - 1; i > 0; i--)
  for (j = 0; j < i; j++) {
    k = j + 1;
    if (a[j] > a[k]) {
      temp = a[j];
      a[j] = a[k];
      a[k] = temp;
    }
  }
```

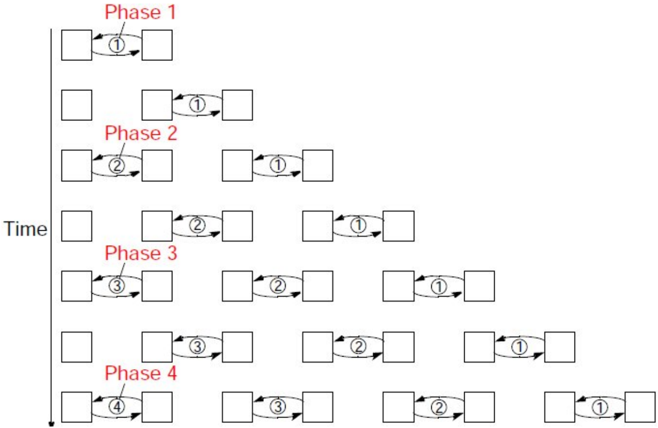
The total number of compare-and-exchange operations is $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$, which corresponds to a time complexity of $\mathcal{O}(n^2)$.

Bubble Sort


Original sequence: 4 2 7 8 5 1 3 6
4 2 7 8 5 1 3 6
2 4 7 8 5 1 3 6
2 4 7 8 5 1 3 6
2 4 7 8 5 1 3 6
2 4 7 5 8 1 3 6
2 4 7 5 1 8 3 6
2 4 7 5 1 3 8 6
2 4 7 5 1 3 6 8
2 4 7 5 1 3 6 8
2 4 7 5 1 3 6 8
2 4 5 7 1 3 6 8
Phase 1
Place largest number
Phase 2
Place next largest number
Time

Parallel Bubble Sort

A possible idea is to **run multiple iterations in a pipeline fashion**, i.e., start the bubbling action of the next iteration before the preceding iteration has finished in such a way that it does not overtakes it.



Odd-Even Transposition Sort

Odd-even transposition sort is a variant of bubble sort which operates in two alternating phases:

- **Even Phase:** even processes exchange values with right neighbors ($P_0 \leftrightarrow P_1$, $P_2 \leftrightarrow P_3$, ...)
- **Odd Phase:** odd processes exchange values with right neighbors ($P_1 \leftrightarrow P_2$, $P_3 \leftrightarrow P_4$, ...)

For sequential programming, odd-even transposition sort has no particular advantage over normal bubble sort. However, its parallel implementation corresponds to a time complexity of $\mathcal{O}(n)$.

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

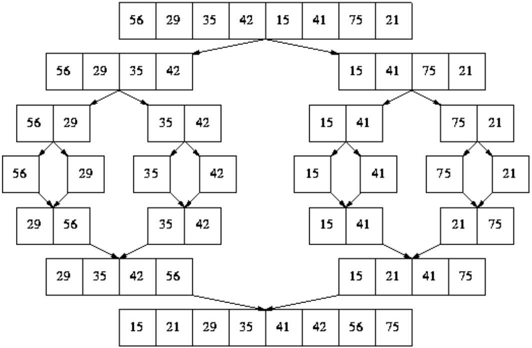
Time

Odd-Even Transposition Sort

```
rank = process_id();
A = initial_value();
for (i = 0; i < N; i++) {
  if (i % 2 == 0) {                         // even phase
    if (rank % 2 == 0) {                    // even process
      recv(B, rank + 1); send(A, rank + 1);
      A = min(A,B);
    } else {                                // odd process
      send(A, rank - 1); recv(B, rank - 1);
      A = max(A,B);
    }
  } else if (rank > 0 && rank < N - 1) { // odd phase
    if (rank % 2 == 0) {                    // even process
      recv(B, rank - 1); send(A, rank - 1);
      A = max(A,B);
    } else {                                // odd process
      send(A, rank + 1); recv(B, rank + 1);
      A = min(A,B);
    }
}
```

Mergesort

Mergesort is a classical sorting algorithm using a **divide-and-conquer** approach. The initial unsorted list is first divided in half, each half sublist is then applied the same division method until individual elements are obtained. Pairs of adjacent elements/sublists are then **merged into sorted sublists** until the one fully merged and sorted list is obtained.



Mergesort

Computations only occur when merging the sublists. In the worst case, it takes $2s - 1$ steps to merge two sorted sublists of size s. If we have $m = \frac{n}{s}$ sorted sublists in a merging step, it takes

$$\frac{m}{2}(2s - 1) = ms - \frac{m}{2} = n - \frac{m}{2}$$

steps to merge all sublists (two by two).

Since in total there are $\log(n)$ merging steps, this corresponds to a time complexity of $\mathcal{O}(n \log(n))$.

Parallel Mergesort

The idea is to **take advantage of the tree structure of the algorithm** to assign work to processes.