Chapter 9 (Secs. 9.1, 9.3, 9.4) Sorting Algorithms

A. Grama, A. Gupta, G. Karypis, and V. Kumar

To accompany the text "Introduction to Parallel Computing", Addison Wesley, 2003.

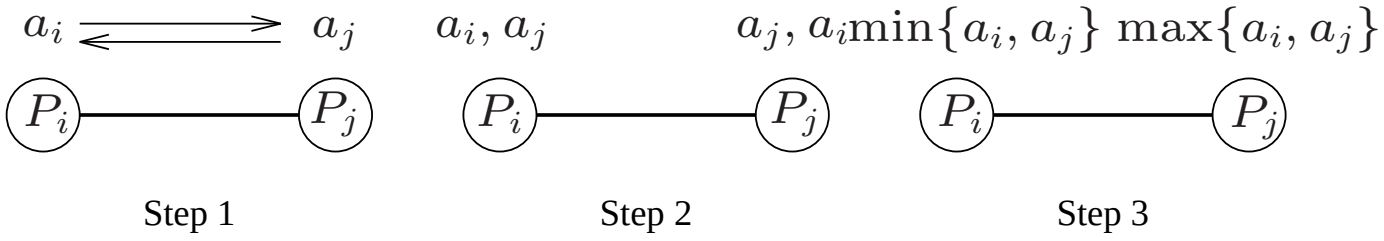
Topic Overview

- Issues in Sorting on Parallel Computers
- Bubble Sort and its Variants
- Quicksort

Sorting: Basics

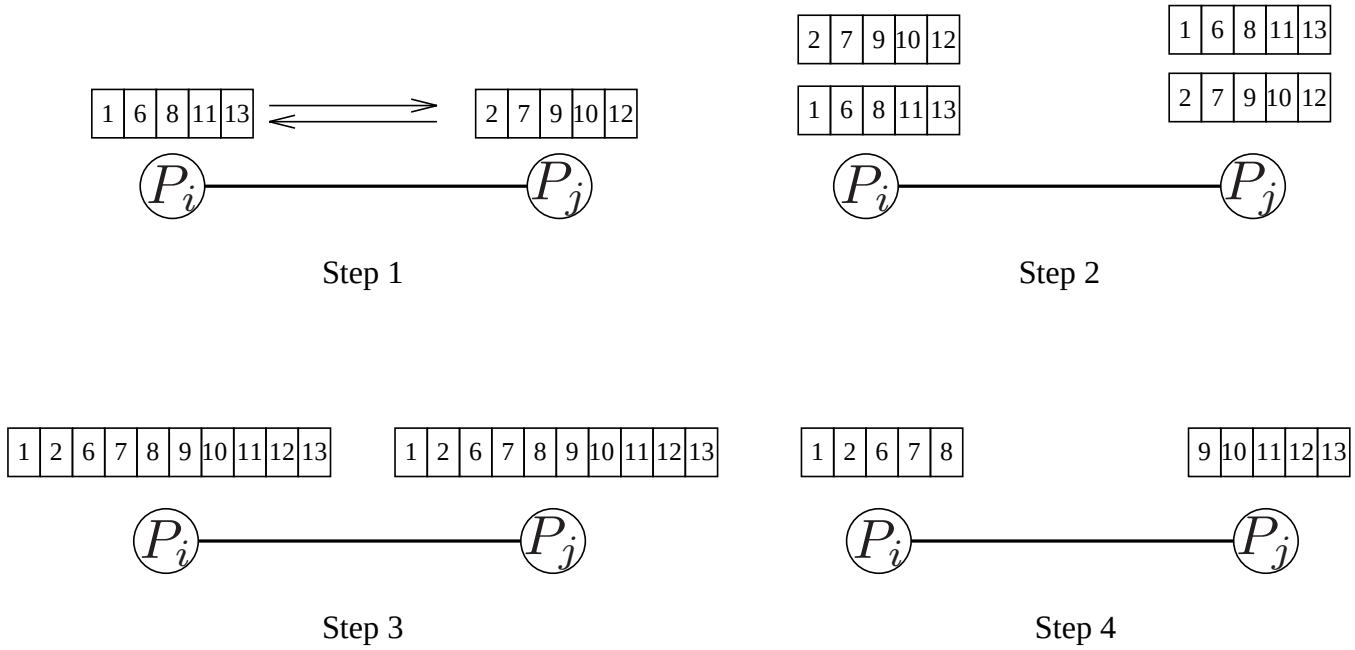
- One of the most commonly used and well-studied kernels.
- The fundamental operation of comparison-based sorting is *compare-exchange*.
- The lower bound on any comparison-based sort of n numbers is $\Theta(n \log n)$ on a serial computer.
- In case of parallel sorting, the sorted list is partitioned among a number of processors, such that (1) each sublist is sorted (2) for $i < j$, each element in processor P_i's sublist is less than those in P_j's sublist.

Sorting: Parallel Compare Exchange Operation



A parallel compare-exchange operation (each process is responsible for one element). 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 Split Operation



A compare-split operation (each process is responsible for a block of elements). Each process sends all its elements to another 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_j retains the larger elements.

Bubble Sort and its Variants

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.

Bubble Sort and its Variants

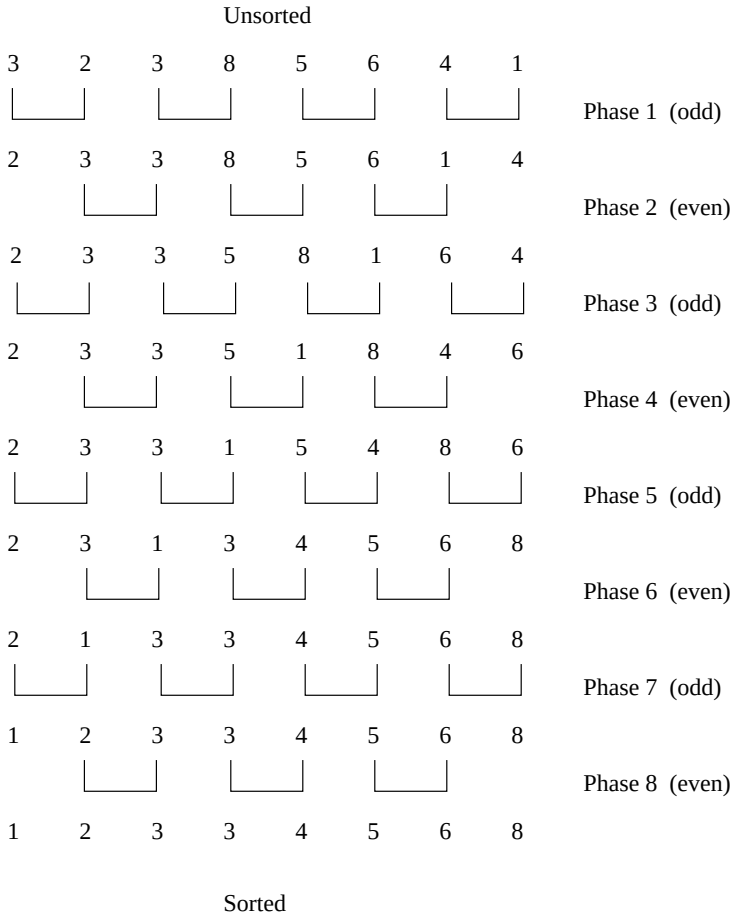
- The complexity of bubble sort is $\Theta(n^2)$.
- Bubble sort is difficult to parallelize since the algorithm has no concurrency.
- A simple variant, though, uncovers the concurrency.

Odd-Even Transposition

```
1.   procedure ODD-EVEN(n)
2.   begin
3.       for i := 1 to n do
4.       begin
5.           if i is odd then
6.               for j := 0 to n/2 − 1 do
7.                   compare-exchange(a_{2j+1}, a_{2j+2});
8.           if i is even then
9.               for j := 1 to n/2 − 1 do
10.                  compare-exchange(a_{2j}, a_{2j+1});
11.      end for
12.  end ODD-EVEN
```

Sequential odd-even transposition sort algorithm.

Odd-Even Transposition



Sorting $n = 8$ elements, using the odd-even transposition sort algorithm. During each phase, $n = 8$ elements are compared.

Odd-Even Transposition

- 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 Transposition

- Consider the one item per processor case.
- There are n iterations, in each iteration, each processor does one compare-exchange.
- The parallel run time of this formulation is $\Theta(n)$.
- This is cost optimal with respect to the base serial algorithm but not the optimal one.

Parallel Odd-Even Transposition

```
1.    procedure ODD-EVEN_PAR(n)
2.    begin
3.        id := process's label
4.        for i := 1 to n do
5.        begin
6.            if i is odd then
7.                if id is odd then
8.                    compare-exchange_min(id + 1);
9.                else
10.                   compare-exchange_max(id - 1);
11.           if i is even then
12.               if id is even then
13.                   compare-exchange_min(id + 1);
14.               else
15.                   compare-exchange_max(id - 1);
16.       end for
17.   end ODD-EVEN_PAR
```

Parallel formulation of odd-even transposition.

Parallel Odd-Even Transposition

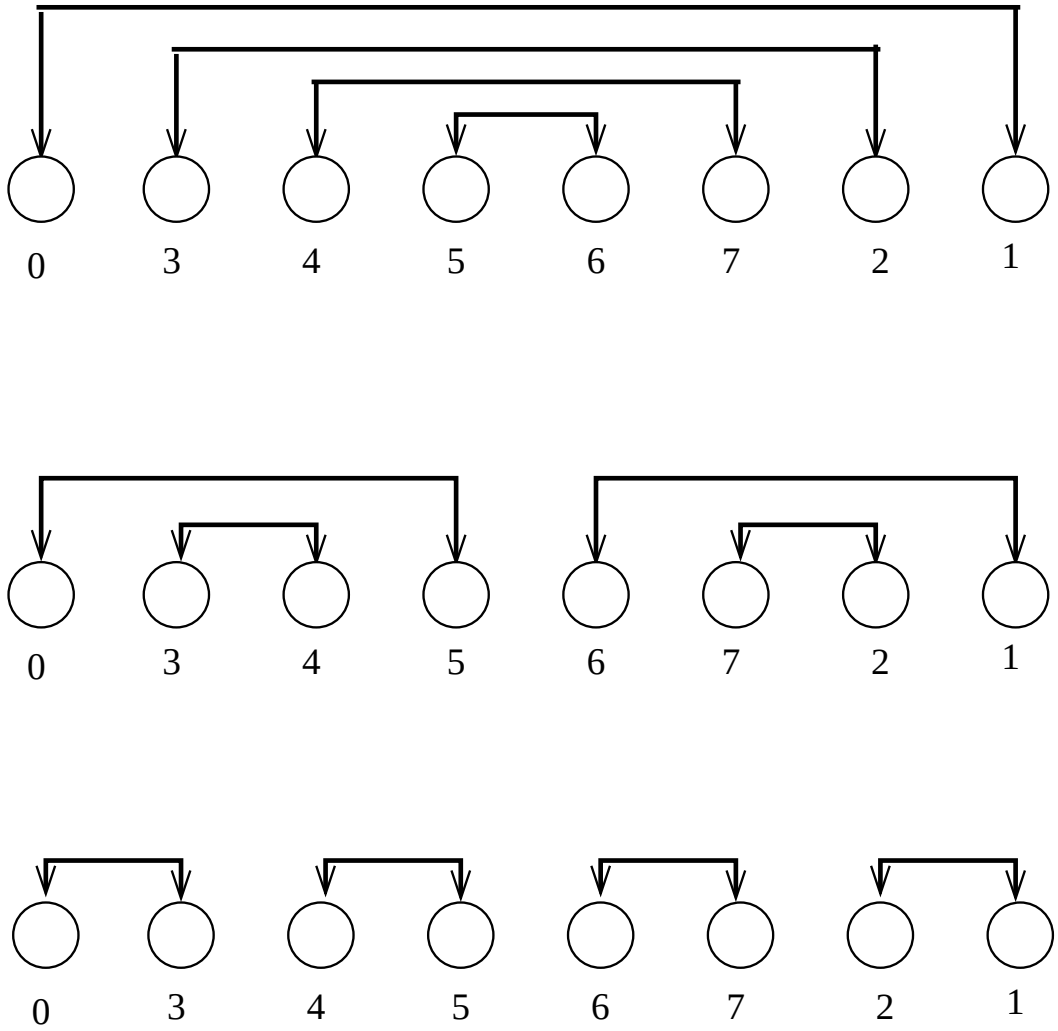
- Consider a block of n/p elements per processor.
- The first step is a local sort.
- In each subsequent step, the compare exchange operation is replaced by the compare split operation.
- The parallel run time of the formulation is

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

Shellsort

- Let n be the number of elements to be sorted and p be the number of processes.
- During the first phase, processes that are far away from each other in the array compare-split their elements.
- During the second phase, the algorithm switches to an odd-even transposition sort.

Parallel Shellsort



An example of the first phase of parallel shellsort on an eight-process array.

Parallel Shellsort

- Each process performs $d = \log p$ compare-split operations.
- With $O(p)$ bisection width, the each communication can be performed in time $\Theta(n/p)$ for a total time of $\Theta((n \log p)/p)$.
- In the second phase, l odd and even phases are performed, each requiring time $\Theta(n/p)$.
- The parallel run time of the algorithm is:

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{local sort}} + \overbrace{\Theta\left(\frac{n}{p}\log p\right)}^{\text{first phase}} + \overbrace{\Theta\left(l\frac{n}{p}\right)}^{\text{second phase}}. \qquad (1)$$

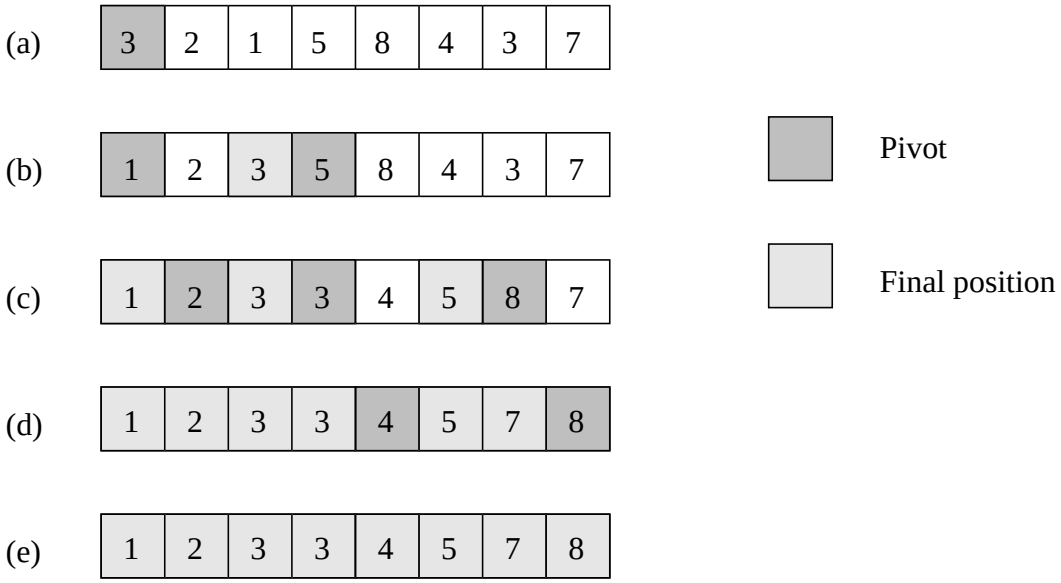
Quicksort

- Quicksort is one of the most common sorting algorithms for sequential computers because of its simplicity, low overhead, and optimal average complexity.

- Quicksort selects one of the entries in the sequence to be the pivot and divides the sequence into two – one with all elements less than the pivot and other greater.

- The process is recursively applied to each of the sublists.

Quicksort



Example of the quicksort algorithm sorting a sequence of size $n = 8$.

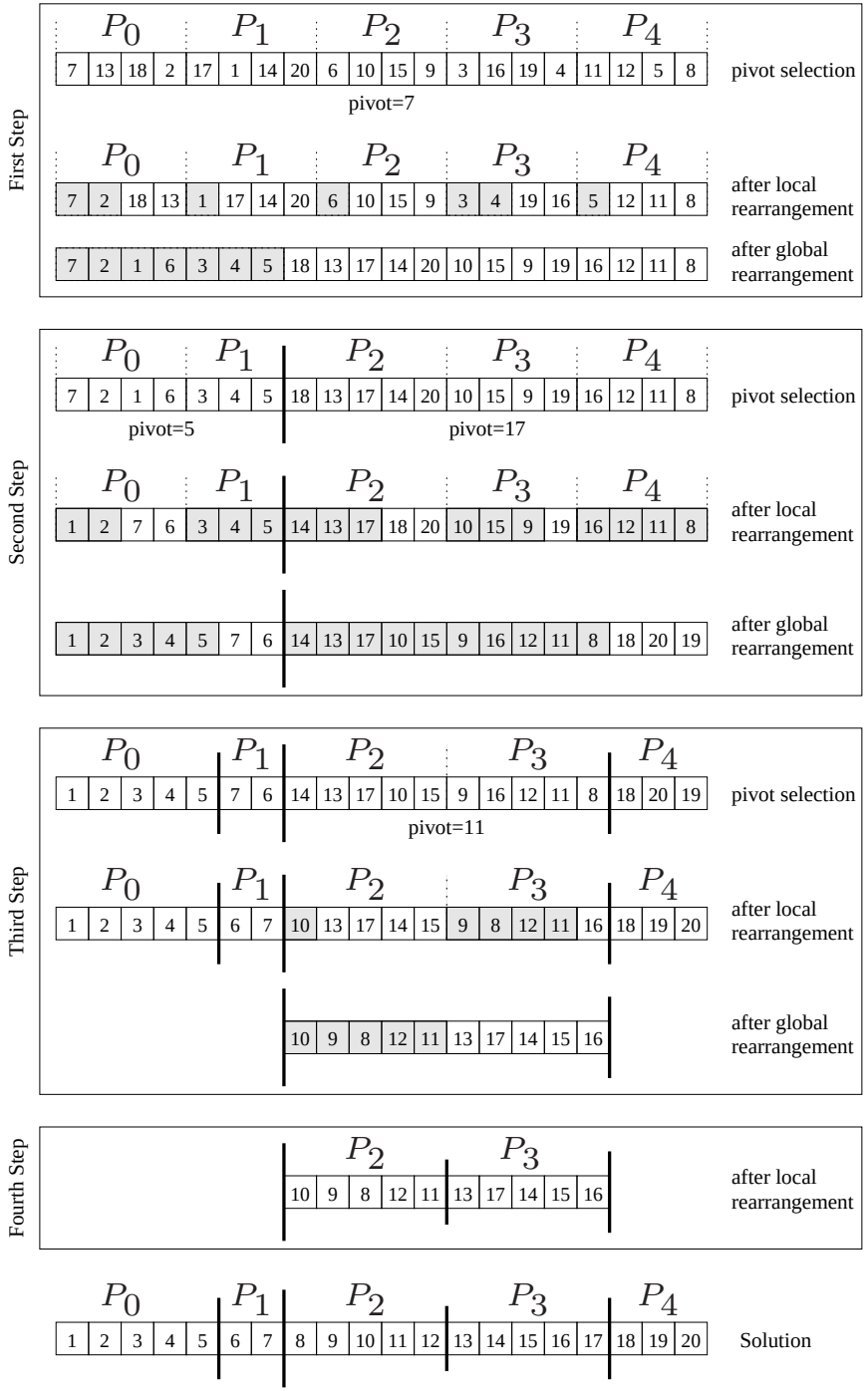
Quicksort

- The performance of quicksort depends critically on the quality of the pivot.
- In the best case, the pivot divides the list in such a way that the larger of the two lists does not have more than αn elements (for some constant α).
- In this case, the complexity of quicksort is $O(n \log n)$.

Parallelizing Quicksort: Shared Address Space Formulation

- Consider a list of size n equally divided across p processors.
- A pivot is selected by one of the processors and made known to all processors.
- Each processor partitions its list into two, say S_i and L_i, based on the selected pivot.
- All of the S_i lists are merged and all of the L_i lists are merged separately.
- The set of processors is partitioned into two (in proportion of the size of lists S and L). The process is recursively applied to each of the lists.

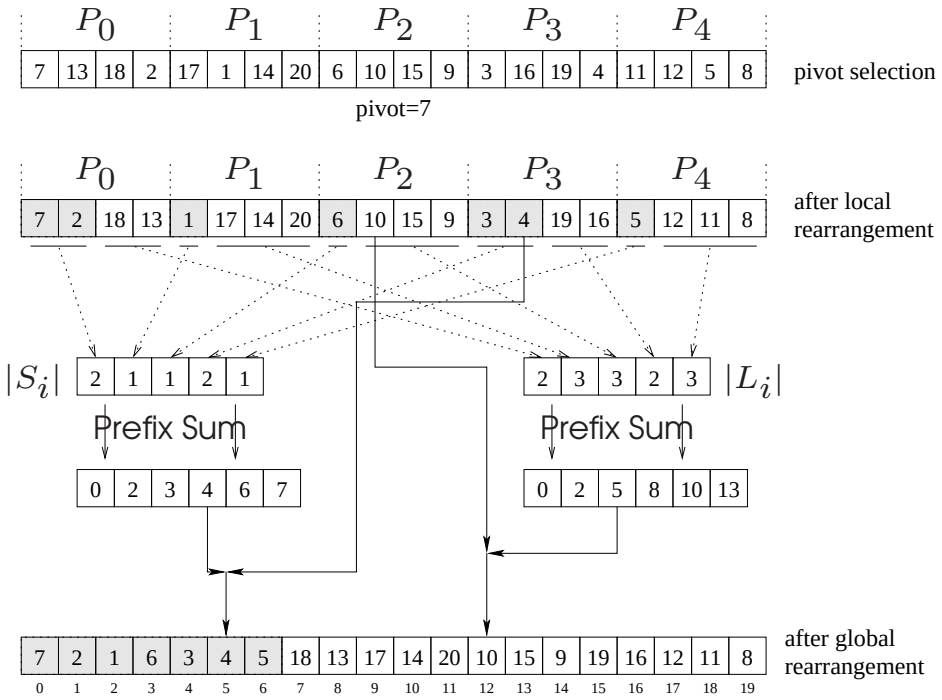
Shared Address Space Formulation



Parallelizing Quicksort: Shared Address Space Formulation

- How to globally merge the local lists (S_0, L_0, S_1, L_1, ...) to form S and L?
- Each processor needs to determine the right location for its elements in the merged list.
- Each processor first counts the number of elements locally less than and greater than pivot.
- It then computes two sum-scans to determine the starting location for its elements in the merged S and L lists.
- Once it knows the starting locations, it can write its elements safely.

Parallelizing Quicksort: Shared Address Space Formulation



Efficient global rearrangement of the array.

Parallelizing Quicksort: Shared Address Space Formulation

- The parallel time depends on the split and merge time, and the quality of the pivot.
- The latter is an issue independent of parallelism, so we focus on the first aspect, assuming ideal pivot selection.
- The algorithm executes in four steps: (i) determine and broadcast the pivot; (ii) locally rearrange the array assigned to each process; (iii) determine the locations in the globally rearranged array that the local elements will go to; and (iv) perform the global rearrangement.
- The first step takes time $\Theta(\log p)$, the second, $\Theta(n/p)$, the third, $\Theta(\log p)$, and the fourth, $\Theta(n/p)$.
- The overall complexity of splitting an n-element array is $\Theta(n/p) + \Theta(\log p)$.

Parallelizing Quicksort: Shared Address Space Formulation

- The process recurses until there are p lists, at which point, the lists are sorted locally.

- Therefore, the total parallel time is:

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{local sort}} + \overbrace{\Theta\left(\frac{n}{p}\log p\right) + \Theta(\log^2 p)}^{\text{array splits}}. \tag{2}$$

- The corresponding isoefficiency is $\Theta(p\log^2 p)$ due to broadcast and scan operations.

Parallelizing Quicksort: Message Passing Formulation

- A simple message passing formulation is based on the recursive halving of the machine.
- Assume that each processor in the lower half of a p processor ensemble is paired with a corresponding processor in the upper half.
- A designated processor selects and broadcasts the pivot.
- Each processor splits its local list into two lists, one less (S_i), and other greater (L_i) than the pivot.
- A processor in the low half of the machine sends its list L_i to the paired processor in the other half. The paired processor sends its list S_i.
- It is easy to see that after this step, all elements less than the pivot are in the low half of the machine and all elements greater than the pivot are in the high half.

Parallelizing Quicksort: Message Passing Formulation

- The above process is recursed until each processor has its own local list, which is sorted locally.
- The time for a single reorganization is $\Theta(\log p)$ for broadcasting the pivot element, $\Theta(n/p)$ for splitting the locally assigned portion of the array, $\Theta(n/p)$ for exchange and local reorganization.
- We note that this time is identical to that of the corresponding shared address space formulation.
- It is important to remember that the reorganization of elements is a bandwidth sensitive operation.