Lesson 9: Sorting algorithms

G Stefanescu — University of Bucharest

Parallel & Concurrent Programming
Fall, 2014

Parallel sorting algorithms

Potential speedup:

- best sequential sorting algorithms (for arbitrary sequences of numbers) have average time complexity $O(n\ log\ n)$
- hence, the best speedup one can expect from using n processors is
$$\frac{O(n\ log\ n)}{n} = O(log\ n)$$
- there are such parallel algorithms, but the hidden constant is very large (Leighton'84)
- generally, a practical, useful $O(log\ n)$ algorithm may be difficult to find

Rank sort

Rank sort:

- count the number of numbers that are smaller than a number a in the list
- this gives the position of a in the sorted list
- this procedure has to be repeated for all elements of the list; hence the time complexity is $n(n-1) = O(n^2)$ (not so good sequential algorithm)

Rank sort, sequential code

Rank sort: sequential code

```
for (i=0; i<n; i++){
  x=0;
  for (j=0; j<n; j++)
    if (a[i] > a[j]) x++;
  b[x] = a[i];
}
```

- work well if there are no repetitions of the numbers in the list (in the case of repetitions one has to change slightly the code)

Rank sort, parallel code

Rank sort: parallel code, using n processors

```
forall (i=0; i<n; i++){
  x=0;
  for (j=0; j<n; j++)
    if (a[i] > a[j]) x++;
  b[x] = a[i];
}
```

- n processors work in parallel to find the ranks of all numbers of the list
- parallel time complexity is $O(n)$, better than any sequential sorting algorithm

..Rank sort, parallel code

Rank sort: parallel code, using n^2 processors

- in the case n^2 processors may be used, the comparison of each `a[0],...,a[n-1]` with `a[i]` may be done in parallel, as well
- incrementing the counter is still sequential, hence the overall computation requires $1+n$ steps;
- if a tree structure is used to increment the counter, then the overall computation time is $O(log\ n)$ (but, as one expects, processor efficiency is very low)

These are just theoretical results: it is not efficient to use n or n^2 processors to sort n numbers.

Compare-and-exchange sorting algorithms

Compare-and-exchange:

- Compare-and-exchange is a basis for many sequential sorting algorithms
- sequential "compare-and-exchange":

```
if (a > b) {
  tmp = A;
  A = B;
  B = tmp;
}
```

..Compare-and-exchange sorting

Asymmetric parallel "compare-and-exchange":

- process P1 sends *A* to P2;
- process P2 compares *A* with its values *B*; if *B* is larger than *A* it sends *B* to P1, otherwise it sends *A* back to P1
- code for process P1:

```
send(&A, P2);
recv(&A, P2);
```

- code for process P2:

```
recv(&A, P1);
if (A > B) {
  send(&B, P1);
  B = A;
} else
  send(&A, P1);
```

..Compare-and-exchange sorting

Symmetric parallel “compare-and-exchange”:

- each process sends its number to the other
- code for process P1:
```
send(&A, P2);
recv(&B, P2);
if (A > B) A = B;
```
- code for process P2:
```
recv(&A, P1);
send(&B, P1);
if (A > B) B = A;
```

  (alternated send-recv are used here to avoid deadlock)

- *Duplicated computation:* the result of the comparison should be the same on each processor

Data partitioning

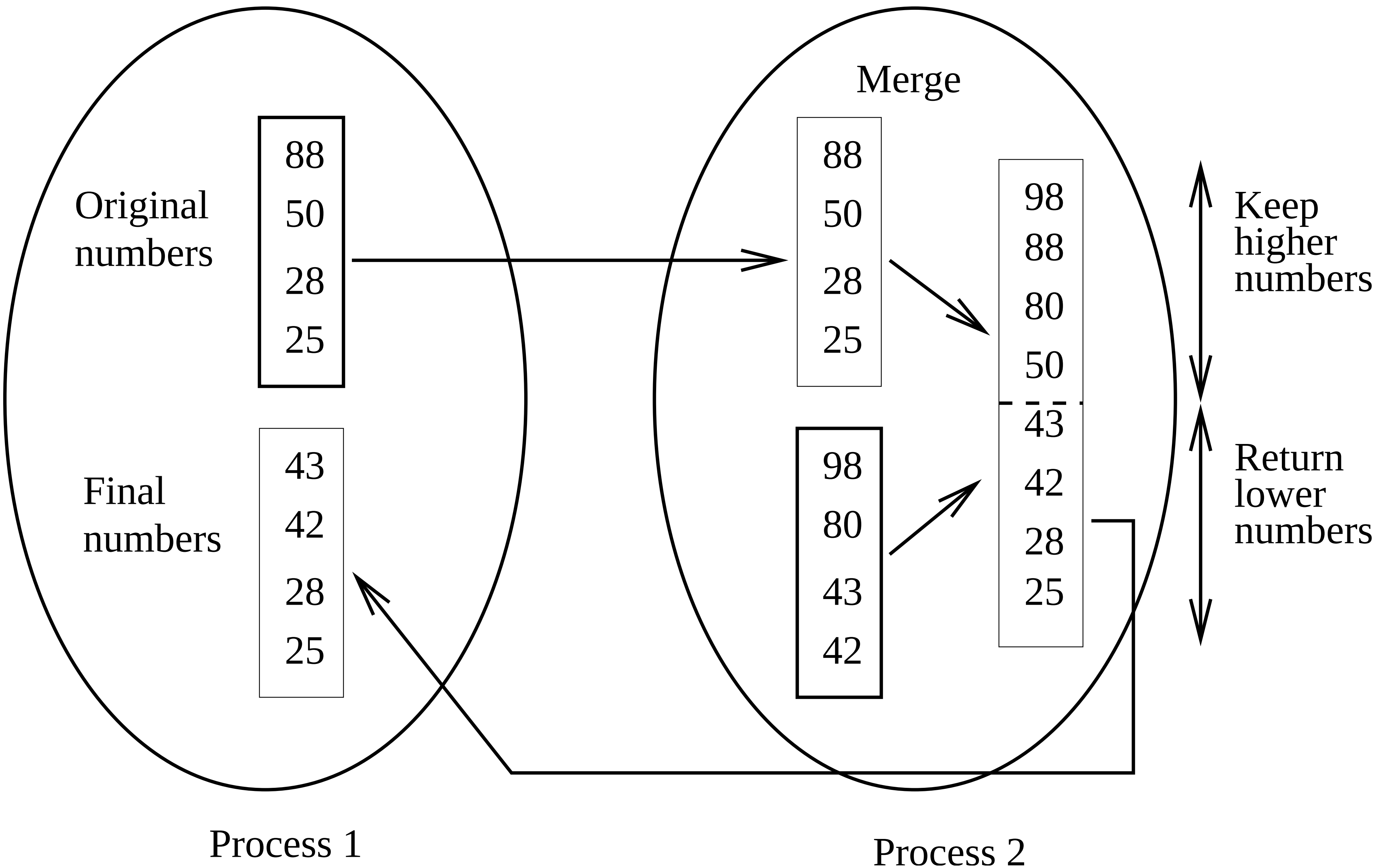
Data partitioning:

- usually, the number n of numbers is much larger than the number p of processes

- in such cases, each process will handle a group of data, here a sorted sublist

- the algorithm is the same, but now each process
—concatenate its list with that received from the other process,
—then sort it, and
—finally keep the corresponding (top or bottom) half of it

Merging two sublists



Bubble sort

Bubble sort:

- simple, but not too efficient, sequential algorithm
- sequential code;

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

- time complexity: $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$

Parallel bubble sort

Parallel bubble sort: based on the idea that the bodies of the main loop may be overlapped

Odd-even transposition sort:

- Even phase (0,2,4,6,...)

```
Pi (i=0,2,..; even):        Pi (i=1,3,..; odd):
recv(&newA, P(i+1));        send(&A, P(i-1));
send(&A, P(i+1));           recv(&newA, P(i-1));
if(newA < A) A = newA;      if(newA > A) A = newA;
```

- Odd phase (1,3,5,...)

```
Pi (i=2,4,..; even):        Pi (i=1,3,..; odd):
recv(&newA, P(i-1));        send(&A, P(i+1));
send(&A, P(i-1));           recv(&newA, P(i+1));
if(newA > A) A = newA;      if(newA < A) A = newA;
```

parallel bubble sort

Example (sorting 8 numbers):

Step	P_0		P_1		P_2		P_3		P_4		P_5		P_6		P_7
0	4	$\longleftrightarrow$	2		7	$\longleftrightarrow$	8		5	$\longleftrightarrow$	1		3	$\longleftrightarrow$	6
1	2		4	$\longleftrightarrow$	7		8	$\longleftrightarrow$	1		5	$\longleftrightarrow$	3		6
2	2	$\longleftrightarrow$	4		7	$\longleftrightarrow$	1		8	$\longleftrightarrow$	3		5	$\longleftrightarrow$	6
3	2		4	$\longleftrightarrow$	1		7	$\longleftrightarrow$	3		8	$\longleftrightarrow$	5		6
4	2	$\longleftrightarrow$	1		4	$\longleftrightarrow$	3		7	$\longleftrightarrow$	5		8	$\longleftrightarrow$	6
5	1		2	$\longleftrightarrow$	3		4	$\longleftrightarrow$	5		7	$\longleftrightarrow$	6		8
6	1	$\longleftrightarrow$	2		3	$\longleftrightarrow$	4		5	$\longleftrightarrow$	6		7	$\longleftrightarrow$	8
7	1		2	$\longleftrightarrow$	3		4	$\longleftrightarrow$	5		6	$\longleftrightarrow$	7		8

Two dimensional sorting

Shearsort:

- the goal is to sort a two-dimensional array/mesh in *snakelike-style*, i.e., 1st line increasing, 2nd line decreasing, 3rd line increasing, and so on
- in *odd phases* rows are sorted in snakelike-style (alternated directions)
- in *even phases* columns are sorted increasingly from top to bottom (all columns are sorted in the same direction)
- result: after $log\, n + 1$ phases the mash is snakelike-style sorted

..Sheresort

Shearsort / Example:

4	14	8	2
10	3	13	16
7	15	1	5
12	6	11	9

Original numbers

2	4	8	14
16	13	10	3
1	5	7	15
12	11	9	6

Phase 1 - row sort

1	4	7	3
2	5	8	6
12	11	9	14
16	13	10	15

Phase 2 - col sort

1	3	4	7
8	6	5	2
9	11	12	14
16	15	13	10

Phase 3 - row sort

1	3	4	2
8	6	5	7
9	11	12	10
16	15	13	14

Phase 4 - col sort

1	2	3	4
8	7	6	5
9	10	11	12
16	15	14	13

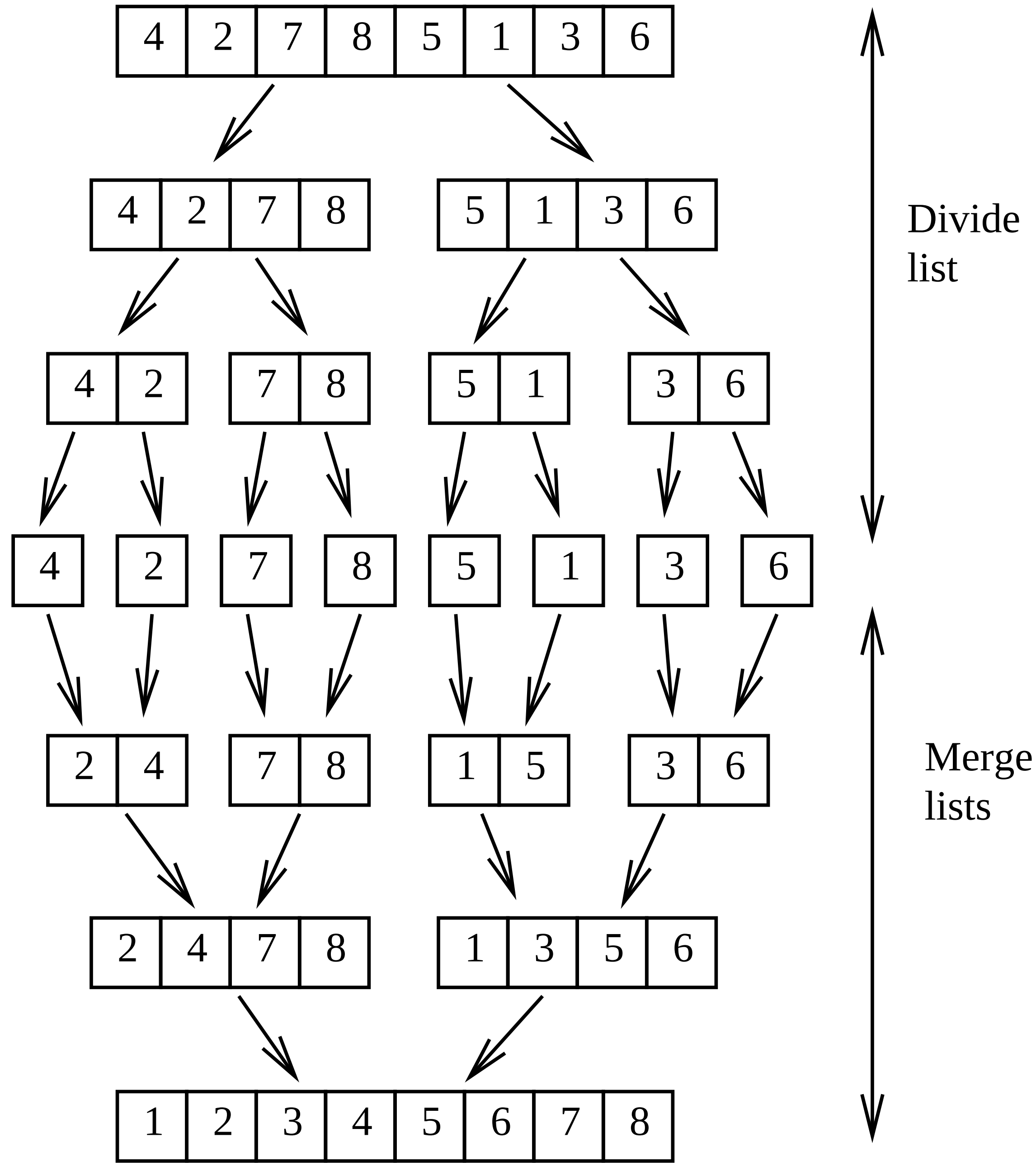
Final phase - row sort

Using transpositions, one may arrange to use only line sorting ("transpose, then line sorting, then transpose" may replace column sorting).

Mergesort

Mergesort:
is an optimal $O(n \log n)$ sequential sorting algorithm;

an example:



..Mergesort

Mergesort / Analysis:

Sequential: $O(n\ log\ n)$

Parallel: $(t_s = t_{startup}; t_d = t_{data})$

- communication: splitting data using $log\ n$ steps
  $(t_s + (n/2)t_d) + (t_s + (n/4)t_d) + (t_s + (n/8)t_d) + \ldots$
  and merging phase ($log\ n$ steps, again)
  $\ldots + (t_s + (n/8)t_d) + (t_s + (n/4)t_d) + (t_s + (n/2)t_d)$
  total communication time: $t_{comm} \approx 2(log\ n)t_{startup} + 2nt_{data}$

- computation (merging lists):
  $t_{comp} = \sum_{i=1}^{log\ n}(2^i - 1) = O(n)$

The overall time complexity (using n processors) is $O(n)$.

Quicksort

Quicksort:

- another optimal sequential sorting algorithm (sequential time complexity is $O(n\,log\,n)$)
- select a number r, called *pivot*, and split the list into two sub-lists: one with all the elements at most equal to r, the other holding all the elements grater than r
- the procedure is recursively applied till one element lists are obtained (which are sorted)
- example:

4	2	7	8	5	1	3	6$^{\leftarrow}$
3	$^{\rightarrow}$2	7	8	5	1	**4**	6
3	2	**4**	8	5	1$^{\leftarrow}$	7	6
3	2	1	$^{\rightarrow}$8	5	**4**	7	6
3	2	1	**4**	5$^{\leftarrow}$	8	7	6

..Quicksort

Quicksort:

- sequential code: `list[]` holds the numbers; `pivot` is the index of the pivot

```
quicksort(list, start, end){
  if (start < end) {
    partition(list, start, end, pivot);
    quicksort(list, start, pivot-1);
    quicksort(list, pivot+1, end);
  }
}
```

..Quicksort

Parallelizing quicksort:

- the recursive shape of the algorithm suggests to apply a divide-and-conquer parallelization method
- the main problem of the approach is that the tree distribution (induced by the lengths of the sublists) heavily depends on pivot selection; in the worst case, the tree may consist of a single path
- analysis: provided an equal distribution of numbers within sub-lists is assured, one gets
  —computation: $t_{comp} = n + (n/2) + (n/4) + \ldots \approx 2n$
  —communication: $(t_s + (n/2)t_d) + (t_s + (n/4)t_d) + (t_s + (n/8)t_d) + \ldots \approx (log\ n)t_s + nt_d$

Quicksort on a hypercube

Quicksort on a hypercube I: a root, say 00..0, is supposed to *hold all numbers*

- quicksort fits quite well for a hypercube implementation
- each processor receives a part of the list, divides it using a locally selected pivot, and submits the sublists to other nodes
- the selection of nodes where the sublists are to be submitted is similar to that used in the hypercube broadcast procedure (see next slide)

..Quicksort on a hypercube

Quicksort on a hypercube: example, using the standard binary coding of nodes

1st step:

000→100: 000 passes to 100 the numbers greater than p1 and keeps the others

2nd step:

000→010: 000 passes to 010 the numbers greater than p2 and keeps the others

100→110: 100 passes to 110 the numbers greater than p3 and keeps the others

3rd step:

000→001: 000 passes to 001 the numbers greater than p4 and keeps the others

010→011: 010 passes to 011 the numbers greater than p5 and keeps the others

100→101: 100 passes to 101 the numbers greater than p6 and keeps the others

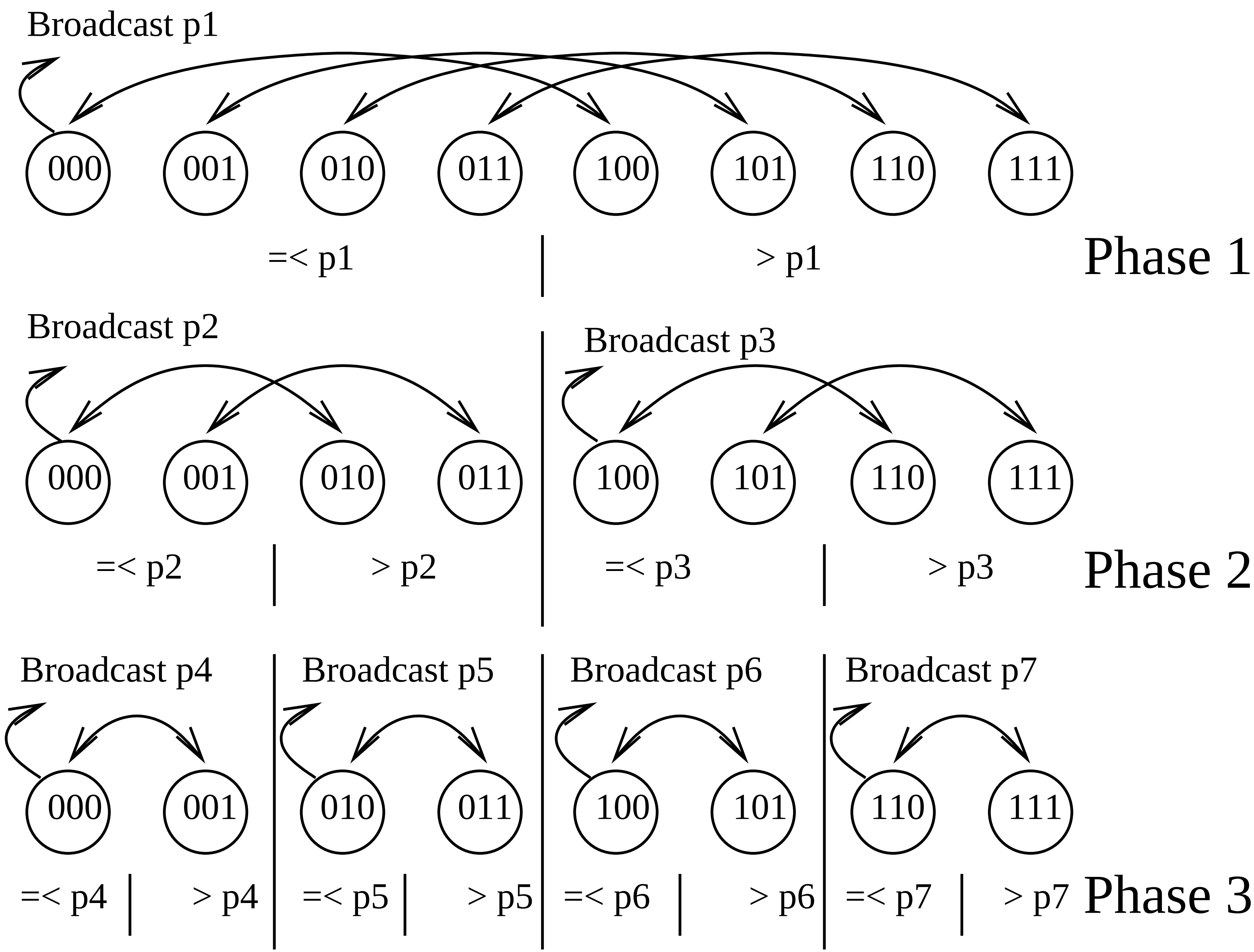
110→111: 110 passes to 111 the numbers greater than p7 and keeps the others

..Quicksort on a hypercube

Quicksort on a hypercube II: the list of numbers is *distributed* across all processors

- phase 1: —one processor, say 00..0, selects a pivot $p1$ an broadcast it to all nodes in the hypercube
  —pairs of complementary nodes P, Q (namely, $P = 0i_2..i_k$ and $Q = 1i_2..i_k$) exchange numbers such that finally P holds all common numbers less than or equal to $p1$ and Q holds those common numbers greater than $p1$
- phase 2,3,..: repeat the above procedure for all the sub-hypercubes of smaller dimensions $k-1, k-2, ..$ (it each pahse, each sub-hypercube selects its own pivot)
- finally, each node has a small list to sort; one gets the final sorted list concatenating these small sorted lists

..Quicksort on a hypercube



..Quicksort on a hypercube

Pivot selection:

- pivot selection is important for having equally loaded nodes
- in the sequential algorithm, usually the first number is selected;
- another possibility may be to take a sample, compute the mean value, and select the median as pivot;
- the latter is slightly more time consuming when the numbers are distributed across processes: one needs extra communication to select a sample
- the problems are simplified if each node keeps it small list *sorted*

Hyperquicksort

Hyperquicksort: quicksort on a hypercube with *distributed and sorted* lists across all processors

- each processor sorts its list sequentially
- the phases of the algorithm are as before (general quicksort on a hypercube), but
  —after the exchange of data, each node will merge its half list with the half list received to keep its numbers sorted

Analysis (hyperquicksort): suppose there are d dimensions (hence, $p = 2^d$ processors) and each processor initially holds n/p numbers

- computations: initial sorting $(n/p)log\ (n/p)$ comparisons

Hyperquicksort

..Analysis (hyperquicksort):

- computation: pivot selection $O(1)$ (sorted lists, just take a middle list element)
- communication (pivot broadcast):
  —broadcast one pivot in a k hypercube $k(t_s + t_d)$
  —total $(d + \ldots + 1)(t_s + t_d) = \dfrac{d(d-1)}{2}(t_s + t_d)$
- computation (split data using the pivot): for x numbers in a sorted list this requires $log\ x$
- communication (exchange $x/2$ numbers): $2(t_s + (x/2)t_d)$
- computation (merge sorted sublists): this requires $x/2$ comparisons, if the longest list has $x/2$ numbers

Hyperquicksort

..Analysis (hyperquicksort):

- total time complexity: add the above items
- it is quite tedious to count the sum of the last 3 items (the lengths of the sublists is not known in advance; the processes are synchronized, hence only the longest time counts as the phase time)
- in the ideal case when all list splits are into equal sublists, just add the last 3 items for x equal to $(n/p),(n/p)/2,(n/p)/4,\ldots$

Odd-even mergesort

Odd-even mergesort:

- the basic step is to merge two sorted lists $a_1,\ldots,a_n$ and $b_1,\ldots,b_n$ into one sorted list using the following parallel method
  —merge the elements of odd indices of each list
  —merge the elements of even indices of each list (to be done in parallel with the previous action)
  —finally, compare-and-exchange (in parallel) positions $(2,n+1),(3,n+2),\ldots,(n,2n-1)$.

- the lists should be of equal lengths

- moreover, usually n is a power of 2 in order to apply recursively the basic step

..Odd-even mergesort

Odd-even mergesort Example: - basic step

2, 4, **5**, 8   **1**, 3, **6**, 7

1, 2, 5, 6   3, 4, 7, 8

1, 2, 3, 4, 5, 6, 7, 8

Bitonic mergesort

Bitonic sequences: a *bitonic sequence* of numbers is

- a sequence consisting of two subsequences (of consecutive numbers), one increasing and one decreasing; e.g.,

$$3, 5, 8, \mathbf{19}, 17, 14, 12, 11 \quad (*)$$

- or a sequence which may be brought to such a form $(*)$ by a circular shifting of the elements of the sequence; e.g.,

$$\mathbf{12}, \mathbf{11}, 3, 5, 8, 19, 17, 14$$

..Bitonic mergesort

Bitonic sequences have the following useful mathematical property:

- if one preforms compare-and-exchange of elements $(1, 1 + n/2), (2, 2 + n/2), \ldots$ of a bitonic sequence $a_1, \ldots, a_n$, then
  —two bitonic subsequences $b_1, \ldots, b_{n/2}$ and $b_{(n/2)+1}, \ldots, b_n$ are obtained and
  —all numbers in one subsequence are less than all numbers in the other

Example: 12, 11, 3, 5, **8, 19, 17, 14**

8, 11, 3, 5, 12, 19, 17, 14

..Bitonic mergesort

Bitonic mergesort: the above observations lead to the following sorting method

1. Sorting a bitonic sequence (SBS): by a repeated application of the procedure explained in the above example one gets a sorted list

2. Creating bitonic sequences: here we can apply the same procedure (SBS)
—start with two-element lists (they are bitonic)
—sort adjacent lists (using SBS), but in opposite directions
—concatenate two adjacent lists to get a longer bitonic sequence;
—repeate the procedure till the full list becomes a bitonic sequence

..Bitonic mergesort

Bitonic mergesort / Example:

```
8   3   4   7   9   2   1   5
```

1. given list; mark starting 2-elem bitonic sublists

```
[8   3]   [4   7]   [9   2]   [1   5]
```

2. alternate (increasing - decreasing) sorting of 2-elem bitonic lists

```
[3   8]   [7   4]   [2   9]   [5   1]
```

3. mark obtained 4-elem bitonic lists

```
[3   8   7   4]   [2   9   5   1]
```

4. alternate (increasing - decreasing) sorting of 4-elem bitonic lists

```
[3   4   7   8]   [9   5   2   1]
```

5. mark obtained 8-elem bitonic list

```
[3   4   7   8   9   5   2   1]
```

6. sort (increasing) the 8-elem bitonic list

```
1   2   3   4   5   7   8   9
```

..Bitonic mergesort

..Bitonic mergesort / Example:

- Sorting of bitonic sequences (of steps 2,4,6) is explained here by expanding into microsteps the last case (step 6):

  ```
  [3  4  7  8  9  5  2  1]
  ```

  6.1. given 8-elem bitonic list; split into two 4-elem bitonic sublists

  ```
  [3  4  2  1]  [9  5  7  8]
  ```

  6.2. split each 4-elem bitonic sublists into two 2-elem bitonic sublists

  ```
  [2  1]  [3  4]  [7  5]  [9  8]
  ```

  6.3. sort (increasing) all 2-elem bitonic sublists

  ```
  1  2  3  4  5  7  8  9
  ```

..Bitonic mergesort

Analysis (bitonic mergesort):

- with $n = 2^k$, there are k phases, each involving $1, 2, \ldots, k$ steps, respectively;
- the total number of steps is

$$\sum_{i=1}^{k} i = \frac{k(k+1)}{2} = O(k^2) = O(log^2 n)$$