

Parallel Sorting

Sathish Vadhiyar

Parallel Sorting Problem

- The input sequence of size N is distributed across P processors
- The output is such that
 - elements in each processor P_i is sorted
 - elements in P_i is greater than elements in P_{i-1} and lesser than elements in P_{i+1}

Parallel quick sort

- Naïve approach
- Start with a single processor; divide array into two sub-arrays
- Now involve one more processor
- Both the processors perform the next step of quick sort within their local subarrays
- And so on....till the number of subarrays equal the number of processors
- Disadvantage: Inefficient utilization of processors

Another algorithm

- This algorithm involves all the processors in all the iterations
- One of the processors, P_0 , begins by broadcasting one of its elements as the pivot element to all the processors
- Each processor then divides its local array into two sub-arrays
 - L_i : elements less than the pivot
 - G_i : elements greater than the pivot

Parallel Quick Sort

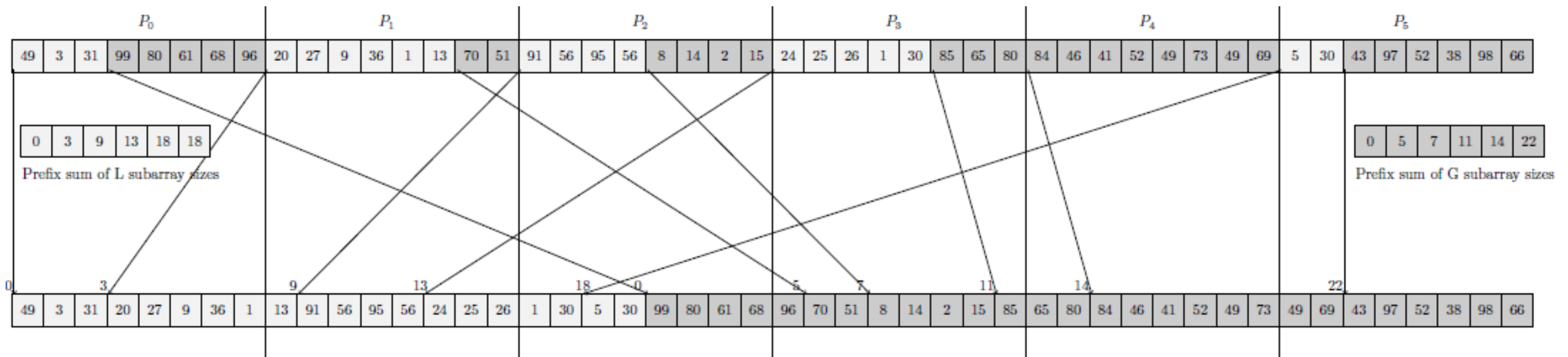
- Processors then divided into two groups:
 - First group will process the subsequent steps with L_i s
 - Second group with G_i s
- The sizes of the processor groups must be in the ratio of the number of elements in L s and G s to achieve load balance
- These number of elements can be found using an allreduce operation

Shared memory implementation

- All L 's are formed in the first part of the array; all G 's in the second part
- Each processor needs to know the locations in the shared memory where it has to write its L_i and G_i
- Prefix sums of the sizes of the subarrays can be used
- Prefix sum can be done in $O(\log P)$

Example: Prefix sum illustration

- In this example, 36 is the pivot element



Message Passing Version

- A processor should know which elements in its L_i and G_i it should send to which processor
- Distributed prefix sum is used
- A processor can then deduce its destination processor for sending its L array using:
 - Total number of elements of L subarrays
 - prefix sums of sizes
 - Size of the processor group that will be responsible for L subarray
- Similarly for the G subarray
- In worst case, this requires all-to-all with time complexity $O(N/P)$

Parallel Quick sort

- The process now repeats with the subgroups
- Until the number of subgroups equal the number of processors
- At this stage, each processor performs a local quick sort:
 $O(N/P \log(N/P))$

Complexity and analysis

- $\log P$ times:
 - Broadcast: $O(\log P)$
 - Allreduce: $O(\log P)$
 - Prefix sum and all-to-all: $O(\log P + N/P)$
- Then local quick sort: $O(N/P \cdot \log P)$
- Total: $O(N/P \cdot \log P) + O(\log^2 P + N/P \cdot \log P)$
- Weaknesses: Load imbalance and under-utilization

Bitonic Sort

Bitonic sequence

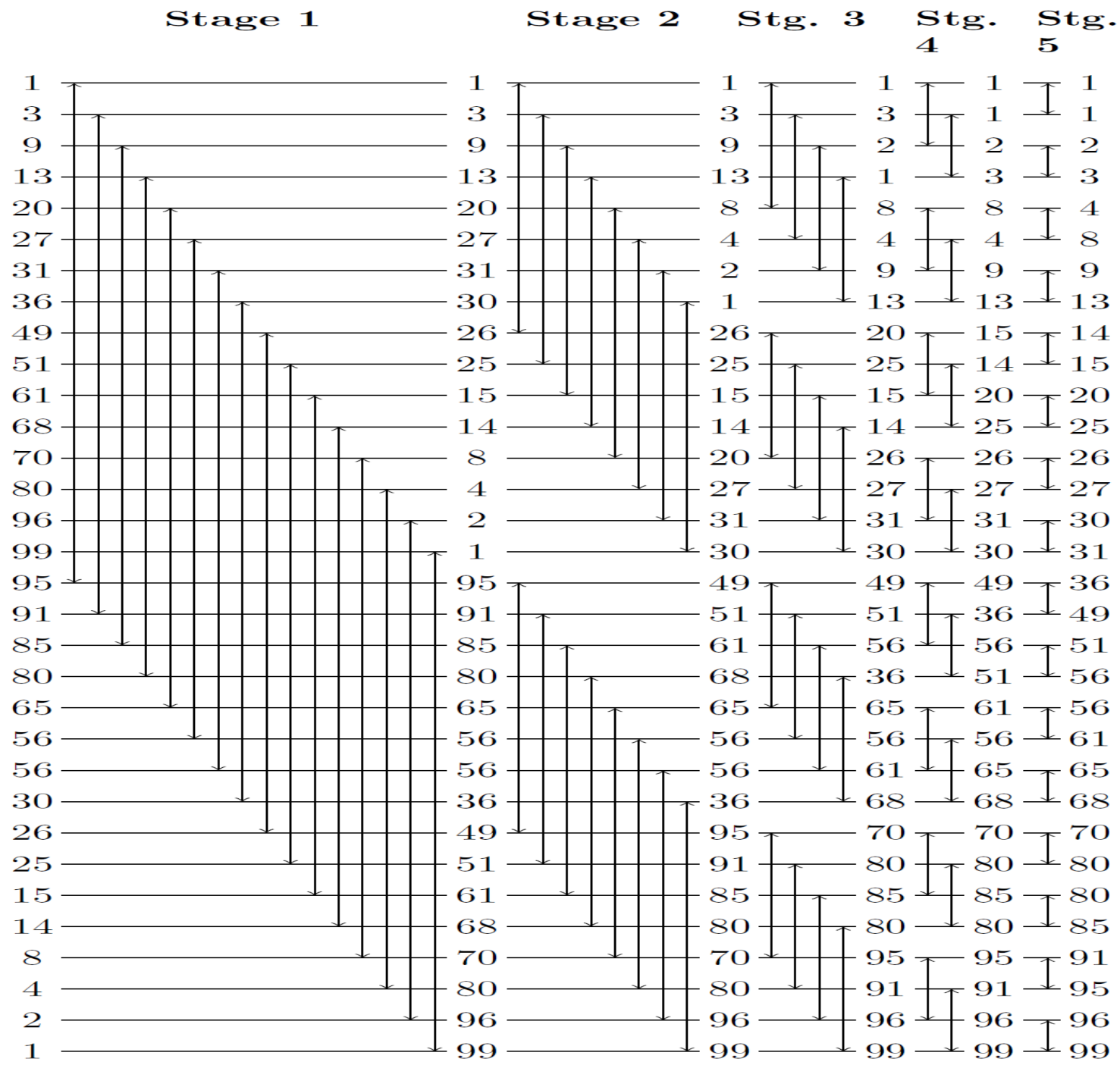
- A sequence of length n is a bitonic sequence if
 - for an element i
 - elements $a_1 \leq a_2 \leq a_3 \leq \dots \leq a_i$ and
 - Elements $a_i \geq a_{i-1} \geq a_{i-2} \geq \dots \geq a_n$
 - Any cyclic rotation of such a sequence is also a bitonic sequence

Bitonic property

- Given a bitonic sequence A , let us form another sequence B such that:
 - $B[i] = \min(A[i], A[i+N/2])$
 - $B[i+N/2] = \max(A[i], A[i+N/2])$
- It is easy to prove that:
 - Lower half $B[0] \dots B[N/2-1] \leq$ upper half $B[N/2] \dots B[N-1]$
 - Both the halves themselves are bitonic sequences of lengths $N/2$

Converting bitonic sequence into a sorted sequence

- To convert bitonic sequence of length N into a sorted sequence, we repeat the above recursively:
- In the first stage, form two bitonic sub-sequences of $N/2$ each
- In the second stage, form four bitonic sub-sequences of $N/4$ each
-
- After $\log N$ stages, a sorted sequence is formed
- This process is called **bitonic merge**



Bitonic sort

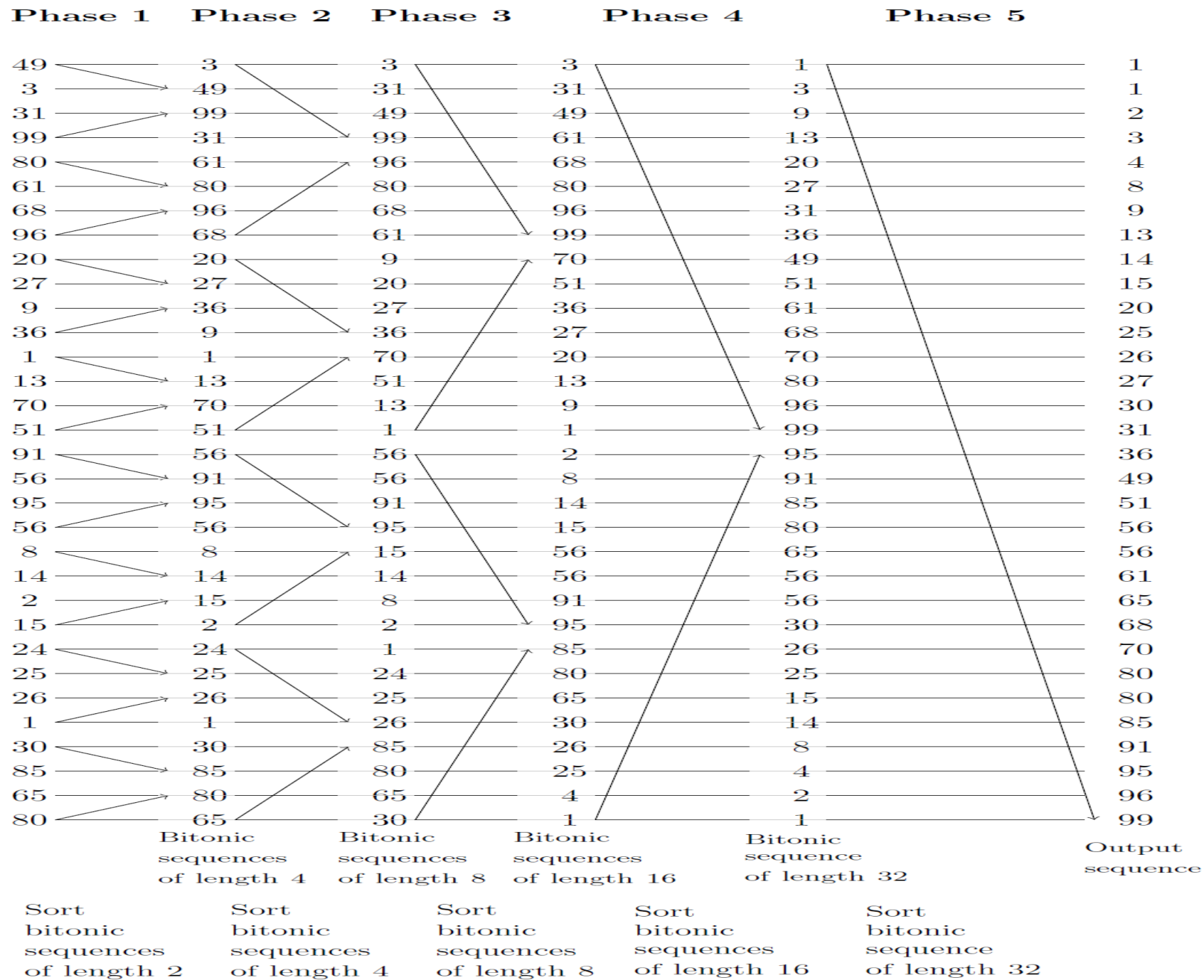
- Convert the original unsorted sequence into a bitonic sequence, then use the above procedure to convert to a sorted sequence
- Converting unsorted sequence of length N into a bitonic sequence of length N :
- Larger and larger bitonic sequences are built starting from sequences of lengths 2
- Note that any sequence of length 2 is a bitonic sequence

Bitonic sort

- In the first phase:
 - Sort two consecutive sub-sequences of lengths 2 such that
 - the first subsequence is sorted in ascending order, second in descending order
 - Now the two sorted sub-sequences are merged to form a bitonic sequence of length 4.
-
- In the second phase:
 - Consider two consecutive sub-sequences of lengths 4
 - Sort them into ascending and descending
 - Merge them into bitonic sequence of length 8

Bitonic sort

- So on....
- At the end of $\log N$ phases, a bitonic sequence of length N formed, which is converted into a sorted sequence
- Time complexity:
- $\log N$ phases
- Phase i has i stages
- $O(\log^2 N)$



Sequential complexity

- Has $\log N$ phases
- Each phase i has i stages
- Each stage i performs N compare-exchange operations
- Hence $O(N \log^2 N)$

Parallelization

Hypercube and mesh networks

- Maps well to hypercubes
- Processors are mapped to corresponding hypercube nodes
- Processors that need to interact for compare-exchange operations in the phases are mapped to hypercube nodes that have direct connections
- For mesh networks, a shuffle-row mapping is used

Parallel implementation

General networks

- Array distributed into block distribution across P processors
- The last $\log P$ of the $\log N$ phases require communications for exchanging elements
- In the last phase, out of the $\log N$ stages, the first $\log P$ stages involve communications
- Each communication is a compare-and-exchange
- Hence $O(\log^2 P)$ communication steps
- $O(N/P \cdot \log^2 P)$ communications
- $O(N/P \log^2 N)$ computations

Observations

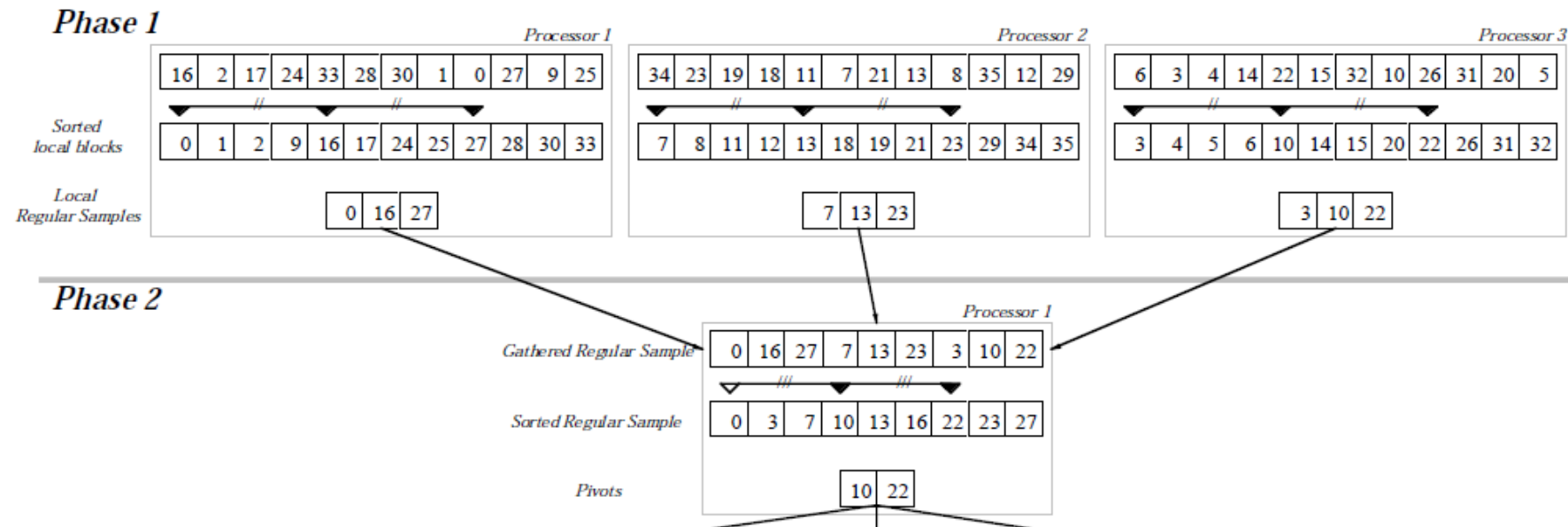
- In general, applied to small sequences due to high computation complexity
- Has poor speedup for greater than thousand processors due to high communication complexities

- Sample Sort

Parallel Sorting by Regular Sampling (PSRS)

1. Each processor sorts its local data
2. Each processor selects a sample vector of size $p-1$; k th element is $(n/p * (k+1)/p)$
3. Samples are sent and merge-sorted on processor 0
4. Processor 0 defines a vector of $p-1$ *splitters* starting from $p/2$ element; i.e., k th element is $p(k+1/2)$; broadcasts to the other processors

Example



PSRS

5. Each processor sends local data to correct destination processors based on splitters; all-to-all exchange
6. Each processor merges the data chunk it receives

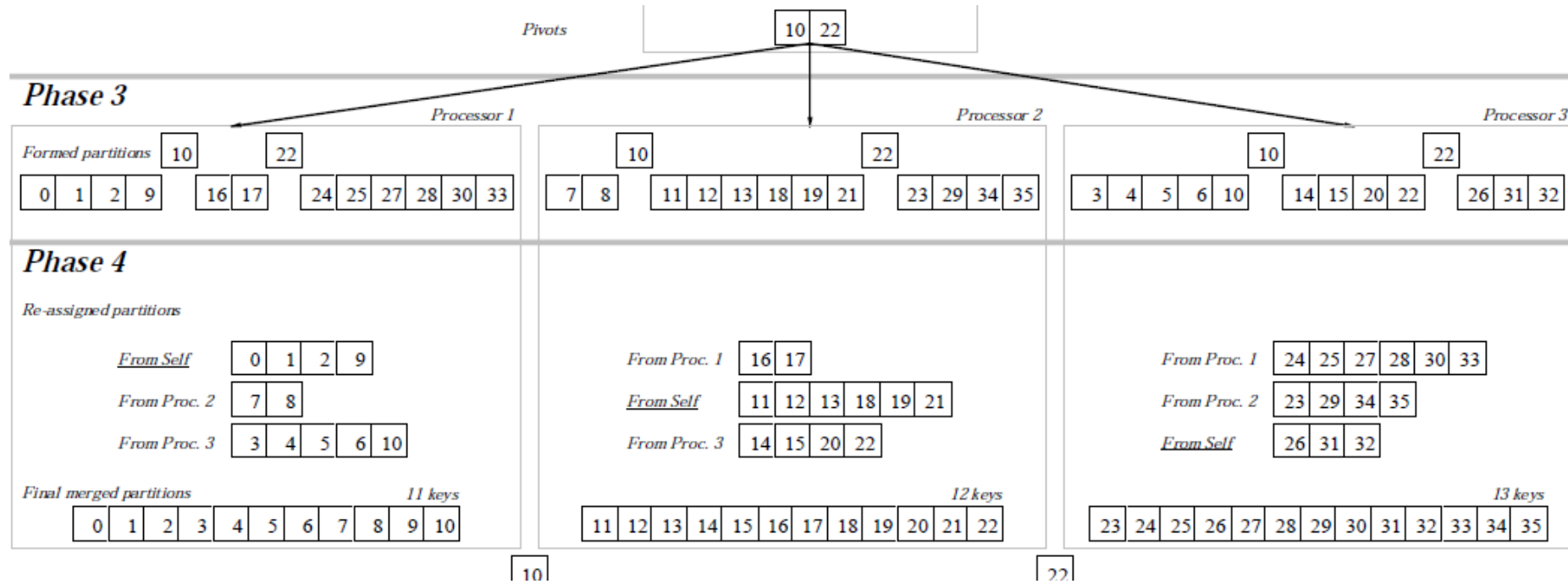
Step 5

- Each processor finds where each of the $p-1$ pivots divides its list, using a binary search
- i.e., finds the index of the largest element number larger than the j th pivot
- At this point, each processor has p sorted sublists with the property that each element in sublist i is greater than each element in sublist $i-1$ in any processor

Step 6

- Each processor i performs a p -way merge-sort to merge the i th sublists of p processors

Example Continued



Analysis

- The first phase of local sorting takes $O((n/p)\log(n/p))$
- 2nd phase:
 - Sorting $p(p-1)$ elements in processor 0 - $O(p^2\log p^2)$
 - Each processor performs $p-1$ binary searches of n/p elements - $p\log(n/p)$
- 3rd phase: Each processor merges $(p-1)$ sublists
 - Size of data merged by any processor is no more than $2n/p$ (proof)
 - Complexity of this merge sort $2(n/p)\log p$
- Summing up: $O((n/p)\log n)$

Analysis

- 1st phase - no communication
- 2nd phase - $p(p-1)$ data collected; $p-1$ data broadcast
- 3rd phase: Each processor sends $(p-1)$ sublists to other $p-1$ processors; processors work on the sublists independently

Analysis

Not scalable for large number of processors

Merging of $p(p-1)$ elements done on one processor; 16384 processors require 16 GB memory

Sorting by Random Sampling

- An interesting alternative; random sample is flexible in size and collected randomly from each processor's local data
- Advantage
 - A random sampling can be retrieved before local sorting; overlap between sorting and splitter calculation

Radix Sort

- During every step, the algorithm puts every key in a bucket corresponding to the value of some subset of the key's bits
- A k -bit radix sort looks at k bits every iteration
- Easy to parallelize - assign some subset of buckets to each processor
- Load balance - assign variable number of buckets to each processor

Radix Sort – Load Balancing

- Each processor counts how many of its keys will go to each bucket
- Sum up these histograms with reductions
- Once a processor receives this combined histogram, it can adaptively assign buckets

Radix Sort - Analysis

- Requires multiple iterations of costly all-to-all
- Cache efficiency is low - any given key can move to any bucket irrespective of the destination of the previously indexed key
- Affects communication as well

Histogram Sort

- Another splitter-based method
- Histogram also determines a set of $p-1$ splitters
- It achieves this task by taking an iterative approach rather than one big sample
- A processor broadcasts k ($> p-1$) initial splitter guesses called a probe
- The initial guesses are spaced evenly over data range

Histogram Sort

Steps

1. Each processor sorts local data
2. Creates a histogram based on local data and splitter guesses
3. Reduction sums up histograms
4. A processor analyzes which splitter guesses were satisfactory (in terms of load)
5. If unsatisfactory splitters, the , processor broadcasts a new probe, go to step 2; else proceed to next steps

Histogram Sort

Steps

6. Each processor sends local data to appropriate processors
- all-to-all exchange
7. Each processor merges the data chunk it receives

Merits:

- Only moves the actual data once
- Deals with uneven distributions

Sources/References

- On the versatility of parallel sorting by regular sampling. Li et al. Parallel Computing. 1993.
- Parallel Sorting by regular sampling. Shi and Schaeffer. JPDC 1992.
- Highly scalable parallel sorting. Solomonik and Kale. IPDPS 2010.