Parallel Programming
in C with MPI and OpenMP

Michael J. Quinn

Chapter 14

Sorting

Outline

- Sorting problem
- Sequential quicksort
- Parallel quicksort
- Hyperquicksort
- Parallel sorting by regular sampling

Sorting Problem

- Permute: unordered sequence $\Rightarrow$ ordered sequence
- Typically key (value being sorted) is part of record with additional values (satellite data)
- Most parallel sorts designed for theoretical parallel models: not practical
- Our focus: internal sorts based on comparison of keys

Sequential Quicksort

17	14	65	4	22	63	11

Unordered list of values

Sequential Quicksort

17	14	65	4	22	63	11

Choose pivot value

Sequential Quicksort

14	4	11	17	65	22	63

Low list
(≤ 17)

High list
(> 17)

Sequential Quicksort

4	11	14	17	65	22	63

Recursively apply quicksort to low list

Sequential Quicksort

4	11	14	17	22	63	65

Recursively apply quicksort to high list

Sequential Quicksort

4	11	14	17	22	63	65

Sorted list of values

Attributes of Sequential Quicksort

- Average-case time complexity: $\Theta(n \log n)$
- Worst-case time complexity: $\Theta(n^2)$
  - Occurs when low, high lists maximally unbalanced at every partitioning step
- Can make worst-case less probable by using sampling to choose pivot value
  - Example: "Median of 3" technique

Quicksort Good Starting Point for Parallel Algorithm

- Speed
  - Generally recognized as fastest sort in average case
  - Preferable to base parallel algorithm on fastest sequential algorithm
- Natural concurrency
  - Recursive sorts of low, high lists can be done in parallel

Definitions of "Sorted"

- Definition 1: Sorted list held in memory of a single processor
- Definition 2:
  - Portion of list in every processor's memory is sorted
  - Value of last element on P_i's list is less than or equal to value of first element on P_{i+1}'s list
- We adopt Definition 2: Allows problem size to scale with number of processors

Parallel Quicksort

75, 91, 15, 64, 21, 8, 88, 54	P_0
50, 12, 47, 72, 65, 54, 66, 22	P_1
83, 66, 67, 0, 70, 98, 99, 82	P_2
20, 40, 89, 47, 19, 61, 86, 85	P_3

Parallel Quicksort

75, 91, 15, 64, 21, 8, 88, 54	P_0
50, 12, 47, 72, 65, 54, 66, 22	P_1
83, 66, 67, 0, 70, 98, 99, 82	P_2
20, 40, 89, 47, 19, 61, 86, 85	P_3

Process P_0 chooses and broadcasts randomly chosen pivot value

Parallel Quicksort

75, 91, 15, 64, 21, 8, 88, 54	P_0
50, 12, 47, 72, 65, 54, 66, 22	P_1
83, 66, 67, 0, 70, 98, 99, 82	P_2
20, 40, 89, 47, 19, 61, 86, 85	P_3

Exchange “lower half” and “upper half” values”

Parallel Quicksort

Lower "half"	75, 15, 64, 21, 8, 54, 66, 67, 0, 70	P_0
	50, 12, 47, 72, 65, 54, 66, 22, 20, 40, 47, 19, 61	P_1
Upper "half"	83, 98, 99, 82, 91, 88	P_2
	89, 86, 85	P_3

After exchange step

Parallel Quicksort

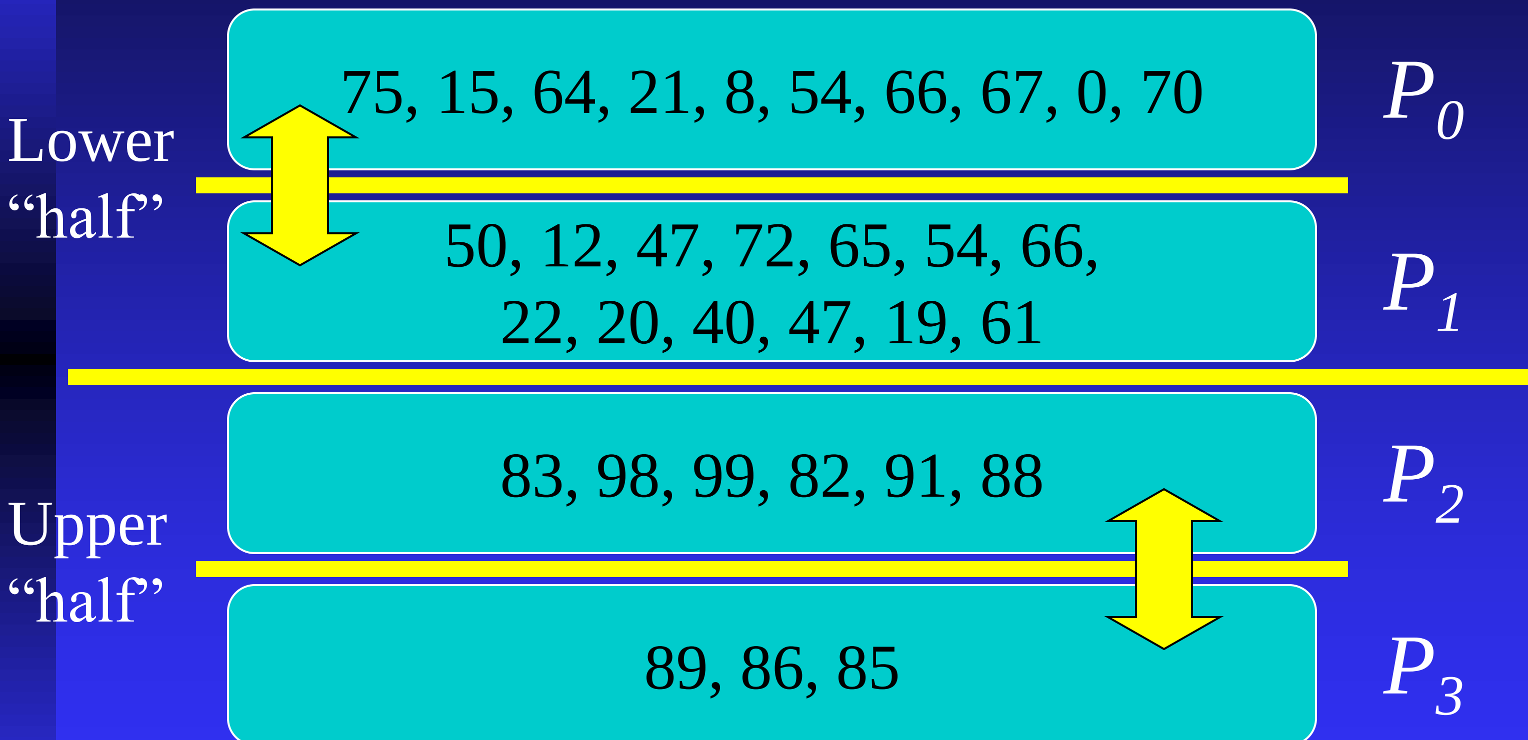
Lower "half"

Lower "half"	75, 15, 64, 21, 8, 54, 66, 67, 0, 70	P_0
	50, 12, 47, 72, 65, 54, 66, 22, 20, 40, 47, 19, 61	P_1
Upper "half"	83, 98, 99, 82, 91, 88	P_2
	89, 86, 85	P_3

Upper "half"

Processes P0 and P2 choose and broadcast randomly chosen pivots

Parallel Quicksort



Exchange values

Parallel Quicksort

Lower "half" of lower "half"	15, 21, 8, 0, 12, 20, 19	P_0
Upper "half" of lower "half"	50, 47, 72, 65, 54, 66, 22, 40, 47, 61, 75, 64, 54, 66, 67, 70	P_1
Lower "half" of upper "half"	83, 82, 91, 88, 89, 86, 85	P_2
Upper "half" of upper "half"	98, 99	P_3

Exchange values

Parallel Quicksort

Lower “half” of lower “half”	0, 8, 12, 15, 19, 20, 21	P_0
Upper “half” of lower “half”	22, 40, 47, 47, 50, 54, 54, 61, 64, 65, 66, 66, 67, 70, 72, 75	P_1
Lower “half” of upper “half”	82, 83, 85, 86, 88, 89, 91	P_2
Upper “half” of upper “half”	98, 99	P_3

Each processor sorts values it controls

Analysis of Parallel Quicksort

- Execution time dictated by when last process completes
- Algorithm likely to do a poor job balancing number of elements sorted by each process
- Cannot expect pivot value to be true median
- Can choose a better pivot value

Hyperquicksort

- Start where parallel quicksort ends: each process sorts its sublist
- First "sortedness" condition is met
- To meet second, processes must still exchange values
- Process can use median of its sorted list as the pivot value
- This is much more likely to be close to the true median

Hyperquicksort

75, 91, 15, 64, 21, 8, 88, 54	P_0
50, 12, 47, 72, 65, 54, 66, 22	P_1
83, 66, 67, 0, 70, 98, 99, 82	P_2
20, 40, 89, 47, 19, 61, 86, 85	P_3

Number of processors is a power of 2

Hyperquicksort

8, 15, 21, 54, 64, 75, 88, 91	P_0
12, 22, 47, 50, 54, 65, 66, 72	P_1
0, 66, 67, 70, 82, 83, 98, 99	P_2
19, 20, 40, 47, 61, 85, 86, 89	P_3

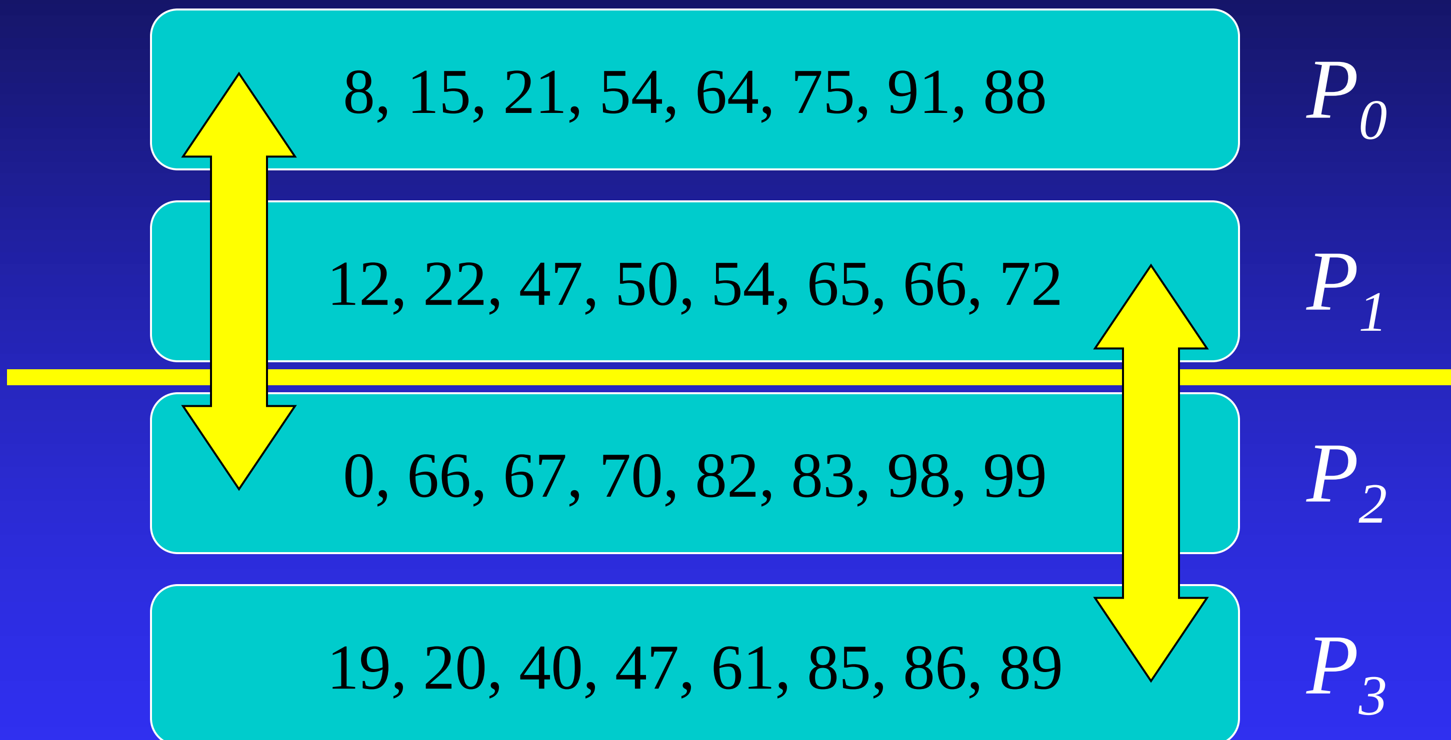
Each process sorts values it controls

Hyperquicksort

8, 15, 21, 54, 64, 75, 91, 88	P_0
12, 22, 47, 50, 54, 65, 66, 72	P_1
0, 66, 67, 70, 82, 83, 98, 99	P_2
19, 20, 40, 47, 61, 85, 86, 89	P_3

Process P_0 broadcasts its median value

Hyperquicksort



Processes will exchange “low”, “high” lists

Hyperquicksort

0, 8, 15, 21, 54	P_0
12, 19, 20, 22, 40, 47, 47, 50, 54	P_1
64, 66, 67, 70, 75, 82, 83, 88, 91, 98, 99	P_2
61, 65, 66, 72, 85, 86, 89	P_3

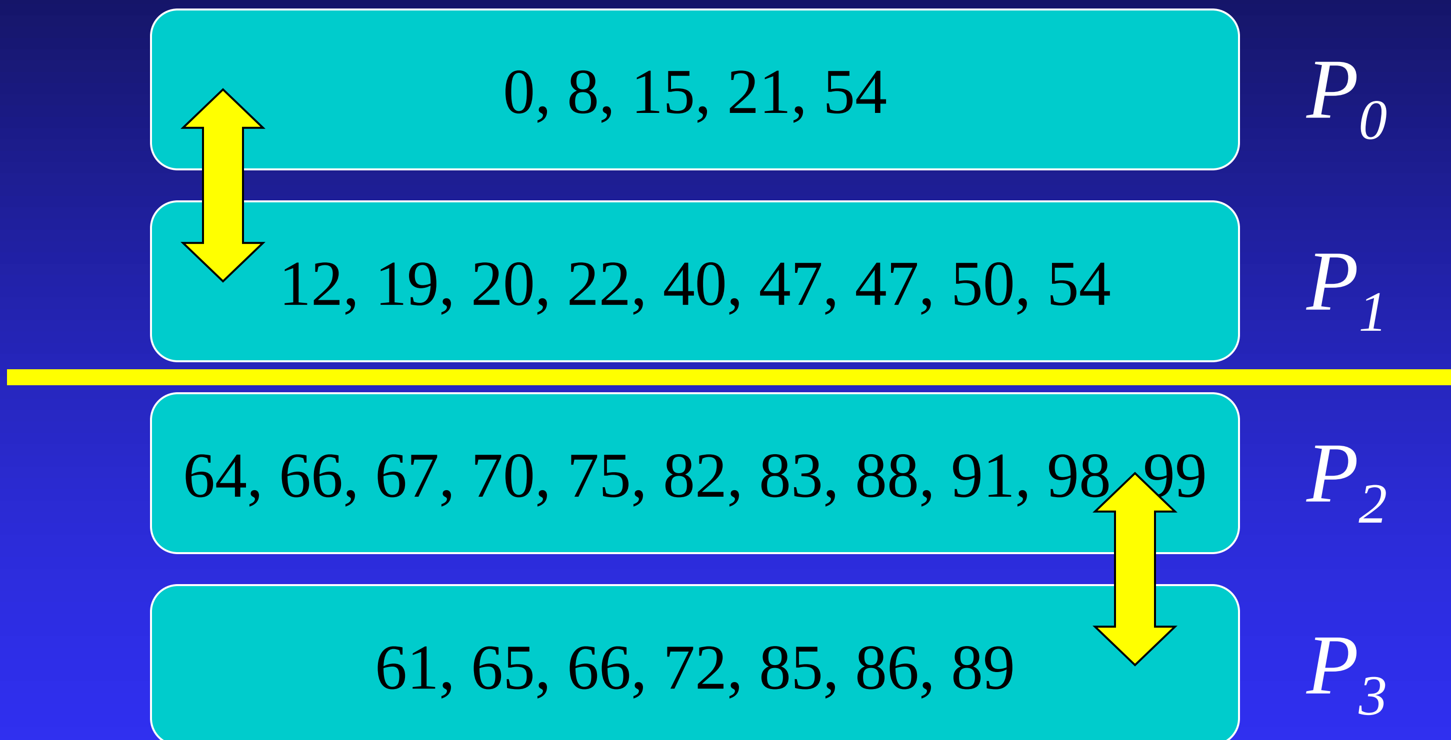
Processes merge kept and received values.

Hyperquicksort

0, 8, 15, 21, 54	P_0
12, 19, 20, 22, 40, 47, 47, 50, 54	P_1
64, 66, 67, 70, 75, 82, 83, 88, 91, 98, 99	P_2
61, 65, 66, 72, 85, 86, 89	P_3

Processes P_0 and P_2 broadcast median values.

Hyperquicksort



Communication pattern for second exchange

Hyperquicksort

0, 8, 12, 15	P_0
19, 20, 21, 22, 40, 47, 47, 50, 54, 54	P_1
61, 64, 65, 66, 66, 67, 70, 72, 75, 82	P_2
83, 85, 86, 88, 89, 91, 98, 99	P_3

After exchange-and-merge step

Complexity Analysis Assumptions

- Average-case analysis
- Lists stay reasonably balanced
- Communication time dominated by message transmission time, rather than message latency

Complexity Analysis

- Initial quicksort step has time complexity $\Theta((n/p) \log (n/p))$
- Total comparisons needed for log p merge steps: $\Theta((n/p) \log p)$
- Total communication time for log p exchange steps: $\Theta((n/p) \log p)$

Isoefficiency Analysis

- Sequential time complexity: $\Theta(n \log n)$
- Parallel overhead: $\Theta(n \log p)$
- Isoefficiency relation:
  $n \log n \geq C\, n \log p \Rightarrow \log n \geq C \log p \Rightarrow \mathrm{n} \geq p^C$

$$M(p^C)/p = p^C/p = p^{C-1}$$

- The value of C determines the scalability. Scalability depends on ratio of communication speed to computation speed.

Another Scalability Concern

- Our analysis assumes lists remain balanced
- As *p* increases, each processor's share of list decreases
- Hence as *p* increases, likelihood of lists becoming unbalanced increases
- Unbalanced lists lower efficiency
- Would be better to get sample values from all processes before choosing median

Parallel Sorting by Regular Sampling (PSRS Algorithm)

- Each process sorts its share of elements
- Each process selects regular sample of sorted list
- One process gathers and sorts samples, chooses pivot values from sorted sample list, and broadcasts these pivot values
- Each process partitions its list into p pieces, using pivot values
- Each process sends partitions to other processes
- Each process merges its partitions

PSRS Algorithm

75, 91, 15, 64, 21, 8, 88, 54	P_0
50, 12, 47, 72, 65, 54, 66, 22	P_1
83, 66, 67, 0, 70, 98, 99, 82	P_2

Number of processors does not have to be a power of 2.

PSRS Algorithm

8, 15, 21, 54, 64, 75, 88, 91	P_0
12, 22, 47, 50, 54, 65, 66, 72	P_1
0, 66, 67, 70, 82, 83, 98, 99	P_2

Each process sorts its list using quicksort.

PSRS Algorithm

8, 15 21, 54 64, 75 88, 91	P_0
12, 22 47, 50 54, 65 66, 72	P_1
0, 66 67, 70 82, 83 98, 99	P_2

Each process chooses *p* regular samples.

PSRS Algorithm

8, 15, 21, 54, 64, 75, 88, 91	P_0
12, 22, 47, 50, 54, 65, 66, 72	P_1
0, 66, 67, 70, 82, 83, 98, 99	P_2
15, 54, 75, 22, 50, 65, 66, 70, 83	

One process collects p^2 regular samples.

PSRS Algorithm

8, 15, 21, 54, 64, 75, 88, 91 — P_0

12, 22, 47, 50, 54, 65, 66, 72 — P_1

0, 66, 67, 70, 82, 83, 98, 99 — P_2

15, 22, 50, 54, 65, 66, 70, 75, 83

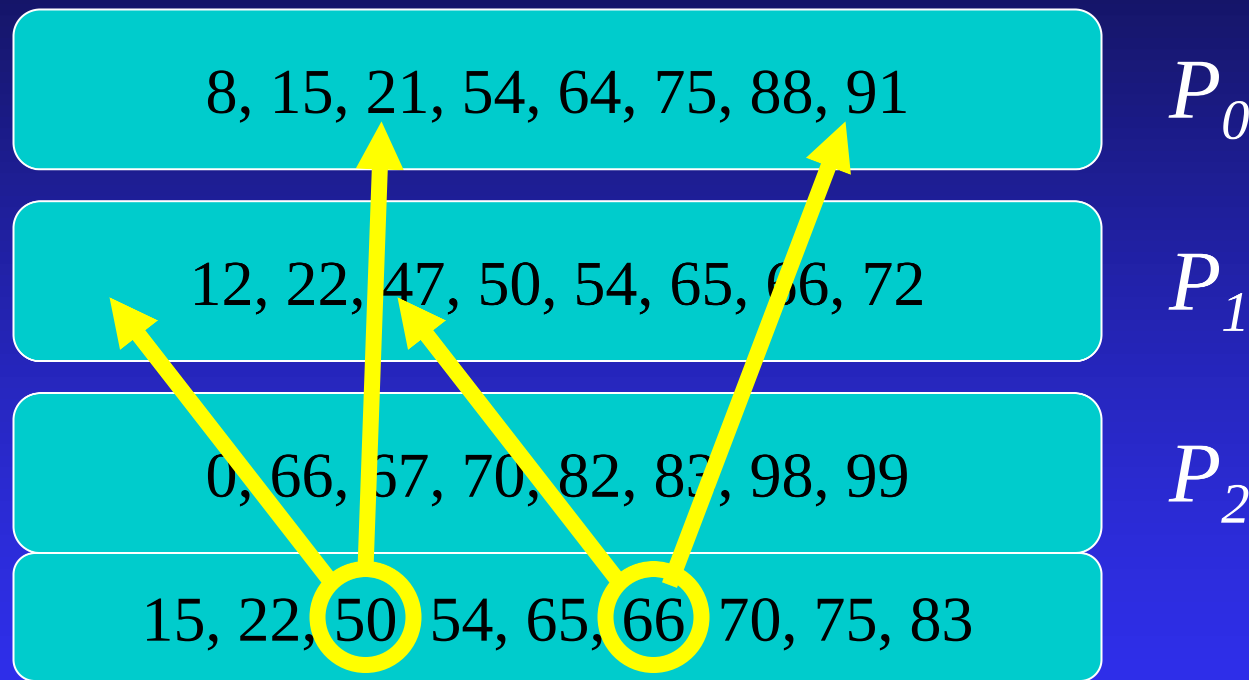
One process sorts p^2 regular samples.

PSRS Algorithm

8, 15, 21, 54, 64, 75, 88, 91	P_0
12, 22, 47, 50, 54, 65, 66, 72	P_1
0, 66, 67, 70, 82, 83, 98, 99	P_2
15, 22, 50, 54, 65, 66, 70, 75, 83	

One process chooses *p*-1 pivot values.

PSRS Algorithm



One process broadcasts *p*-1 pivot values.

PSRS Algorithm

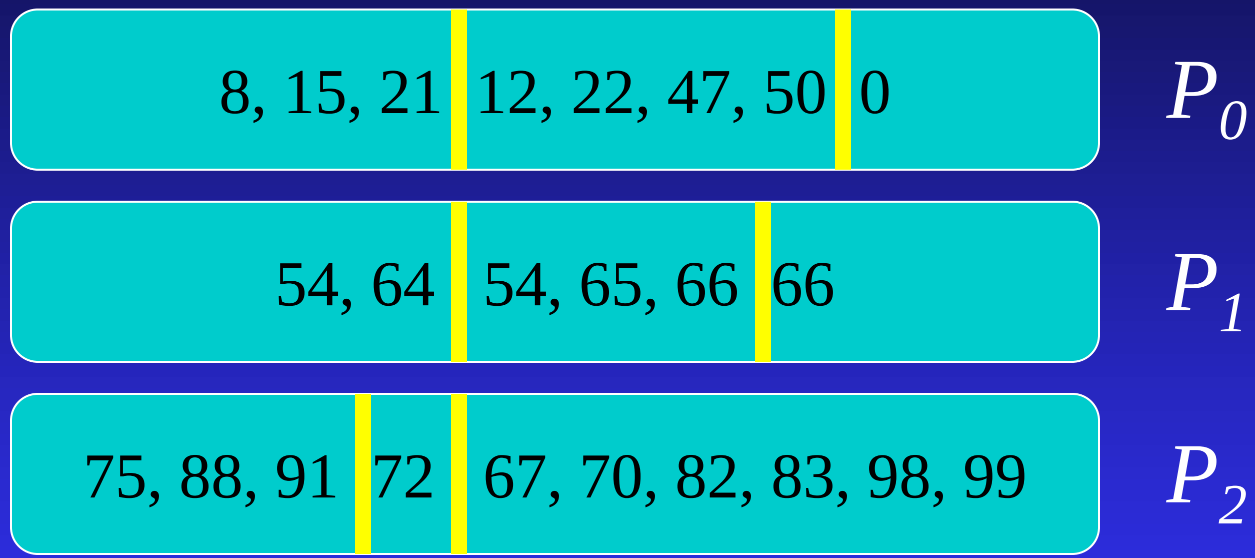
8, 15, 21, 54, 64, 75, 88, 91 — P_0

12, 22, 47, 50, 54, 65, 66, 72 — P_1

0, 66, 67, 70, 82, 83, 98, 99 — P_2

Each process divides list, based on pivot values.

PSRS Algorithm



Each process sends partitions to correct destination process.

PSRS Algorithm

0, 8, 12, 15, 21, 22, 47, 50	P_0
54, 54, 64, 65, 66, 66	P_1
67, 70, 72, 75, 82, 83, 88, 91, 98, 99	P_2

Each process merges *p* partitions.

Assumptions

- Each process ends up merging close to n/p elements
- Experimental results show this is a valid assumption
- Processor interconnection network supports *p* simultaneous message transmissions at full speed
- 4-ary hypertree is an example of such a network

Time Complexity Analysis

- Computations
  - Initial quicksort: $\Theta((n/p)\log(n/p))$
  - Sorting regular samples: $\Theta(p^2 \log p)$
  - Merging sorted sublists: $\Theta((n/p)\log p$
  - Overall: $\Theta((n/p)(\log n + \log p) + p^2\log p)$
- Communications
  - Gather samples, broadcast pivots: $\Theta(\log p)$
  - All-to-all exchange: $\Theta(n/p)$
  - Overall: $\Theta(n/p)$

Isoefficiency Analysis

- Sequential time complexity: $\Theta(n \log n)$
- Parallel overhead: $\Theta(n \log p)$
- Isoefficiency relation:
  $n \log n \geq Cn \log p \Rightarrow \log n \geq C \log p$
- Scalability function same as for hyperquicksort
- Scalability depends on ratio of communication to computation speeds

Summary

- Three parallel algorithms based on quicksort
- Keeping list sizes balanced
  - Parallel quicksort: poor
  - Hyperquicksort: better
  - PSRS algorithm: excellent
- Average number of times each key moved:
  - Parallel quicksort and hyperquicksort: log p / 2
  - PSRS algorithm: $(p-1)/p$