Parallel Sort

Parallel and Distributed Computing

Department of Computer Science and Engineering (DEI)
Instituto Superior Técnico

November 29, 2012

Outline

- Parallel Sort
  - Hyperquicksort
  - PSRS, Parallel Sorting by Regular Sampling
  - Odd-Even Transposition Sort

Sorting Problem

Sorting Problem

Given an unordered sequence, obtain an ordered one through permutations of the elements in the sequence.

Typically the value being sorted (key) is part of record with additional values (satellite data).

Efficiency of sorting is particularly important as it is used as part of many algorithms.

Sorting Algorithms

Name	Average	Worst	Memory	Stable
Bubble sort	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Selection sort	$O(n^2)$	$O(n^2)$	$O(1)$	No
Insertion sort	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(1)$	No
Quicksort	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	No

Sorting Algorithms

Name	Average	Worst	Memory	Stable
Bubble sort	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Selection sort	$O(n^2)$	$O(n^2)$	$O(1)$	No
Insertion sort	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(1)$	No
Quicksort	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	No

Smaller hidden constants in Quicksort make it popular.

Quicksort

```
procedure quicksort(array, left, right)
     if right > left
         select a pivot index (e.g. pivotIndex := left)
         pivotNewIndex := partition(array, left, right, pivotIndex)
         quicksort(array, left, pivotNewIndex - 1)
         quicksort(array, pivotNewIndex + 1, right)
```

Quicksort

```
procedure quicksort(array, left, right)
     if right > left
         select a pivot index (e.g. pivotIndex := left)
         pivotNewIndex := partition(array, left, right, pivotIndex)
         quicksort(array, left, pivotNewIndex - 1)
         quicksort(array, pivotNewIndex + 1, right)

function partition(array, left, right, pivotIndex)
     pivotValue := array[pivotIndex]
     swap array[pivotIndex] and array[right] // Move pivot to end
     storeIndex := left
     for i  from  left to right - 1
         if array[i] <= pivotValue
             swap array[i] and array[storeIndex]
             storeIndex := storeIndex + 1
     // Move pivot to its final place
     swap array[storeIndex] and array[right]
     return storeIndex
```

Quicksort

18	15	66	10	23	64	11	35

Quicksort

35	15	66	10	23	64	11	18

Quicksort

35	15	66	10	23	64	11	18

Quicksort

15	35	66	10	23	64	11	18

Quicksort

15	35	66	10	23	64	11	18

Quicksort

15	10	66	35	23	64	11	18

Quicksort

15	10	66	35	23	64	11	18

Quicksort

15	10	66	35	23	64	11	18

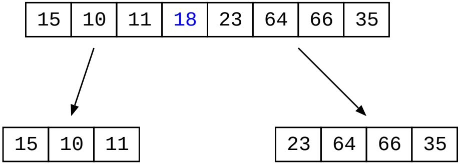
Quicksort

15	10	11	35	23	64	66	18

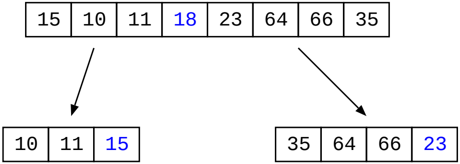
Quicksort

15	10	11	18	23	64	66	35

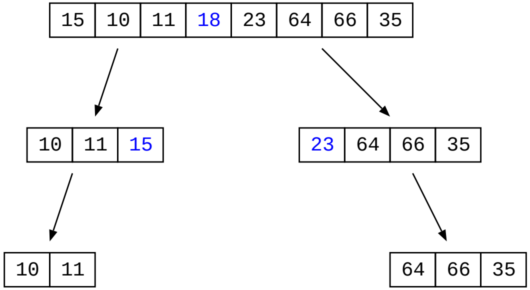
Quicksort



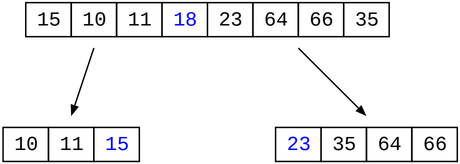
Quicksort



Quicksort



Quicksort

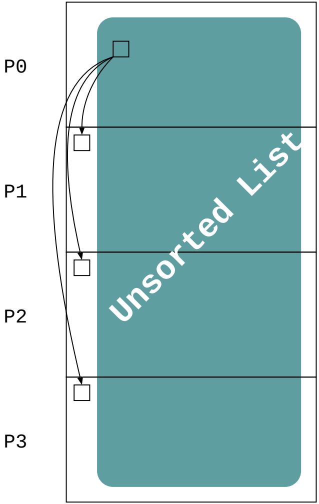


Quicksort

10	11	15	18	23	35	64	66

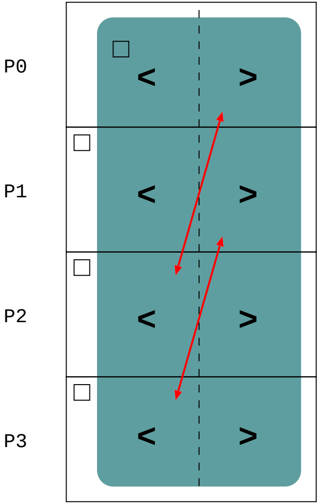
Parallel Quicksort

1. one process broadcast initial pivot to all processes



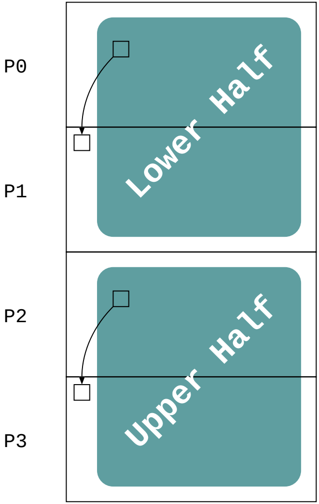
Parallel Quicksort

1. one process broadcast initial pivot to all processes
2. each process in the upper half swaps with a partner in the lower half



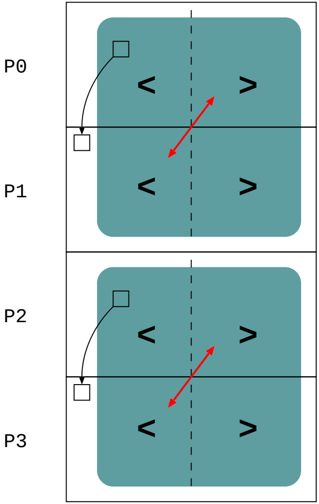
Parallel Quicksort

1. one process broadcast initial pivot to all processes
2. each process in the upper half swaps with a partner in the lower half
3. recurse on each half



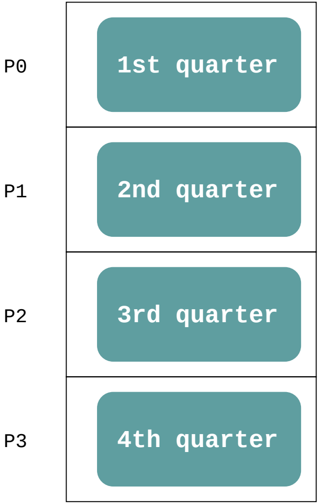
Parallel Quicksort

1. one process broadcast initial pivot to all processes
2. each process in the upper half swaps with a partner in the lower half
3. recurse on each half
4. swap among partners in each half



Parallel Quicksort

1. one process broadcast initial pivot to all processes
2. each process in the upper half swaps with a partner in the lower half
3. recurse on each half
4. swap among partners in each half
5. each process uses quicksort on local elements



Hyperquicksort

Limitation of parallel quicksort: poor balancing of list sizes.

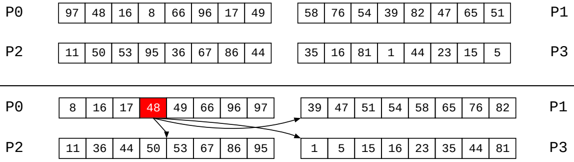
Hyperquicksort: sort elements before broadcasting pivot.

1. sort elements in each process
2. select median as pivot element and broadcast it
3. each process in the upper half swaps with a partner in the lower half
4. recurse on each half

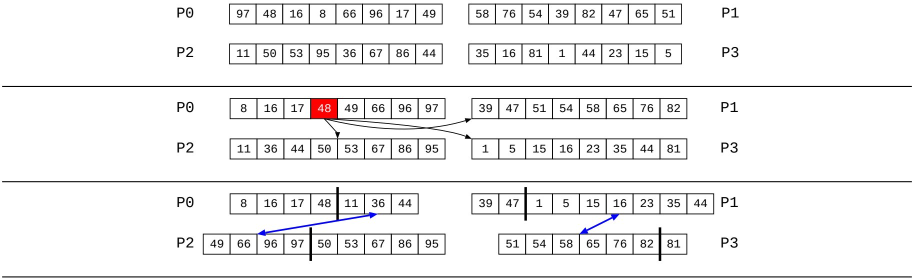
Hyperquicksort

P0	97	48	16	8	66	96	17	49		58	76	54	39	82	47	65	51	P1
P2	11	50	53	95	36	67	86	44		35	16	81	1	44	23	15	5	P3

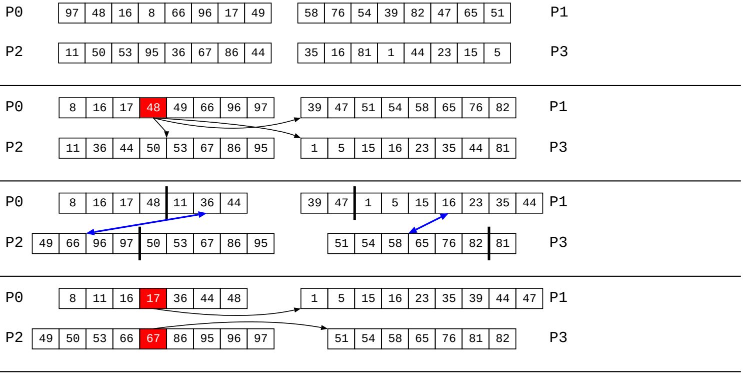
Hyperquicksort



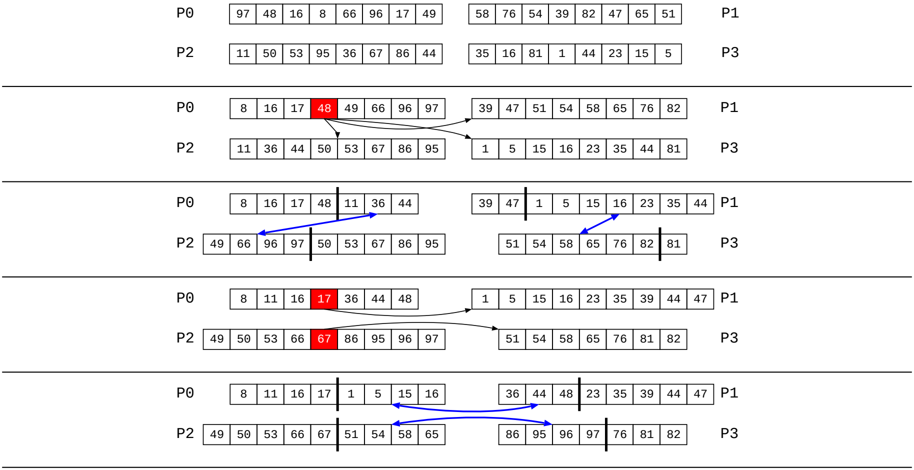
Hyperquicksort



Hyperquicksort



Hyperquicksort



Hyperquicksort

P0	97 48 16 8 66 96 17 49	58 76 54 39 82 47 65 51	P1
P2	11 50 53 95 36 67 86 44	35 16 81 1 44 23 15 5	P3

P0	8 16 17 **48** 49 66 96 97	39 47 51 54 58 65 76 82	P1
P2	11 36 44 50 53 67 86 95	1 5 15 16 23 35 44 81	P3

P0	8 16 17 48 | 11 36 44	39 47 | 1 5 15 16 23 35 44	P1
P2	49 66 96 97 | 50 53 67 86 95	51 54 58 65 76 82 | 81	P3

P0	8 11 16 **17** 36 44 48	1 5 15 16 23 35 39 44 47	P1
P2	49 50 53 66 **67** 86 95 96 97	51 54 58 65 76 81 82	P3

P0	8 11 16 17 | 1 5 15 16	36 44 48 | 23 35 39 44 47	P1
P2	49 50 53 66 67 | 51 54 58 65	86 95 96 97 | 76 81 82	P3

P0	1 5 8 11 15 16 16 17	23 35 36 39 44 44 47 48	P1
P2	49 50 51 53 54 58 65 66 67	76 81 82 86 95 96 97	P3

Complexity Analysis of Hyperquicksort

1. sort elements in each process
2. select median as pivot element and broadcast it
3. each process in the upper half swaps with a partner in the lower half
4. recurse on each half

Computation complexity:

- initial quicksort step:

Complexity Analysis of Hyperquicksort

1. sort elements in each process
2. select median as pivot element and broadcast it
3. each process in the upper half swaps with a partner in the lower half
4. recurse on each half

Computation complexity:

- initial quicksort step: $O(\frac{n}{p} \log \frac{n}{p})$
- remaining sorts:

Complexity Analysis of Hyperquicksort

1. sort elements in each process
2. select median as pivot element and broadcast it
3. each process in the upper half swaps with a partner in the lower half
4. recurse on each half

Computation complexity:

- initial quicksort step: $O(\frac{n}{p} \log \frac{n}{p})$
- remaining sorts: $O(\frac{n}{p})$

Communication time:

- pivot broadcast:

Complexity Analysis of Hyperquicksort

1. sort elements in each process
2. select median as pivot element and broadcast it
3. each process in the upper half swaps with a partner in the lower half
4. recurse on each half

Computation complexity:

- initial quicksort step: $O(\frac{n}{p} \log \frac{n}{p})$
- remaining sorts: $O(\frac{n}{p})$

Communication time:

- pivot broadcast: $O(\log p)$
- array exchange:

Complexity Analysis of Hyperquicksort

1. sort elements in each process
2. select median as pivot element and broadcast it
3. each process in the upper half swaps with a partner in the lower half
4. recurse on each half

Computation complexity:

- initial quicksort step: $O(\frac{n}{p} \log \frac{n}{p})$
- remaining sorts: $O(\frac{n}{p})$

Communication time:

- pivot broadcast: $O(\log p)$
- array exchange: $O(\frac{n}{p})$

Number of iterations:

Complexity Analysis of Hyperquicksort

1. sort elements in each process
2. select median as pivot element and broadcast it
3. each process in the upper half swaps with a partner in the lower half
4. recurse on each half

Computation complexity:

- initial quicksort step: $O(\frac{n}{p} \log \frac{n}{p})$
- remaining sorts: $O(\frac{n}{p})$

Communication time:

- pivot broadcast: $O(\log p)$
- array exchange: $O(\frac{n}{p})$

Number of iterations: $\log p$

Total time: $O(\frac{n}{p} \log n) \qquad n \gg p$

Isoefficiency Analysis of Hyperquicksort

Isoefficiency analysis: $T(n,1) \geq CT_0(n,p)$

($T(n,1)$ sequential time; $T_0(n,p)$ parallel overhead)

Sequential time complexity: $T(n,1) = O(n \log n)$

Parallel overhead dominated by exchanges: $O(\frac{n}{p} \log p)$

$$T_0(n,p) = p \times O(\frac{n}{p} \log p) = O(n \log p)$$

Isoefficiency Analysis of Hyperquicksort

Isoefficiency analysis: $T(n, 1) \geq CT_0(n, p)$

($T(n, 1)$ sequential time; $T_0(n, p)$ parallel overhead)

Sequential time complexity: $T(n, 1) = O(n \log n)$

Parallel overhead dominated by exchanges: $O(\frac{n}{p} \log p)$

$$T_0(n, p) = p \times O(\frac{n}{p} \log p) = O(n \log p)$$

$$n \log n \geq Cn \log p \quad \Rightarrow \quad n \geq p^C$$

Isoefficiency Analysis of Hyperquicksort

Isoefficiency analysis: $T(n,1) \geq CT_0(n,p)$

($T(n,1)$ sequential time; $T_0(n,p)$ parallel overhead)

Sequential time complexity: $T(n,1) = O(n \log n)$

Parallel overhead dominated by exchanges: $O(\frac{n}{p} \log p)$

$$T_0(n,p) = p \times O(\frac{n}{p} \log p) = O(n \log p)$$

$$n \log n \geq Cn \log p \quad \Rightarrow \quad n \geq p^C$$

Scalability function: $M(f(p))/p$

$$M(n) = n \quad \Rightarrow \quad \frac{M(p^C)}{p} = p^{C-1}$$

Isoefficiency Analysis of Hyperquicksort

Isoefficiency analysis: $T(n, 1) \geq CT_0(n, p)$

($T(n, 1)$ sequential time; $T_0(n, p)$ parallel overhead)

Sequential time complexity: $T(n, 1) = O(n \log n)$

Parallel overhead dominated by exchanges: $O(\frac{n}{p} \log p)$

$$T_0(n, p) = p \times O(\frac{n}{p} \log p) = O(n \log p)$$

$$n \log n \geq Cn \log p \quad \Rightarrow \quad n \geq p^C$$

Scalability function: $M(f(p))/p$

$$M(n) = n \quad \Rightarrow \quad \frac{M(p^C)}{p} = p^{C-1}$$

$\Rightarrow$ Scalability is only good for $C \leq 2$.

Isoefficiency Analysis of Hyperquicksort

Isoefficiency analysis: $T(n,1) \geq CT_0(n,p)$

($T(n,1)$ sequential time; $T_0(n,p)$ parallel overhead)

Sequential time complexity: $T(n,1) = O(n \log n)$

Parallel overhead dominated by exchanges: $O(\frac{n}{p} \log p)$

$$T_0(n,p) = p \times O(\frac{n}{p} \log p) = O(n \log p)$$

$$n \log n \geq Cn \log p \quad \Rightarrow \quad n \geq p^C$$

Scalability function: $M(f(p))/p$

$$M(n) = n \quad \Rightarrow \quad \frac{M(p^C)}{p} = p^{C-1}$$

$\Rightarrow$ Scalability is only good for $C \leq 2$.

$$C = \frac{\varepsilon(n,p)}{1 - \varepsilon(n,p)} \leq 2 \Rightarrow \varepsilon(n,p) \leq \frac{2}{3}$$

Cannot maintain high efficiency!

Limitations on the Scalability of Hyperquicksort

- 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 lowers efficiency

A better solution is to get sample values from all processes before choosing median.

Parallel Sorting by Regular Sampling

Parallel Sorting by Regular Sampling, PSRS:

- each process sorts its share of elements
- each process selects regular samples 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

Parallel Sorting by Regular Sampling

P0

15	46	48	6	93	39	72	91	14

P1

36	69	40	89	61	97	12	21	54

P2

53	97	84	58	32	27	33	72	20

Parallel Sorting by Regular Sampling

P0

15	46	48	6	93	39	72	91	14

P1

36	69	40	89	61	97	12	21	54

P2

53	97	84	58	32	27	33	72	20

P0

6	14	15	39	46	48	72	91	93

P1

12	21	36	40	54	61	69	89	97

P2

20	27	32	33	53	58	72	84	97

Parallel Sorting by Regular Sampling

P0									
	15	46	48	6	93	39	72	91	14

P1									
	36	69	40	89	61	97	12	21	54

P2									
	53	97	84	58	32	27	33	72	20

P0									
	6	14	15	39	46	48	72	91	93

P1									
	12	21	36	40	54	61	69	89	97

P2									
	20	27	32	33	53	58	72	84	97

---

6 39 72 12 40 69 20 33 72 $\longrightarrow$ 6 12 20 $\boxed{33}$ 39 40 $\boxed{69}$ 72 72

Parallel Sorting by Regular Sampling

P0	15	46	48	6	93	39	72	91	14
P1	36	69	40	89	61	97	12	21	54
P2	53	97	84	58	32	27	33	72	20

P0	**6**	14	15	**39**	46	48	**72**	91	93
P1	**12**	21	36	**40**	54	61	**69**	89	97
P2	**20**	27	32	**33**	53	58	**72**	84	97

---

6 39 72 12 40 69 20 33 72 $\longrightarrow$ 6 12 20 $\boxed{33}$ 39 40 $\boxed{69}$ 72 72

P0	6	14	15	| 33	39	46	48	| 69	72	91	93
P1	12	21	| 33	36	40	54	61	69	| 69	89	97
P2	20	27	32	33	| 33	53	58	| 69	72	84	97

Parallel Sorting by Regular Sampling

P0	15	46	48	6	93	39	72	91	14
P1	36	69	40	89	61	97	12	21	54
P2	53	97	84	58	32	27	33	72	20

P0	**6**	14	15	**39**	46	48	**72**	91	93
P1	**12**	21	36	**40**	54	61	**69**	89	97
P2	**20**	27	32	**33**	53	58	**72**	84	97

---

6 39 72 12 40 69 20 33 72 $\longrightarrow$ 6 12 20 $\boxed{33}$ 39 40 $\boxed{69}$ 72 72

P0: 6 14 15 | 39 46 48 | 72 91 93 (33, 69)

P1: 12 21 | 36 40 54 61 69 | 89 97 (33, 69)

P2: 20 27 32 33 | 53 58 | 72 84 97 (33, 69)

P0: 6 14 15 | 12 21 | 20 27 32 33

P1: 39 46 48 | 36 40 54 61 69 | 53 58

P2: 72 91 93 | 89 97 | 72 84 97

Parallel Sorting by Regular Sampling

P0								
15	46	48	6	93	39	72	91	14

P1								
36	69	40	89	61	97	12	21	54

P2								
53	97	84	58	32	27	33	72	20

P0								
6	14	15	**39**	46	48	**72**	91	93

P1								
12	21	36	**40**	54	61	**69**	89	97

P2								
20	27	32	**33**	53	58	**72**	84	97

---

6 39 72 12 40 69 20 33 72 $\longrightarrow$ 6 12 20 $\boxed{33}$ 39 40 $\boxed{69}$ 72 72

P0								
6	14	15	39	46	48	72	91	93

33 | 69

P1								
12	21	36	40	54	61	69	89	97

33 | 69

P2								
20	27	32	33	53	58	72	84	97

33 | 69

P0								
6	14	15	12	21	20	27	32	33

P1									
39	46	48	36	40	54	61	69	53	58

P2							
72	91	93	89	97	72	84	97

P0								
6	12	14	15	20	21	27	32	33

P1									
36	39	40	46	48	53	54	58	61	69

P2							
72	72	84	89	91	93	97	97

Complexity Analysis of PSRS

Computation complexity:

- initial quicksort step:

Complexity Analysis of PSRS

Computation complexity:

- initial quicksort step: $O(\frac{n}{p}\log\frac{n}{p})$
- sorting samples:

Complexity Analysis of PSRS

Computation complexity:

- initial quicksort step: $O(\frac{n}{p}\log\frac{n}{p})$
- sorting samples: $O(p^2 \log p)$
- merging subarrays:

Complexity Analysis of PSRS

Computation complexity:

- initial quicksort step: $O(\frac{n}{p}\log\frac{n}{p})$
- sorting samples: $O(p^2 \log p)$
- merging subarrays: $O(\frac{n}{p}\log p)$

Communication time:

- gather samples:

Complexity Analysis of PSRS

Computation complexity:

- initial quicksort step: $O(\frac{n}{p}\log\frac{n}{p})$
- sorting samples: $O(p^2\log p)$
- merging subarrays: $O(\frac{n}{p}\log p)$

Communication time:

- gather samples: $O(p\log p)$
- pivot broadcast:

Complexity Analysis of PSRS

Computation complexity:

- initial quicksort step: $O(\frac{n}{p}\log\frac{n}{p})$
- sorting samples: $O(p^2\log p)$
- merging subarrays: $O(\frac{n}{p}\log p)$

Communication time:

- gather samples: $O(p\log p)$
- pivot broadcast: $O(p\log p)$
- array exchange:

Complexity Analysis of PSRS

Computation complexity:

- initial quicksort step: $O(\frac{n}{p}\log\frac{n}{p})$
- sorting samples: $O(p^2\log p)$
- merging subarrays: $O(\frac{n}{p}\log p)$

Communication time:

- gather samples: $O(p\log p)$
- pivot broadcast: $O(p\log p)$
- array exchange: $O(\frac{n}{p})$

Total time: $O(\frac{n}{p}\log n)$

Isoefficiency Analysis of PSRS

Isoefficiency analysis: $T(n,1) \geq CT_0(n,p)$

Sequential time complexity: $T(n,1) = O(n \log n)$

Parallel overhead:
communication dominated by exchanges: $O(\frac{n}{p})$
redundant computation of last merge: $O(\frac{n}{p} \log p)$

$$T_0(n,p) = p \times O(\frac{n}{p} \log p) = O(n \log p)$$

$$n \log n \geq Cn \log p \quad \Rightarrow \quad n \geq p^C$$

Scalability function: $M(f(p))/p$

$$M(n) = n \quad \Rightarrow \quad \frac{M(p^C)}{p} = p^{C-1}$$

$\Rightarrow$ Same scalability as hyperquicksort.

Comparison of Parallel QuickSorting Algorithms

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: $\frac{\log p}{2}$
- PSRS algorithm: $\frac{p-1}{p}$

Odd-Even Transposition Sort

- Based on Bubble Sort
- Consists of two distinct phases:

  Even phase

  compare and swaps are executed on pairs (a[0],a[1]); (a[2],a[3]); (a[4],a[5]);...

  Odd phase

  compare and swaps are executed on pairs (a[1],a[2]); (a[3],a[4]); (a[5],a[6]);...
- Sorting is complete after at most n phases

Odd-Even Transposition Sort Example

18	15	22	10	23	11

Odd-Even Transposition Sort Example

	18	15	22	10	23	11
even phase	15	18	10	22	11	23

Odd-Even Transposition Sort Example

	18	15	22	10	23	11
even phase	15	18	10	22	11	23
odd phase	15	10	18	11	22	23

Odd-Even Transposition Sort Example

	18	15	22	10	23	11
even phase	15	18	10	22	11	23
odd phase	15	10	18	11	22	23
even phase	10	15	11	18	22	23

Odd-Even Transposition Sort Example

	18	15	22	10	23	11
even phase	15	18	10	22	11	23
odd phase	15	10	18	11	22	23
even phase	10	15	11	18	22	23
odd phase	10	11	15	18	22	23

Odd-Even Transposition Sort Example

	18	15	22	10	23	11
even phase	15	18	10	22	11	23
odd phase	15	10	18	11	22	23
even phase	10	15	11	18	22	23
odd phase	10	11	15	18	22	23
even phase	10	11	15	18	22	23

Odd-Even Transposition Sort Program

```
void Odd_even_sort(int a[], int n)
{
  int phase, i;

  for (phase = 0; phase < n; phase++)
    if (phase % 2 == 0) /* even phase */
      for (i = 1; i < n; i += 2)
        if (a[i-1] > a[i])
          swap(&a[i], &a[i-1]);
    else /* odd phase */
      for (i = 1; i < n-1; i += 2)
        if (a[i] > a[i+1])
          swap(&a[i], &a[i+1]);
}
```

Odd-Even Transposition Sort Program

```
void Odd_even_sort(int a[], int n)
{
  int phase, i;

  for (phase = 0; phase < n; phase++)
    if (phase % 2 == 0) /* even phase */
      for (i = 1; i < n; i += 2)
        if (a[i-1] > a[i])
          swap(&a[i], &a[i-1]);
    else /* odd phase */
      for (i = 1; i < n-1; i += 2)
        if (a[i] > a[i+1])
          swap(&a[i], &a[i+1]);
}
```

Complexity of sequencial algorihtm:

Odd-Even Transposition Sort Program

```
void Odd_even_sort(int a[], int n)
{
  int phase, i;

  for (phase = 0; phase < n; phase++)
    if (phase % 2 == 0) /* even phase */
      for (i = 1; i < n; i += 2)
        if (a[i-1] > a[i])
          swap(&a[i], &a[i-1]);
    else /* odd phase */
      for (i = 1; i < n-1; i += 2)
        if (a[i] > a[i+1])
          swap(&a[i], &a[i+1]);
}
```

Complexity of sequencial algorihtm: $O(n^2)$

Can we do better in the parallel algorithm?

Parallel Odd-Even Transposition Sort

Partitioning:

Parallel Odd-Even Transposition Sort

Partitioning:

Primitive task is to determine value of $a[i]$ at the end of each phase

Parallel Odd-Even Transposition Sort

Partitioning:

Primitive task is to determine value of $a[i]$ at the end of each phase

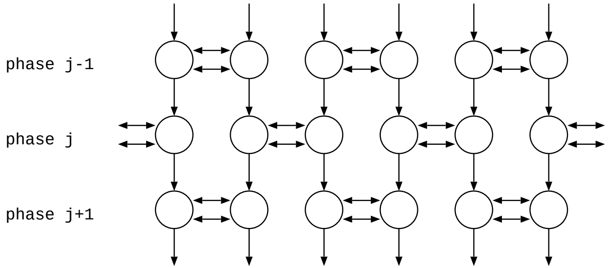
Communication:

Parallel Odd-Even Transposition Sort

Partitioning:

Primitive task is to determine value of $a[i]$ at the end of each phase

Communication:

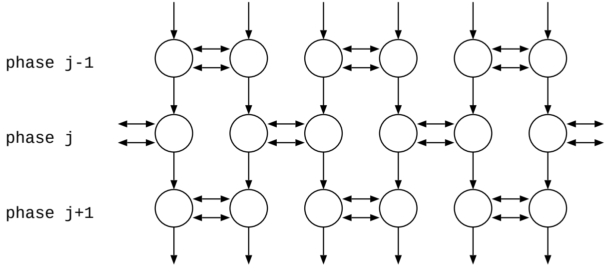


Parallel Odd-Even Transposition Sort

Partitioning:

Primitive task is to determine value of $a[i]$ at the end of each phase

Communication:



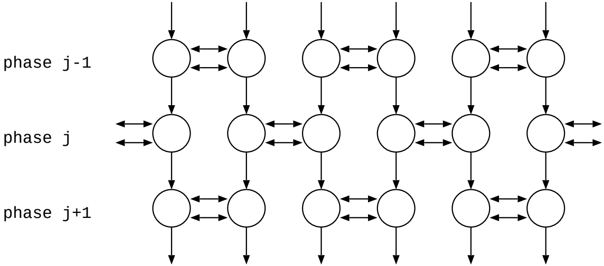
Aglomeration and Mapping:

Parallel Odd-Even Transposition Sort

Partitioning:

Primitive task is to determine value of $a[i]$ at the end of each phase

Communication:



Aglomeration and Mapping:

Distibute values of array through processors

Parallel Odd-Even Transposition Sort Example

	P0	P1	P2	P3
Start	15, 11, 9, 16	3, 14, 8, 7	4, 6, 12, 10	5, 2, 13, 1

Parallel Odd-Even Transposition Sort Example

	P0	P1	P2	P3
Start	15, 11, 9, 16	3, 14, 8, 7	4, 6, 12, 10	5, 2, 13, 1
After local sort	9, 11, 15, 16	3, 7, 8, 14	4, 6, 10, 12	1, 2, 5, 13

Parallel Odd-Even Transposition Sort Example

	P0	P1	P2	P3
Start	15, 11, 9, 16	3, 14, 8, 7	4, 6, 12, 10	5, 2, 13, 1
After local sort	9, 11, 15, 16	3, 7, 8, 14	4, 6, 10, 12	1, 2, 5, 13
After phase 0	3, 7, 8, 9	11, 14, 15, 16	1, 2, 4, 5	6, 10, 12, 13

Parallel Odd-Even Transposition Sort Example

	P0	P1	P2	P3
Start	15, 11, 9, 16	3, 14, 8, 7	4, 6, 12, 10	5, 2, 13, 1
After local sort	9, 11, 15, 16	3, 7, 8, 14	4, 6, 10, 12	1, 2, 5, 13
After phase 0	3, 7, 8, 9	11, 14, 15, 16	1, 2, 4, 5	6, 10, 12, 13
After phase 1	3, 7, 8, 9	1, 2, 4, 5	11, 14, 15, 16	6, 10, 12, 13

Parallel Odd-Even Transposition Sort Example

	P0	P1	P2	P3
Start	15, 11, 9, 16	3, 14, 8, 7	4, 6, 12, 10	5, 2, 13, 1
After local sort	9, 11, 15, 16	3, 7, 8, 14	4, 6, 10, 12	1, 2, 5, 13
After phase 0	3, 7, 8, 9	11, 14, 15, 16	1, 2, 4, 5	6, 10, 12, 13
After phase 1	3, 7, 8, 9	1, 2, 4, 5	11, 14, 15, 16	6, 10, 12, 13
After phase 2	1, 2, 3, 4	5, 7, 8, 9	6, 10, 11, 12	13, 14, 15, 16

Parallel Odd-Even Transposition Sort Example

	P0	P1	P2	P3
Start	15, 11, 9, 16	3, 14, 8, 7	4, 6, 12, 10	5, 2, 13, 1
After local sort	9, 11, 15, 16	3, 7, 8, 14	4, 6, 10, 12	1, 2, 5, 13
After phase 0	3, 7, 8, 9	11, 14, 15, 16	1, 2, 4, 5	6, 10, 12, 13
After phase 1	3, 7, 8, 9	1, 2, 4, 5	11, 14, 15, 16	6, 10, 12, 13
After phase 2	1, 2, 3, 4	5, 7, 8, 9	6, 10, 11, 12	13, 14, 15, 16
After phase 3	1, 2, 3, 4	5, 6, 7, 8	9, 10, 11, 12	13, 14, 15, 16

Complexity Analysis of Odd-Even Sort

1. sort elements in each process
2. send/receive values to/from partner process
3. if rank of process is smaller than rank of partner then keep smaller values
4. otherwise keep larger values

Computational complexity:

- initial quicksort:

Complexity Analysis of Odd-Even Sort

1. sort elements in each process
2. send/receive values to/from partner process
3. if rank of process is smaller than rank of partner then keep smaller values
4. otherwise keep larger values

Computational complexity:

- initial quicksort: $O(\frac{n}{p} \log \frac{n}{p})$
- sorting smaller/larger values in each phase:

Complexity Analysis of Odd-Even Sort

1. sort elements in each process
2. send/receive values to/from partner process
3. if rank of process is smaller than rank of partner then keep smaller values
4. otherwise keep larger values

Computational complexity:

- initial quicksort: $O(\frac{n}{p} \log \frac{n}{p})$
- sorting smaller/larger values in each phase: $O(\frac{n}{p})$

Communication time:

- Sending/receving values in each phase:

Complexity Analysis of Odd-Even Sort

1. sort elements in each process
2. send/receive values to/from partner process
3. if rank of process is smaller than rank of partner then keep smaller values
4. otherwise keep larger values

Computational complexity:

- initial quicksort: $O(\frac{n}{p} \log \frac{n}{p})$
- sorting smaller/larger values in each phase: $O(\frac{n}{p})$

Communication time:

- Sending/receving values in each phase: $O(\frac{n}{p})$

Number of phases:

Complexity Analysis of Odd-Even Sort

1. sort elements in each process
2. send/receive values to/from partner process
3. if rank of process is smaller than rank of partner then keep smaller values
4. otherwise keep larger values

Computational complexity:

- initial quicksort: $O(\frac{n}{p} \log \frac{n}{p})$
- sorting smaller/larger values in each phase: $O(\frac{n}{p})$

Communication time:

- Sending/receving values in each phase: $O(\frac{n}{p})$

Number of phases: p

Total time: $O(\frac{n}{p} \log n + n)$

Isoefficiency Analysis of Odd-Even Sort

Isoefficiency analysis: $T(n, 1) \geq CT_0(n, p)$

Sequential time complexity: $T(n, 1) = O(n \log n)$

Parallel overhead dominated by exchanges: $O(n)$

$$T_0(n, p) = p \times O(n) = O(pn)$$

$$n \log n \geq Cpn \quad \Rightarrow \quad n \geq e^{Cp}$$

Scalability function: $M(f(p))/p$

$$M(n) = n \quad \Rightarrow \quad \frac{M(e^{Cp})}{p} = \frac{e^{Cp}}{p}$$

$\Rightarrow$ Poor scalability.

Comparison of Parallel Sorting Algorithms

Hyperquicksort

Total time: $O(\frac{n}{p}\log n)$

Scalability function: p^{C-1}

$\Rightarrow$ Scalability is only good for $C \leq 2$.

PSRS

Total time: $O(\frac{n}{p}\log n)$

Scalability function: p^{C-1}

$\Rightarrow$ Same scalability as hyperquicksort.

Odd-Even Sort

Total time: $O(\frac{n}{p}\log n + n)$

Scalability function: $\frac{e^{Cp}}{p}$

$\Rightarrow$ Poor scalability.

Review

- Parallel Sort
  - Hyperquicksort
  - PSRS, Parallel Sorting by Regular Sampling
  - Odd-Even Transposition Sort

Next Class

- Efficient parallelization of numerical algorithms