Parallel Sorting

CPS343

Parallel and High Performance Computing

Spring 2020

Outline

1. Overview of Sorting
   - Sorting Characteristics
   - Bubble Sort
   - Quicksort
   - Mergesort
2. Parallel Sorting
   - Assumptions
   - Parallel Quicksort
   - Parallel Sample Sort
   - Parallel Sorting by Regular Sampling (PSRS)

Outline

Sorting Terms

- Sorting involves rearranging **records** into some specified order.

Sorting Terms

- Sorting involves rearranging **records** into some specified order.
- The **key** is used to determine the order; typically we sort so that they keys are in nondecreasing order. If a_i is the key for the i^{th} record, then after the sort $a_i \leq a_j$ if $i < j$.

Sorting Terms

- Sorting involves rearranging **records** into some specified order.
- The **key** is used to determine the order; typically we sort so that they keys are in nondecreasing order. If a_i is the key for the i^{th} record, then after the sort $a_i \leq a_j$ if $i < j$.
- The other data in the record is called **satellite data**.

Sorting Terms

- Sorting involves rearranging **records** into some specified order.
- The **key** is used to determine the order; typically we sort so that they keys are in nondecreasing order. If a_i is the key for the i^{th} record, then after the sort $a_i \leq a_j$ if $i < j$.
- The other data in the record is called **satellite data**.
- If the entire sorting operation can be carried out using main memory, the sort is called an **internal sort**.

Sorting Terms

- Sorting involves rearranging **records** into some specified order.
- The **key** is used to determine the order; typically we sort so that they keys are in nondecreasing order. If a_i is the key for the i^{th} record, then after the sort $a_i \leq a_j$ if $i < j$.
- The other data in the record is called **satellite data**.
- If the entire sorting operation can be carried out using main memory, the sort is called an **internal sort**.
- If the data set is too large for this, then must use an **external sort**.

Sorting Terms

- Sorting involves rearranging **records** into some specified order.
- The **key** is used to determine the order; typically we sort so that they keys are in nondecreasing order. If a_i is the key for the i^{th} record, then after the sort $a_i \leq a_j$ if $i < j$.
- The other data in the record is called **satellite data**.
- If the entire sorting operation can be carried out using main memory, the sort is called an **internal sort**.
- If the data set is too large for this, then must use an **external sort**.
- A sorting algorithm is called **in place** if no additional memory is required for the sort. (Well, a small amount of additional memory may be required, but the amount is not a function of the size of the list to be sorted.)

Sorting Terms

- Sorting involves rearranging **records** into some specified order.
- The **key** is used to determine the order; typically we sort so that they keys are in nondecreasing order. If a_i is the key for the i^{th} record, then after the sort $a_i \leq a_j$ if $i < j$.
- The other data in the record is called **satellite data**.
- If the entire sorting operation can be carried out using main memory, the sort is called an **internal sort**.
- If the data set is too large for this, then must use an **external sort**.
- A sorting algorithm is called **in place** if no additional memory is required for the sort. (Well, a small amount of additional memory may be required, but the amount is not a function of the size of the list to be sorted.)
- A sorting algorithm is called **stable** if the relative order of elements with equal keys is unchanged by the sorting algorithm.

Compare and Exchange

- A basic operation in many sorting algorithms is *compare and exchange*

Compare and Exchange

- A basic operation in many sorting algorithms is *compare and exchange*
- Suppose we're trying to sort a list in increasing order. The key operation (assuming $i < j$) is

$$
\begin{aligned}
&\text{if } a_i > a_j \text{ then} \\
&\qquad t = a_i \\
&\qquad a_i = a_j \\
&\qquad a_j = t \\
&\text{endif}
\end{aligned}
$$

Compare and Exchange

- A basic operation in many sorting algorithms is *compare and exchange*
- Suppose we're trying to sort a list in increasing order. The key operation (assuming $i < j$) is

$$\begin{array}{l} \text{if } a_i > a_j \text{ then} \\ \qquad t = a_i \\ \qquad a_i = a_j \\ \qquad a_j = t \\ \text{endif} \end{array}$$

- This operation is called a *swap* and is written swap(x,y)

Compare and Exchange

- A basic operation in many sorting algorithms is *compare and exchange*
- Suppose we're trying to sort a list in increasing order. The key operation (assuming $i < j$) is

$$
\begin{aligned}
&\text{if } a_i > a_j \text{ then} \\
&\qquad t = a_i \\
&\qquad a_i = a_j \\
&\qquad a_j = t \\
&\text{endif}
\end{aligned}
$$

- This operation is called a *swap* and is written swap(x,y)
- An example that uses this approach is the bubble sort

Compare and Exchange

- A basic operation in many sorting algorithms is *compare and exchange*
- Suppose we're trying to sort a list in increasing order. The key operation (assuming $i < j$) is

if $a_i > a_j$ then
$t = a_i$
$a_i = a_j$
$a_j = t$
endif

- This operation is called a *swap* and is written swap(x,y)
- An example that uses this approach is the bubble sort
- Other sorts (e.g. mergesort) do not have compare and exchange as a basic operation.

Outline

1. Overview of Sorting
   - Sorting Characteristics
   - Bubble Sort
   - Quicksort
   - Mergesort
2. Parallel Sorting
   - Assumptions
   - Parallel Quicksort
   - Parallel Sample Sort
   - Parallel Sorting by Regular Sampling (PSRS)

Bubble Sort

- The Bubble sort gets its name from the way smaller list elements "bubble" up in the list throughout the sort

Bubble Sort

- The Bubble sort gets its name from the way smaller list elements "bubble" up in the list throughout the sort
- Given an n element list of integers a_i, $i = 0, 1, \ldots, n-1$:

```
done = false
last = n − 2
while not done do
    done = true
    for i = 0 to last do
        if a_i > a_{i+1} then
            done = false
            swap(a_i, a_{i+1})
        endif
    endfor
    last = last − 1
endwhile
```

Bubble Sort

For example, consider the bubble sort applied to a list of eight numbers. The number in **bold** in each column represents a_i and is compared with the next number in the list.

5
7
2
8
4
1
3
6

Bubble Sort

For example, consider the bubble sort applied to a list of eight numbers. The number in **bold** in each column represents a_i and is compared with the next number in the list.

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

Bubble Sort

For example, consider the bubble sort applied to a list of eight numbers. The number in **bold** in each column represents a_i and is compared with the next number in the list.

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

Bubble Sort

For example, consider the bubble sort applied to a list of eight numbers. The number in **bold** in each column represents a_i and is compared with the next number in the list.

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

Bubble Sort

For example, consider the bubble sort applied to a list of eight numbers. The number in **bold** in each column represents a_i and is compared with the next number in the list.

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

Bubble Sort

For example, consider the bubble sort applied to a list of eight numbers. The number in **bold** in each column represents a_i and is compared with the next number in the list.

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

Bubble Sort

For example, consider the bubble sort applied to a list of eight numbers. The number in **bold** in each column represents a_i and is compared with the next number in the list.

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

Bubble Sort

For example, consider the bubble sort applied to a list of eight numbers. The number in **bold** in each column represents a_i and is compared with the next number in the list.

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

Bubble Sort

The second sweep looks like

5
2
7
4
1
3
6
8

Bubble Sort

The second sweep looks like

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

Bubble Sort

The second sweep looks like

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

Bubble Sort

The second sweep looks like

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

Bubble Sort

The second sweep looks like

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

Bubble Sort

The second sweep looks like

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

Bubble Sort

The second sweep looks like

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

Bubble Sort

The second sweep looks like

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

Notice that we have one fewer comparison since the largest element in the list is already at the bottom.

Bubble Sort

The third sweep looks like

2
5
4
1
3
6
7
8

Bubble Sort

The third sweep looks like

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

Bubble Sort

The third sweep looks like

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

Bubble Sort

The third sweep looks like

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

Bubble Sort

The third sweep looks like

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

Bubble Sort

The third sweep looks like

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

Bubble Sort

The third sweep looks like

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

Now the last three elements are all in place. After four more sweeps the list will be sorted.

Bubble Sort

- Although simple in concept, the bubble sort is not very efficient.

Bubble Sort

- Although simple in concept, the bubble sort is not very efficient.
- In the worst case the outer loop is done $n - 1$ times.

Bubble Sort

- Although simple in concept, the bubble sort is not very efficient.
- In the worst case the outer loop is done $n - 1$ times.
- The first time through the inner loop body is executed $n - 1$ times.

Bubble Sort

- Although simple in concept, the bubble sort is not very efficient.
- In the worst case the outer loop is done $n - 1$ times.
- The first time through the inner loop body is executed $n - 1$ times.
- Each subsequent time the inner loop body is executed one fewer time.

Bubble Sort

- Although simple in concept, the bubble sort is not very efficient.
- In the worst case the outer loop is done $n - 1$ times.
- The first time through the inner loop body is executed $n - 1$ times.
- Each subsequent time the inner loop body is executed one fewer time.
- This means that in the worst case the number comparisons is $(n-1) + (n-2) + \cdots + 2 + 1 = n(n-1)/2$ which is $O(n^2)$.

Bubble Sort

- Although simple in concept, the bubble sort is not very efficient.
- In the worst case the outer loop is done $n-1$ times.
- The first time through the inner loop body is executed $n-1$ times.
- Each subsequent time the inner loop body is executed one fewer time.
- This means that in the worst case the number comparisons is $(n-1)+(n-2)+\cdots+2+1 = n(n-1)/2$ which is $O(n^2)$.
- The number of comparisons in the average case is also $O(n^2)$.

Outline

1. Overview of Sorting
   - Sorting Characteristics
   - Bubble Sort
   - Quicksort
   - Mergesort
2. Parallel Sorting
   - Assumptions
   - Parallel Quicksort
   - Parallel Sample Sort
   - Parallel Sorting by Regular Sampling (PSRS)

Quicksort

- Uses divide and conquer strategy.

Quicksort

- Uses divide and conquer strategy.
- Given a list of n elements, choose one element as a *pivot*.

Quicksort

- Uses divide and conquer strategy.
- Given a list of n elements, choose one element as a *pivot*.
- Reorder list so all elements smaller than the pivot come before it and all elements larger than the pivot come after it.

Quicksort

- Uses divide and conquer strategy.
- Given a list of n elements, choose one element as a *pivot*.
- Reorder list so all elements smaller than the pivot come before it and all elements larger than the pivot come after it.
- Pivot is now in its proper place and the list has been partitioned into two sublists that must still be sorted.

Quicksort

- Uses divide and conquer strategy.
- Given a list of n elements, choose one element as a *pivot*.
- Reorder list so all elements smaller than the pivot come before it and all elements larger than the pivot come after it.
- Pivot is now in its proper place and the list has been partitioned into two sublists that must still be sorted.
- Quicksort is naturally recursive.

Quicksort

- Uses divide and conquer strategy.
- Given a list of n elements, choose one element as a *pivot*.
- Reorder list so all elements smaller than the pivot come before it and all elements larger than the pivot come after it.
- Pivot is now in its proper place and the list has been partitioned into two sublists that must still be sorted.
- Quicksort is naturally recursive.
- Average number of comparisons is $O(n \log n)$.

Quicksort

- Uses divide and conquer strategy.
- Given a list of n elements, choose one element as a *pivot*.
- Reorder list so all elements smaller than the pivot come before it and all elements larger than the pivot come after it.
- Pivot is now in its proper place and the list has been partitioned into two sublists that must still be sorted.
- Quicksort is naturally recursive.
- Average number of comparisons is $O(n \log n)$.
- Worst case number of comparisons is $O(n^2)$.

Quicksort

Often the first element in the list is used as the pivot, although other mechanisms can be used. Ideally pivot is median of sorted list.

Here red indicates the selected pivot and elements in blue are previous pivots that now partition the list.

5
7
2
8
4
1
3
6

Quicksort

Often the first element in the list is used as the pivot, although other mechanisms can be used. Ideally pivot is median of sorted list.

Here red indicates the selected pivot and elements in blue are previous pivots that now partition the list.

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

Quicksort

Often the first element in the list is used as the pivot, although other mechanisms can be used. Ideally pivot is median of sorted list.

Here red indicates the selected pivot and elements in blue are previous pivots that now partition the list.

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

Quicksort

Often the first element in the list is used as the pivot, although other mechanisms can be used. Ideally pivot is median of sorted list.

Here red indicates the selected pivot and elements in blue are previous pivots that now partition the list.

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

Quicksort

Often the first element in the list is used as the pivot, although other mechanisms can be used. Ideally pivot is median of sorted list.

Here red indicates the selected pivot and elements in blue are previous pivots that now partition the list.

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

Quicksort

Often the first element in the list is used as the pivot, although other mechanisms can be used. Ideally pivot is median of sorted list.

Here red indicates the selected pivot and elements in blue are previous pivots that now partition the list.

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

Quicksort

Often the first element in the list is used as the pivot, although other mechanisms can be used. Ideally pivot is median of sorted list.

Here red indicates the selected pivot and elements in blue are previous pivots that now partition the list.

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

Outline

Mergesort

- Like Quicksort, Mergesort uses a divide and conquer strategy, and is naturally recursive.

Mergesort

- Like Quicksort, Mergesort uses a divide and conquer strategy, and is naturally recursive.
- Basic algorithm:

Mergesort

- Like Quicksort, Mergesort uses a divide and conquer strategy, and is naturally recursive.
- Basic algorithm:
  1. if list length is 0 or 1 then return.

Mergesort

- Like Quicksort, Mergesort uses a divide and conquer strategy, and is naturally recursive.
- Basic algorithm:
  1. if list length is 0 or 1 then return.
  2. partition list into two sublists of (nearly) equal size.

Mergesort

- Like Quicksort, Mergesort uses a divide and conquer strategy, and is naturally recursive.
- Basic algorithm:
  1. if list length is 0 or 1 then return.
  2. partition list into two sublists of (nearly) equal size.
  3. sort each sublist using mergesort.

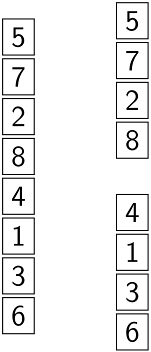
Mergesort

- Like Quicksort, Mergesort uses a divide and conquer strategy, and is naturally recursive.
- Basic algorithm:
  1. if list length is 0 or 1 then return.
  2. partition list into two sublists of (nearly) equal size.
  3. sort each sublist using mergesort.
  4. merge both sublists into one sorted list.

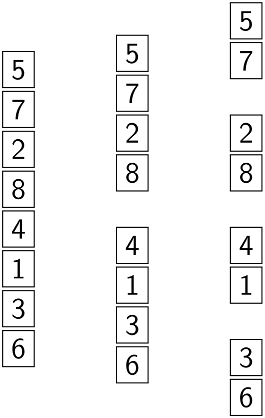
Mergesort

$\boxed{5}$
$\boxed{7}$
$\boxed{2}$
$\boxed{8}$
$\boxed{4}$
$\boxed{1}$
$\boxed{3}$
$\boxed{6}$

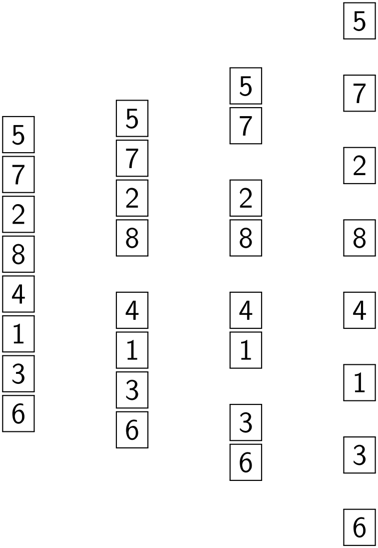
Mergesort

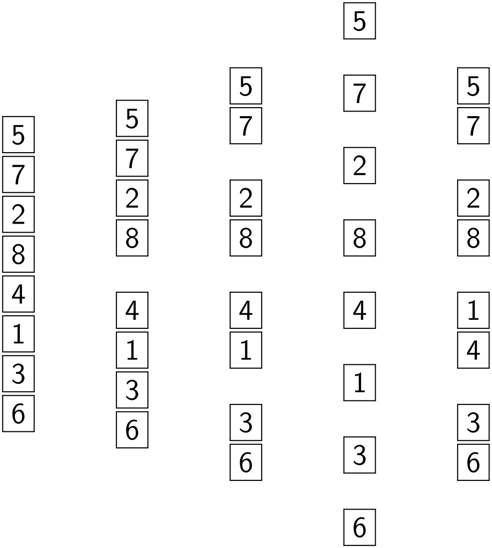


Mergesort

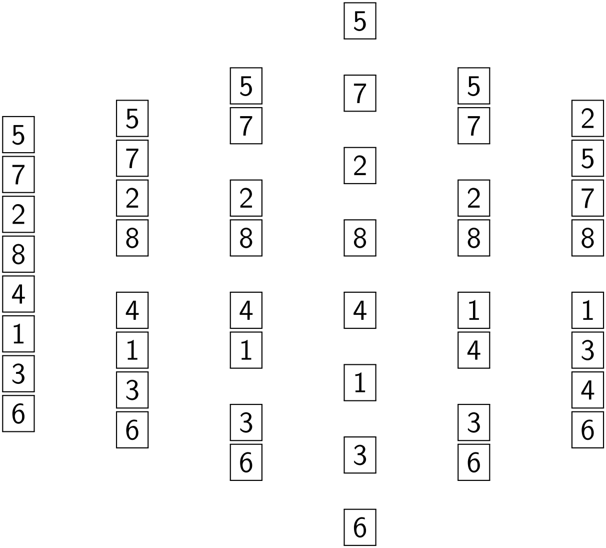


Mergesort

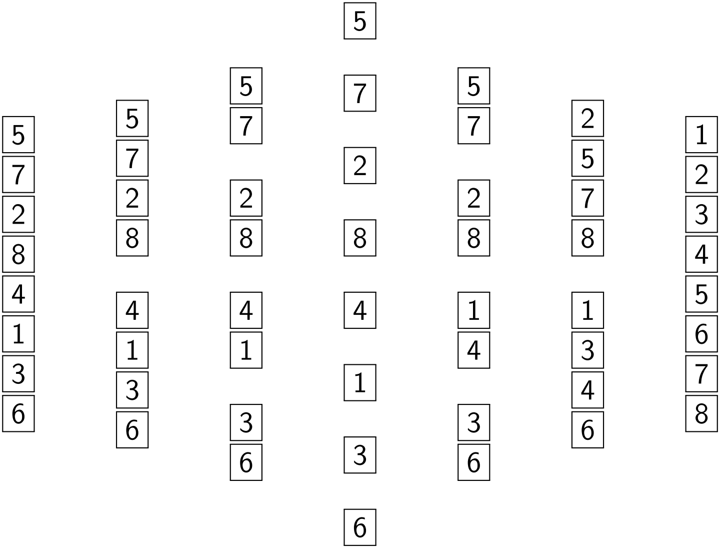



5 7 2 8 4 1 3 6
5 7 2 8
4 1 3 6
5 7
2 8
4 1
3 6
5
7
2
8
4
1
3
6
5 7
2 8
1 4
3 6

Mergesort



Mergesort


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

Mergesort

- Like the Quicksort the average number of comparisons is $O(n \log n)$.

Mergesort

- Like the Quicksort the average number of comparisons is $O(n \log n)$.
- Unlike the Quicksort, the worst case number of comparisons is also $O(n \log n)$.

Outline

General Assumptions

We begin by stating exactly the sorting problem we are considering.

- computer is a parallel cluster with p nodes.

General Assumptions

We begin by stating exactly the sorting problem we are considering.

- computer is a parallel cluster with p nodes.
- list of length n is stored such that each cluster node has a approximately $\lceil n/p \rceil$ elements of the list.

General Assumptions

We begin by stating exactly the sorting problem we are considering.

- computer is a parallel cluster with p nodes.
- list of length n is stored such that each cluster node has a approximately $\lceil n/p \rceil$ elements of the list.
- sort keys are integers

Outline

Parallel Quicksort Algorithm Assumptions

- Assume number of processes p and total length of the list n are both powers of 2.

Parallel Quicksort Algorithm Assumptions

- Assume number of processes p and total length of the list n are both powers of 2.
- The process list can be partitioned into two halves using the most significant bit of process ID. Processes with a leading 0 in their ID are in the lower half of the process list while those with a leading 1 are in the upper half.

Parallel Quicksort Algorithm Assumptions

- Assume number of processes p and total length of the list n are both powers of 2.
- The process list can be partitioned into two halves using the most significant bit of process ID. Processes with a leading 0 in their ID are in the lower half of the process list while those with a leading 1 are in the upper half.
- *Partner* processes have the same ID except for the leading bit.

Parallel Quicksort Algorithm Assumptions

- Assume number of processes p and total length of the list n are both powers of 2.
- The process list can be partitioned into two halves using the most significant bit of process ID. Processes with a leading 0 in their ID are in the lower half of the process list while those with a leading 1 are in the upper half.
- *Partner* processes have the same ID except for the leading bit.
- For example, suppose there are eight processes. Then we'd have

lower half	upper half
000	100
001	101
010	110
011	111

Parallel Quicksort Algorithm

- One process selects the pivot and broadcasts to all processes.

Parallel Quicksort Algorithm

- One process selects the pivot and broadcasts to all processes.
- Each process in the lower half of the process list sends list elements greater than the pivot to their partners in the upper half of the list.

Parallel Quicksort Algorithm

- One process selects the pivot and broadcasts to all processes.
- Each process in the lower half of the process list sends list elements greater than the pivot to their partners in the upper half of the list.
- Meanwhile, each process in the upper half of the process list sends list elements less than the pivot to their partners in the lower half.

Parallel Quicksort Algorithm

- One process selects the pivot and broadcasts to all processes.
- Each process in the lower half of the process list sends list elements greater than the pivot to their partners in the upper half of the list.
- Meanwhile, each process in the upper half of the process list sends list elements less than the pivot to their partners in the lower half.
- The upper and lower halves of the process list are now treated as separate process lists each with $p/2$ processes. Appropriate upper and lower half lists are identified and the algorithm recurses.

original lower half		*original upper half*	
lower half	upper half	lower half	upper half
000	010	100	110
001	011	101	111

Parallel Quicksort Algorithm

- One process selects the pivot and broadcasts to all processes.
- Each process in the lower half of the process list sends list elements greater than the pivot to their partners in the upper half of the list.
- Meanwhile, each process in the upper half of the process list sends list elements less than the pivot to their partners in the lower half.
- The upper and lower halves of the process list are now treated as separate process lists each with $p/2$ processes. Appropriate upper and lower half lists are identified and the algorithm recurses.

original lower half		*original upper half*	
lower half	upper half	lower half	upper half
000	010	100	110
001	011	101	111

- After $\log_2 p$ recursions each process has its final portion of the entire list; all that remains is to sort it using a sequential sort.

Parallel Quicksort Computation Cost

- During the first phase each process makes n/p comparisons.

Parallel Quicksort Computation Cost

- During the first phase each process makes n/p comparisons.
- In the ideal case each process continues to be responsible for n/p list items and subsequent phases of the sort also require n/p comparisons. Since there are $\log_2 p$ steps, the comparison count will be $(n \log_2 p)/p$.

Parallel Quicksort Computation Cost

- During the first phase each process makes n/p comparisons.
- In the ideal case each process continues to be responsible for n/p list items and subsequent phases of the sort also require n/p comparisons. Since there are $\log_2 p$ steps, the comparison count will be $(n \log_2 p)/p$.
- Unfortunately, the sublist sizes in each process are unlikely to remain uniform. This means that the comparison work will become unbalanced and, in the worst case where one process is responsible for a majority of the list, approach the level for a sequential algorithm.

Parallel Quicksort Computation Cost

- During the first phase each process makes n/p comparisons.
- In the ideal case each process continues to be responsible for n/p list items and subsequent phases of the sort also require n/p comparisons. Since there are $\log_2 p$ steps, the comparison count will be $(n \log_2 p)/p$.
- Unfortunately, the sublist sizes in each process are unlikely to remain uniform. This means that the comparison work will become unbalanced and, in the worst case where one process is responsible for a majority of the list, approach the level for a sequential algorithm.
- Of course, once the list elements have all been collected in the proper processes, they must still be sorted using a sequential sort.

Parallel Quicksort Communication Cost

- As already noted, there are $\log_2 p$ steps required to rearrange the list elements between the processes. Each step consists of broadcasting the pivot and exchanging half the list data (on average).

Parallel Quicksort Communication Cost

- As already noted, there are $\log_2 p$ steps required to rearrange the list elements between the processes. Each step consists of broadcasting the pivot and exchanging half the list data (on average).
- The communication cost for each broadcast is

$$t_{\text{broadcast}} = t_{\text{startup}} + 1 \cdot t_{\text{data}}$$

Parallel Quicksort Communication Cost

- As already noted, there are $\log_2 p$ steps required to rearrange the list elements between the processes. Each step consists of broadcasting the pivot and exchanging half the list data (on average).
- The communication cost for each broadcast is

$$t_{\text{broadcast}} = t_{\text{startup}} + 1 \cdot t_{\text{data}}$$

- The average communication cost for the data exchanges is

$$t_{\text{exchange}} = 2\left(t_{\text{startup}} + \frac{1}{2}\frac{n}{p}t_{\text{data}}\right) = 2t_{\text{startup}} + \frac{n}{p}t_{\text{data}}$$

(This assumes that send/receive operations are not overlapped.)

Parallel Quicksort Communication Cost

- As already noted, there are $\log_2 p$ steps required to rearrange the list elements between the processes. Each step consists of broadcasting the pivot and exchanging half the list data (on average).
- The communication cost for each broadcast is

$$t_{\text{broadcast}} = t_{\text{startup}} + 1 \cdot t_{\text{data}}$$

- The average communication cost for the data exchanges is

$$t_{\text{exchange}} = 2\left(t_{\text{startup}} + \frac{1}{2}\frac{n}{p}t_{\text{data}}\right) = 2t_{\text{startup}} + \frac{n}{p}t_{\text{data}}$$

  (This assumes that send/receive operations are not overlapped.)
- Summing these two costs we find the communication cost per step is

$$t_{\text{comm per step}} = t_{\text{broadcast}} + t_{\text{exchange}} = 3t_{\text{startup}} + \left(1 + \frac{n}{p}\right)t_{\text{data}}.$$

Parallel Quicksort Communication Cost

- Since there are $\log_2 p$ steps, the total communication cost for all steps is
$$t_{\text{total comm}} = \left[3t_{\text{startup}} + \left(1 + \frac{n}{p}\right) t_{\text{data}} \right] \log_2 p.$$

Parallel Quicksort Communication Cost

- Since there are $\log_2 p$ steps, the total communication cost for all steps is
$$t_{\text{total comm}} = \left[3t_{\text{startup}} + \left(1 + \frac{n}{p}\right) t_{\text{data}} \right] \log_2 p.$$
- As already mentioned, however, the average sublist size in each process can vary dramatically as the algorithm proceeds, meaning this analysis may not apply to the typical case.

Outline

Seeking a More Balanced Workload

- The main problem with the parallel Quicksort lies in the choice of the pivot.

Seeking a More Balanced Workload

- The main problem with the parallel Quicksort lies in the choice of the pivot.
- The Sample Sort and its relatives seek to mitigate this problem by choosing $p - 1$ values, called **splitters**, that are used to partition the data into p parts, one for each process.

Seeking a More Balanced Workload

- The main problem with the parallel Quicksort lies in the choice of the pivot.
- The Sample Sort and its relatives seek to mitigate this problem by choosing $p - 1$ values, called **splitters**, that are used to partition the data into p parts, one for each process.
- Basic Algorithm

Seeking a More Balanced Workload

- The main problem with the parallel Quicksort lies in the choice of the pivot.
- The Sample Sort and its relatives seek to mitigate this problem by choosing $p - 1$ values, called **splitters**, that are used to partition the data into p parts, one for each process.
- Basic Algorithm
  1. Each process sorts its original portion of the list.

Seeking a More Balanced Workload

- The main problem with the parallel Quicksort lies in the choice of the pivot.
- The Sample Sort and its relatives seek to mitigate this problem by choosing $p - 1$ values, called **splitters**, that are used to partition the data into p parts, one for each process.
- Basic Algorithm
  1. Each process sorts its original portion of the list.
  2. Splitters are chosen and distributed.

Seeking a More Balanced Workload

- The main problem with the parallel Quicksort lies in the choice of the pivot.
- The Sample Sort and its relatives seek to mitigate this problem by choosing $p-1$ values, called **splitters**, that are used to partition the data into p parts, one for each process.
- Basic Algorithm
  1. Each process sorts its original portion of the list.
  2. Splitters are chosen and distributed.
  3. Each process partitions its sorted data using the splitters and exchanges data with other processes.

Seeking a More Balanced Workload

- The main problem with the parallel Quicksort lies in the choice of the pivot.
- The Sample Sort and its relatives seek to mitigate this problem by choosing $p - 1$ values, called **splitters**, that are used to partition the data into p parts, one for each process.
- Basic Algorithm
  1. Each process sorts its original portion of the list.
  2. Splitters are chosen and distributed.
  3. Each process partitions its sorted data using the splitters and exchanges data with other processes.
  4. Each process sorts its final portion of the list.

Selecting Splitters for the Sample Sort

- The name "Sample Sort" comes from the fact that splitters are determined using a random sample from each process' sublist.

Selecting Splitters for the Sample Sort

- The name "Sample Sort" comes from the fact that splitters are determined using a random sample from each process' sublist.
- Each process collects some number s, often 32 or 64, of samples from its portion of the original list data.

Selecting Splitters for the Sample Sort

- The name "Sample Sort" comes from the fact that splitters are determined using a random sample from each process' sublist.
- Each process collects some number s, often 32 or 64, of samples from its portion of the original list data.
- One process gathers all $s \cdot p$ samples and produces a sorted list.

Selecting Splitters for the Sample Sort

- The name "Sample Sort" comes from the fact that splitters are determined using a random sample from each process' sublist.
- Each process collects some number s, often 32 or 64, of samples from its portion of the original list data.
- One process gathers all $s \cdot p$ samples and produces a sorted list.
- The $p - 1$ splitters are then found in this list at locations $1s, 2s, 3s, \ldots, (p-1)s$.

Selecting Splitters for the Sample Sort

- The name "Sample Sort" comes from the fact that splitters are determined using a random sample from each process' sublist.
- Each process collects some number s, often 32 or 64, of samples from its portion of the original list data.
- One process gathers all $s \cdot p$ samples and produces a sorted list.
- The $p - 1$ splitters are then found in this list at locations $1s, 2s, 3s, \ldots, (p-1)s$.
- The splitters are then broadcast to all processes.

Selecting Splitters for the Sample Sort

- The name "Sample Sort" comes from the fact that splitters are determined using a random sample from each process' sublist.
- Each process collects some number s, often 32 or 64, of samples from its portion of the original list data.
- One process gathers all $s \cdot p$ samples and produces a sorted list.
- The $p - 1$ splitters are then found in this list at locations $1s, 2s, 3s, \ldots, (p-1)s$.
- The splitters are then broadcast to all processes.
- One potential complication is that splitters must be unique; elements in the original list can be "tagged" in some way so that list can be uniquely ordered.

Sample Sort Example

- Original list distributed on two processors ($p = 2$)

12	14	8	16	2	13	4	11

1	6	15	7	10	5	9	3

Sample Sort Example

- Original list distributed on two processors ($p = 2$)

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

- Each process sorts its portion of the list

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

Sample Sort Example

- Original list distributed on two processors ($p = 2$)

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

- Each process sorts its portion of the list

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

- Two random samples ($s = 2$) are chosen in by each process

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

Sample Sort Example

- Original list distributed on two processors ($p = 2$)

$\boxed{12}\ \boxed{14}\ \boxed{8}\ \boxed{16}\ \boxed{2}\ \boxed{13}\ \boxed{4}\ \boxed{11}$ $\quad$ $\boxed{1}\ \boxed{6}\ \boxed{15}\ \boxed{7}\ \boxed{10}\ \boxed{5}\ \boxed{9}\ \boxed{3}$

- Each process sorts its portion of the list

$\boxed{2}\ \boxed{4}\ \boxed{8}\ \boxed{11}\ \boxed{12}\ \boxed{13}\ \boxed{14}\ \boxed{16}$ $\quad$ $\boxed{1}\ \boxed{3}\ \boxed{5}\ \boxed{6}\ \boxed{7}\ \boxed{9}\ \boxed{10}\ \boxed{15}$

- Two random samples ($s = 2$) are chosen in by each process

$\boxed{2}\ \boxed{4}\ \boxed{8}\ \boxed{11}\ \boxed{12}\ \boxed{13}\ \boxed{14}\ \boxed{16}$ $\quad$ $\boxed{1}\ \boxed{3}\ \boxed{5}\ \boxed{6}\ \boxed{7}\ \boxed{9}\ \boxed{10}\ \boxed{15}$

- Samples are collected on single processor

$\boxed{8}\ \boxed{14}\ \boxed{7}\ \boxed{10}$

Sample Sort Example

- Original list distributed on two processors ($p = 2$)

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

- Each process sorts its portion of the list

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

- Two random samples ($s = 2$) are chosen in by each process

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

- Samples are collected on single processor

8	14	7	10

- Samples are then sorted

7	8	10	14

Sample Sort Example (continued)

The sorted samples: $\boxed{\boxed{7}\ \boxed{8}\ \boxed{10}\ \boxed{14}}$

- Since there are only two processes, there is only one splitter, which is located at position $1s$. As $s = 2$, the single element splitter list contains only the element 10.

Sample Sort Example (continued)

The sorted samples: | 7 | 8 | 10 | 14 |

- Since there are only two processes, there is only one splitter, which is located at position $1s$. As $s = 2$, the single element splitter list contains only the element 10.
- The splitter list is broadcast to the processes, who then partition their data.

2	4	8	11	12	13	14	16

1	3	5	6	7	9	10	15

Sample Sort Example (continued)

The sorted samples: | 7 | 8 | 10 | 14 |

- Since there are only two processes, there is only one splitter, which is located at position $1s$. As $s = 2$, the single element splitter list contains only the element 10.
- The splitter list is broadcast to the processes, who then partition their data.

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

- Processes exchange partitioned data

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

Sample Sort Example (continued)

The sorted samples: | 7 | 8 | 10 | 14 |

- Since there are only two processes, there is only one splitter, which is located at position $1s$. As $s = 2$, the single element splitter list contains only the element 10.
- The splitter list is broadcast to the processes, who then partition their data.

2	4	8	11	12	13	14	16

1	3	5	6	7	9	10	15

- Processes exchange partitioned data

2	4	8	1	3	5	6	7	9

11	12	13	14	16	10	15

- Finally, each process sequentially sorts its portion of the list.

1	2	3	4	5	6	7	8	9

10	11	12	13	14	15	16

Discussion of Sample Sort

- Each process must carry out two sequential sorts

Discussion of Sample Sort

- Each process must carry out two sequential sorts
- One process must collect samples sort them, choose splitters, and broadcast the splitters.

Discussion of Sample Sort

- Each process must carry out two sequential sorts
- One process must collect samples sort them, choose splitters, and broadcast the splitters.
- Most literature suggests that the Radix Sort be used for the sequential sorts; Quicksort can also be used.

Discussion of Sample Sort

- Each process must carry out two sequential sorts
- One process must collect samples sort them, choose splitters, and broadcast the splitters.
- Most literature suggests that the Radix Sort be used for the sequential sorts; Quicksort can also be used.
- As our example illustrates, it is often the case that the sorting operation will end with an imbalance in distribution of the sorted list. Usually this will be smaller than with the parallel Quicksort.

Discussion of Sample Sort

- Each process must carry out two sequential sorts
- One process must collect samples sort them, choose splitters, and broadcast the splitters.
- Most literature suggests that the Radix Sort be used for the sequential sorts; Quicksort can also be used.
- As our example illustrates, it is often the case that the sorting operation will end with an imbalance in distribution of the sorted list. Usually this will be smaller than with the parallel Quicksort.
- The number of splitters s can impact the performance of the algorithm; in general smaller s values lead to greater load imbalances. As s increases the sequential work to determine the splitters increases.

Discussion of Sample Sort

- Each process must carry out two sequential sorts
- One process must collect samples sort them, choose splitters, and broadcast the splitters.
- Most literature suggests that the Radix Sort be used for the sequential sorts; Quicksort can also be used.
- As our example illustrates, it is often the case that the sorting operation will end with an imbalance in distribution of the sorted list. Usually this will be smaller than with the parallel Quicksort.
- The number of splitters s can impact the performance of the algorithm; in general smaller s values lead to greater load imbalances. As s increases the sequential work to determine the splitters increases.
- During the data exchange, processes must be prepared to accept an unknown amount of data. To accommodate this we can use an additional global communication (e.g. using `MPI_Allgather()`) to distribute the partition sizes each process is preparing to send.

Outline

Parallel Sorting by Regular Sampling (PSRS)

- Rather than using a random sample to determine the splitters, the PSRS selects specific elements of the locally sorted list as samples.

Parallel Sorting by Regular Sampling (PSRS)

- Rather than using a random sample to determine the splitters, the PSRS selects specific elements of the locally sorted list as samples.
- The phrase “regular sampling” implies that the elements are selected at regular intervals in the local list.

Parallel Sorting by Regular Sampling (PSRS)

- Rather than using a random sample to determine the splitters, the PSRS selects specific elements of the locally sorted list as samples.
- The phrase "regular sampling" implies that the elements are selected at regular intervals in the local list.
- Since each locally sorted list has roughly n/p elements, choosing p regularly spaced samples can be done by selecting samples with indices

$$0, \left(\frac{1}{p} \cdot \frac{n}{p}\right), \left(\frac{2}{p} \cdot \frac{n}{p}\right), \ldots, \left(\frac{p-1}{p} \cdot \frac{n}{p}\right)$$

which reduces to

$$0, \frac{n}{p^2}, \frac{2n}{p^2}, \ldots, \frac{(p-1)n}{p^2}.$$