

## 8 Parallel Sorting

### 8.1 Heapsort?

- Heapsort is  $O(n \log n)$  both worst case and average case
- Algorithm: First make a heap, then sort the heap
- A heap is a data structure in which  $a(n) \leq a(2n)$  and  $a(n) \leq a(2n + 1)$
- Problem: ONE HUGE ARRAY being accessed randomly
- This won't work for distributed memory machines like `daniel`

### 8.2 Parallel Sorting by Random Sampling

- The problem in sorting on distributed memory is the movement of data to do a merge
- We minimize the problem in the average case by using sampling in hopes of moving only as much data as is necessary to move
- Assume that the array of length  $N$  is distributed among  $p$  nodes into partitions of size  $N/p$
- We gamble that the different partitions have the same distribution of numbers to be sorted

### 8.2.1 The Algorithm

1. Distribute the array into partitions of length  $n$
2. Sort locally using the best local method
3. Sample the  $p$  elements subscripted

$$1, \quad 1 + n/p, \quad 1 + 2n/p, \quad \dots, \quad 1 + (p - 1)n/p,$$

4. Collect the local samples on the head node
5. Sort the  $p^2$  samples on the head node
6. Sample the  $p - 1$  elements subscripted

$$p^2/(p - 1), \quad 2p^2/(p - 1), \quad \dots, \quad (p - 1)p^2/(p - 1),$$

7. Broadcast these samples to all nodes
8. Partition the local arrays into  $p$  subpartitions of size roughly  $n/p$  (assuming random distribution of elements)
9. Send the  $0th$  partition to node 0, the first partition to node 1, etc.
10. Sort locally (this is a merge sort)
11. Read off elements from node 0, then node 1, etc.

### 8.2.2 Code `psort.c` for PSRS

### 8.3 Parallel Sorting Algorithms

- Notes follow Quinn, chapter 10
- Also Wilkinson, chapter 9
- Sorting is a fundamentally horrible thing to do on a parallel computer, especially one with distributed memory
- It's also a fundamentally necessary thing to do—vast numbers of problems have a huge sort at their heart
- It's also dominated by data movement—anything that looks like a compare-exchange algorithm has to do two data fetches (expensive data movement) for every comparison (cheap CPU cost)
- Many of the sort algorithms are “architectural” algorithms
- Note that due to the trivial computation involved (one comparison per “operation”), it actually isn't crazy to think about sorts with rather large numbers of processors
- Sorting is known to be  $O(n \log n)$
- So with  $n$  processors,  $O(\log n)$  is the best one could do
- This is known to be possible: Leighton 1994 based on Ajtai, Komlós, and Szemerédi 1983
- But the constants are huge and the algorithms unworkable

### 8.3.1 An Expensive Parallel Sort

- Muller and Preparata, 1975
- Sort  $n$  elements using  $n^2$  processes
- Takes  $\Theta(\log n)$  time to spawn the processes and then  $O(1)$  time to sort
- We assume we're on a CRCW PRAM in which simultaneous writes to a fixed memory location cause the sum to be computed

\*\*\*\*\*

spawn  $n^2$  processes  $P_{\{i,j\}}$  for  $0 \leq i, j < n$

for all  $P_{\{i,j\}}$  where  $0 \leq i, j < n$  do

    position[i] = 0

    if  $a[i] < a[j]$  or  $(a[i] == a[j]$  and  $i < j)$  then

        position[i] = 1

    endif

endfor

for all  $P_{\{i,0\}}$  where  $0 \leq i < n$  do

    sorted[position[i]] = a[i]

endfor

\*\*\*\*\*

- Spawn the processes in a log tree in log time
- Each process compares two elements
- position[.] accumulates the rank of element  $i$  in the list

**8.3.2 A Variation**

- $n$  processors,  $O(n)$  time
- Processor  $i$  just counts the number of elements smaller than  $a_i$
- Then read off the enumerated sequence

### 8.3.3 Odd Even Exchange Sort

- Assume  $n$  data items and  $n$  comparator boxes arranged in a linear array
  1. Odd subscript boxes compare their elements with their neighbors to the right and exchange if out of order
  2. Even subscript boxes compare their elements with their neighbors to the right and exchange if out of order
  3. Go to Step 1
- With  $n/2$  steps the array is sorted

### 8.3.4 Two Dimensional Shearsort

- By Scherson, Sen, Shamir 1986
- Use a two dimensional mesh of processors of size  $\sqrt{n} \times \sqrt{n}$
- 1. Sort each row, with rows 0,2,4,... ascending and 1,3,5,... descending
  2. Sort each column, with all columns ascending
  3. Go to Step 1
- At the end of  $\log n + 1$  pairs of row/col sorts, the data are snakelike sorted in ascending order
- figure 9-12, page 279

### 8.3.5 Systolic Sort

- The above two sorts ignore loading the data into the array (maybe  $n$  steps?)
- We can actually sort the data in the same time we load it, if we load it one item at a time
- Concept of “systolic” algorithm popularized by H. T. Kung in 1980s as algorithms with arrays of processors through which data get “pumped” like blood through the body
- Consider an array of  $n$  comparator boxes
  1. Every box zeros its local storage location
  2. Every box
    - (a) receives one item from its predecessor
    - (b) compares this with its local storage value
    - (c) sends on the smaller to its successor
  3. Repeat step 2
- After  $n$  steps the data are loaded into the array and have been arranged into increasing order

### 8.3.6 Ken Batchter's Bitonic Sort

- Classic 1968 algorithm
- The basic idea has been incorporated into many algorithms and contexts
- Takes  $\Theta(\log^2 n)$  time with  $O(n)$  processors
- **Definition** A *bitonic sequence* is a sequence of values  $a_0, \dots, a_{n-1}$  for which there exists an index  $i$  with  $0 \leq i < n$  such that the elements  $a_0$  through  $a_i$  are monotonically increasing and the elements  $a_i$  through  $a_{n-1}$  are monotonically decreasing, or there exists a cyclic shift of the sequence for which this condition holds.
- Basically a bitonic sequence has at most one peak and one valley
- 

**Lemma 8.1.** *If  $n$  is even, then  $n/2$  comparators are sufficient to transform a bitonic sequence of  $n$  values,*

$$a_0, \dots, a_{n-1}$$

*into two bitonic sequences of  $n/2$  values*

$$\min(a_0, a_{n/2}), \min(a_1, a_{n/2+1}), \dots, \min(a_{n/2-1}, a_{n-1}),$$

$$\max(a_0, a_{n/2}), \max(a_1, a_{n/2+1}), \dots, \max(a_{n/2-1}, a_{n-1}),$$

- Use the  $n/2$  comparators to sort the sequence exactly as in the lemma
- Recursively apply the lemma  $\lg n$  steps to sort the sequence
- page 264, Figure 10-10 of Quinn

### 8.3.7 Bitonic Merge

- Essentially the reverse of the bitonic sort
- Treat an unsorted list of  $n$  items as  $n/2$  bitonic sequences of 2 items
- Recursively merge longer and longer sequences
- **Definition** A *perfect shuffle network* is a network of  $n = 2^k$  nodes numbered 0 through  $n - 1$  with *shuffle* links that connect nodes  $i$  and  $2i \pmod{n - 1}$  and *exchange* links that connect pairs 0, 1, 2, 3, 4, 5, etc., with node  $n - 1$  linked to itself.
- In a perfect shuffle network, a datum at a node with address  $a_{k-1}a_{k-2}\dots a_1a_0$  will wind up at a node with address  $a_{k-2}\dots a_1a_0a_{k-1}$  after one shuffle operation.
- Stone's perfect shuffle implementation of a bitonic merge sort, page 266  
Figure 10-13 of Quinn