

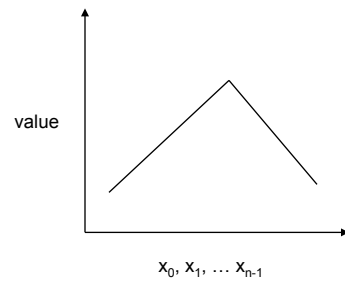
Brief Bitonic sort review

9/29/09

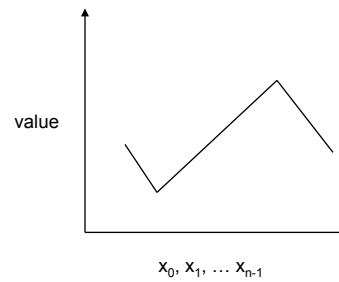
Bitonic sequences

- A monotonic sequence is one such that it is entirely non-increasing or non-decreasing.
- In contrast, a bitonic sequence is one that either:
 - x_1, x_2, \dots, x_{k-1} is non-decreasing and $x_k, x_{k+1}, \dots, x_{n-1}$ is a non-increasing sequence.
 - x_1, x_2, \dots, x_{k-1} is non-increasing and $x_k, x_{k+1}, \dots, x_{n-1}$ is a non-decreasing sequence.
 - There is a cyclic shift of the sequence that satisfies either of these properties.

Examples



Single maximum



Single max and single min

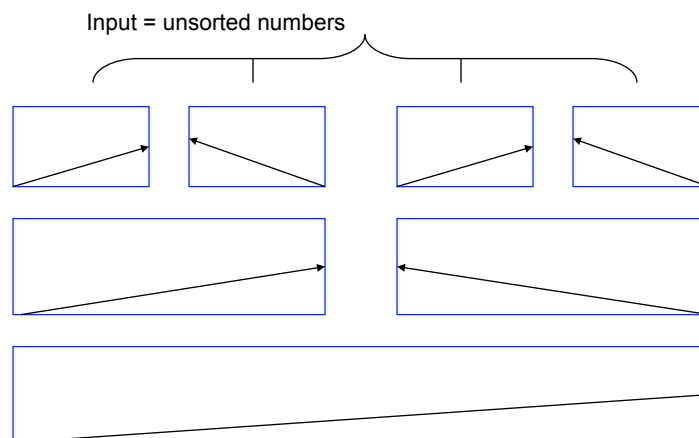
Key observation last lecture

- Note the list from last Tues with I_2 concatenated on I_1 is a bitonic sequence.
- Further, the bitonic split operation that creates I_{min} and I_{max} also produces bitonic sequences.

Bitonic sort

- Suppose the bitonic split operation always creates two new bitonic sequences such that no element in the mins is greater than the maxes.
- If so, we can sort a bitonic sequence by simply performing repeated splits.
- Although this may not always hold recursively, a cyclic shift exists such that this property is true,

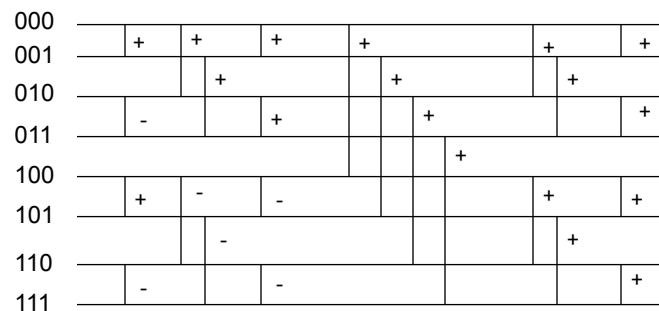
Illustration



Run time

- Using the bitonic sorting algorithm, the time to sort n elements is:
 - $T_s(n) = T_s(n/2) + T_m(n)$
 - $T_s(n) = \sum_{i=1}^{\log n} i \Rightarrow O(\log^2 n)$
- Therefore, our overall speedup would be: $n / \log n$

Bitonic merge circuit for an 8-element sequence



- + leaves the smaller number up top
- leaves the smaller number on bottom

MPI collective communication

9/29/09

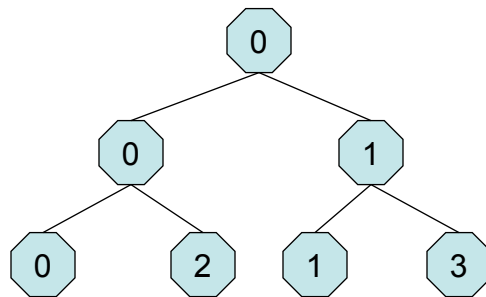
Tree-structured communication

- Suppose we would like to broadcast specific data from one processor to the remaining $p - 1$ processors.
- As mentioned in a previous class, this can be done using a tree.

Basic framework

- Given that we have p processors, our tree structure will have p leaves.
- Each processor will be involved in at most $\log(p) + 1$ stages and send to at most one other processor per step.

Example



Advantages

- By structuring our communication as illustrated (left child is the same proc), there is only one communication per step.
- For example if $p = 8$:
 - Stage 1: 0 sends to 1
 - Stage 2: 0 sends to 2; 1 sends to 3
 - Stage 3: 0 sends to 4; 1 sends to 5; 2 sends to 6; 3 sends to 7

Generalized formula

- If $2^{\text{stage}} \leq \text{my_rank} \leq 2^{\text{stage}+1}$
 - Receive from $\text{my_rank} - 2^{\text{stage}}$
- Otherwise if $\text{my_rank} < 2^{\text{stage}}$
 - Send to $\text{my_rank} + 2^{\text{stage}}$

Our code vs. MPI

- Based on the programming assignment, we may know how to use MPI_Send and MPI_Recv.
- Suppose we implement our communication strategy using a loop and the preceding observations. Is this better than MPI?

Pacheco's implementation

- Pp 67-71; v2 is the MPI-based solution

P	<i>nCUBE2</i>		<i>Paragon</i>		<i>SP2</i>	
	<i>v1</i>	<i>v2</i>	<i>v1</i>	<i>v2</i>	<i>v1</i>	<i>v2</i>
2	0.59	0.69	0.21	0.43	0.15	0.16
8	4.7	1.9	0.84	0.93	0.55	0.35
32	19.0	3.0	3.2	1.3	2.0	0.57

Reduction

- Collective communication called reductions are the same as the ones we discussed related to parallel prefix.
- These are also called “*scan*” in parallel terminology.

Usage

- `int MPI_Reduce (`
 - `void* operand,`
 - `void* result, // stored on root processor`
 - `int count,`
 - `MPI_datatype datatype,`
 - `MPI_Op operator,`
 - `int root,`
 - `MPI_Comm comm)`

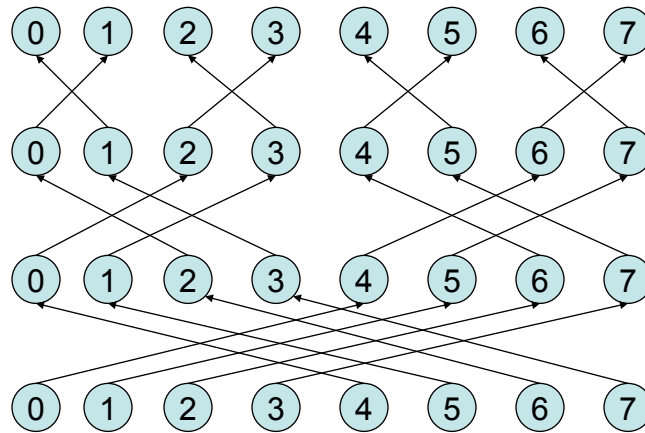
Predefined operators

Operator	Function
MPI_MAX	Maximum
MPI_SUM	Sum
MPI_LAND	Logical AND
MPI_PROD	Product
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

Allreduce

- In class, we discussed performing parallel prefix using hypercubic permutations.
- Today, we discussed tree-based communications for broadcast.
- How could we perform a tree-structured reduce rooted at all of the processors simultaneously?

Butterfly



Discussion

- Suppose we add another line connecting a processor to itself in adjacent rows in the preceding slide.
- The result is a tree rooted at each process as before!

Usage

- int MPI_AllReduce (
 - void* operand,
 - void* result, // stored on all processors
 - int count,
 - MPI_datatype datatype,
 - MPI_Op operator,
 - MPI_Comm comm)

Tags and collective communication

- Note that none of the collective communications involve tags, as the communication in your assignment may have.
- This is because they all should be *synchronous* operations, or one that can not complete until all others have started.
- The implication is all processors must execute the same operation in the same order.

Common bugs in parallel code

- Trying to receive before sending, or trying to receive when there is no send.
 - Called “deadlock”
- Incorrect parameters to send/receive
- Serial bugs

Helpful hints

- Ensure communications are behaving using IO messages
 - Use fflush to clear the buffer as needed
- Start small where possible.
 - Serial before parallel
 - 2 processors before 16

Sample sort

9/29/09

Bucket Sort

- In a bucket sort, we divide the range of the input numbers into equal sized intervals called buckets.
- If the numbers are uniformly distributed, each bucket can be expected to have roughly identical number of elements placed in them.
- Elements in the buckets are locally sorted.

In class exercise

- In your small groups, discuss ways of partitioning elements for a bucket sort.
- HINT: One way would be to take a sample of some kind. How many numbers do you think would work?

Parallel Bucket Sort

- Parallelizing bucket sort is relatively simple. We can select p intervals.
- A given processor is responsible for a range of values and each individual processor runs through its local list to assign each of its elements to the corresponding processor.
- The elements are sent to the destination processors using a single all-to-all personalized communication.
- Each then processor sorts all of its elements.

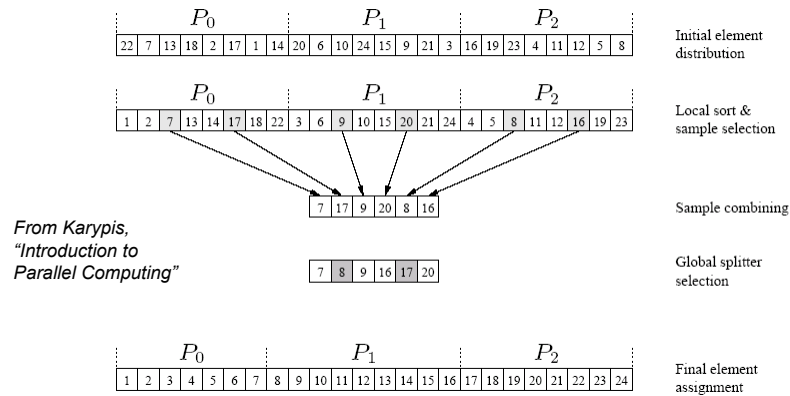
Picking splitters

- The most important aspect of a bucket sort is assigning appropriate numeric ranges to processors.
- Ideally, the n elements should be divided into p blocks of size $O(n/p)$ and sorted using our favorite serial algorithm (e.g., quicksort).
- To do so, we choose $p - 1$ evenly spaced elements per processor.

Splitters

- The $p(p - 1)$ elements selected from all the blocks represent the sample used to determine the buckets.
- This scheme guarantees that the number of elements ending up in each bucket is less than $2n/p$ (proof outside scope of lecture).

Parallel Sample Sort



Sample sort performed on an array with 24 elements using three processors.

Sample Sort

- The overall election scheme can run concurrently.
- Each processor generates the $p-1$ local splitters in parallel.
- All processors share their splitters using a single all-to-all broadcast operation.
- Each processor sorts the $p(p-1)$ elements it receives and selects $p-1$ uniformly spaced splitters from them.

Analysis

- Sorting n/p elements requires time $\Theta((n/p)\log(n/p))$
- The selection of the $p-1$ sample requires time $\Theta(p)$ and the time for an all-to-all broadcast is $\Theta(p^2)$
- The time required to sort the $p(p-1)$ sample elements is $\Theta(p^2 \log p)$
- Each process can *insert* these $p-1$ splitters in its local sorted block of size n/p by performing $p-1$ binary searches in time $\Theta(p \log(n/p))$.

Parallel Sample Sort

- The total time is given by:

$$T_P = \overbrace{\Theta\left(\frac{n}{p} \log \frac{n}{p}\right)}^{\text{local sort}} + \overbrace{\Theta(p^2 \log p)}^{\text{sort sample}} + \overbrace{\Theta\left(p \log \frac{n}{p}\right)}^{\text{block partition}} + \overbrace{\Theta(n/p)}^{\text{communication}} \quad (5)$$