Lecture 16: Sorting

CS178: Programming Parallel and Distributed Systems

April 2, 2001
Steven P. Reiss

I. Overview

A. Before break we started talking about parallel computing and MPI

1. Basic idea of a network of machines
2. There is some sort of network topology
3. Machines cooperate by passing messages\

B. MPI

1. Large and powerful library to support messages
2. Support several types of sends
3. Supports concrete and virtual topologies

C. This week and next I want to cover

1. Basic techniques for parallel programming
2. Basic parallel programming algorithms

D. You're used to a sequential world

1. You've learned lots of sequential algorithms and data structures
2. You've learned lots of control flow and coding patterns for sequential code
3. Many of these don't work in a parallel world (or they don't work as well)
4. You have to think differently

II. Sorting

A. Motivation

1. Sorting is a problem you should all be familiar with
2. Many large problems develop (or accumulate or read) data at each node

a) But the data should be processed homogeneously

b) This means sorting it within the network

3. A distributed vector is sorted on a set of processors P1 to Pn iff

a) Within each processor a[i] <= a[j], i <= j

b) For i< j, all of A in Pi <= all of A in Pj

4. Sorting (and similar algorithms) arise as subproblems of many scientific computations

B. What algorithms might be appropriate?

1. The best sequential algorithms are often not the most appropriate

a) In many cases the simpler algorithms are easier to parallelize than the more complex ones

2. Actually this question isn't fair

a) One needs to understand the network to understand the algorithm

b) This is true for parallel algorithms in general

(1) What's good for a linear array is different than whats good for a hypercube

C. How do you think about parallel sorting

1. First, sorting the numbers within a node is simple

a) Use quicksort or any other algorithm

b) Don't have to worry about parallelization

2. So what we are left with is a vector in each node

a) Need to merge those vectors

b) Can think of this as a sorting problem on vectors

3. What are the basic sorting operations

a) Compare & Exchange

(1) Given A and B, compare A to B and exchange them if A < B (or A > B)

(2) This is used extensively in most sorting algorithms

b) Can you do this on arrays between processors?

4. Then you can think of sorting using arrays as elements

III. Compare and Exchange

A. Basic idea

1. Processors i and j have arrays of size N
2. Each passes their whole array to the other
3. Each sorts the array and keeps their portion, discarding the rest

B. Issues that arise

1. They better agree on computing A < B
2. If the arrays are partially sorted, then complete sorting might not be necessary
3. Different sort algorithms might be appropriate given you know you only want the upper/lower half

C. Basic code

```
void mergeSplit(WhichPart which,int pid)
{
   MPI_Status status;
   MPI_Sendrecv(sort_base,data_size,SORT_DATA_TYPE,pid,0,
                temp_base,data_size,SORT_DATA_TYPE,pid,0,
                mpi_comm,&status);
   if (which == HIGH) mergeDataHigh();
   les mergeDataLow();
}
```

1. Note the use of MPI_Sendrecv
2. What might the merge look like
   a) Suppose we know both are sorted

IV. Quicksort

A. Recall how quicksort works

1. How can this be parallelized
2. Assume # processors is a power of two

B. First broadcast a pivot to all processors

1. Then pair up the processors (one high, one low)
2. These do a compare/exchange based on the pivot
3. Result is values < in first half, > in second half

C. This then is repeated recursively on smaller sets

1. This assumes a hypercube organization

D. Notes

1. **Faster if you sort the nodes at each processor locally before doing the split**
2. **Choice of pivot is crucial**
   a) What happens if nodes are unbalanced

V. Mergesort

A. Normal merge sort doesn't scale

1. **Problem is that we only want to compare pairs of processors**
2. **Would have a sequential bottleneck at top level**

B. Sort should only compare pairs

1. **Notion of a sorting network**
2. **Rows for each processor**
3. **Connections indicate which compares are done**
4. **Multiple compares can be done at once if independent**

C. We can still do this:

1. **Suppose we have sequencess of size n that are sorted**
   a) Let this be A[1] ... A[n], B[1] ... B[n]
   b) Then we want to construct C[1..2n]
2. **Recursively merge the odd indices and even ones**
   a) D[1...n] = MERGE(A[1,3,5,...],B[1,3,5,...])
   b) E[1...n] = MERGE(A[2,4,6,...],B[2,4,6,...])
3. **Pairwise compare-exchange the result**
   a) C[1] = D[1]
   b) C[2*i] = MIN(D[i+1,E[i])
   c) C[2i+1] = MIN(D[i+1,E[i])
   d) C[2n] = E[n]

D. Show this on a network of size 8

VI. Bitonic mergesort

A. Bitonic merging

1. **A bitonic sequence is increasing, then decreasing**

a) Or a rotation of that (i.e. can start anywhere)

b) Can be constructed by concatenating two sorted lists

2. Why is this useful

a) Any pair of numbers is a bitonic sequence

b) What happens if you compare-exchange A[i] w A[i+n/2] for all i

c) We get 2 bitonic sequences, but the numbers in one are all larger than the numbers in the other

3. Merging

a) Given a bitonic sequence, do this compare-exchange on n/2, all n/4, all n/8, ...

b) The result is an ordered sequence

c) Show this via a network

B. Bitonic sorting

1. We can start with unordered and form larger bitonic sequences

a) Suppose we have A sorted up, B sorted down

b) Then A-B is a bitonic sequence

c) We can merge A-B to form a sorted AB

2. Procedure

a) Sort the elements in each processor

(1) Odd increasing, even descreasing

b) Merge pairs of processors to form a larger sequence

(1) Again alternate decreasing and increasing

c) Repeat until you only have one sequence

3. Show this via a network

4. Notes

a) The merge only requires pairwise compare-exchange operations

b) We can use binary representation of the processor ID to determine increasing vs decreasing at each step

c) We can use binary representation of the processor ID to determine who to swap with

C. Implementation