Lecture 17: Array Algorithms

CS178: Programming Parallel and Distributed Systems

April 4, 2001

Steven P. Reiss

I. Overview

A. We talking about constructing parallel programs

1. Last time we discussed sorting algorithms

2. Looking at various techniques

B. The emphasis here shouldn't be on sorting per se

1. But rather on the underlying techniques

2. Divide and conquer (split the problem)

3. Finding algorithms that allow maximal parallelism

a) This is what bitonic merge does for us

4. Treating blocks as units

C. Today I want to look at additional techniques

1. Numerical (array) problems

2. Many of the same techniques in a new context

3. New techniques -- pipelining

II. Matrix Computations

A. Motivation

1. Many scientific problems involve matrices

a) We've seen one example with the heat distribution example we did before break

(1) Value at one point is related to the value at other points

(2) This generalizes to multiple values

b) The homework has another example

(1) Maxtrix to store responses

(2) Need to compute over that

c) Other problems can be cast as matrix problems

(1) Web searching

(2) Image processing

2. Many of these problems can be large

a) Matrices of $10^5 * 10^5 * 10^5$ for weather, flow, ...

b) Number of web pages

c) Number of users

3. If we want to do these practically, we need to do them in parallel

B. What are the basic problems

1. Matrix multiplication

a) Occurs in many problems

b) Part of many other problems (transitive closure)

c) Same algorithm as many other problems

d) Matrix-vector multiplication is a common subset

(1) Lots of vectors done at once

e) Recall the standard sequential algorithm

2. Gaussian elimination

a) Solving systems of linear equations

b) We've seen instances of this already

(1) Heat distribution problem with sparse matrices

(2) This was solved using iterative methods

(3) These don't work as well for full matrices

c) Related to matrix inversion

C. How is the array stored

1. This is going to depend on the problem

2. But also is going to control the algorithm

3. Alternatives

a) All in one node

b) All in all nodes

c) On disk (to be read by one or more nodes)

d) Spread out over all the nodes (e.g. heat flow problem)

III. Matrix Multiplication

A. Obvious ways of parallelizing

1. **The inner loop can be done in parallel**
2. **Actually any of the loops can be**
   a) This is a standard technique
3. **But this means that each node needs the whole array**
4. **This is what is done with parallel fortran, etc.**
5. **This corresponds to row or column orientation**
   a) We'll get back to this

B. Can break the matrix up into blocks

1. **Note that this works mathematically**
   a) Block sizes need to be compatible x*y and y*z
2. **This is again a standard technique**
   a) We used it with sorting
3. **But even so, how do we multiply**

C. Recursive implementation

1. **Break the matrix in 4 blocks**
2. **If block is one element, then just multiply**
3. **Else**
   a) Compute the 8 cross products recursively
   b) Add the submatrices to get the results
4. **8 recursive calls can be done in parallel**
   a) Can be repeated until you run out of processors
   b) Seems to imply that the whole matrix is available or storable
5. **Message based approaches shouldn't assume this**

D. Cannon's algorithm

1. **Assume a wraparound mesh topology**
   a) Assume there are P^2 processors $P_{i,j}$ holding submatrices (or elements)
   b) Initially $P_{i,j}$ holds $a_{i,j}$ and $b_{i,j}$

c) We then shift rows and columns around the matrix to do the multiplication

2. Algorithm

a) Initialize: move items to “aligned” position

```
Move Row i of A i places left
Move Column j of B j places upward
```

(1) Then $P_{i,j}$ contains $a_{i,j+i}$ and $b_{i+j,.j}$

(2) This is a part of the sum

b) Do the initial multiplication

```
c = a*b for each processor
```

(1) c, a, b represent the elements (matrices) held by that processor at that time

c) Shift row i of A one place left; shift row j of B one place up

(1) This gives the next component of the sum

d) Accumulate the new result

```
c += a*b
```

e) Repeat c) and d) n-1 times to get the final result

E. Fox’s algorithm

1. As an alternative to moving whole rows and columns, we can broadcast elements

2. Code for Processor $P_{i,j}$

```
dest = [i-1 mod n, j]
src = [i+1 mod n, j]
for (stage = 0; stage < n; ++stage) {
   kbar = (i+stage) mod n
   Broadcast A[i,kbar] across process row i
   C[i,j] += A[i,kbar]*B[kbar,j]
   Send B[kbar,j] to dest
   Receive B[kbar+1 mod n,j] from source
 }
```

F. Pipelined processing

1. Another way of doing the computation is to pipeline the processing

a) This is another example of a general technique

b) Program has send - compute cycles

c) All processor operate in sync on those cycles

2. Basic idea:

a) Send $a_{0,0}$, $a_{0,1}$, $a_{0,2}$, ... to first row, one per cycle

b) Send $b_{0,0}$, $b_{0,1}$, $b_{0,2}$, ... to first column, one per cycle

c) Send 0, $a_{1,0}$, $a_{1,1}$, $a_{1,2}$, to second row; etc

d) At each step, processor computes product of its inputs and accumulates

e) After 2N steps, everything is done

3. Note how this works

4. Note this can be applied to matrix-vector processing as well

IV. Systems of Linear Equations

A. Gaussian Elimination

1. Recall the sequential algorithm

a) Set the diagonal to ones

b) Set everything below the diagonal to zeros

c) At each stage:

(1) Compute pivot (why and how)

(2) Compute multiplier m for each row $A_{j,i}$ / $A_{i,i}$

(3) Subtract row i * m from row j

(4) Do the same for B_i

d) Back substitute at the end

2. How might you parallelize this (what techniques)

3. Partitioning

a) Blocks don't work

b) Want to partition into rows (or sets of rows)

B. Pipelining

1. First row is broadcast to all other processors

a) Each computes multiplier and then updates its row

2. Then second row is broadcast to remaining processors

a) Etc.

b) Note that this can be pipelined

(1) Processor gets data

(2) Processor resends data; computes; sends result

C. Partitioning

1. This assumed one processor per row

2. What happens if we have fewer (more typical)

3. Blocks of rows (strips)

a) Here processors become idle

4. Cyclic partitioning

a) Assign rows sequentially to processors

D. What about pivoting

1. This can be done by finding the pivot row

a) Finding max element over processors, return index

b) Then swapping the two rows

2. It can also be done by maintaining an index array

a) This effectively swaps the rows -- index tells row number

b) But this information needs to be shared

c) This makes back substitution more difficult

3. Can imagine other approaches that pass rows thru a mesh

a) Pass original row down to proper position

b) Pass pivot row up and down to proper positions

c) Compute as you get it

V. Gettting the data to the nodes

A. We assume that the nodes hold portions of the array

1. How do these portions get there in the first place

a) Could be computed there

b) Often, however, data must be read from a file or computed centrally

2. File I/O

a) Don't want all nodes reading from same file (why?)

b) Separate files means knowing configuration in advance

3. Assume one node reads file (or has data initially)

a) How to send it out

B. MPI Has scatter/gather facilities for this

```
MPI_Scatter(void * sendptr,int sendcnt,MPI_Datatype sendtype,
         void * recvptr,int recvcnt,MPI_Datatype recvtype,
         int root,MPI_Comm comm)
MPI_Gather(void * sendptr,int sendcnt,MPI_Datatype sendtype,
         void * recvptr,int recvcnt,MPI_Datatype recvtype,
         int root,MPI_Comm comm)
```

1. Gather

a) Each process sends info from its send area to the root

b) Root receives the data and stores it in rank order

c) Note that root receive area needs to be big enough

2. Scatter does the opposite

a) Sends from root to all processors

3. This can be used for reading and distributing array

4. Suppose we want all nodes to end up with the array

```
MPI_Allgather(...) [no root argument]
```

C. MPI also has facilities for broadcasting

```
MPI_Bcast(void * message,int count,MPI_Datatype type,
            int root,MPI_Comm comm)
```

1. This is a send-recv type call

2. Root is doing the send, all others are receive

3. At the end, all will have the message in their buffer