Lecture 14

Collective Communication
Announcements

• Project Progress report, due next Weds 11/28
Today’s lecture

- Collective Communication algorithms
- Sorting
Collective communication

• Collective operations are called by all processes within a communicator

• Basic collectives seen so far
  ◆ Broadcast: distribute data from a designated root process to all the others
  ◆ Reduce: combine data from all processes returning the result to the root process
  ◆ Will revisit these

• Other Useful collectives
  ◆ Scatter/gather
  ◆ All to all
  ◆ Allgather

• Diverse applications
  ◆ Fast Fourier Transform
  ◆ Sorting
Underlying assumptions

- Fast interconnect structure
  - All nodes are equidistant
  - Single-ported, bidirectional links

- Communication time is $\alpha + \beta n$ in the absence of contention
  - Determined by bandwidth $\beta^{-1}$ for long messages
  - Dominated by latency $\alpha$ for short messages
• Tree like algorithm to broadcast the message to blocks of processes, and a linear algorithm to broadcast the message within each block
• Block size may be configured at installation time
• If there is hardware support (e.g. Blue Gene), then it is given responsibility to carry out the broadcast
• Polyalgorithms apply different algorithms to different cases, i.e. long vs. short messages, different machine configurations
• We’ll use hypercube algorithms to simplify the special cases when $P=2^k$, $k$ an integer
Details of the algorithms

• Broadcast
• AllReduce
• Scatter/gather
• Allgather
• All to all
Broadcast

• The root process transmits of $m$ pieces of data to all the $p-1$ other processors
• Spanning tree algorithms are often used
• We’ll look at a similar algorithm with logarithmic running time: the hypercube algorithm
• With the linear ring algorithm this processor performs $p-1$ sends of length $m$
  ♦ Cost is $(p-1)(\alpha + \beta m)$
Sidebar: what is a hypercube?

• A hypercube is a d-dimensional graph with $2^d$ nodes
• A 0-cube is a single node, 1-cube is a line connecting two points, 2-cube is a square, etc
• Each node has d neighbors
Properties of hypercubes

- A hypercube with \( p \) nodes has \( \lg(p) \) dimensions
- *Inductive construction*: we may construct a \( d \)-cube from two \((d-1)\) dimensional cubes
- **Diameter**: What is the maximum distance between any 2 nodes?
- **Bisection bandwidth**: How many cut edges (mincut)
Bookkeeping

• Label nodes with a binary reflected grey code. See: http://www.nist.gov/dads/HTML/graycode.html

• Neighboring labels differ in exactly one bit position. $001 = 101 \oplus e_2$, $e_2 = 100$
Hypercube broadcast algorithm with $p=4$

- Processor 0 is the root, sends its data to its hypercube “buddy” on processor 2 (10)
- Proc 0 & 2 send data to respective buddies
Details of the algorithms

• Broadcast
• AllReduce
• Scatter/gather
• Allgather
• All to all
Reduction

• We may use the hypercube algorithm to perform reductions as well as broadcasts

• Another variant of reduction provides all processes with a copy of the reduced result

  \texttt{Allreduce( )}

• Equivalent to a \texttt{Reduce + Bcast}

• A clever algorithm performs an \texttt{Allreduce} in one phase rather than having perform separate reduce and broadcast phases
Allreduce

- Can take advantage of duplex connections
Details of the algorithms

- Broadcast
- AllReduce
- Scatter/gather
- Allgather
- All to all
Scatter/Gather

$P_0$  $P_1$  $P_{p-1}$

Gather

Scatter

Root

©2012 Scott B. Baden / CSE 260/ Fall 2012
Scatter

• Simple linear algorithm
  ◆ Root processor sends a chunk of data to all others
  ◆ Reasonable for long messages

\[(p - 1)\alpha + \frac{p - 1}{p} n\beta\]

• Similar approach taken for Reduce and Gather
• For short messages, we need to reduce the complexity of the latency (\(\alpha\)) term
Minimum spanning tree algorithm

- Recursive hypercube-like algorithm with \([ \log P \] steps
  - Root sends half its data to process \((\text{root} + p/2) \mod p\)
  - Each receiver acts as a root for corresponding half of the processes
  - MST: organize communication along edges of a minimum-spanning tree covering the nodes
- Requires \(O(n/2)\) temp buffer space on intermediate nodes
- Running time:
  \[
  [\lg P] \alpha + \frac{p - 1}{p} n \beta
  \]
Details of the algorithms

- Broadcast
- AllReduce
- Scatter/gather
- Allgather
- All to all
AllGather

- Equivalent to a gather followed by a broadcast
- All processors accumulate a chunk of data from all the others
AllGather
Allgather

- Use the all to all recursive doubling algorithm
- For $P$ a power of two, running time is

$$\lfloor \lg P \rfloor \alpha + \frac{p-1}{p} n \beta$$
Details of the algorithms

• Broadcast
• AllReduce
• Scatter/gather
• Allgather
• All to all
All to all

• Also called *total exchange* or *personalized communication*: a transpose
• Each process sends a different chunk of data to each of the other processes
• Used in sorting and the Fast Fourier Transform
Exchange algorithm

- \( n \) elements / processor (\( n \) total elements)
- \( p - 1 \) step algorithm
  - Each processor exchanges \( n/p \) elements with each of the others
  - In step \( i \), process \( k \) exchanges with processes \( k \pm i \)

```plaintext
for i = 1 to p-1
    src = (rank - i + p) mod p
    dest = (rank + i) mod p
    sendrecv( from src to dest )
end for
```

- Good algorithm for long messages
- Running time:

\[
(p - 1)\alpha + (p - 1)\frac{n}{p} \beta \approx n\beta
\]
Recursive doubling for short messages

- In each of \([\log p]\) phases all nodes exchange \(\frac{1}{2}\) their accumulated data with the others
- Only \(P/2\) messages are sent at any one time

\[
D = 1
\]

\[
\text{while } (D < p)
\]

- Exchange & accumulate data with rank \(\otimes D\)
- Left shift D by 1

\[
\text{end while}
\]

- Optimal running time for short messages

\[
[\lg P]\alpha + nP\beta \approx [\lg P]\alpha
\]
Flow of information
Flow of information

©2012 Scott B. Baden /CSE 260/ Fall 2012
Flow of information
Summarizing all to all

- Short messages: $[\lg P] \alpha$
- Long messages: $\frac{P-1}{P^n} \beta$
“Vector” All to AL1

- Generalize all-to-all, gather, etc.
- Processes supply varying length datum
- Vector all-to-all
  
  ```c
  MPI_Alltoallv ( 
    void *sendbuf, int sendcounts[], int sDispl [],
    MPI_Datatype sendtype,
    void* recvbuf, int recvcounts[], int rDispl[],
    MPI_Datatype recvtype, MPI_Comm comm )
  ```
- Used in sample sort (coming)
Alltoally used in sample sort

Initial element distribution

Local sort & sample selection

Sample combining

Global splitter selection

Final element assignment

Details of the algorithms

- Broadcast
- AllReduce
- Scatter/gather
- Allgather
- All to all
- Revisiting Broadcast
Revisiting Broadcast

- P may not be a power of 2
- We use a binomial tree algorithm
- We’ll use the hypercube algorithm to illustrate the special case of $P=2^k$
- Hypercube algorithm is efficient for short messages
- We use a different algorithm for long messages
Strategy for long messages

• Based van de Geijn’s strategy
• Scatter the data
  ◆ Divide the data to be broadcast into pieces, and fill the machine with the pieces
• Do an Allgather
  ◆ Now that everyone has a part of the entire result, collect on all processors
• Faster than MST algorithm for long messages

$$2^{\frac{p-1}{p}} n\beta \ll [\log p] n\beta$$
Algorithm for long messages

The scatter step

P₀  P₁  Pₚ₋₁  Root

Scatter
Algorithm for long messages

P₀ P₁ ... P_{p-1}

AllGather step
Today’s lecture

• Collective Communication algorithms
• Sorting
Rank sorting

• Compute the rank of each input value
• Move each value in sorted position according to its rank
• On an ideal parallel computer, the forall loops parallelize perfectly

\[
\text{forall } i=0:n-1, j=0:n-1 \\
\text{ if } ( x[i] > x[j] ) \text{ then } \text{rank}[i] += 1 \text{ end if} \\
\text{forall } i=0:n-1 \\
\text{ y[rank[i]] = x[i]} 
\]
In search of a fast and practical sort

• Rank sorting is impractical on real hardware
• Let’s borrow the concept: compute the processor owner for each key
• Communicate data in sorted order in one step
• But how do we know which processor is the owner?
• Depends on the distribution of keys
Bucket sort

- Divide key space into equal subranges and associate a bucket with each subrange
- Unsorted input data distributed evenly over processors
- Each processor maintains $p$ local buckets
  - Assigns each key to a local bucket: $\left\lfloor \frac{p \times \text{key}}{K_{\text{max}} - 1} \right\rfloor$
  - Routes the buckets to the correct owner (each local bucket has $\sim \frac{n}{p^2}$ elements)
  - Sorts all incoming data into a single bucket

![Diagram showing bucket sort process](Wikipedia)
Running time

• Assume that the keys are distributed uniformly over 0 to $K_{\text{max}} - 1$
• Local bucket assignment: $O(n/p)$
• Route each local bucket to the correct owner
  All to all: $O(n)$
• Local sorting: $O(n/p)$
  ♦ Radix sort
Scaling study

- IBM SP3 system: 16-way servers w/ Power 3 CPUs
- Weak scaling: 1M points per processor

Local sort: quicksort \( O(n/p \log(n/p)) \)

All-to-allv \( O(n) \)
Worst case behavior

- What is the worst case?
- Mapping of keys to processors based on knowledge of $K_{max}$
- If keys are in range $[0, Q-1]$ …
  … processor $k$ has keys in the range $[k*Q/P : (k+1)*Q/P]$
- For $Q=2^{30}$, $P=64$, each processor gets $2^{24} = 16$ M elements
- What if keys $\in [0, 2^{24} -1] \subset [0, 2^{30} -1]$?
- But if they keys are distributed non-uniformly, we need more information to ensure that the keys (and communication) are balanced over the processors
- Sample sort is an algorithm that collects such information and improves worst case behavior
Improving on bucket sort

• *Sample sort* remedies the problem
The idea behind sample sort

- Use a heuristic to estimate the distribution of the global key range over the $p$ processors processor so that...
- …each processor gets about the same number of keys
- Sample the keys to determine a set of $p-1$ splitters that partition the key space into $p$ disjoint intervals [sample size parameter: $s$]
- Each interval is assigned a unique processor mapped to a bucket
- Once each processor knows the splitters, it can distribute its keys to the others accordingly
- Processors sort incoming keys
Alltoally used in sample sort

Splitter selection: regular sampling

- After sorting local keys, each processor chooses \( p \) evenly spaced samples
- Each processor “deals” its sorted data into one of \( p \) bins
  - The \( k^{th} \) item is placed into position \([k/p]\) of bin \( k \mod p\)
  - When done, each sends bin \( j \) to processor \( j \)
- This is like a transpose with block sizes = \( n/p^2 \)
- Each processor receives \( p \) sorted subsequences
- Processor \( p-1 \) determines the splitters
  - It samples each sorted subsequence, taking every \((kn/(p^2s))^{th}\) element \((1 \leq k \leq s-1)\), where \( p \leq s \leq n/p^2 \)
  - Merges the sampled sequences, and collects \( p-1 \) regularly spaced splitters
  - Broadcasts the splitters to all processors
- Processors route (exchange) sorted subsequences according to the splitters (transpose)
- The data are unshuffled
Performance

- Assuming $n \geq p^3$ and $p \leq s \leq n/p^2$
- Running time is $\approx O((n/p) \log n)$
- With high probability … no processor holds more than $(n/p + n/s - p)$ elements
- Duplicates $d$ do not impact performance unless $d = O(n/p)$
- Tradeoff: increasing $s$ …
  - Spreads the final distribution more evenly over the processors
  - Increases the cost of determining the splitters
- For some inputs, communication patterns can be highly irregular with some pairs of processors communicating more heavily than others
- This imbalance degrades communication performance

©2012 Scott B. Baden /CSE 260/ Fall 2012
The collective calls

• Processes transmit varying amounts of information to the other processes
• This is an MPI_Alltoallv
  \(( SKeys, \text{send\_counts, send\_displace}, MPI\_INT, RKeys, \text{recv\_counts, recv\_displace}, MPI\_INT, MPI\_COMM\_WORLD ) )\)
• Prior to making this call, all processes must cooperate to determine how much information they will exchange
  ◆ The send list describes the number of keys to send to each process \( k \), and the offset in the local array
  ◆ The receive list describes the number of incoming keys for each process \( k \) and the offset into the local array
Determining the send and receive lists

- After sorting, each process scans its local keys from left to right, marking where the splitters divide the keys, in terms of send counts
- Perform an all to all to transpose these send counts into receive counts

\[
\text{MPI\_Alltoall}(\text{send\_counts, 1, MPI\_INT, recv\_counts, 1, MPI\_INT, MPI\_COMM\_WORLD})
\]

- A simple loop determines the displacements

\[
\begin{align*}
\text{for } (p = 1; p < \text{nodes}; p++)\{ \\
\text{s\_displ}[p] &= \text{s\_displ}[p-1] + \text{send\_counts}[p-1]; \\
\text{r\_displ}[p] &= \text{r\_displ}[p-1] + \text{rend\_counts}[p-1]; \\
\}\end{align*}
\]
Fin