Lecture 7

More collective operations
Advanced Sorting

Announcements

• Assignment #2 due in class on Thursday
• Today’s readings include a handout on Bitonic sort
  www.cse.ucsd.edu/classes/sp03/cse160/Lectures/Lec07/bitonic.pdf
More on collective communication

• We’ve looked at two collective communication operations so far
  – Reduce
  – Broadcast
• We’ll look at total exchange next
• More generally, see
  www.netlib.org/utk/papers/mpi-book/node92.html#SECTION00510000000000000000
  www.tacc.utexas.edu/resources/user_guides/mpi.c/collective.php#gather_scatter

Total exchange

• MPI_Alltoall()
• Also called personalized communication
• Each processor sends a different chunk of data to each of the other processors
Transposition

- Consider a computation which has two separate phases that operate on a 2D array
- In one phase, data are distributed row-wise
- In the other, column-wise
- Transpose the data between phases

![Diagram showing data distribution and transposition]

How do we implement this?

- Simplest algorithm:
- Using point-to-point communication …
- … each processor sends data to all others

```plaintext
for i = 0 to p-1
  irecv(src = i, …)
  isend(dest = i, …)
end for
Waitall()
```
Performance

• Each processor sends P-1 messages, each of size n/P (Let n = N^2)
• Message passing time = (P-1)(α + β (n/P))
  = α(P-1) + β (n(P-1))/P
• For short messages, this may be reasonable
• But for long messages, we’ll flood the network with O(P^2) messages

Another approach: the ring algorithm

• Let’s modify our ring passing algorithm to route the data in P steps
• Each processor passes a unique set of data to its neighbors downstream along the ring
• Data received from the upstream neighbor is passed downstream..
• But before sending the data on its way, the processor picks off the top chunk of data, sending only the remainder
The ring algorithm in action
The ring algorithm in action
Performance

- In the first step, all send P-1 chunks of data
- Then P-2, P-3, down to 1 chunk
- Each chunk has size \((n/P^2)\)
- What is the asymptotic running time?
- \[
\begin{align*}
    \text{Sum (from } i=1 \text{ to } P-1 (\alpha + \beta (n/P^2)(P - i)) &= \alpha(P-1) + (1/2) \text{ Sum (from } i=1 \text{ to } P-1 (i \beta n/P^2) \\
    &= (P-1)\alpha + ((P-1)/P)n\beta/2 = ((P-1)/P)n\beta\n\end{align*}
\]
- Same running time as the linear broadcast, but only \(O(P)\) messages sent at any one time

Another approach: the hypercube algorithm

- Consider the case of \(P=2^d\)
- Recall that we may split a \(d\)-cube into two \((d-1)\) cubes
- Let’s split the cube into a left and right half
- Each processor in the left half swaps half its data with the corresponding neighbor in the right half
- Now each processor has all the data that its neighbor intended to send to the processor’s half of the cube
- Repeat along the other directions
Flow of information
Flow of information

Bookkeeping

- Recall: in a hypercube, the address of neighbor along dimension \( k \) is \( \text{proc} \otimes \text{e}_k \)
  - \( \otimes \) is exclusive-or (XOR)
  - \( \text{e}_k \) is a bit vector of zeroes with the \( k \)th position set to 1
- Consider proc=5 (101\(_2\)), neighbor along dim 2
  - \( \text{e}_2 = 100 \)
  - \( 101 \otimes 100 = 001 \)
Relative costs

- Hypercube algorithm
  \[ \alpha \lg P + \frac{(P-1)}{P} N^2 \beta \]

- The Ring algorithm
  \[ \alpha (P-1) + \frac{(P-1)}{P} N^2 \beta \]

More sorting
Shell sort

- We’ve talked about compare exchange sorts
- Slow running times result from swaps that only involve a small window
- Let’s configure the processors in a hypercube, and carry out swaps along each dimension of the hypercube
- This new algorithm is called *shell sort*

Shell sort algorithm

1. Sort locally
2. Compare/split along each dimension in the hypercube

At this point the list is nearly sorted
3. Perform final sorting with another method i.e. Odd/Even merge sort
Running time of shell sort

1. Local sort is $O(N/P) \log N/P$
2. Compare/split along each dimension in hypercube
   for $i = 1$ to $(\log P)$
   compare/split with partner along dimension $I$
   Cost = $(N/P) \log P$
3. Final sorting with odd/even merge sort
   $L$ rounds: $L \times (N/P)$
   In the worst case $L = P$
   When does this occur?

In search of a faster sort algorithm

- The problem with shell sort is that data moves incrementally to its final destination
- Ideally, we should compute the processor that will own a given value
- We can then communicate in one step
- But how do we know which processor should own a given value?
- The specific owner depends on the distribution of keys
A first pass

- Assume that the keys are distributed uniformly over 0 to $K_{\text{max}}-1$
- Assign each key to processor 
  \[ P \times \frac{\text{key}}{(K_{\text{max}}-1)} \]
- The assignment of keys to processors is based only on the knowledge of $K_{\text{max}}$
- But if the keys are distributed non-uniformly, then this approach will result in an imbalance
- In the worst case, all the keys could go to one processor

Bucket sorting

- Assuming that the keys are evenly distributed over the range ….
- Divide the range of keys into equal subranges and associate a bucket with each range
- Examine each key, and place in the appropriate bucket $O(N)$
- Sort the buckets $O(N \log (N/m))$
  - If the keys are evenly distributed, then each bucket has $N/m$ elements
- Merge the buckets $O(N)$
- Total running time is $O(N \log(N/m))$
Parallel algorithm

- A processor can have keys over the full range of possible key values
- Each of P processors maintains P local buckets
  - Assigns each key to a local bucket
  - Routes each local bucket to the correct owner
    (each local bucket has about \(N/P^2\) elements)
  - Sorts all incoming data into a single bucket

Running time

- Local bucket assignment: \(N/P\)
- Route each local bucket to the correct owner
  All to all : what does this cost?
- Local sorting : \(N/P \log(N/P)\)
Worst case behavior

- We’ve assumed that the keys are distributed uniformly over the range
- If the keys are integers in the range \([0,Q-1]\) we can assign each processor \(k\) to a subrange \([k\times Q/P,(k+1)\times Q/P-1]\)
- E.g. for \(Q=2^{30}\), \(P=64\), each processor gets \(2^{24}=16\) M elements
- But what if all the keys are in the range \([0, 2^{24}-1]\)?
- We’ll next look at an algorithm called \textit{sample sort}, which is designed to remedy the problem

The idea behind sample sort

- Uses a heuristic to determine the key range for each processor, such that each processor will get about the same number of keys
- Sample the keys to determine a set of \(P-1\) \textit{splitters} that partition the key space into \(P\) disjoint regions
- Each region is assigned to processor, and is treated as a bucket
- Once each processor knows the splitters, it can distribute its keys to the other buckets accordingly
- Each processor sorts the keys sent it
Sample sort

• We’ll look at a few variations
• For details see:
  http://www.umiacs.umd.edu/research/EXPAR/papers/spaa96/spaa96.html

Splitter selection

• Each processor chooses $s < N/P$ keys at random and sorts them into a list of candidate splitters
• Candidate splitters are collected by one processor and then sorted
• The sorted list is sampled at uniform positions $0, s, 2s, \ldots (P-1)s$ to generate the splitter list
• The splitter list is distributed to the other processors
Limitations

• Tradeoff: as $s$ increases…
  – the distribution of the final sorted keys over the processors becomes more even
  – the cost of determining the splitters increases

• For some inputs, communication patterns can be highly irregular with some pairs of processors communicating more heavily than others

• This imbalance degrades communication performance

Enhancement: random sample sorting

• Mix up the keys as a preprocessing step
• Each processor randomly assigns each of its $n/p$ elements to one of $p$ buckets
• We treat this communication like a transpose
• Each processor sorts its assigned values locally
• A distinguished processor selects the splitters and broadcasts to the others
• Each processor collects its local keys into $p$ buckets and routes $p-1$ of these to the other processors (another transpose)
• Each processor merges the incoming keys (use radix sort)
Performance bounds

- \( T(n,p) = T_{\text{comp}}(n,p) + T_{\text{comm}}(n,p) \)
- With “high probability” \( 1 - n^{-\epsilon} \)
- No processors exchange \( > c_2 n / p^2 \) keys when
  for any \( c_2 \geq 3.10, p^2 \leq n/(3 \ln n) \)
- No processor will obtain more than \( \alpha n / p \) keys
  for any \( \alpha \geq 1.77, p^2 \leq n/(3 \ln n) \)

A word on probabilities

- For \( \alpha \geq 1.77, p^2 \leq n/(3 \ln n) \) …
- Each bucket will have no more than \( L \leq \alpha n / p \) keys
  with probability \( n^{-\epsilon} \)
- \( \epsilon = (1-1/\alpha)^2 s / 2 \)
- For \( \alpha=3, s=32, n=100,000 \), \( n^{-\epsilon} \approx 5 \times 10^{-5} \)
- For \( \alpha=3, s=64, n=1,000,000 \), \( n^{-\epsilon} \approx 3 \times 10^{-13} \)
Radix sort

- We need a good sorting algorithm to do the local sorts
- With integers keys, radix sort is the candidate of choice
- We sort the keys in passes, choosing an r-bit block at a time

A simple example

- Following an example in the NIST Dictionary of Algorithms and Data Structures http://www.nist.gov/dads/
- Uses buckets to sort the keys in passes
- Running time is $O(cn)$, $c$ depends on size of the keys and the number of buckets
- Need a stable sort: output preserves order of inputs that have the same value
Radix sort in action

• Consider the input keys
  34, 12, 42, 32, 44, 41, 34, 11, 32, and 23
• Use 4 buckets
• Sort on each digit in succession, least significant to most significant

• After pass 1
  41 11 12 42 32 32 23 34 44 34
Radix sort in action

• Consider the input keys
  34, 12, 42, 32, 44, 41, 34, 11, 32, and 23
• Use 4 buckets
• Sort on each digit in succession, least significant to most significant
• After pass 1
  41 11 12 42 32 32 23 34 44 34
• After pass 2
  11 12 23 32 32 34 34 41 42 44

Bitonic sort

• Classic parallel sorting algorithm O(log^2 n) time on n processors (for details, see on-line notes)
• Definition: A bitonic sequence is a sequence of numbers a_0, a_1...a_{n-1} with the following properties
  – There exists an index i where
    a_0 \leq a_1 \leq a_i \ldots \leq a_i \textbf{ and } a_i \geq a_{i+1} \geq a_{i+1} \ldots \geq a_{n-1}
  – We may cyclically shift the a_k while maintaining this relationship

  1,2,4,7,6,0

  7,6,0,1,2,4
Merge property of a bitonic sequence

- We can merge two bitonic sequences just like we merge two **monotonic** sequences.

Splitting property of bitonic sequences

- We can split a bitonic sequence $y$ into two bitonic sequences $L(y)$ and $R(y)$.

  $L(y) = \langle \min\{a_0, a_{n/2}\}, \min\{a_1, a_{n/2+1}\}, \ldots, \min\{a_{n/2+1}, a_{n-1}\} \rangle$

  $R(y) = \langle \max\{a_0, a_{n/2}\}, \max\{a_1, a_{n/2+1}\}, \ldots, \max\{a_{n/2+1}, a_{n-1}\} \rangle$

- See the notes for a proof.

  All values in $L(y) < R(y)$

  $L(y): 3 \ 4 \ 2 \ 1$

  $R(y): 7 \ 5 \ 8 \ 9$
Sorting a bitonic sequence

- Split the bitonic sequence \( y \) into two bitonic subsequences \( L(y) \) & \( R(y) \)
- Sort \( L(y) \) and \( R(y) \) recursively
- Merge the two sorted lists
- Since all values in \( L(y) \) are smaller than all values in \( R(y) \) we don’t need to exchange values in \( L(y) \) and \( R(y) \)

Bitonic sort algorithm

- Create a bitonic sequence \( y \) from an unsorted list
- Apply the previous algorithm to sort the bitonic sequence
- We need an algorithm to create the bitonic sequence \( y \)
Additonal properties of bitonic sequences

- Any 2 element sequence is a bitonic sequence
- We can trivially construct a bitonic sequence from two monotonic sequences, one sorted in increasing order, the other in decreasing order

\[ \vec{v} + \vec{w} = \vec{v} \]

Inductive construction of the initial bitonic sequence

- Form matched pairs of 2-element bitonic sequences, pointing up and down
- Merge these into 4-element bitonic sequences
- Continue until we have one sequence
Bitonic merging network

- Converts a bitonic sequence into a sorted sequence

From *Introduction to Parallel Computing*, V. Kumar et al, Benjamin Cummings, 1994

Bitonic conversion network

Converts an unordered sequence into a bitonic sequence

From *Introduction to Parallel Computing*, V. Kumar et al, Benjamin Cummings, 1994