Lecture 8

Advanced Parallel Sorting
Bitonic sorting
Bin Sort
Announcements

• Beware of using Break (CTRL/C) when running jobs on Valkyrie: please clean up after yourself

• Code for Cannon’s algorithm has been posted in $PUB/examples/Cannon
Advice on programming lab reports

- Develop the discussions: goals, methods, commentary, speculation, reflection
- Code fragments and pseudo code
- State the obvious
- Number the figures and tables
- Don’t present data that adds no new information
- Put related information together in the same plot
- Tabular data
- How will your plots look in Black and White
Revisiting Correctness issues

• Correction: when there are multiple outstanding iRecvs, each iRecv will get a unique message
• Advice: never use ANY_SOURCE or ANY_TAG unless non-determinism is part of the algorithm

```c
MPI_Request req1, req2;
MPI_Status status;
MPI_Irecv(buff, len, CHAR, nextnode, TYPE, WORLD,&req1);
MPI_Irecv(buff2,len, CHAR, prevnode, TYPE, WORLD,&req2);
MPI_Send(buff, len, CHAR, nextnode, TYPE, WORLD);
MPI_Send(buff2, len, CHAR, prevnode, TYPE, WORLD);
MPI_Wait(&req1, &status);
MPI_Wait(&req2, &status);
```
Outputting distributed arrays

- Recall the blocked decomposition used in Canon’s algorithm.
- The use of row and column communicators
Gather

int MPI_Gather ( void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm )

P_0

P_{p-1}

Gather

Scatter

Root
Code

$N \times N = $ size of array, $q = \sqrt{p}$

myRow = myRank % q, myCol = myRank / q;

MPI_Comm_split(comm, myRow, rank_key, &row_comm);
MPI_Comm_split(comm, myCol, rank_key, &col_comm);

MPI_Comm_rank(row_comm, &myRankR);
MPI_Comm_rank(col_comm, &myRankC);

\[ \frac{N}{\sqrt{P}} \]
Code

\( N \times N = \text{size of array}, \ q = \sqrt{p}, \ nsq = (N/q)^2 \)

\( \text{ROOT} = 0; \)

// Each row root gathers its rows
float* Local_Matrix[N/q][N/q], rows = new float[q*nsq];
MPI_Gather ( Local_Matrix, nsq, MPI_FLOAT, rows, nsq, MPI_FLOAT, ROOT, row_comm);

// Gather all the data from each root
float* cols = new float[N*N];
MPI_Gather ( rows, q*nsq, MPI_FLOAT, cols, q*nsq, MPI_FLOAT, ROOT, col_comm);
How to (and not to) expose trends
Packet time

![Graph showing packet time vs. bytes for Valkyrie network]
Focusing in (every 128)
Packet time

![Graph showing packet time vs. bytes for Valkyrie]
Packet time - short messages
Parallel Sorting
Parallel Sorting

- We now consider advanced methods
  - Bitonic sort
  - Bin sort

- In practice, we sort on external media, i.e. disk
  - Sort benchmarking: http://research.microsoft.com/barc/SortBenchmark
  - 125 M records (116GB): record for number of 100-byte records (with 10-byte keys) that can be sorted in a minute
Bitonic sort

- Classic parallel sorting algorithm $O(\log^2 n)$ time on $n$ processors
- **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 $a_0 \leq a_1 \leq a_i \ldots \leq a_i$ 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

$$1, 2, 4, 7, 6, 0 \quad \uparrow \quad \downarrow \quad 7, 6, 0, 1, 2, 4$$
Merge property of a bitonic sequence

• We may merge two bitonic sequences in much the same way as 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 is easy

- 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 $R(y)$
- When $|L(.)| < 3$, sorting is trivial
- We designate $S(n)$ to be sort on of an $n$-element bitonic sequence
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 $y$
Additional 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
Inductive construction of the initial bitonic sequence

- Form matched pairs of 2-element bitonic sequences, pointing up and down \([B(2)]\)
- Trivially merge these into 4-element bitonic sequences
- Now form matched pairs of 4-element sequences \([B(4)]\)
- Apply \(S(4)\) to each sequence, sorting the first upward, the second downward
- Trivially merge into an 8-element bitonic sequence
- Continue until there is just one sequence
Implementing the bitonic sort algorithm

- Create a bitonic sequence $y$ from an unsorted list, $B(n)$
- Apply the previous algorithm to sort the bitonic sequence, $S(n)$
- We use comparators to re-order data
- We use a shuffle exchange network to form $L(y)$ and $R(y)$
  - This network shuffles an $n$-element sequence by interleaving $x_0$, $x_{n/2}$, $x_1$, $x_{n/2+1}$, …
Comparators

- Given two values x & y, produce two outputs
  - For an increasing comparator, the output is\[\text{min}[x,y], \text{max}[x,y]\]
  - For a decreasing comparator, the output is\[\text{max}[x,y], \text{min}[x,y]\]
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

\[ B(4) = S(4) + S(2) \]

From Introduction to Parallel Computing, V. Kumar et al, Benjamin Cummings, 2003
Bin Sort
Recall rank sort

- 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

```plaintext
forall i=0:n-1, j=0:n-1
    if ( x[i] > x[j] ) then rank[i] += 1 end if
forall i=0:n-1
    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
1st attempt: bucket sort

- Divide the range of keys into equal subranges and associate a bucket with each range
- Each of p processors maintains p local buckets
  - Assigns each key to a local bucket:
    \[ p \times \frac{\text{key}}{K_{\text{max}} - 1} \]
  - 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
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

- Runs on an IBM SP3 system: 16-way SMP nodes w/ Power 3 CPUs
- Scaled runs: 1M points per processor

Local sort: quicksort
\[ \Theta(n/p \log(n/p)) \]

All-to-allv
\[ \Theta(n) \]
The send and receive lists

- Send list for process 0
  - \{0->0\}, send count, displ: 634 0
  - \{0->1\}, send count, displ: 601 634
  - \{0->2\}, send count, displ: 638 1235
  - \{0->3\}, send count, displ: 627 1873

- Send list for process 1
  - \{1->0\}, send count, displ: 616 0
  - \{1->1\}, send count, displ: 622 616
  - \{1->2\}, send count, displ: 642 1238
  - \{1->3\}, send count, displ: 620 1880

- Receive list for process 0
  - \{0<-0\}, recv count, displ: 634 0
  - \{0<-1\}, recv count, displ: 616 634
  - \{0<-2\}, recv count, displ: 671 1250
  - \{0<-3\}, recv count, displ: 659 1921

- Receive list for process 1
  - \{1<-0\}, recv count, displ: 601 0
  - \{1<-1\}, recv count, displ: 622 601
  - \{1<-2\}, recv count, displ: 598 1223
  - \{1<-3\}, recv count, displ: 624 1821

<table>
<thead>
<tr>
<th>Receiver</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>Receive Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>634</td>
<td>616</td>
<td>671</td>
<td>659</td>
<td>2580</td>
</tr>
<tr>
<td>1</td>
<td>601</td>
<td>622</td>
<td>598</td>
<td>624</td>
<td>2445</td>
</tr>
<tr>
<td>2</td>
<td>638</td>
<td>642</td>
<td>--</td>
<td>--</td>
<td>2515</td>
</tr>
<tr>
<td>3</td>
<td>627</td>
<td>620</td>
<td>--</td>
<td>--</td>
<td>2460</td>
</tr>
</tbody>
</table>
The collective calls

• Processes transmit varying amounts of information to the other processes
• This is an MPI_Alltoallv
  ( SKeys, send_counts, send_displace, MPI_INT, RKeys, 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

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

• A simple loop determines the displacements

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

\]
Worst case behavior

- The assignment of keys to processors is based only on the knowledge of $K_{\text{max}}$
- If keys are in range $[0, Q-1]$ … $[k\frac{Q}{P}, (k+1)\frac{Q}{P}]$ … processor $k$ has keys in the range
- For $Q=2^{30}$, $P=64$, each processor gets $2^{24} = 16$ M elements
- What if keys $\in [0, 2^{24} - 1] \subset [0, 2^{30} - 1]$
- For a non-uniform distribution, we need more information to balance keys (and communication) over the processors
- In the worst case, all the keys could go to one processor
Improving on bucket sort

- *Sample sort* remedies the problem
- “Parallel Algorithms for Personalized Communication and Sorting With an Experimental Study.”
- “Parallel Sorting by Regular Sampling.” H. Shi and J. Schaeffer.
- “Parallel sorting on a shared-nothing architecture using probabilistic splitting.” D. J. DeWitt, J. F. Naughton, J. F., and D. A. Schneider,
- “Samplesort: A Sampling Approach to Minimal Storage Tree Sorting,”
The idea behind sample sort

- Uses a heuristic to estimate the distribution of the global key range over the p processors
- Sample the keys to determine a set of p-1 splitters that partition the key space into p disjoint intervals
- Each interval 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
- Processors sort incoming keys
Splitter selection: regular sampling

• Shi and Schaeffer [1992]
• Each processor sorts its local keys, then selects $s$ evenly spaced samples at uniform positions $0, s, 2s, \ldots (p-1)s$
• These candidate splitters are collected by one processor
  ▶ Sorted
  ▶ Sampled at uniform positions to generate a $p-1$ element splitter list
  ▶ Broadcasted to the other processors
• All use these splitters to route their data to others
Performance

• Assuming $n \geq p^3$ …

• $T_P = O((n/p) \log n)$

• If $s = p$, each processor will merge not more than
  $$2n/p + n/s - p$$
  elements

• If $s > p$, each processor will merge not more than
  $$\frac{3}{2}(n/p) - \frac{n}{ps} + 1 + d$$
  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
Limits to Performance

• Bottleneck is the redistribution phase: all to all gather

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

• This imbalance degrades communication performance

\[
\begin{array}{cccccccc}
  k & k & k & k & k & k & k & k \\
  + & + & + & + & + & + & + & + \\
  1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

\[
\begin{array}{c}
  \ldots \\
  + \\
\end{array}
\]

\[
\begin{array}{c}
  n \\
\end{array}
\]

\[
\begin{array}{c}
  k \\
\end{array}
\]
Improved regular sampling

- We want all pairs of processors to communicate about the same amount of data
- Let’s use a cyclic scattering strategy to break up key concentrations
- Hellman et al. [1996]
- Keys distributed nearly uniformly across the processors
- Introduces an additional global communication stage

<table>
<thead>
<tr>
<th>k</th>
<th>k</th>
<th>k</th>
<th>k</th>
<th>k</th>
<th>k</th>
<th>k</th>
<th>k</th>
<th>.....</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>.....</td>
<td>+</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>.....</td>
<td>n</td>
</tr>
</tbody>
</table>
Improved regular sampling algorithm

- Processors sort their assigned input
- Each processor “deals” its sorted data into the $p$ bins
  - The $k^{th}$ item is placed into position $[k/p]$ of bin $k \mod p$
  - When done, routes bin $j$ to processor $j$
  - 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

• Guarantees that
  – No processor will have more than $n/p + n/s - p$ elements
  – The number of values exchanged at most $n/p^2 + n/sp - 1$
• For $s=64$, $n >> p$, each processor holds more than about 1.6% above the ideal
• First communication stage has a uniform communication pattern
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
Radix sort

• We need a **stable** sorting algorithm to do the local sorts: the output preserves the order of inputs having the same associated key

• **radix sort** meets our needs: sort the keys in passes, choosing an r-bit block at a time, O(cn) running time, c depends on size of the keys and # of buckets

• See NIST’s *Dictionary of Algorithms and Data Structures*
  http://www.nist.gov/dads/HTML/radixsort.html

• Explanation with a demo
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
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