Algorithms at Scale
(Week 11)

Map-Reduce (MPC) Algorithms
Summary

Today: Map-Reduce

Map-Reduce Model
- Cluster computing

Some simple examples
- Word count
- Join

Algorithms
- Bellman-Ford
- PageRank

Last Week: Multicore

Models of Parallelism
- Fork-Join model
- Work and Span
- Greedy schedulers

Algorithms
- Sum
- MergeSort
- Parallel Sets
- BFS
- Prefix-Sum
- (Luby’s)
Announcements / Reminders

Today:

MiniProject explanatory section due today.

Next week:

MiniProject talk due
Recap: Prefix Sum

Algorithm 2: \textsc{PrefixSumPartOne}($A, level, begin, end$)

1 \hspace{1em} \text{if} \ (begin = end) \ \text{then}
2 \hspace{1em} \hspace{1em} \text{return} \ A[begin]
3 \hspace{1em} \text{else}
4 \hspace{1em} \hspace{1em} \text{mid} = (begin + end)/2
5 \hspace{1em} \hspace{1em} \text{in parallel}
6 \hspace{1em} \hspace{1em} \hspace{1em} (1) \ s_1 = \textsc{PrefixSumPartOne}($A, level - 1, begin, mid$)
7 \hspace{1em} \hspace{1em} \hspace{1em} (2) \ s_2 = \textsc{PrefixSumPartOne}($A, level - 1, mid + 1, end$)
8 \hspace{1em} S[level, end, left] = s_1
9 \hspace{1em} S[level, end, right] = s_2
10 \hspace{1em} \text{return} \ s_1 + s_2
Recap: Prefix Sum

Algorithm 3: \texttt{PREFIX\_SUM\_PART\_TWO}(A, level, sum, begin, end)

\begin{verbatim}
1  if (begin = end) then
3  else
4      mid = (begin + end)/2
5         in parallel
6          (1) PREFIX\_SUM\_PART\_TWO(A, level - 1, sum, begin, mid)
7          (2) PREFIX\_SUM\_PART\_TWO(A, level - 1, sum + S[level, end, left], mid + 1, end)
8  return
\end{verbatim}
Recap: Prefix Sum

Algorithm 3: PREFIX_SUM_PART_TWO(A, level, sum, begin, end)

1. if (begin = end) then
3. else
4. \[ mid = (begin + end)/2 \]
5. in parallel
6. (1) PREFIX_SUM_PART_TWO(A, level - 1, sum, begin, mid)
7. (2) PREFIX_SUM_PART_TWO(A, level - 1, sum + S[level, end, left], mid + 1, end)
8. return
Recap: Binary Prefix Sum

A[j] = number of 1's in A[1..j]
Recap: Partition

Goal: partition array around key $k$

Example: $k = 4$

$$
\begin{array}{cccccccc}
7 & 9 & 3 & 2 & 5 & 8 & 4 & 2 \\
\end{array}
$$

$$
\begin{array}{cccccccc}
3 & 2 & 2 & 4 & 7 & 9 & 5 & 8 \\
\end{array}
$$
Recap: Partition

Step 1: mark items < k

Example: k = 4

<table>
<thead>
<tr>
<th>7</th>
<th>9</th>
<th>3</th>
<th>2</th>
<th>3</th>
<th>8</th>
<th>4</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Work: O(n)
Span: O(log n)
Recap: Partition

Step 2: prefix sums

Example: $k = 4$

\[
\begin{array}{cccccccc}
7 & 9 & 3 & 2 & 3 & 8 & 4 & 2 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\
0 & 0 & 1 & 2 & 3 & 0 & 0 & 4 \\
\end{array}
\]

Work: $O(n)$
Span: $O(\log n)$
Recap: Partition

Step 2: prefix sums

Example: $k = 4$

\[
\begin{array}{cccccccc}
7 & 9 & 3 & 2 & 3 & 8 & 4 & 2 \\
0 & 0 & 1 & 2 & 3 & 0 & 0 & 4 \\
\end{array}
\]

Work: $O(n)$
Span: $O(\log n)$
Recap: Partition

Step 3: mark items ≥ k

Example: k = 4

Work: $O(n)$
Span: $O(\log n)$
Recap: Partition

Step 4: prefix sum

Example: $k = 4$

Work: $O(n)$
Span: $O(\log n)$
Recap: Partition

Step 4: prefix sum

Example: $k = 4$

\[
\begin{array}{cccccccc}
7 & 9 & 3 & 2 & 3 & 8 & 4 & 2 \\
0 & 0 & 1 & 2 & 3 & 0 & 0 & 4 \\
1 & 2 & 0 & 0 & 0 & 3 & 4 & 0 \\
\end{array}
\]

Work: $O(n)$
Span: $O(\log n)$
Recap: Partition

Step 5: add size from \((< k)\) prefix-sum to \((\geq k)\) prefix sum.

Example: \(k = 4\)

<table>
<thead>
<tr>
<th>7</th>
<th>9</th>
<th>3</th>
<th>2</th>
<th>3</th>
<th>8</th>
<th>4</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>

Work: \(O(n)\)
Span: \(O(\log n)\)
Recap: Partition

Step 5: add size from \((< k)\) prefix-sum to \((\geq k)\) prefix sum.

Example: \(k = 4\)

\[
\begin{array}{cccccccc}
7 & 9 & 3 & 2 & 3 & 8 & 4 & 2 \\
0 & 0 & 1 & 2 & 3 & 0 & 0 & 4 \\
5 & 6 & 0 & 0 & 0 & 7 & 8 & 0 \\
\end{array}
\]

Work: O(n)
Span: O(log n)
Recap: Partition

Step 6: compress

Example: $k = 4$

\[
\begin{array}{cccccccc}
7 & 9 & 3 & 2 & 3 & 8 & 4 & 2 \\
0 & 0 & 1 & 2 & 3 & 0 & 0 & 4 \\
5 & 2 & 0 & 0 & 0 & 7 & 7 & 0 \\
5 & 6 & 1 & 2 & 3 & 7 & 8 & 4
\end{array}
\]

Work: $O(n)$
Span: $O(\log n)$
Recap: Partition

Step 6: compress

Example: $k = 4$

<table>
<thead>
<tr>
<th>7</th>
<th>9</th>
<th>3</th>
<th>2</th>
<th>3</th>
<th>8</th>
<th>4</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>9</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>

Work: $O(n)$
Span: $O(\log n)$
Recap: Partition

Step 6: copy to final location

Example: $k = 4$

Work: $O(n)$  
Span: $O(\log n)$
Recap: Partition

Partition around $k$:

<table>
<thead>
<tr>
<th>Work: $O(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Span: $O(\log n)$</td>
</tr>
</tbody>
</table>

Example: $k = 4$

```
7  9  3  2  3  8  4  2
  5  6  1  2  3  7  8  4
  3  2  3  2  7  9  8  4
```

```
1  2  3  4  5  6  7  8
```
Recap: Partition

Exercise:
Write down the algorithm precisely for each of the steps.
(Combine several steps together!)

Do the work and span analysis.

<table>
<thead>
<tr>
<th>7</th>
<th>9</th>
<th>3</th>
<th>2</th>
<th>3</th>
<th>8</th>
<th>4</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3</th>
<th>2</th>
<th>3</th>
<th>2</th>
<th>7</th>
<th>9</th>
<th>8</th>
<th>4</th>
</tr>
</thead>
<tbody>
<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>
</tr>
</tbody>
</table>
Recap: QuickSort

**QuickSort**\( (A, \text{begin}, \text{end}) \)

\[
\text{pivot} = \text{random}(\text{begin}, \text{end})
\]

\[
\text{split} = \text{partition}(A, \text{begin}, \text{end}, \text{pivot})
\]

\[
\text{mid} = (\text{begin}+\text{end})/2
\]

**in parallel:**

1. **QuickSort**\( (A, \text{begin}, \text{mid}) \)
2. **QuickSort**\( (A, \text{mid}+1, \text{end}) \)
Recap: QuickSort Work

**QuickSort(A, begin, end)**

- **pivot** = random(begin, end)  \( \text{O}(1) \)
- **split** = partition\( (A, \text{begin}, \text{end}, \text{pivot}) \)  \( \text{O}(n) \)
- **mid** = (begin+end)/2  \( \text{O}(1) \)

**in parallel:**

1. QuickSort\( (A, \text{begin}, \text{mid}) \)  \( 2W(n/2) \)
2. QuickSort\( (A, \text{mid+1}, \text{end}) \)

\[
W(n) = 2W(n/2) + O(n) = O(n \log n)
\]

**Assume random pivot is the exact median. Precise randomized analysis is identical to the sequential version.**
Recap: QuickSort Span

QuickSort(A, begin, end)

pivot = random(begin, end) \[\text{O}(1)\]

split = partition(A, begin, end, pivot) \[\text{O}(\log n)\]

mid = (begin + end) / 2 \[\text{O}(1)\]

in parallel:
1. QuickSort(A, begin, mid) \[S(n/2)\]
2. QuickSort(A, mid+1, end)

\[S(n) = S(n/2) + O(\log n) = O(\log^2 n)\]

** Assume random pivot is the exact median.
Precise randomized analysis is identical to the sequential version.
Recap: QuickSort Span

QuickSort(A, begin, end)

\[
\text{pivot} = \text{random}(\text{begin}, \text{end})
\]

\[
\text{split} = \text{partition}(A, \text{begin}, \text{end}, \text{pivot})
\]

\[
\text{mid} = (\text{begin}+\text{end})/2
\]

**in parallel:**

1. QuickSort(A, begin, mid)
2. QuickSort(A, mid+1, end)

Exercise:

Modify the algorithm to efficiently sort arrays with repeated elements. (As described, this is very slow for an array of all 1’s.)
Fork-Join algorithms

Assumptions:
- Tightly synchronized
- Shared memory

Advantages:
- Simple algorithm design
- Focus on parallelism (*computational*)
- Easy analysis: work and span is enough!
- Minimizes race conditions, deadlocks, etc.
Yahoo TeraSort:

- Each node has:
  - 8 cores: 2GHz
  - 8 GB RAM
  - 4 disks: 4TB each

- 40 nodes / rack (interconnect: 1GB/s switch)
- 25-100 racks (interconnect: 8GB/s switch)

⇒ ~16,000 cores
High Performance Clusters

Yahoo TeraSort:
- Each node has:
  - 8 cores: 2GHz
  - 8 GB RAM
  - 4 disks: 4TB each
- 40 nodes / rack (interconnect: 1GB/s switch)
- more racks (interconnect: 8GB/s switch)

⇒ 50,400 cores

2013:
Yahoo (Hadoop) sorts 100TB of data in 72 minutes.
High Performance Clusters

DataBricks TeraSort:
- 206 nodes
- 6,592 cores

DataBricks PetaSort:
- 190 nodes
- 6,080 cores

Record (2014):
DataBricks (Spark) sorts 100TB of data in 23 minutes.

Record (2014):
DataBricks (Spark) sorts 1PB of data in 234 minutes.
High Performance Clusters

Assumptions:
- Loosely synchronized
- No shared memory
- Data exchanged over fast interconnect

Fork/Join is not a good model for clusters.
High Performance Clusters

Assumptions:

- Loosely synchronized
- No shared memory
- Data exchanged over fast interconnect

Issues:

- Communication cost?
- Coordination among cores?
- Fine-grained parallelism?

Fork/Join is not a good model for clusters.
Recap: Prefix Sum

Algorithm 2: \textsc{PrefixSumPartOne}(A, level, begin, end)


1 \begin{verbatim}
if (begin = end) then
    return A[begin]
else
    mid = (begin + end)/2
    in parallel
    (1) s_1 = \textsc{PrefixSumPartOne}(A, level - 1, begin, mid)
    (2) s_2 = \textsc{PrefixSumPartOne}(A, level - 1, mid + 1, end)
    S[level, end, left] = s_1
    S[level, end, right] = s_2
    return s_1 + s_2
\end{verbatim}
Recap: Prefix Sum

send to node 1

send to node 2
Recap: Prefix Sum

Open question:
Could a scheduler translate fork-join algorithms to a cluster?
High Performance Clusters

Assumptions:
- Loosely synchronized
- No shared memory
- Data exchanged over fast interconnect

Issues:
- Communication cost?
- Coordination among cores?
- Fine-grained parallelism?

Fork/Join is not a good model for clusters.
High Performance Clusters

Map-Reduce Model:
- Target: high-performance clusters
- Focus: data (not computation)

Inventor: Google
- processing web data

Today: ubiquitous (Amazon, Yahoo, Facebook, etc.,)
- Hadoop, etc.
Map-Reduce Model

Data: (key, value) pairs

- All data is stored as key/value pairs.
- Initially stored on some shared disk.
  - e.g., GFS (Google File System), HDFS (Hadoop FS)
- During the computation, route (key/value) pairs to different servers to perform the computation.
Basic round:

1. **Map**: process each (key, value) pair
2. **Shuffle**: group items by key
3. **Reduce**: process items with same key together

Plan:

Load data from disk.

Execute several rounds.

Save (key, value) pairs, sorted by key.
Map-Reduce Example

Input: $A = [3, 2, 1, 6, 4]$  

Compute: $\sum_{j \in \text{odd}} A[j]^2, \sum_{j \in \text{even}} A[j]^2$
Input: $A = [3, 2, 1, 6, 4]$

Compute: $\sum_{j \in \text{odd}} A[j]^2, \sum_{j \in \text{even}} A[j]^2$

Step 1: Load (key, value) pairs: $[A[j] \rightarrow (j, A[j])]$

$\{(1, 3), (2, 2), (3, 1), (4, 6), (5, 4)\}$

key = position  value = array entry
Map-Reduce Example

map(key, value) ➔ (key, value)

Step 2:

map(key, value)

if (key is even)
    then emit(2, value*value)

else if (key is odd)
    then emit(1, value*value)
Map-Reduce Example

Properties of map function:
• processes one (key, value) pair at a time
• no saved state
• scheduler allocates map processes to cores

map(key, value)

if (key is even)
    then emit(2, value*value)
else if (key is odd)
    then emit(1, value*value)
Map-Reduce Example

Input: \( A = [3, 2, 1, 6, 4] \)

Compute: \( \sum_{j \in \text{odd}} A[j]^2, \sum_{j \in \text{even}} A[j]^2 \)

Step 2: Map

\[ (1, 3), (2, 2), (3, 1), (4, 6), (5, 4) \]

\[ (1, 9), (2, 4), (1, 1), (2, 36), (1, 16) \]
Map-Reduce Example

Input: $A = [3, 2, 1, 6, 4]$

Compute: $\sum_{j \in \text{odd}} A[j]^2$, $\sum_{j \in \text{even}} A[j]^2$

Step 3: Shuffle

$(1, 9), (2, 4), (1, 1), (2, 36), (1, 16)$

$(1, 9), (1, 1), (1, 16), (2, 4), (2, 36)$
Map-Reduce Example

reduce(key, \([v_1, v_2, \ldots]\)) \rightarrow (key, \text{value}) \text{ pair(s)}

Step 3:

\[
\text{reduce(key, V[\ldots])}
\]

\[
\text{sum} = 0
\]

\[
\text{for (j = 1 to } |V|) \quad \text{sum} = \text{sum} + V[j]
\]

\[
\text{emit(key, sum)}
\]
Map-Reduce Example

Properties of reduce function:
- processes all values with the same key
- scheduler allocates reduce processes to cores
- scheduler routes all (key, *) pairs to that reducer

\[
\text{reduce(key, V[...])} \\
\text{sum} = 0 \\
\text{for } (j = 1 \text{ to } |V|) \\
\quad \text{sum} = \text{sum} + V[j] \\
\text{emit(key, sum)}
\]
Map-Reduce Example

Input: \( A = [3, 2, 1, 6, 4] \)

Compute: \( \sum_{j \in \text{odd}} A[j]^2, \sum_{j \in \text{even}} A[j]^2 \)

Step 4: Reduce

\[(1, 9), (1, 1), (1, 16), (2, 4), (2, 36)\]

\[(1, 26), (2, 40)\]
Map-Reduce Example

Input: $A = [3, 2, 1, 6, 4]$

Compute: $\sum_{j \in \text{odd}} A[j]^2$, $\sum_{j \in \text{even}} A[j]^2$

Step 5: Write back to disk

(1, 26), (2, 40)

Out = [26, 40]
Map-Reduce Schematic
Note:
Just like Fork-Join, Map-Reduce relies on a scheduler to assign Map and Reduce processes to cores.
Map-Reduce

Metric: number of rounds

Example: 1 round

Goal: algorithms that run in $O(1)$ rounds.

→ Each map-reduce round is expensive.
Map-Reduce

There exists a 1 round Map-Reduce algorithm for every computable problem.
There exists a 1 round Unrestricted Map-Reduce algorithm for every computable problem.

Algorithm:
1. Map all data to key 1.
2. Reduce key 1: compute the answer on a single core.

Not very useful!
Not very parallel!
(Real) Map-Reduce

Restrictions:
(Real) Map-Reduce

Restriction on computation:

Each Map and Reduce process should be efficient, fast, polynomial time.

- Cannot solve NP-hard problems.
- Map and Reduce processes should not be expensive.
Restriction on memory:

Each Map and Reduce process should use “sublinear” memory in the size of the problem.

- If the data is initially size $n$, no map or reduce process should use more than $O(n^\varepsilon)$ memory.
- For example: no more then $O(\sqrt{n})$ memory.

(Sometimes we relax this restriction, but the memory use should be much smaller than the entire dataset.)
Restriction on communication:

Each Map and Reduce process should input/output a “sublinear” number of (key, value) pairs.

- If the data is initially size $n$, no map or reduce process should take as input more than $O(n^\epsilon)$ pairs.
- If the data is initially size $n$, no map or reduce process should emit more than $O(n^\epsilon)$ pairs.
- For example: no more then $O(\sqrt{n})$ key/value pairs.

(Sometimes we relax this restriction, but the number of keys should be much smaller than the entire dataset.)
(Real) Map-Reduce

Restriction on communication:

Each (key, value) pairs should not be too big.

- A (key, value) pair should be size $O(\text{polylog } n)$.
- Should not store too much information in a single key/value pair.
Map-Reduce

What is the speed bottleneck?

- Data movement
- Communication bandwidth
- Shuffling
- Reading / writing from disk
Basic round:

1. **Map**: process each (key, value) pair
2. **Shuffle**: group items by key
3. **Reduce**: process items with same key together

Plan:

Load data from disk.

Execute several rounds.

Save (key, value) pairs, sorted by key.
Example 1: Word Count

Input:
- File IN where IN[j] is a word

Output:
- File OUT where OUT[j] is a (word, count) pair.
- Each pair indicates how many times the word appears in the input file.
Example 1: Word Count

map(key, value)

emit(word, 1)
Example 1: Word Count

map(key, value)
emit(word, 1)

Notes:
• File is translated into (key, value) pairs.
Example 1: Word Count

\[
\text{map(key, value)} \\
\text{emit(word, 1)}
\]

Notes:
- File is translated into (key, value) pairs.
- Using a string as a key.
Example 1: Word Count

map(key, value)
emitting(word, 1)

Notes:

• File is translated into (key, value) pairs.
• Using a string as a key.
• Assumes a hash function translates strings to integers.
Example 1: Word Count

\[
\text{reduce(word, count[...])}
\]

\[
\text{sum} = 0
\]

\[
\text{for (i=1 to |count|)}
\]

\[
\text{sum} = \text{sum} + \text{count}[i]
\]

\[
\text{emit(word, count)}
\]
Example 1: Word Count

\[
\text{reduce}(\text{word}, \text{count}[\ldots])
\]

\[
\begin{align*}
\text{sum} &= 0 \\
\text{for } (i=1 \text{ to } |\text{count}|) & \quad \text{sum} = \text{sum} + \text{count}[i] \\
\text{emit}(\text{word}, \text{count})
\end{align*}
\]

Problem: what if all the words in the input file are the same?

Size is not sublinear!
Example 1: Word Count

reduce(word, count[...])

$$\text{sum} = 0$$

for $$i = 1$$ to $$|\text{count}|$$

$$\text{sum} = \text{sum} + \text{count}[i]$$

emit(word, count)

Reduce function is associative!

Scheduler can call reduce function on a few keys at a time.
Example 1: Word Count

\[
\text{reduce}(\text{word}, \text{count}[]) \\
\text{sum} = 0 \\
\text{for } (i=1 \text{ to } |\text{count}|) \\
\quad \text{sum} = \text{sum} + \text{count}[i] \\
\text{emit(\text{word}, \text{count})}
\]

(“gaa”, 1), (”gaa”, 1), (”gaa”, 1), (”gaa”, 1)  
(“gaa”, 2), (”gaa”, 2)
Example 1: Word Count

```plaintext
reduce(word, count[...])

sum = 0
for (i=1 to |count|)
    sum = sum + count[i]
emit(word, count)
```

- ("gaa", 2), ("gaa", 2)
- ("gaa", 4)
Example 1: Word Count

\[
\text{reduce}(\text{word, count}[\ldots])
\]

\[
\begin{align*}
\text{sum} &= 0 \\
\text{for } (i=1 \text{ to } |\text{count}|) & \quad \text{sum} = \text{sum} + \text{count}[i] \\
\text{emit}(\text{word, count})
\end{align*}
\]

Reduce function is associative!

Scheduler can call reduce function on a few keys at a time.
Example 1: Word Count

\[
\text{reduce}(\text{word}, \text{count}[\ldots])
\]

\[
\text{sum} = 0
\]

\[
\text{for} \ (i=1 \ \text{to} \ |\text{count}|)
\]

\[
\text{sum} = \text{sum} + \text{count}[i]
\]

\[
\text{emit}(\text{word}, \text{count})
\]

Note: analogous to a summation tree in the fork-join model.
Example 2: Join

Input:
- Set $A = (x_1, y_1), (x_2, y_2), (x_3, y_3), \ldots$
- Set $B = v_1, v_2, v_3, v_4, \ldots$

Output:
- Items in $A$ selected by keys in $B$.
- More precisely:

  \[ \{ y_i : \exists j, x_i = v_j \} \]
Example 2: Join

Input:

- Set $A = (x_1, y_1), (x_2, y_2), (x_3, y_3), \ldots$
- Set $B = v_1, v_2, v_3, v_4, \ldots$

Output:

- Items in $A$ selected by keys in $B$.
- More precisely:
  \[ \{ y_i : \exists j, x_i = v_j \} \]

Sequential solution:
- double-loop
- hashing
- etc.
Example 2: Join

mapA(key, (x,y))
emit(x, y)

mapB(key, (x,y))
emit(v, BVALUE)
Example 2: Join

\[
\text{mapA(key, (x,y))} \\
\text{emit(x, y)}
\]

\[
\text{mapB(key, (x,y))} \\
\text{emit(v, BVALUE)}
\]

Notes:
- Set A and set B map to different keys.
- Use key to indicate which mapper to use.
Example 2: Join

map(key, (x,y))
    if (key = A) then…
    else if (key = B) then…

Notes:
• Set A and set B map to different keys.
• Use key to indicate which mapper to use.
Example 2: Join

mapA(key, (x,y))
emit(x, y)

mapB(key, (x,y))
emit(v, BVALUE)

Notes:
• Set A and set B map to different keys.
• Use key to indicate which mapper to use.
Example 2: Join

```
reduce(key, values[...])
    if BVALUE in values
        for j = 1 to |values|
            if values[j] != BVALUE then
                emit(key, values[i])
```
Example 2: Join

\[
\text{reduce}(\text{key}, \text{values}[]) \\
\quad \text{if } \text{BVALUE in values} \\
\quad \quad \text{for } j = 1 \text{ to } |\text{values}| \\
\quad \quad \quad \text{if } \text{values}[j] \neq \text{BVALUE} \text{ then} \\
\quad \quad \quad \quad \text{emit}(\text{key}, \text{values}[i])
\]

Is this associative?
Example 2: Join

reduce(key, values[…])

if BVALUE in values
    for j = 1 to |values|
        if values[j] != BVALUE then
            emit(key, values[i])

Is this associative?

No! Not as written.
If BVALUE is processed by a different reducer, then important values may be lost.
Example 2: Join

reduce(key, v₁, v₂, v₃, …)

\[
\text{if } \text{BVALUE} = v₁ \\
\text{for each } v_j \\
\text{if } v_j \neq \text{BVALUE} \text{ then} \\
\text{emit(key, } v_j) \\
\]

Reducer can process values in a stream:

(“gaa”, BVALUE”), ("gaa", 2), ("gaa", 7), ("gaa", 1), …

As long as BVALUE is the first (key, value) pair in stream.
Example 3: Sorting

Input:
- Array $A = [x_1, x_2, x_3, x_4, x_5, x_6, ...]$

Output:
- Sorted array
Example 3: Sorting

map (key, value)
   emit(value, value)

reduce(key, V)
   for (v in V)
      emit(v, v)
Example 3: Sorting

map (key, value)
  emit(value, value)

reduce(key, V)
  for (v in V)
    emit(v, v)

Notes:
• Map and Reduce functions do nothing.
• Sorting occurs inside the framework.
• Shuffle and output phases do sort.
Map-Reduce Model

Basic round:

1. **Map**: process each (key, value) pair
2. **Shuffle**: group items by key
3. **Reduce**: process items with same key together

Plan:

- Load data from disk.
- Execute several rounds.
- Save (key, value) pairs, sorted by key.
Example 3: Bucket Sort

map (key, value)

choose j : (jB ≤ value < (j+1)B)

emit(j, value)

reduce(key, V)

sort(V)

for (j = 1 to |V|)

emit(key*B+j, v)

Fix B = number of buckets.
Example 3: Bucket Sort

map (key, value)

    choose j : (jB ≤ value < (j+1)B)
    emit(j, value)

reduce(key, V)

    sort(V)
    for (j = 1 to |V|)
        emit(key*B+j, v)

Only reasonable if: B is large (e.g., n^{1/2})
values are well distributed
Map-Reduce and Graphs
Single-Source Shortest Paths

- graph $G = (V,E)$, $n=|V|$, $m=|E|$
- source $s \in V$
- weights $w : V \rightarrow \mathbb{R}$

Output:

For each vertex $v$: distance $d(v)$ from the source.
Map-Reduce and Graphs

Bellman-Ford

\[ BF(V, E, s, w) \]

\[ s.\text{est} = 0 \]

\[ \text{for each node } u: \quad u.\text{est} = \infty \]

\[ \text{repeat } |V| \text{ times:} \]

\[ \quad \text{for each node } u: \]

\[ \quad \quad \text{for each neighbor } v \text{ of } u: \]

\[ \quad \quad \quad \text{if } v.\text{est} > u.\text{est} + w(u,v) \]

\[ \quad \quad \quad v.\text{est} = u.\text{est} + w(u,v) \]
Map-Reduce and Graphs

Bellman-Ford

- Time: $O(nm)$
- Order of edge relaxation does not matter.
- Easy to parallelize: can relax all edges at the same time.
Bellman-Ford

What keys should we use?

– Each node has a nodeID.
– Use nodeID as the key.
Bellman-Ford

What keys should we use?
- Each node has a nodeID.
- Use nodeID as the key.

What should the value be?
- nodeID
- est
- nbrIDs = [x₁, x₂, …]
- nbrWeights = [w₁, w₂, …]
What keys should we use?

– Each node has a nodeID.
– Use nodeID as the key.

What should the value be?

– nodeID
– est
– nbrIDs = [x₁, x₂, …]
– nbrWeights = [w₁, w₂, …]

Bellman-Ford

What if this is too big?

How else do you want to store the adjacency list?
Bellman-Ford

What keys should we use?

- Each node has a nodeID.
- Use nodeID as the key.

What should the value be?

- nodeID
- est
- nbrID = [x₁, x₂, …]
- nbrWeight = [w₁, w₂, …]

What if this is too big?

How else do you want to store the adjacency list?

Remember how we stored the graph as a list of edges to build cache-efficient algs?
Bellman-Ford

map (nodeID, u)

emit(nodeID, u)

for i = 1 to |u.nbrIDs|

emit(u.nbrID[i], u.est+u.nbrWeight[i])
Bellman-Ford

\[
\text{map (nodeID, u)} \\
\quad \text{emit(nodeID, u)} \\
\quad \text{for } i = 1 \text{ to } |u.\text{nbrIDs}| \\
\quad \quad \text{emit(u.nbrID}[i], u.\text{est}+u.\text{nbrWeight}[i]) \\
\]

re-output same (key, value) pair
Bellman-Ford

map (nodeID, u)

emit(nodeID, u)

for i = 1 to |u.nbrIDs|

emit(u.nbrID[i], u.est+u.nbrWeight[i])

re-output same (key, value) pair

Two types of (key, value) pairs emitted:
1. Node type
2. estimate type
Bellman-Ford

map(nodeID, u)

emit(nodeID, u)

for i = 1 to |u.nbrIDs|

emit(u.nbrID[i], u.est+u.nbrWeight[i])

re-output same (key, value) pair

Two types of (key, value) pairs emitted:
1. Node type
2. estimate type

send (estimate+weight) to neighbor

if (v.est > u.est + w(u,v)) then...
Bellman-Ford

reduce(nodeID, val[...])

Let \( w \) be the “node” in the array val[...].

for \( i = 1 \) to \(|val|\)

if \( \text{val}[i] \) is not a “node”

\[
\text{if } w.\text{est} > \text{val}[i] \text{ then } w.\text{est} = \text{val}[i]
\]

emit(nodeID, \( w \))
Bellman-Ford

reduce(nodeID, val[...])

Let $w$ be the “node” in the array $val[...]$.

for $i = 1$ to $|val|

if $val[i]$ is not a “node”

if $w.est > val[i]$ then $w.est = val[i]$

emit(nodeID, $w$)

Note: assumes we can distinguish the two different types of (key, value) pairs.
Bellman-Ford

reduce(nodeID, val[...])

Let $w$ be the “node” in the array val[...].

for $i = 1$ to $|val|$

    if val[i] is not a “node”

        if $w$.est > val[i] then $w$.est = val[i]

emit(nodeID, $w$)

Each node “receives” possible estimates from all of its neighbors.

It chooses the minimum possible estimate among them.
Bellman-Ford

reduce(nodeID, val[...])

Let \( w \) be the “node” in the array \( \text{val}[...] \).

for \( i = 1 \) to \( \mid \text{val} \mid \)

    if \( \text{val}[i] \) is not a “node”
        if \( w.\text{est} > \text{val}[i] \) then \( w.\text{est} = \text{val}[i] \)

emit(nodeID, \( w \))

At the end, it re-outputs the node.
Bellman-Ford

\[ \text{reduce}(\text{nodeID}, \text{val}[,\ldots]) \]

Let \( w \) be the “node” in the array \( \text{val}[,\ldots] \).

\[ \text{for } i = 1 \text{ to } |\text{val}| \]

\[ \text{if } \text{val}[i] \text{ is not a “node”} \]

\[ \text{if } w.\text{est} > \text{val}[i] \text{ then } w.\text{est} = \text{val}[i] \]

\[ \text{emit}(\text{nodeID}, w) \]

What if the degree is large?
Bellman-Ford

reduce(nodeID, val[...])

Let \( w \) be the “node” in the array \( \text{val}[...] \).

\[
\text{for } i = 1 \text{ to } |\text{val}|
\]

if \( \text{val}[i] \) is not a “node”

\[
\text{if } w.\text{est} > \text{val}[i] \text{ then } w.\text{est} = \text{val}[i]
\]

emit(nodeID, \( w \))

What if the degree is large?

The val array will be too large! Is it associative?
Bellman-Ford

\[
\text{reduce}(\text{nodeID}, \text{val}[\ldots])
\]

Let \( w \) be the “node” in the array \( \text{val}[\ldots] \).

\[
\text{for } i = 1 \text{ to } |\text{val}|
\]

\[
\text{if } \text{val}[i] \text{ is not a “node”}
\]

\[
\text{if } w.\text{est} > \text{val}[i] \text{ then } w.\text{est} = \text{val}[i]
\]

\[
\text{emit}(\text{nodeID}, w)
\]

What if the degree is large?

The \( \text{val} \) array will be too large! Is it associative? No!

But can handle streams of edges, if the “node” key is first.
Bellman-Ford: one iteration

map(nodeID, u)

  emit(nodeID, u)

  for i = 1 to |u.nbrIDs|
      emit(u.nbrID[i], u.est+u.nbrWeight[i])

reduce(nodeID, val[...])

  Let w be the “node” in the array val[...].
  for i = 1 to |val|
      if val[i] is not a “node”
          if w.est > val[i] then w.est = val[i]
  emit(nodeID, w)
Bellman-Ford

How many iterations?
Bellman-Ford

Simple version: $n$ iterations

Running time: $n$ Map-Reduce steps.
Bellman-Ford

Better version: stop early

Can stop if no estimates change during one iteration.

*Exercise*: design a “termination detection” step.
Bellman-Ford

With termination detection

Running time: 2D Map-Reduce steps

\[ D = \text{diameter of the graph} \]

*Is this any good?*
Map-Reduce and PageRank
Map-Reduce and PageRank

Goal:

- graph $G = (V,E)$
- PageRank assigns a value to each node in the graph
Map-Reduce and PageRank

Goal:

- graph $G = (V,E)$
- PageRank assigns a value to each node in the graph

$\text{PageRank}(v) =$ probability that a random walks ends at node $v$. 
Map-Reduce and PageRank

PageRank(G)

Choose a random node $v$ (uniformly) from $G$
Repeat many times:

1. With probability $\frac{1}{2}$: stay at node $v$.
2. With probability $\frac{1}{2}$: choose a neighbor of $v$ uniformly at random and go to that neighbor.

Assign to each node $u$ the probability that you are at node $u$ when the process terminates.
Map-Reduce and PageRank

Goal:

- graph $G = (V,E)$
- PageRank assigns a value to each node in the graph

$\text{PageRank}(v) = \text{probability that a random walks ends at node } v.$
Map-Reduce and PageRank

Goal:
  – graph $G = (V,E)$
  – PageRank assigns a value to each node in the graph

PageRank$\left( v \right) = \text{probability that a random walks ends at node } v.$

Several equivalent formulations (e.g., related to the second eigenvalue of the Laplacian/adjacency matrix).
Map-Reduce and PageRank

Goal:

- graph $G = (V,E)$
- PageRank assigns a value to each node in the graph

$\text{PageRank}(v) = \text{probability that a random walks ends at node } v.$

Several equivalent formulations (e.g., related to the second eigenvalue of the Laplacian/adjacency matrix).
Inductive calculation:

• Assume we have already calculated the probability distribution after \( t \) steps of the random walk.

• Compute the distribution after step \((t+1)\).

Notation:

\[ p(v)_t = \text{probability random walk is at } v \text{ after step } t \]
Initially, uniform distribution:

\[ p(v)_0 = \frac{1}{n} \]
PageRank

Initially, uniform distribution:

\[ p(v)_0 = \frac{1}{n} \]

Iterative computation:

\[ p(v)_{t+1} = \frac{1}{2} p(v)_t + \frac{1}{2} \sum_{u \in v.nbrs} \frac{p(u)_t}{|v.nbrs|} \]

- Probability \( \frac{1}{2} \), stay at node \( v \)
- Probability \( \frac{1}{2} \), used to be at node \( u \) and chose to come to \( v \).
PageRank

PageRank(G)

Initialize, for all $v$: $p(v)_0 = 1/n$

Repeat many times:

For all $v$ do:

\[ p(v)_{t+1} = \frac{1}{2} p(v)_t + \frac{1}{2} \sum_{u \in v.nbrs} \frac{p(u)_t}{|v.nbrs|} \]
PageRank

\[
\text{map}(\text{nodeID}, u) \\
\text{emit}(\text{nodeID}, u) \\
\text{for } i = 1 \text{ to } |u.\text{nbrIDs}| \\
\text{emit}(u.\text{nbrID}[i], u.\text{est}/|u.\text{nbrID}|)
\]

Estimate est stores probability random walk is at u.

Send critical info to nbrs.

\[
p(v)_{t+1} = \frac{1}{2} p(v)_t + \frac{1}{2} \sum_{u \in v.\text{nbrs}} \frac{p(u)_t}{|v.\text{nbrs}|}
\]
PageRank

reduce(nodeID, val[...])

Let \( w \) be the “node” in the array \( \text{val[...]} \).

\[
\text{sum} = 0
\]

for \( i = 1 \) to \(|\text{val}|\)

if \( \text{val}[i] \) is not a “node”

if \( \text{w.est} > \text{val}[i] \) then

\[
\text{sum} = \text{sum} + \text{val}[i]
\]

\( \text{w.est} = (1/2)\text{w.est} + (1/2)\text{sum} \)

emit(nodeID, \( w \))
PageRank

Let \( w \) be the “node” in the array \( \text{val}[\ldots] \).

\[
p(v)_{t+1} = \frac{1}{2}p(v)_t + \frac{1}{2} \sum_{u \in v.nbrs} \frac{p(u)_t}{|v.nbrs|}
\]

\[
\text{sum} = 0
\]

\[
\text{for } i = 1 \text{ to } |\text{val}|
\]

\[
\text{if } \text{val}[i] \text{ is not a “node”}
\]

\[
\text{if } w.\text{est} > \text{val}[i] \text{ then}
\]

\[
\text{sum} = \text{sum} + \text{val}[i]
\]

\[
w.\text{est} = (1/2)w.\text{est} + (1/2)\text{sum}
\]

\[
\text{emit(\text{nodeID, w})}
\]
PageRank

Conclusion:

After (enough) iterations, the estimates are equal to the PageRank of the nodes in the graph.
Conclusion:

After (enough) iterations, the estimates are equal to the PageRank of the nodes in the graph. Depends on the mixing time of the graph.

- For random graphs, $O(\log n)$ steps.
- For worst-case graphs, $O(n^3)$ steps.
- For cliques, $O(\log n)$ steps.
Map-Reduce

Discussion:

Is this a good framework for building high-performance cluster computing solutions?

Pros:
– It has been very successful (e.g., at Google).
– There exist (pretty) good implementations.

Cons:
– Other frameworks may be easier today.
– E.g., SPARK…
– Better for some types of problems than others.
Map-Reduce

Discussion:

Is this a good way to design parallel algorithms?

Pros:
– Simple model of parallelism.
– Easy to analyze, to think about.

Cons:
– Tedious to carefully move data around.
– Does not really capture the costs of data management. (See: sorting example.)
– Not easy to adjust parallelism (e.g., high-degree nodes)
Summary

Today: Map-Reduce

Map-Reduce Model
- Cluster computing

Some simple examples
- Word count
- Join

Algorithms
- Bellman-Ford
- PageRank

Last Week: Multicore

Models of Parallelism
- Fork-Join model
- Work and Span
- Greedy schedulers

Algorithms
- Sum
- MergeSort
- Parallel Sets
- BFS
- Prefix-Sum
- (Luby’s)
Design Some Algorithms

Design Map-Reduce algorithms for:

- BFS (Breadth-First-Search)
- Lubys (Maximal Independent Set)
- Prefix-Sum

A little more:

Can you design a Map-Reduce algorithm for Bellman-Ford where key/value pairs are small (i.e., do not contain adjacency lists) and all functions are associative or streamable?

How would you add termination detection to Bellman-Ford?

Can you design an MIS algorithm? (Next week...)

What about Dijkstra’s? (Open...)

Design a k-median or an (iterative) k-means clustering algorithm for Map-Reduce.
Map-Reduce

Discussion:

Is this a good way to design parallel algorithms?

Pros:
– Simple model of parallelism.
– Easy to analyze, to think about.

Cons:
– Tedious to carefully move data around.
– Does not really capture the costs of data management. (See: sorting example.)
– Not easy to adjust parallelism (e.g., high-degree nodes)