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

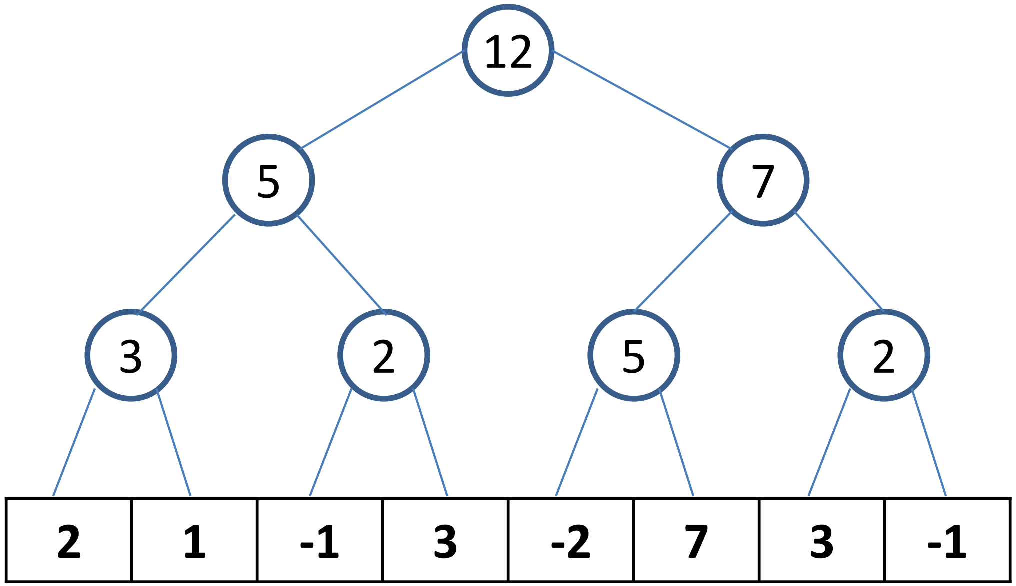
Today:

MiniProject explanatory section due today.

Next week:

MiniProject talk due

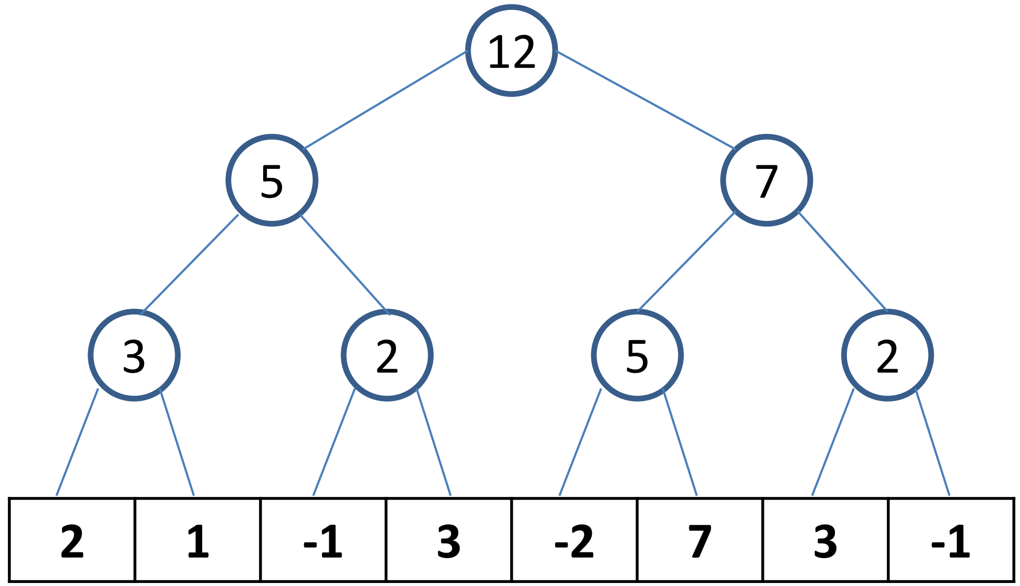
Recap: Prefix Sum



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

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 $s_1 + s_2$

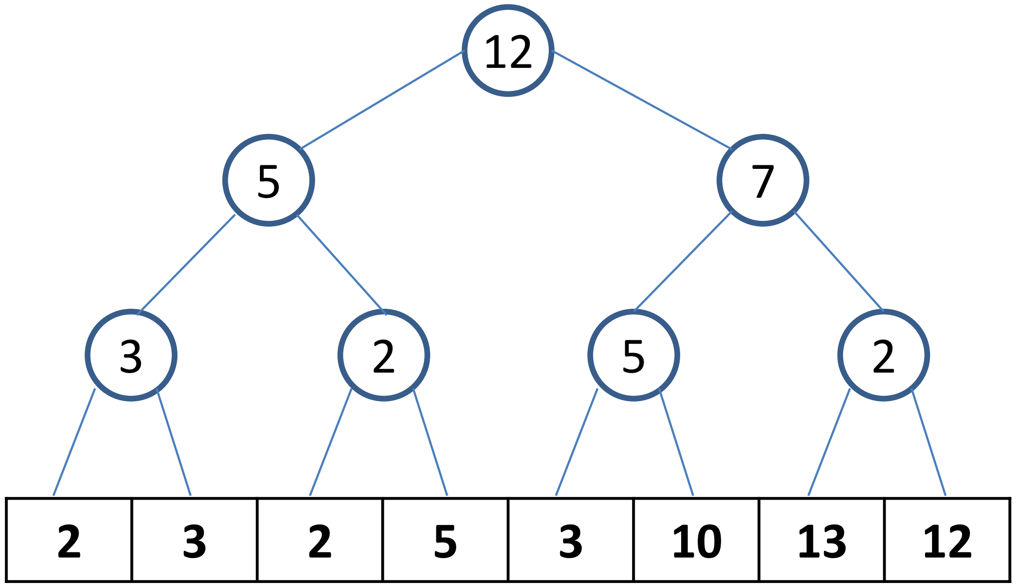
Recap: Prefix Sum



Algorithm 3: $\textsc{PrefixSumPartTwo}(A, level, sum, begin, end)$

if $(begin = end)$ **then**
    $A[begin] = sum + A[begin]$
else
    $mid = (begin + end)/2$
    in parallel
        (1) $\textsc{PrefixSumPartTwo}(A, level - 1, sum, begin, mid)$
        (2) $\textsc{PrefixSumPartTwo}(A, level - 1, sum + S[level, end, left], mid + 1, end)$
    return

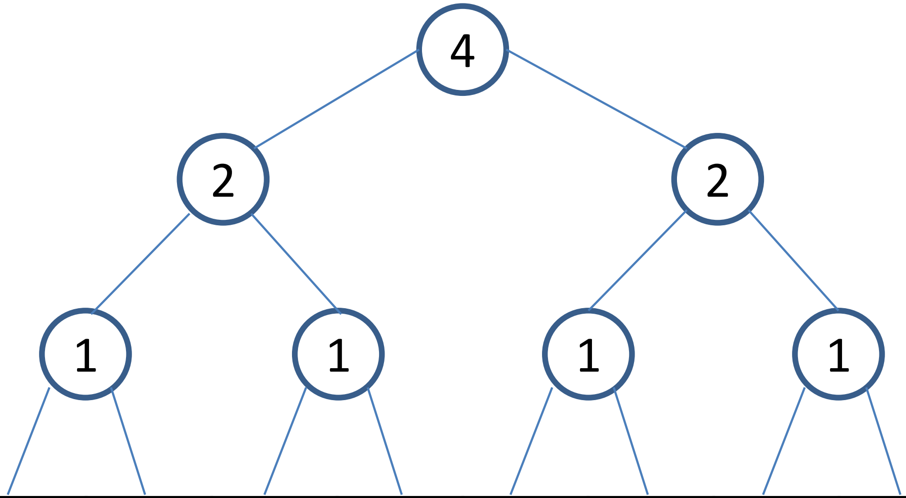
Recap: Prefix Sum



Algorithm 3: $\textsc{PrefixSumPartTwo}(A, level, sum, begin, end)$

```
if (begin = end) then
    A[begin] = sum + A[begin]
else
    mid = (begin + end)/2
    in parallel
        (1) PrefixSumPartTwo(A, level − 1, sum, begin, mid)
        (2) PrefixSumPartTwo(A, level − 1, sum + S[level, end, left], mid + 1, end)
    return
```

Recap: Binary Prefix Sum



0	**1**	**1**	**0**	**0**	**1**	**0**	**1**

⇓

0	**1**	**2**	**0**	**0**	**3**	**0**	**4**

<u>binary prefix sum</u>

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

Recap: Partition

Goal: partition array around key k

Example: k = 4

7	9	3	2	5	8	4	2

3	2	2	4	7	9	5	8

Recap: Partition

Step 1: mark items < k

Example: k = 4

7	**9**	**3**	**2**	**3**	**8**	**4**	**2**

⇓

7	**9**	**3**	**2**	**3**	**8**	**4**	**2**
0	0	1	1	1	0	0	1

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

Recap: Partition

Step 2: prefix sums

Example: k = 4

7	**9**	**3**	**2**	**3**	**8**	**4**	**2**

⇓

7	**9**	**3**	**2**	**3**	**8**	**4**	**2**
0	0	1	1	1	0	0	1
0	0	1	2	3	0	0	4

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

Recap: Partition

Step 2: prefix sums

Example: k = 4

7	**9**	**3**	**2**	**3**	**8**	**4**	**2**

⇓

7	**9**	**3**	**2**	**3**	**8**	**4**	**2**
0	0	1	2	3	0	0	4

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

Recap: Partition

Step 3: mark items ≥ k

Example: k = 4

7	**9**	**3**	**2**	**3**	**8**	**4**	**2**

⇩

7	**9**	**3**	**2**	**3**	**8**	**4**	**2**
0	0	1	2	3	0	0	4
1	1	0	0	0	1	1	0

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

Recap: Partition

Step 4: prefix sum

Example: k = 4

7	**9**	**3**	**2**	**3**	**8**	**4**	**2**

⇓

7	**9**	**3**	**2**	**3**	**8**	**4**	**2**
0	0	1	2	3	0	0	4
1	1	0	0	0	1	1	0
1	2	0	0	0	3	4	0

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

Recap: Partition

Step 4: prefix sum

Example: k = 4

7	**9**	**3**	**2**	**3**	**8**	**4**	**2**

7	**9**	**3**	**2**	**3**	**8**	**4**	**2**
0	0	1	2	3	0	0	4
1	2	0	0	0	3	4	0

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

Recap: Partition

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

Example: k = 4

7	**9**	**3**	**2**	**3**	**8**	**4**	**2**

⇓

7	**9**	**3**	**2**	**3**	**8**	**4**	**2**
0	0	1	2	3	0	0	4
1	2	0	0	0	3	4	0
5	6	0	0	0	7	8	0

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

Recap: Partition

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

Example: k = 4

7	**9**	**3**	**2**	**3**	**8**	**4**	**2**

⇓

7	**9**	**3**	**2**	**3**	**8**	**4**	**2**
0	0	1	2	3	0	0	4
5	6	0	0	0	7	8	0

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

Recap: Partition

Step 6: compress

Example: k = 4

7	**9**	**3**	**2**	**3**	**8**	**4**	**2**

⇓

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

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

Recap: Partition

Step 6: compress

Example: k = 4

7	**9**	**3**	**2**	**3**	**8**	**4**	**2**

⇩

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

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

Recap: Partition

Step 6: copy to final location

Example: k = 4

7	**9**	**3**	**2**	**3**	**8**	**4**	**2**

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

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

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

Recap: Partition

Partition around k:

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

Example: k = 4

7	**9**	**3**	**2**	**3**	**8**	**4**	**2**

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.

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: QuickSort

```
QuickSort(A, begin, end)
    pivot = random(begin, end)
    split = partition(A, begin, end, pivot)
    mid = (begin+end)/2
    in parallel:
    1.  QuickSort(A, begin, mid)
    2.  QuickSort(A, mid+1, end)
```

Recap: QuickSort Work

```
QuickSort(A, begin, end)
    pivot = random(begin, end)                  O(1)
    split = partition(A, begin, end, pivot)     O(n)
    mid = (begin+end)/2                         O(1)
    in parallel:
    1. QuickSort(A, begin, mid)
    2. QuickSort(A, mid+1, end)                 2W(n/2)
```

$$W(n) = 2W(n/2) + O(n) = O(n \log n)$$

** 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)                      O(1)
    split = partition(A, begin, end, pivot)         O(log n)
    mid = (begin+end)/2                             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)
    pivot = random(begin, end)                    O(1)
    split = partition(A, begin, end, pivot)       O(log n)
    mid = (begin+end)/2                           O(1)
    in parallel:
    1. QuickSort(A, begin, mid)                   S(n/2)
    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

Good model for multicore / multithreaded CPUs.

Advantages:

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

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)
- 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)

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

→ 50,400 cores

High Performance Clusters

DataBricks TeraSort:

- 206 nodes
- 6,592 cores

Record (2014):

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

DataBricks PetaSort:

- 190 nodes
- 6,080 cores

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:

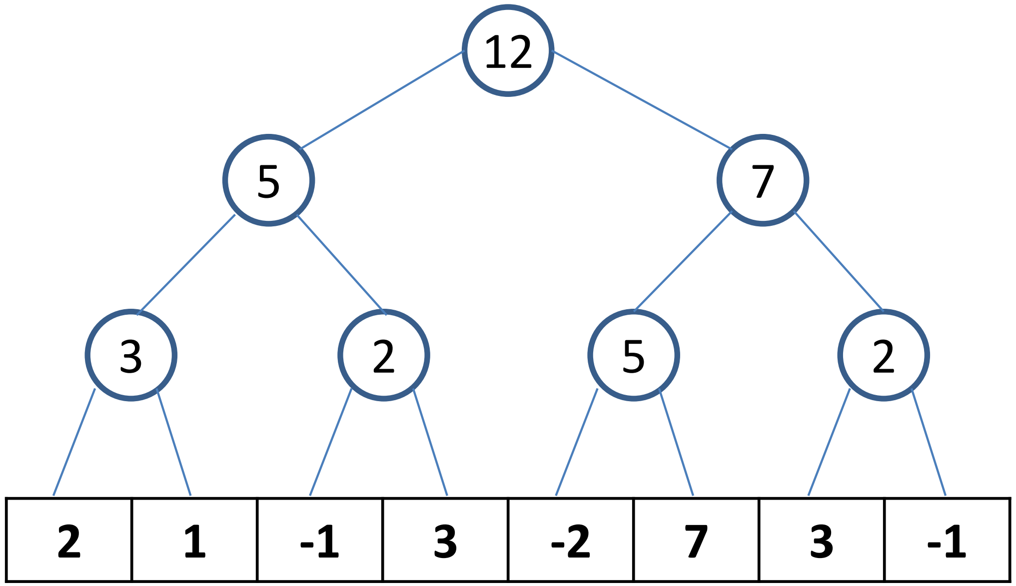
- Loosely synchronized
- No shared memory
- Data exchanged over fast interconnect

Fork/Join is not a good model for clusters.

Issues:

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

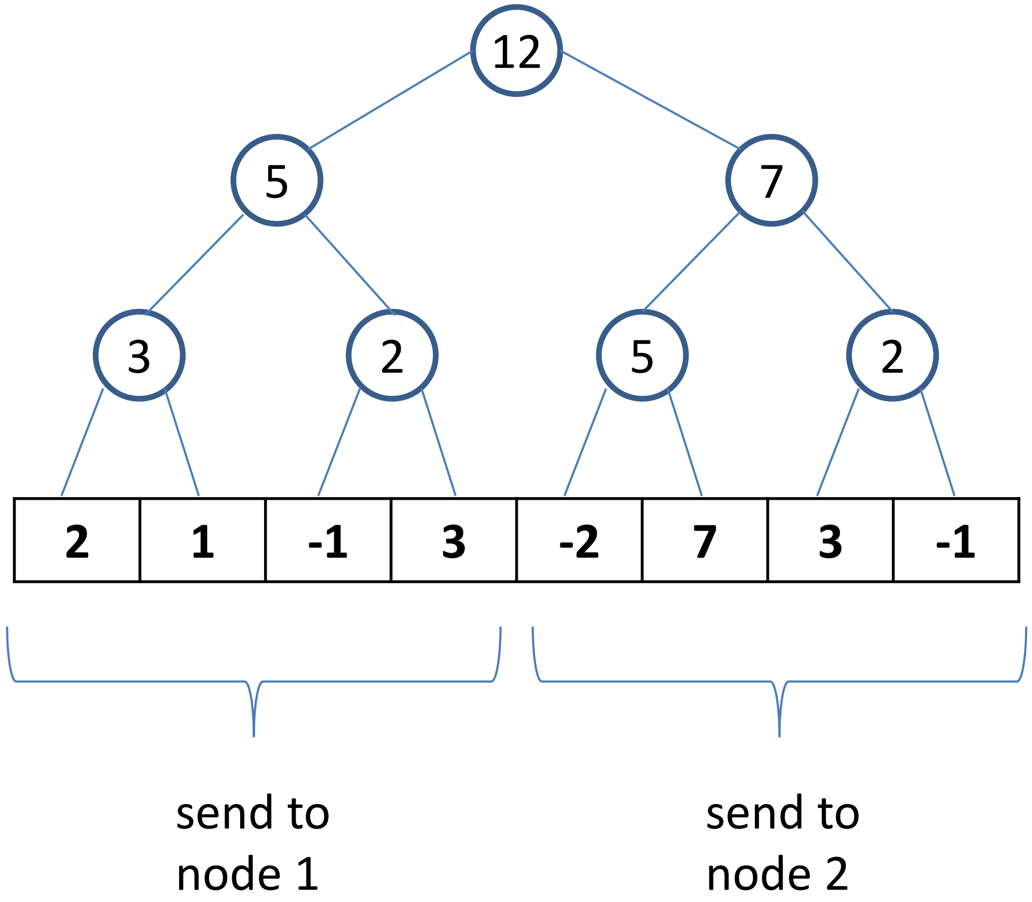
Recap: Prefix Sum



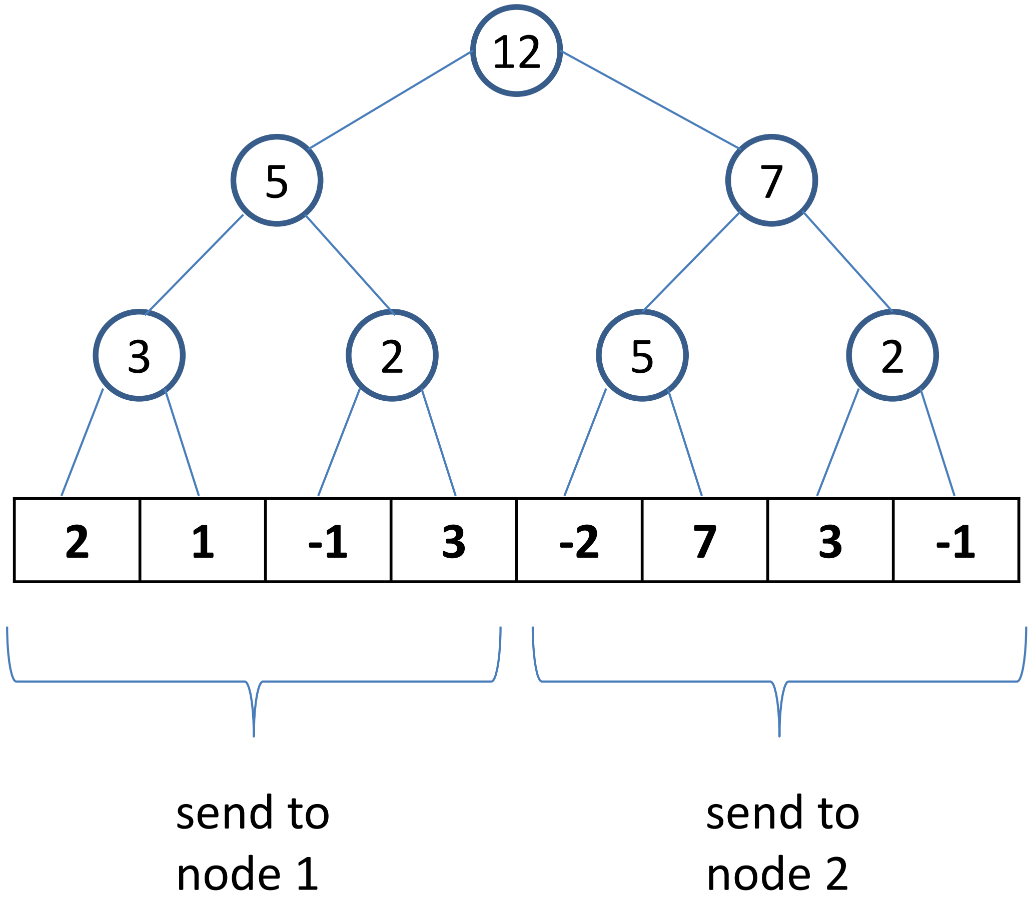
Algorithm 2: $\textsc{PrefixSumPartOne}(A, level, begin, end)$

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 $s_1 + s_2$

Recap: Prefix Sum



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

Fork/Join is not a good model for clusters.

Issues:

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

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.

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.

Map-Reduce Example

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

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

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

(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)

$\Downarrow$

(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)

$\Downarrow$

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

Map-Reduce Example

reduce(key, [v_1, v_2, …]) ➔ (key, value) pair(s)

Step 3:

```
reduce(key, V[…])
    sum = 0
    for (j = 1 to |V|)
        sum = sum + V[j]
    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

```
reduce(key, V[…])
    sum = 0
    for (j = 1 to |V|)
        sum = sum + V[j]
    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)

$\Downarrow$

(1, 26), (2, 40)

Map-Reduce Example

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

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

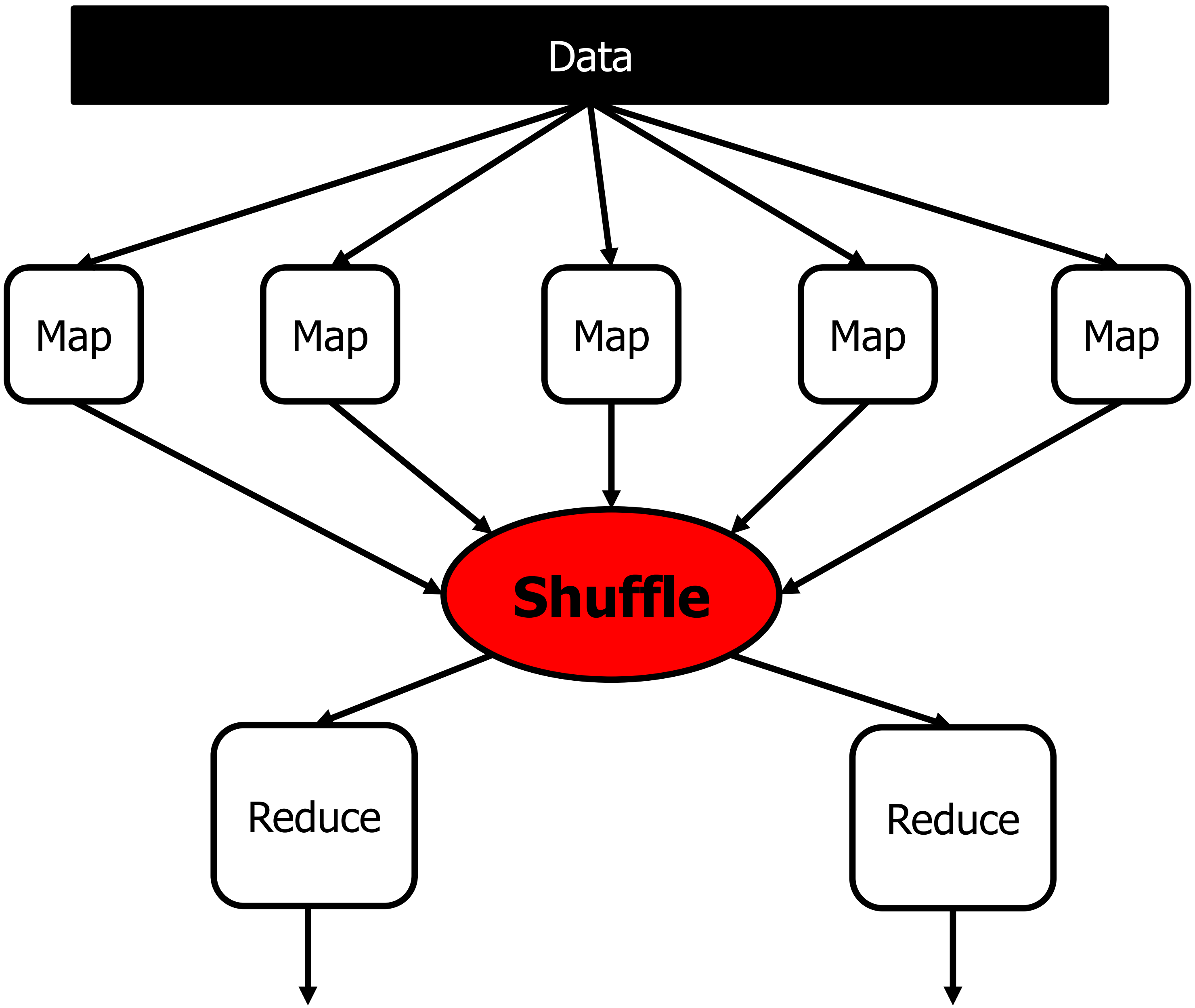
Step 5: Write back to disk

$$\boxed{(1, 26), (2, 40)}$$

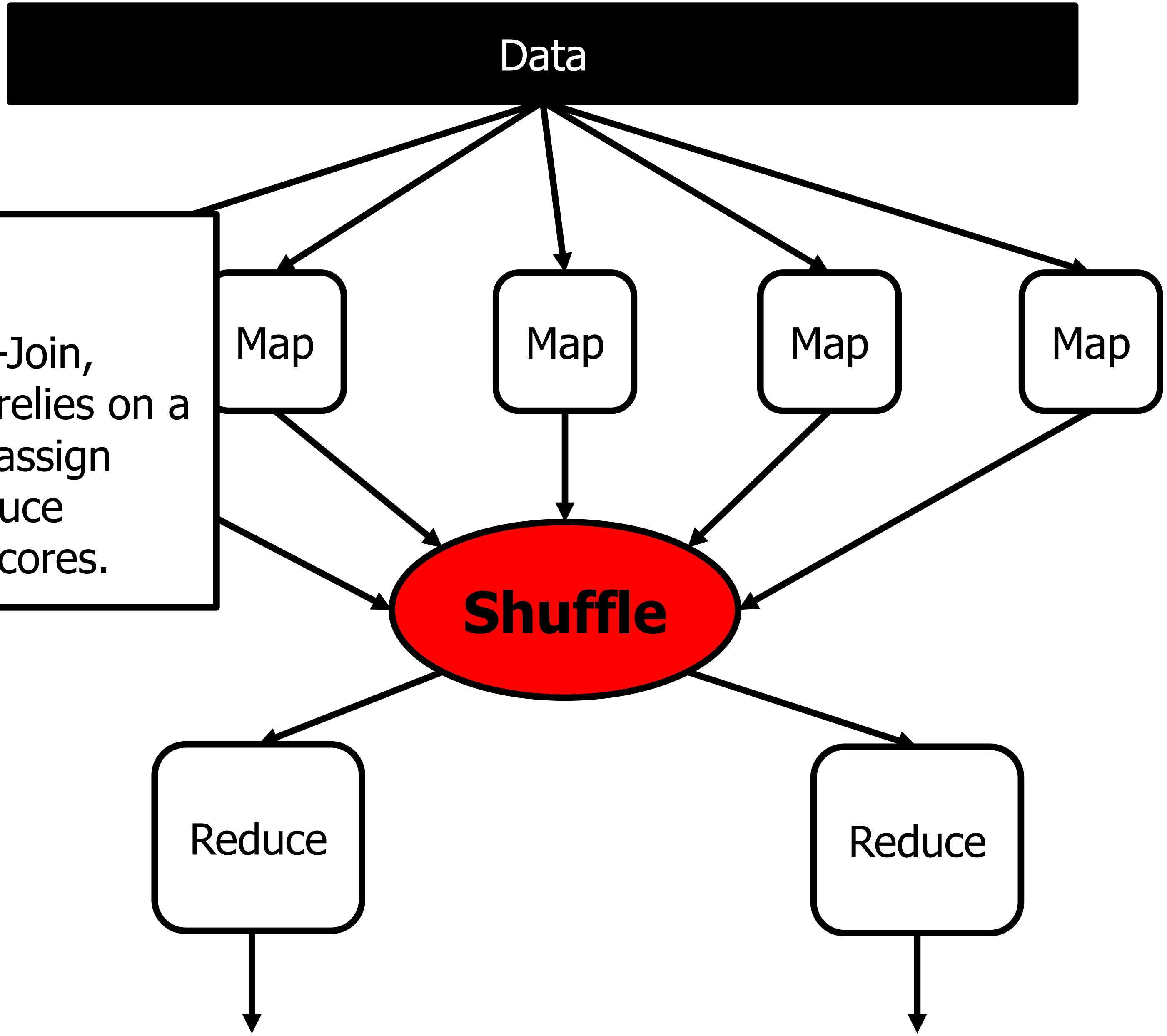
$$\Downarrow$$

Out = [26, 40]

Map-Reduce Schematic



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

analogous to span

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.

Unrestricted Map-Reduce

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.

(Real) Map-Reduce

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.)

(Real) Map-Reduce

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 $O(n^{\varepsilon})$ pairs.
- If the data is initially size n, no map or reduce process should emit more than $O(n^{\varepsilon})$ 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(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

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 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

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

Notes:

- File is translated into (key, value) pairs.
- Using a string as a key.

Example 1: Word Count

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

Notes:

- File is translated into (key, value) pairs.
- Using a string as a key.
- Assumes a hash function translates string to integer.

Example 1: Word Count

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

Example 1: Word Count

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

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

Size is not sublinear!

Example 1: Word Count

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

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

("gaa", 1), ("gaa", 1), ("gaa", 1), ("gaa", 1)

("gaa", 2), ("gaa", 2)

Example 1: Word Count

```
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

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

```
reduce(word, count[…])
    sum = 0
    for (i=1 to |count|)
        sum = sum + count[i]
    emit(word, 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 = $(x_1, y_1), (x_2, y_2), (x_3, y_3),$
- 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

```
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

```
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

```
reduce(key, values[…])
    if BVALUE in values
        for j = 1 to |values|
            if values[j] != BVALUE then
                emit(key, 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, …)
    if BVALUE = v1
        for each vj
            if vj != BVALUE then
                emit(key, vj)
```

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 $A = [x_1, x_2, x_3, x_4, x_5, x_6, \ldots]$

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

Is your Map-Reduce framework any good?

How fast can it sort?

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

Map-Reduce and Graphs

Single-Source Shortest Paths

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

Output:

For each vertex v: distance d(v) from the source.

Map-Reduce and Graphs

Bellman-Ford

```
BF(V, E, s, w)
    s.est = 0
    for each node u: u.est = ∞
    repeat |V| times:
        for each node u:
            for each neighbor v of u:
                if v.est > u.est + w(u,v)
                    v.est = u.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 th ekey.

Bellman-Ford

What keys should we use?

- Each node has a nodeID.
- Use nodeID as th ekey.

What should the value be?

- nodeID
- est
- nbrIDs = $[x_1, x_2, \ldots]$
- nbrWeights = $[w_1, w_2, \ldots]$

Distributed version of adjacency list!

Bellman-Ford

What keys should we use?

- Each node has a nodeID.
- Use nodeID as th ekey.

What should the value be?

- nodeID
- est
- nbrIDs = [x_1, x_2, …]
- nbrWeights = [w_1, w_2, …]

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 th ekey.

What should the value be?

- nodeID
- est
- nbrID = $[x_1, x_2, \ldots]$
- nbrWeight = $[w_1, w_2, \ldots]$

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

```
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

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 val[i] is not a "node"

if *w*.est > val[i] **then** *w*.est = 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 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)
```

At the end, it re-outputs the node.

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)

What if the degree is large?

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)
```

What if the degree is large?

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

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)
```

What if the degree is large?

The 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 = 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

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 ½: stay at node v.
2. With probability ½: 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

PageRank(v) = 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(v) = 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

PageRank(v) = probability that a random walks ends at node v.

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

PageRank

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:

$p(v)_t = \text{probability random walk is at v after step t}$

PageRank

Initially, uniform distribution:

$$p(v)_0 = 1/n$$

PageRank

Initially, uniform distribution:

$$p(v)_0 = 1/n$$

probability ½, used to be at node u
and chose to come to v.

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 ½, stay at node 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

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

Estimate est stores probability random walk is at u.

probability that random walk is at u and goes to u.nbrID[i]

Send critical info to nbrs.

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

PageRank

reduce(nodeID, val[…])

Let w be the "node" in the array val[…].

sum = 0

for i = 1 **to** |val|

if val[i] is not a "node"

if w.est > val[i] **then**

sum = sum + val[i]

w.est = (1/2)w.est + (1/2)sum

emit(nodeID, w)

PageRank

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

reduce(nodeID

Let w be the "node" in the array val[…].

sum = 0

for i = 1 **to** |val|

  if val[i] is not a "node"

    if w.est > val[i] **then**

      sum = sum + val[i]

w.est = (1/2)w.est + (1/2)sum

emit(nodeID, w)

PageRank

Conclusion:

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

PageRank

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

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

What about Dijkstra's? (Open…)

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?

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)