Data Parallelism

Multiprocessor Synchronization
CSCI 176

Lecture 23
29 November 2022

Maurice Herlihy
Brown University
Ever Wonder ...

When did the term *multicore* become popular?

“A multi-core processor is a single computing component with two or more independent actual central processing units, which are the units that read and execute program instructions.”

*Wikipedia*
Let’s Ask Google Ngram!

usage of *multicore* in books by publication year
Let’s Ask Google Ngram!

This part since 2000 is obvious …
Let's Ask Google Ngram!
Let’s Ask Google Ngram!

multicore cable
multicore fiber
...

but we digress ...

Art of Multiprocessor Programming
WordCount

easy to do sequentially …
what about in parallel?

alpha → 8
bravo → 3
charlie → 9
…
zulu → 1
MapReduce

split text among *mapping* threads

chapter 1
chapter 2
chapter k

Art of Multiprocessor Programming
Map Phase

must count words!

must count words!

must count words!

a mapping thread per chapter

chapter 1

chapter 2

... chapter k
Map Phase

each mapper thread produces a stream of key-value pairs…

key: word
value: local count

alpha → 9
juliet → 2,
alpha → 1
tango → 4
abstract class Mapper<IN, K, V> extends RecursiveTask<Map<K, V>> {
    IN input;
    public void setInput(IN anInput) {
        input = anInput;
    }
}
abstract class Mapper<IN, K, V> extends RecursiveTask<Map<K, V>> {
    IN input;
    public void setInput(IN anInput) {
        input = anInput;
    }
}
abstract class Mapper<IN, K, V> extends RecursiveTask<Map<K, V>> {
    IN input;
    public void setInput(IN anInput) {
        input = anInput;
    }
}
abstract class Mapper<IN, K, V> extends RecursiveTask<Map<K, V>> {
    IN input;
    public void setInput(IN anInput) {
        input = anInput;
    }
}

value: local count
abstract class Mapper<IN, K, V> extends RecursiveTask<Map<K, V>> {
  IN input;
  public void setInput(IN anInput) {
    input = anInput;
  }
}

a task that runs in parallel with other tasks
abstract class Mapper<IN, K, V>
    extends RecursiveTask<Map<K, V>> {
    IN input;
    public void setInput(IN anInput) {
        input = anInput;
    }
}
Mapper Class

abstract class Mapper<IN, K, V> extends RecursiveTask<Map<K, V>> {
    IN input;
    public void setInput(IN anInput) {
        input = anInput;
    }
}

initialize input: which document fragment?
class WordCountMapper extends mapreduce.Mapper<
    List<String>, String, Long
> {

    ...

}
class WordCountMapper extends mapreduce.Mapper<List<String>, String, Long> {
    ...
}

document fragment is list of words
WordCount Mapper

class WordCountMapper extends mapreduce.Mapper{
  List<String>, String, Long
  ...
  document fragment is list of words
  map each word ...
}
class WordCountMapper extends mapreduce.Mapper{
  List<String>, String, Long

  document fragment is list of words

  map each word ...

  to its count in the fragment
WordCount Mapper

```java
Map<String, Long> compute() {
    Map<String, Long> map = new HashMap<>();
    for (String word : input) {
        map.merge(word,
                   1L,
                   (x, y) -> x + y);
    }
    return map;
}
```
WordCount Mapper

```java
Map<String, Long> compute() {
    Map<String, Long> map = new HashMap<>();
    for (String word : input) {
        map.merge(word, 1L, (x, y) -> x + y);
    }
    return map;
}
```

the compute() method constructs the local word count
WordCount Mapper

```java
Map<String, Long> compute() {
    Map<String, Long> map = new HashMap<>();
    for (String word : input) {
        map.merge(word,
                   1L,
                   (x, y) -> x + y);
    }
    return map;
}
```

create a map to hold the output
Map<String, Long> compute() {
    Map<String, Long> map = new HashMap<>();
    for (String word : input) {
        map.merge(word, 1L, (x, y) -> x + y);
    }
    return map;
}

WordCount Mapper

examine each word in the document fragment
Map<String, Long> compute() {
    Map<String, Long> map = new HashMap<>();
    for (String word : input) {
        map.merge(word, 1L, (x, y) -> x + y);
    }
    return map;
}
Map<String, Long> compute() {
    Map<String, Long> map = new HashMap<>();
    for (String word : input) {
        map.merge(word, 1L, (x, y) -> x + y);
    }
    return map;
}

when the local count is complete, return the map
Reduce Phase

alpha → 4
bravo → 2
... 
zulu → 1

A reducer thread merges mapper outputs

alpha → 2
juliet → 1
tango → 1
...

alpha → 1
foxtrot → 1
papa → 1
tango → 1
...

alpha → 1
oscar → 1,
bravo → 2...
the reducer task produces a stream of *key-value* pairs...

key: word  value: word count
abstract class Reducer<K, V, OUT> extends RecursiveTask<OUT> {
    K key;
    List<V> valueList;
    public void setInput(K aKey, List<V> aList) {
        key = aKey;
        valueList = aList;
    }
}
abstract class Reducer<K, V, OUT> extends RecursiveTask<OUT> {
    K key;
    List<V> valueList;
    public void setInput(K aKey, List<V> aList) {
        key = aKey;
        valueList = aList;
    }
}
abstract class Reducer<K, V, OUT> extends RecursiveTask<OUT> {
    K key;
    List<V> valueList;
    public void setInput(K aKey, List<V> aList) {
        key = aKey;
        valueList = aList;
    }
}

and a list of associated values (word count per fragment)
abstract class Reducer<K, V, OUT> extends RecursiveTask<OUT> {
    K key;
    List<V> valueList;
    public void setInput(K aKey, List<V> aList) {
        key = aKey;
        valueList = aList;
    }
}

It produces a single summary value (the total count for that word)
normalizing document wordcount gives a *fingerprint* vector
Document Fingerprint

a fingerprint is a point in a high-dimensional space
Clustering

- romance novels
- tango lyrics
- CS textbooks
k-means

Find $k$ clusters from raw data

each point closer to those in same cluster …

than in different clusters.
MapReduce

split points among *mapping* threads

thread 1

thread 2

... 

thread k
k-means

Reducer picks *k “centers”* at random
Reduce Phase

0 \rightarrow c_0
1 \rightarrow c_1
2 \rightarrow c_2

reducer sends key-value pair to mappers

key: cluster number
value: center point

thread 1
thread 2
\ldots
thread k
Each mapper uses centers to assign each point to a cluster.
Each mapper uses centers to assign each point to a cluster.
Mappers

mapper sends key-value stream to reducer

key: point
value: cluster ID

\[ p_0 \rightarrow 2 \]
\[ p_1 \rightarrow 1 \]
\[ p_2 \rightarrow 1 \]
\[ p_3 \rightarrow 0 \]
Back at the Reducer

The reducer merges the streams .... and assembles clusters ...
Back at the Reducer

The reducer computes new centers based on new clusters …

$0 \rightarrow c_0'$

$1 \rightarrow c_1'$

$2 \rightarrow c_2'$
Once is Not Enough

Reducer sends *new centers* to mappers

Process ends when centers become stable

thread 1

thread 2

...  

thread k

0 \rightarrow c_0', 
1 \rightarrow c_1', 
2 \rightarrow c_2'
To Recapitulate

We saw two problems …

`wordcount` & `k-means` …

with similar parallel solutions

Map part is parallel …

Reduce part is sequential.
abstraction
Map Function

$(k_1, v_1) \rightarrow list(k_2, v_2)$

doc, contents

cluster ID, center

word, count

point, cluster ID
Reduce Function

\((k_2, \text{list}(v_2)) \rightarrow \text{list}(v_2)\)

- word, list of counts
- cluster ID, list of points
- count
- new cluster center
Example

Distributed Grep

Map:
line of document

Reduce:
copy line to display
Example

URL Access Frequency

Map:

(URL, local count)

Reduce:

(URL, total count)
Example

Reverse web link graph

Map:
(target link, source page)

Reduce:
(target link, list of source pages)
Other Examples

histogram

matrix multiplication

PageRank

Betweenness centrality
Distributed MapReduce

Google, Hadoop, etc...

Communication by message

Fault-tolerance important

Figure 1: Execution overview
Multicore MapReduce

Phoenix, Phoenix++, Metis ...

Communication by shared memory objects

Fault-tolerance unimportant
Costs

- key-value layout
- cache pressure
- memory allocation
- static vs dynamic
- mechanism overhead
Part Two

Data Streams
Streams

- source
- data
- transformation
- data
- transformation
- data
- consumer

- sequence of transformations
- sometimes in parallel
- no relation to I/O streams
Streams

source
data
transformation
data
transformation
data
consumer

transformations given by mathematical functions
Streams

- Source
  - Data
  - Transformation creates new stream

- Transformation
  - Data
  - No modifications or side-effects

- Consumer
  - Correctness easier?
Functional Programming

functions map old state to new state

old state never changed

no complex side-effects

elegant, easier proofs of correctness
Oh, Really?

“Functional languages are unnatural to use; but so are knives and forks, diplomatic protocols, double-entry bookkeeping, and a host of other things modern civilization has found useful.”

Jim Morris, 1982
Haiku

esthetically pleasing

only works on those who understand Haiku
Karate

esthetically pleasing

works even on those who do not understand Karate
Jim Morris’s Question

Is functional programming more like Haiku or Karate?

1981: Haiku

Today: Karate
Laziness

No computation until absolutely necessary

1,2,3,...

\( x \rightarrow x+1 \)  
add 1 to each element: no computation

\( x \rightarrow 2x \)  
double each element: no computation

Sum  
\( \sum 2(x_i+1) \) is terminal
Laziness

1, 2, 3, ...

x → x + 1

x → 2x

collect in List

move to container is terminal
Laziness permits optimizations.

\[ x \rightarrow x + 1 \]

\[ x \rightarrow 2x \]

collect in List

\[ x \rightarrow 2(x+1) \]
Laziness

Laziness permits infinite streams ...

Stream<Integer> fib = new FibStream();

1 1 2 3 5 8 12 20 32 ...

Art of Multiprocessor Programming 70
Unbounded Random Stream

```java
Stream<Double> randomDoubleStream() {
    return Stream.generate(
        () -> random.nextDouble()
    );
}
```
Unbounded Random Stream

```java
Stream<Double> randomDoubleStream() {
    return Stream.generate(() -> random.nextDouble());
}
```

Unbounded stream of double-precision random numbers
Random Stream

Stream<Double> randomDoubleStream() {
    return Stream.generate(
        () -> random.nextDouble);
}

Stream that generates new elements on the fly
Random Stream

```java
Stream<Double> randomDoubleStream() {
    return Stream.generate(() -> random.nextDouble());
}
```

- **Function to call when generating new element**
- **Example of Java lambda expression (anon method)**
WordCount

List<String> readFile(String fileName) {
...
    return reader
        .lines()
        .map(String::toLowerCase)
        .flatMap(s ->
            pattern.splitAsStream(s))
        .collect(Collectors.toList());
}
WordCount

```java
List<String> readFile(String fileName) {
    ...
    return reader
        .lines()
        .map(String::toLowerCase)
        .flatMap(s ->
            pattern.splitAsStream(s))
        .collect(Collectors.toList());
}
```

puts each word from the document into a List
List<String> readFile(String fileName) {
    // Open the file, create a FileReader

    return reader
        .lines()
        .map(String::toLowerCase)
        .flatMap(s ->
            open the file, create a FileReader
            .splitAsStream(s))
        .collect(Collectors.toList());
}
WordCount

```
List<Stream<String>> readFile(String fileName) {
...
return reader
.lines()
.map(String::toLowerCase)
.flatMap(s ->
    pattern.splitAsStream(s))
.collect(Collectors.toList());
}
```

how a stream program looks

each line creates a new stream
WordCount

List<String> readFile(String fileName) {

    return reader
        .lines()
        .map(String::toLowerCase)
        .flatMap(s ->
            pattern.splitAsStream(s))
        .collect(Collectors.toList());
}

turn the FileReader into a stream of lines, each line a string
WordCount

```java
List<String> readFile(String fileName) {
    return reader
        .lines()
        .map(String::toLowerCase)
        .flatMap(s ->
            pattern.splitAsStream(s))
        .collect(Collectors.toList());
}
```
WordCount

```java
flatMap replaces one stream element with multiple stream elements
return reader
    .lines()
    .map(String::toLowerCase)
    .flatMap(s ->
        pattern.splitAsStream(s))
    .collect(Collectors.toList());
```
WordCount

collect puts stream elements in a container {
  return reader.lines() .map(String::toLowerCase) .flatMap(s -> pattern.splitAsStream(s)) .collect(Collectors.toList());
}
WordCount

List<String> readFile(String fileName) {
    ...
    return reader
        .lines()
        .map(String::toLowerCase)
        .flatMap(s ->
            pattern.splitAsStream(s))
        .collect(Collectors.toList());
}
Map<String, Long> map = text
  .stream()
  .collect(
    Collectors.groupingBy(
      Function.identity(),
      Collectors.counting()));

now let’s count the words
Map<String, Long> map = text
    .stream()
    .collect(
        Collectors.groupingBy(
            Function.identity(),
            Collectors.counting()));
Map<String, Long> map = text
.stream()
collect(
    Collectors.groupingBy(
        Function.identity(),
        Collectors.counting()));

turn List into a stream
WordCount

Map<String, Long> map = text
    .stream()
    .collect(
        Collectors.groupingBy(
            Function.identity()
            ,
            Collectors.counting()));

put stream into a container (Map) word → count
WordCount

Map<String, Long> map = text
    .stream()
    .collect(
        Collectors.groupingBy(
            Function.identity(),
            Collectors.counting()));

each element’s key is that element
Map<String, Long> map = text
    .stream()
    .collect(
        Collectors.groupingBy(
            Function.identity(),
            Collectors.counting()));

each element’s value is the number of times it appears
Map<String, Long> map = text
    .stream()
    .collect(
        Collectors.groupingBy(
            Function.identity(),
            Collectors.counting()));

WordCount

- No loops
- No conditionals
- No mutable objects
k-Means

class Point {
    Point(double x, double y) {...}
    Point plus(Point other) {...}
    Point scale(double x) {...}
    static Point barycenter(
        List<Point> cluster
    ) {...}
}

Art of Multiprocessor Programming 91
class Point {
    Point(double x, double y) {
    }
    Point plus(Point other) {
    }
    Point scale(double x) {
    }
    static Point barycenter(List<Point> cluster) {
    }
}
BaryCenter
Stream Barycenter

Point barycenter(List<Point> cluster) {
    double numPoints = cluster.size();
    Optional<Point> sum = cluster.stream()
        .reduce(Point::plus);
    return sum.get().scale(1 / numPoints);
}
Point barycenter(List<Point> cluster) {
    double numPoints = cluster.size();
    Optional<Point> sum = cluster.stream()
        .reduce(Point::plus);
    return sum.get().scale(1 / numPoints);
}
Stream Barycenter

Point barycenter(List<Point> cluster) {
    double numPoints = cluster.size();
    Optional<Point> sum = cluster.stream()
        .reduce(Point::plus);
    return sum.get().scale(1 / numPoints);
}

turn List into Stream
Reduce

stream.reduce(+):

() → ∅

(a) → a

(a,b) → a+b

(a,b,c) → (a+b)+c

terminal operation

e tc. ...
Point barycenter(List<Point> cluster){
    double numPoints = cluster.size();
    Optional<Point> sum = cluster.stream()
        .reduce(Point::plus);
    return sum.get().scale(1.0/numPoints);
}

k-Means

Optional because sum might be empty!

sum points in cluster
Point barycenter(List<Point> cluster) {
    double numPoints = cluster.size();
    Optional<Point> sum = cluster.stream().reduce(Point::plus);
    return sum.get().scale(1 / numPoints);
}
List<Point> points = readFile("cluster.dat");

centers =
    randomDistinctCenters(points);

double convergence = 1.0;

while (convergence > EPSILON) {
    ...
}

$k$-Means
List<Point> points = readFile("cluster.dat");

centers = randomDistinctCenters(points);

double convergence = 1.0;

while (convergence > EPSILON) {
    ...
}

read points from file
List<Point> points = readFile("cluster.dat");
centers =
    randomDistinctCenters(points);
double convergence = 1.0;
while (convergence > EPSILON) {
    ...
}

pick random centers
List<Point> points = readFile("cluster.dat");
centers = randomDistinctCenters(points);
double convergence = 1.0;
while (convergence > EPSILON) {
    ...
}
while (convergence > EPSILON) {
    Map<Integer, List<Point>> clusters = points
        .stream()
        .collect(Collectors.groupingBy(p -> closestCenter(centers, p)));
}

turn list of points into a Stream
while (convergence > EPSILON) {
    Map<Integer, List<Point>> clusters = points.stream()
        .collect(Collectors.groupingBy(p -> closestCenter(centers, p)));
}
while (convergence > EPSILON) {
...

Map<Integer, Point> newCenters = clusters
    .entrySet()
    .stream()
    .collect(Collectors.toMap(e -> e.getKey(),
                               e -> Point.barycenter(e.getValue()),
                               (k1, k2) -> k1))
    );

convergence = distance(centers, newCenters);

centers = newCenters;
}
while (convergence > EPSILON) {
  ...
  Map<Integer, Point> newCenters = clusters.entrySet().stream()
    .collect(Collectors.toMap(e -> e.getKey(),
                              e -> Point.barycenter(e.getValue())));
  convergence = distance(centers, newCenters);
  centers = newCenters;
}
while (convergence > EPSILON) {
    ...  
    Map<Integer, Point> newCenters
    = clusters
        .entrySet()
        .stream()
        .collect(
            Collectors.toMap(
                e -> e.getKey(),
                e -> Point.barycenter(e.getValue())
            )
        );
    convergence = distance(centers, newCenters);
    centers = newCenters;
}
while (convergence > EPSILON) {
    ...
    Map<Integer, Point> newCenters = clusters.entrySet().stream().collect(Collectors.toMap(e -> e.getKey(),
        e -> Point.barycenter(e.getValue())));
    convergence = distance(centers, newCenters);
    centers = newCenters;
}
while (convergence > EPSILON) {
... Map<Integer, Point> newCenters = clusters.entrySet().stream().collect(Collectors.toMap(e -> e.getKey(), e -> Point.barycenter(e.getValue())));
convergence = distance(centers, newCenters);
centers = newCenters;
}

*k-Means Con’t*
while (convergence > EPSILON) {
    Map<Integer, Point> newCenters = clusters.entrySet().stream().collect(Collectors.<
        Map.Entry<Integer, Point> entry -> entry.getKey(),
        Map.Entry<Integer, Point> entry -> Point.barycenter(entry.getValue())));
    convergence = distance(centers, newCenters);
    centers = newCenters;
}
Functional k-Means

- many fewer lines of code
- easier to read (really!)
- easier to reason
- easier to optimize
Arrays.asList("Arlington", "Berkeley", "Clarendon", "Dartmouth", "Exeter")
.stream()
.forEach(s -> printf("%s\n", s));

So far, Streams are sequential
Parallelism?

Arrays.asList("Arlington", "Berkeley", "Clarendon", "Dartmouth", "Exeter")
.stream()
.forEach(s -> printf("%s\n", s));

make List of strings and turn them into a stream
Arrays.asList("Arlington",
"Berkeley",
"Clarendon",
"Dartmouth",
"Exeter")
.stream()
.forEach(s -> printf("%s\n", s));

**forEach** applies a method to each element (not functional)
Arrays.asList("Arlington", "Berkeley", "Clarendon", "Dartmouth", "Exeter")
.stream().forEach(s -> printf("%s\n", s));

prints:
Arlington
Berkeley
Clarendon
Dartmouth
Exeter
Parallelism?

Arrays.asList("Arlington", "Berkeley", "Clarendon", "Dartmouth", "Exeter")
.parallelStream()
.forEach(s -> printf("%s\n", s));

turn List into a parallel stream
Arrays.asList("Arlington", "Berkeley", "Clarendon", "Dartmouth", "Exeter")

parallelStream().forEach(s -> {
    printf("%s
", s);
});

prints

Arlington
Berkeley
Clarendon
Dartmouth
Exeter
Dartmouth
Berkeley
Exeter
Arlington
Clarendon
Clarendon
Exeter
Berkeley
Exeter
Arrays.asList("Arlington",
"Berkeley",
"Clarendon",
"Dartmouth",
"Exeter")
.parallelStream()
.forEach(s -> printf("%s
", s));

can turn stream into a parallel stream
Pitfalls

```java
list.stream().forEach(
    s -> list.add(0)
);
```
list.stream().forEach(
  s -> list.add(0)
);

lambda (function) must not modify source!
source.parallelStream()
  .forEach(
    s -> target.add(s));

Throw exception if target not thread-safe

order added is non-deterministic
Conclusions

Streams support

functional programming

data parallelism

compiler optimizations
This work is licensed under a Creative Commons Attribution-ShareAlike 2.5 License.

• You are free:
  – to Share — to copy, distribute and transmit the work
  – to Remix — to adapt the work
• Under the following conditions:
  – Attribution. You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors endorse you or your use of the work).
  – Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
• For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
  – http://creativecommons.org/licenses/by-sa/3.0/.
• Any of the above conditions can be waived if you get permission from the copyright holder.
• Nothing in this license impairs or restricts the author's moral rights.