Data Parallelism

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit
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!

Art of Multiprocessor Programming
Let’s Ask Google Ngram!

multicore cable
multicore fiber

but we digress …
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
Map Phase

must count words!

must count words!

must count words!

a mapping thread per chapter

chapter 1

chapter 2

... chapter k
Map Phase

alpha → 9
juliet → 2,
alpha → 1
tango → 4

each mapper thread produces a stream …

of key-value pairs …

key: word
value: local count

carrier
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, 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;
    }
}

produces a map: word → count
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?
WordCount Mapper

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

WordCount Mapper

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

document fragment is list of words
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

    ...
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
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;
}
```

examine each word in the document fragment
WordCount Mapper

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

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...
Reduce Phase

alpha → 3
bravo → 2
... 
zulu → 1

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;
    }
}
Reducer Class

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;
    }
}

each reducer is given a single key (word)
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;
    }
}

It produces a single summary value (the total count for that word)
WordCount

normalizing document wordcount gives a *fingerprint* vector

\[
\begin{array}{c}
0.037 \\
0.002 \\
0.045 \\
\cdots \\
0.000
\end{array}
\]
a fingerprint is a point in a high-dimensional space
k-means

Find $k$ clusters from raw data
k-means

Find \( k \) clusters from raw data

each vector closer to those in same cluster …

than in different clusters.
MapReduce

split points among mapping threads
k-means

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

Reducer sends *key-value* pair to mappers

**key:** cluster number

**value:** center point

thread 1

thread 2 or Programming

thread k
Each mapper uses centers to assign each vector to a cluster.

0 → c₀
1 → c₁
2 → c₂
Mappers

Each mapper uses centers to assign each vector to a cluster

0 → c₀
1 → c₁
2 → c₂
Mappers

mapper sends key-value stream to reducer
key: point
value: cluster ID

p₀ → 2
p₁ → 1
p₂ → 1
p₃ → 0

mapper sends key-value stream to reducer
key: point
value: cluster ID
Back at the Reducer

The reducer merges the streams ... and assembles clusters ...
The reducer computes new centers based on new clusters …
Once is Not Enough

Reducer sends *new centers* to mappers

Process ends when centers become stable

0 → $c_0'$
1 → $c_1'$
2 → $c_2'$
To Recapitulate

We saw two problems …

wordcount & k-means …

with similar parallel solutions

Map part is parallel …

Reduce part is sequential.
abstraction
Map Function

\((k_1, v_1) \mapsto \text{list}(k_2, v_2)\)

- 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

sequence of transformations

sometimes in parallel

no relation to I/O streams
Streams

transformations given by mathematical functions
Streams

transformation creates new stream

no modifications or side-effects

correctness easier?
Functional Programming

*functions* map old state to new state

old state never changed

no complex side-effects

elegant, easier proofs of correctness
“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

1, 2, 3, ...

$x \rightarrow x + 1$

No computation until absolutely necessary

add 1 to each element: no computation

double each element: no computation

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

$1, 2, 3, \ldots$

$x \rightarrow x + 1$

$x \rightarrow 2x$

collect in List

move to container is terminal
Laziness

1, 2, 3, ...

x → x + 1

x → 2x

collect in List

Laziness permits optimizations

x → 2(x + 1)
Laziness

Laziness permits infinite streams …

Stream<Integer> fib = new FibStream();
Unbounded Random Stream

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

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

Unbounded stream of double-precision random numbers
Random Stream

Stream<Double> randomDoubleStream() {
    return Stream.generate(
        () -> random.nextDouble());
}
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());
}
```
List<String> readFile(String fileName) {
    ...
    return reader
        .lines()
        .map(String::toLowerCase)
        .flatMap(s ->
            puts each word from the document into a List
            pattern.splitAsStream(s))
        .collect(Collectors.toList());
}
```
WordCount

List<String> readFile(String fileName) {

    ... 

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

```java
List<String> readFile(String fileName) {
    ...

    return reader
        .lines() // turn the FileReader into a stream of lines, each line a string
        .map(String::toLowerCase)
        .flatMap(s ->
            pattern.splitAsStream(s))
        .collect(Collectors.toList());
}
```
The method `readFile` reads a file named `fileName` and returns a list of strings. It first reads the file using a reader and then maps each line to lowercase. After that, it flat-maps each string into an array of words using a regular expression pattern, and finally collects the results into a list.

```java
WordCount
{
    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());
}
```

**flatMap** replaces one stream element with multiple stream elements

here, splits line into words
WordCount

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

collect puts stream elements in a container here, in a List
terminal operation
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()));
```

**WordCount**

Start with list of words
WordCount

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

turn List into a stream
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()));
class Point {
    Point(double x, double y) {...}
    Point plus(Point other) {...}
    Point scale(double x) {...}
    static Point barycenter(
        List<Point> cluster
    ) {...}
}
k-Means

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);
}
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)
        .get();
    return sum.get().scale(1 / numPoints);
}
Reduce

**stream.reduce(+):**

- \( () \rightarrow \emptyset \)
- \( (a) \rightarrow a \)
- \( (a,b) \rightarrow a+b \)
- \( (a,b,c) \rightarrow (a+b)+c \)

**terminal operation**

etc. …
**k-Means**

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

Optional because sum might be empty!

Sum points in cluster
$k$-Means

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

extract sum and divide by # points
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) {
    ...
List<Point> points = readFile("cluster.dat");

centers = randomDistinctCenters(points);

double convergence = 1.0;
while (convergence > EPSILON) {
    ...
}

pick random centers
k-Means

List<Point> points =
    readFile("cluster.dat");

centers =
    randomDistinctCenters(points);

double convergence = 1.0;

while (convergence > EPSILON) {
    ...
}

keep going until centers are stable
Compute New Clusters

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())));

    convergence = distance(centers, newCenters);
    centers = newCenters;
}

compute new centers map: ID → center
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;
}
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;
}
Functional k-Means

- many fewer lines of code
- easier to read (really!)
- easier to reason
- easier to optimize
Parallelism?

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

So far, Streams are sequential
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)
Parallelism?

```java
Arrays.asList("Arlington", "Berkeley", "Clarendon", "Dartmouth", "Exeter")
.stream()
.forEach(s -> printf("%s
", 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
Parallelism?

```java
Arrays.asList("Arlington", "Berkeley", "Clarendon", "Dartmouth", "Exeter")
      .parallelStream()
      .forEach(System.out::printf); // prints
```

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

exception if target not thread-safe
order added is non-deterministic
Conclusions

- Streams support
- Functional programming
- Data parallelism
- Compiler optimizations