Parallel Programming with Spark

Qin Liu

The Chinese University of Hong Kong
Previously on Parallel Programming

OpenMP: an API for writing multi-threaded applications

- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded programs in Fortran and C/C++
- Standardizes last 20 years of symmetric multiprocessing (SMP) practice
Compute \( \pi \) using Numerical Integration

Let \( F(x) = \frac{4}{1 + x^2} \)

\[
\pi = \int_0^1 F(x) \, dx
\]

Approximate the integral as a sum of rectangles:

\[
\sum_{i=0}^{N} F(x_i) \Delta x \approx \pi
\]

where each rectangle has width \( \Delta x \) and height \( F(x_i) \) at the middle of interval \( i \)
Example: \(\pi\) Program with OpenMP

```c
#include <stdio.h>
#include <omp.h> // header
const long N = 100000000;
#define NUM_THREADS 4 // #threads
int main () {
    double sum = 0.0;
double delta_x = 1.0 / (double) N;
omp_set_num_threads(NUM_THREADS); // set #threads
#pragma omp parallel for reduction(+:sum) // parallel for
    for (int i = 0; i < N; i++) {
        double x = (i+0.5) * delta_x;
        sum += 4.0 / (1.0 + x*x);
    }
double pi = delta_x * sum;
printf("pi is %f\n", pi);
}```
Example: $\pi$ Program with OpenMP

```c
#include <stdio.h>
#include <omp.h> // header
const long N = 100000000;
#define NUM_THREADS 4 // #threads
int main () {
    double sum = 0.0;
    double delta_x = 1.0 / (double) N;
    #pragma omp parallel for reduction(+:sum) // parallel for
    for (int i = 0; i < N; i++) {
        double x = (i+0.5) * delta_x;
        sum += 4.0 / (1.0 + x*x);
    }
    double pi = delta_x * sum;
    printf("pi is %f\n", pi);
}
```

How to parallelize the $\pi$ program on distributed clusters?
Outline

Why Spark?

Spark Concepts

Tour of Spark Operations

Job Execution

Spark MLlib
Why Spark?
# Apache Hadoop Ecosystem

<table>
<thead>
<tr>
<th>Component</th>
<th>Hadoop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource Manager</td>
<td>YARN</td>
</tr>
<tr>
<td>Storage</td>
<td>HDFS</td>
</tr>
<tr>
<td>Batch</td>
<td><strong>MapReduce</strong></td>
</tr>
<tr>
<td>Streaming</td>
<td>Flume</td>
</tr>
<tr>
<td>Columnar Store</td>
<td>HBase</td>
</tr>
<tr>
<td>SQL Query</td>
<td>Hive</td>
</tr>
<tr>
<td>Machine Learning</td>
<td>Mahout</td>
</tr>
<tr>
<td>Graph</td>
<td>Giraph</td>
</tr>
<tr>
<td>Interactive</td>
<td>Pig</td>
</tr>
</tbody>
</table>
Apache Hadoop Ecosystem

<table>
<thead>
<tr>
<th>Component</th>
<th>Hadoop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource Manager</td>
<td>YARN</td>
</tr>
<tr>
<td>Storage</td>
<td>HDFS</td>
</tr>
<tr>
<td>Batch</td>
<td>MapReduce</td>
</tr>
<tr>
<td>Streaming</td>
<td>Flume</td>
</tr>
<tr>
<td>Columnar Store</td>
<td>HBase</td>
</tr>
<tr>
<td>SQL Query</td>
<td>Hive</td>
</tr>
<tr>
<td>Machine Learning</td>
<td>Mahout</td>
</tr>
<tr>
<td>Graph</td>
<td>Giraph</td>
</tr>
<tr>
<td>Interactive</td>
<td>Pig</td>
</tr>
</tbody>
</table>

... mostly focused on large on-disk datasets: great for **batch** but **slow**
Many Specialized Systems

MapReduce doesn’t compose well for large applications, and so *specialized* systems emerged as workarounds

<table>
<thead>
<tr>
<th>Component</th>
<th>Hadoop</th>
<th>Specialized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource Manager</td>
<td>YARN</td>
<td>RAMCloud</td>
</tr>
<tr>
<td>Storage</td>
<td>HDFS</td>
<td></td>
</tr>
<tr>
<td>Batch</td>
<td>MapReduce</td>
<td></td>
</tr>
<tr>
<td>Streaming</td>
<td>Flume</td>
<td>Storm</td>
</tr>
<tr>
<td>Columnar Store</td>
<td>HBase</td>
<td></td>
</tr>
<tr>
<td>SQL Query</td>
<td>Hive</td>
<td></td>
</tr>
<tr>
<td>Machine Learning</td>
<td>Mahout</td>
<td>DMLC</td>
</tr>
<tr>
<td>Graph</td>
<td>Giraph</td>
<td>PowerGraph</td>
</tr>
<tr>
<td>Interactive</td>
<td>Pig</td>
<td></td>
</tr>
</tbody>
</table>
Goals

A new ecosystem

- leverages current generation of commodity hardware
- provides fault tolerance and parallel processing at scale
- easy to use and combines SQL, Streaming, ML, Graph, etc.
- compatible with existing ecosystems
Berkeley Data Analytics Stack

being built by AMPLab to make sense of Big Data

<table>
<thead>
<tr>
<th>Component</th>
<th>Hadoop</th>
<th>Specialized</th>
<th>BDAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource Manager</td>
<td>YARN</td>
<td>RAMCloud</td>
<td>Mesos</td>
</tr>
<tr>
<td>Storage</td>
<td>HDFS</td>
<td>Tachyon</td>
<td></td>
</tr>
<tr>
<td>Batch</td>
<td>MapReduce</td>
<td>Spark</td>
<td></td>
</tr>
<tr>
<td>Streaming</td>
<td>Flume</td>
<td>Storm</td>
<td>Streaming</td>
</tr>
<tr>
<td>Columnar Store</td>
<td>HBase</td>
<td>Parquet</td>
<td></td>
</tr>
<tr>
<td>SQL Query</td>
<td>Hive</td>
<td>SparkSQL</td>
<td></td>
</tr>
<tr>
<td>Approximate SQL</td>
<td></td>
<td>BlinkDB</td>
<td></td>
</tr>
<tr>
<td>Machine Learning</td>
<td>Mahout</td>
<td>DMLC</td>
<td>MLlib</td>
</tr>
<tr>
<td>Graph</td>
<td>Giraph</td>
<td>PowerGraph</td>
<td>GraphX</td>
</tr>
<tr>
<td>Interactive</td>
<td>Pig</td>
<td></td>
<td>built-in</td>
</tr>
</tbody>
</table>

\[1\text{https://amplab.cs.berkeley.edu/software/}\]
Spark Concepts
What is Spark?

Fast and expressive cluster computing system compatible with Hadoop

- Works with many storage systems: local FS, HDFS, S3, SequenceFile, ...
What is Spark?

Fast and expressive cluster computing system compatible with Hadoop

- Works with many storage systems: local FS, HDFS, S3, SequenceFile, ...

Improves efficiency through:
- As much as 30x faster
  - In-memory computing primitives
  - General computation graphs
What is Spark?

Fast and expressive cluster computing system compatible with Hadoop

- Works with many storage systems: local FS, HDFS, S3, SequenceFile, ...

Improves efficiency through:
- As much as 30x faster
- In-memory computing primitives
- General computation graphs

Improves usability through rich Scala/Java/Python APIs and interactive shell
- Often 2-10x less code
Main Abstraction - RDDs

Goal: work with distributed collections as you would with local ones
Main Abstraction - RDDs

Goal: work with distributed collections as you would with local ones

Concept: resilient distributed datasets (RDDs)

- Immutable collections of objects spread across a cluster
Main Abstraction - RDDs

Goal: work with distributed collections as you would with local ones

Concept: resilient distributed datasets (RDDs)

- Immutable collections of objects spread across a cluster
- Built through parallel transformations (map, filter, ...)

Main Abstraction - RDDs

Goal: work with distributed collections as you would with local ones

Concept: resilient distributed datasets (RDDs)

- Immutable collections of objects spread across a cluster
- Built through parallel *transformations* (*map*, *filter*, ...)
- Automatically rebuilt on failure
Main Abstraction - RDDs

Goal: work with distributed collections as you would with local ones

Concept: resilient distributed datasets (RDDs)

- Immutable collections of objects spread across a cluster
- Built through parallel *transformations* (map, filter, ...)
- Automatically rebuilt on failure
- Controllable persistence (e.g. caching in RAM) for reuse
Main Primitives

Resilient distributed datasets (RDDs)

- Immutable, partitioned collections of objects
Main Primitives

Resilient distributed datasets (RDDs)
- Immutable, partitioned collections of objects

Transformations (e.g. map, filter, reduceByKey, join)
- Lazy operations to build RDDs from other RDDs
Main Primitives

Resilient distributed datasets (RDDs)
  • Immutable, partitioned collections of objects

Transformations (e.g. map, filter, reduceByKey, join)
  • Lazy operations to build RDDs from other RDDs

Actions (e.g. collect, count, save)
  • Return a result or write it to storage
Learning Spark

Download the **binary package** and uncompressed it.
Learning Spark

Download the **binary package** and uncompress it

Interactive Shell (**easiest way**): `./bin/pyspark`
  - modified version of Scala/Python interpreter
  - runs as an app on a Spark cluster or can run locally

```
qliu@qliu-office ~/workspace/spark-1.4.1-bin-hadoop2.6 $ ./bin/pyspark 2> /dev/null
Welcome to
Spark version 1.4.1
Using Python version 2.7.6 (default, Jun 22 2015 17:58:13)
SparkContext available as sc, HiveContext available as sqlContext.
>>> lines = sc.textFile("./data/log.txt")
>>> `
Learning Spark

Download the **binary package** and uncompress it

Interactive Shell (**easiest way**): `. /bin/pyspark`

- modified version of Scala/Python interpreter
- runs as an app on a Spark cluster or can run locally

```
Welcome to  
Spark  
version 1.4.1
Using Python version 2.7.6 (default, Jun 22 2015 17:58:13)  
SparkContext available as sc, HiveContext available as sqlContext.  
```  

Standalone Programs: `. /bin/spark-submit <program>`

- Scala, Java, and Python

This talk: mostly Python
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

DEMO:

```python
lines = sc.textFile("hdfs://...") # load from HDFS

# transformation
errors = lines.filter(lambda s: s.startswith("ERROR"))

# transformation
messages = errors.map(lambda s: s.split(' ')[1])

messages.cache()

# action; compute messages now
messages.filter(lambda s: "life" in s).count()

# action; reuse cached messages
messages.filter(lambda s: "work" in s).count()
```
RDD Fault Tolerance

RDDs track the series of transformations used to build them (their *lineage*) to recompute lost data.

```
msgs = sc.textFile("hdfs://...").filter(lambda s: s.startswith("ERROR"))
    .map(lambda s: s.split('\t')[1])
```
Spark vs. MapReduce

- Spark keeps intermediate data in memory
- Hadoop only supports map and reduce, which may not be efficient for join, group, ...
- Programming in Spark is easier
Tour of Spark Operations
Spark Context

- Main entry point to Spark functionality

- Created for you in **Spark shell** as variable `sc`

- In standalone programs, you’d make your own:

```python
from pyspark import SparkContext

sc = SparkContext(appName="ExampleApp")
```
Creating RDDs

- Turn a local collection into an RDD
  
  ```python
  rdd = sc.parallelize([1, 2, 3])
  ```

- Load text file from local FS, HDFS, or other storage systems
  
  ```python
  sc.textFile("file:///path/file.txt")
  sc.textFile("hdfs://namenode:9000/file.txt")
  ```

- Use any existing Hadoop InputFormat
  
  ```python
  sc.hadoopFile(keyClass, valClass, inputFmt, conf)
  ```
nums = sc.parallelize([1, 2, 3])

# Pass each element through a function
squares = nums.map(lambda x: x*x)
# => {1, 4, 9}

# Keep elements passing a predicate
even = squares.filter(lambda x: x%2 == 0)
# => {4}

# Map each element to zero or more others
nums.flatMap(lambda x: range(x))
# => {0, 0, 1, 0, 1, 2}
Basic Actions

```python
nums = sc.parallelize([1, 2, 3])

# Retrieve RDD contents as a local collection
nums.collect()  # => [1, 2, 3]

# Return first K elements
nums.take(2)  # => [1, 2]

# Count number of elements
nums.count()  # => 3

# Merge elements with an associative function
nums.reduce(lambda a, b: a+b)  # => 6

# Write elements to a text file
nums.saveAsTextFile("hdfs://host:9000/file")
```
Example: $\pi$ Program in Spark

Compute

$$\sum_{i=0}^{N} F(x_i) \Delta x \approx \pi$$

where $F(x) = \frac{4}{1 + x^2}$

$N = 100000000$
$\delta_x = 1.0 / N$

```python
print sc.parallelize(xrange(N))  # i
    .map(lambda i: (i+0.5) * delta_x)  # x_i
    .map(lambda x: 4 / (1 + x**2))  # F(x_i)
    .reduce(lambda a, b: a+b) * delta_x  # pi
```

24
Working with Key-Value Pairs

A few special transformations operate on RDDs of key-value pairs: `reduceByKey`, `join`, `groupByKey`, ...

Python pair (2-tuple) syntax:

```
pair = (a, b)
```

Accessing pair elements:

```
pair[0]  # => a
pair[1]  # => b
```
Some Key-Value Operations

```scala
val pets = sc.parallelize([("cat", 1), ("dog", 1), ("cat", 2)])
pets.reduceByKey(lambda a, b: a+b)
# => [("cat", 3), ("dog", 1)]
pets.groupByKey()
# => [("cat", [1, 2]), ("dog", [1])]  
pets.sortByKey()
# => [("cat", 1), ("cat", 2), ("dog", 1)]
```
Example: Word Count

```scala
lines = sc.textFile("...
counts = lines.flatMap(lambda s: s.split())
  .map(lambda w: (w, 1))
  .reduceByKey(lambda a, b: a+b)
```

```
"to be or"
  |  "to"
  |  "be"
  |  "or"
  |  (to, 1)
  |  (be, 1)
  |  (or, 1)

"not to be"
  |  "not"
  |  "to"
  |  "be"
  |  (not, 1)
  |  (to, 1)
  |  (be, 1)
```

```
  "be, 2"
  "not, 1"
  "or, 1"
  "to, 2"
```
Other RDD Operations

sample(): deterministically sample a subset

join(): join two RDDs

union(): merge two RDDs

cartesian(): cross product

pipe(): pass through external program

Job Execution
Software Components

- Spark runs as a library in your program (1 instance per app)
- Runs tasks locally or on cluster
  - Mesos, YARN or standalone mode
- Accesses storage systems via Hadoop InputFormat API
  - Can use HBase, HDFS, S3, ...
Task Scheduler

- General task graphs
- Automatically pipelines functions
- Data locality aware
- Partitioning aware to avoid shuffles
Advanced Features

• Controllable partitioning
  ▶ Speed up joins against a dataset

• Controllable storage formats
  ▶ Keep data serialized for efficiency, replicate to multiple nodes, cache on disk

• Shared variables: broadcasts, accumulators

• See online docs for details!
Launching on a Cluster

On a private cloud

- **Standalone Deploy Mode**: simplest Spark cluster
  
  ```
  vim conf/slaves # add hostnames of slaves
  ./sbin/start-all.sh
  ```

- **Mesos**
- **YARN**

Running Spark on EC2

- Prepare your AWS account
  
  ```
  ./ec2/spark-ec2 -k <keypair> -i <key-file> -s <num-slaves> launch <cluster-name>
  ```
Spark MLlib
A scalable machine learning library consisting of common learning algorithms and utilities

These libraries are implemented using Spark APIs in Scala and included in Spark codebase
Functionality of Spark MLlib

Classification
- Logistic regression
- Naive Bayes
- Streaming logistic regression
- Linear SVMs
- Decision trees
- Random forests
- Gradient-boosted trees

Regression
- Ordinary least squares
- Ridge regression
- Lasso
- Isotonic regression
- Decision trees
- Random forests
- Gradient-boosted trees
- Streaming linear methods

Clustering
- Gaussian mixture models
- K-Means
- Streaming K-Means
- Latent Dirichlet Allocation
- Power Iteration Clustering

Recommendation
- Alternating Least Squares

Feature extraction & selection
- Word2Vec
- Chi-Squared selection
- Hashing term frequency
- Inverse document frequency
- Normalizer
- Standard scaler
- Tokenizer

Statistics
- Pearson correlation
- Spearman correlation
- Online summarization
- Chi-squared test
- Kernel density estimation

Linear algebra
- Local dense & sparse vectors & matrices
- Distributed matrices
  - Block-partitioned matrix
  - Row matrix
  - Indexed row matrix
  - Coordinate matrix
- Matrix decompositions

Frequent itemsets
- FP-growth

Model import/export
Example: k-means clustering

Given \((x_1, x_2, \ldots, x_n)\), partition the \(n\) samples into \(k\) sets \(S = \{S_1, S_2, \ldots, S_k\}\) so as to minimize the within-cluster sum of squares (WCSS):

\[
\arg\min_S \sum_{i=1}^{k} \sum_{x \in S_i} \|x - \mu_i\|^2
\]

where \(\mu_i\) is the mean of points in \(S_i\).

**Algorithm:** initialize \(\mu_i\), then iterate till converge

- **Assignment:** assign each sample to the cluster with nearest mean
- **Update:** calculate the new means
Example: k-means clustering

Main API: `pyspark.mllib.clustering.KMeans.train()`

Parameters:
- `rdd`: stores training samples
- `k`: number of clusters
- `maxIterations`: maximum number of iterations
- `initializationMode`: random or k-means||
- `runs`: number of times to run k-means
- `initializationSteps`: number of steps in k-means||
- `epsilon`: distance threshold of convergence
Example: k-means clustering

```python
from pyspark import SparkContext
from pyspark.mllib.clustering import KMeans, KMeansModel
from numpy import array
from math import sqrt

sc = SparkContext(appName = "K-Means")

# Load and parse the data
data = sc.textFile("data/mllib/kmeans_data.txt")
parsedData = data.map(lambda line: array(map(float, line.split())))
```
Example: k-means clustering

```python
# Build the model (cluster the data)
clusters = KMeans.train(parsedData, 2, maxIter=10,
    runs=10, initializationMode="random")

# Evaluate clustering by computing WCSS
def error(point):
    center = clusters.centers[clusters.predict(point)]
    return sqrt(sum([x**2 for x in (point - center)]))

WCSS = parsedData.map(error).reduce(lambda x, y: x + y)
print("Within Set Sum of Squared Error = " + str(WCSS))

# Save and load model
clusters.save(sc, "myModelPath")
sameModel = KMeansModel.load(sc, "myModelPath")
```
References


• Spark Docs: [link](#)

• Spark Programming Guide: [link](#)

• Example code: [link](#)

• Parallel Programming with Spark (Part 1 & 2) - Matei Zaharia: [YouTube](#)