Parallel and Distributed Computing - P1

紧密耦合的并行计算 (tightly-coupled parallel computing)
shared-memory symmetric multiprocessor (SMP) system
统一内存访问 (uniform memory access)
memory conection on single bus
cache consistency
非统一内存访问 (non-uniform memory access, NUMA)
single memory space (local / non-local memory access)
memory bus and switch architecture

松散耦合的分布式架构 (loosely-coupled distributed architecture)
use multi SMP system connected by high-speed network
major challenge : software to utilize this architecture

典型的数据中心组织结构 (typical datacenter Organization)
服务器 (server) : single SMP computer with network connection and storage
机架 (rack) : physical stack of servers, with network switch
通道 (aisle) : set of physically adjacent racks, with network switch

高可靠性硬件 (high-reliability hardware)
every computer/switch/etc. has redundant components (冗余备份)
high availability of each components (高度可用)
high initial cost, high operating cost (高配置/运行成本)
rare failure of computer server

商用硬件 (commodity hardware)
off-the-shelf component (现成组件)
low initial cost, no idle redundant components
typical unit of failure : sever
more frequent, software is designed to deal with

其他设计因素, (design consideration)
individual server processing and I/O speed, sever density
power, cooling, internal network latency, external network connectivity

Parallel and Distributed Computing - P2

弹性软件(resilient software)，设计用于商用硬件系统
designed to cope with loss of servers without restarting massive computations
restart only the portion of the computation running on the lost server
files must reside on multiple servers and provide seamless switchover

分布式算法(distributed algorithm)
same algorithm suitable for SMP, not suitable for distributed solution

典型的数据中心软件组织(example datacenter software organization)
sever: Linux operating system
Hadoop scheduler Yarn: on one or more nodes
each compute server: Hadoop job scheduling node software
Distributed file system (HDFS) master/server: on one or more nodes
Distributed database server (HBase, Cassandra): on one or more nodes
User → upload data into the file and database system
→ submit job through Yarn

计算集群(compute cluster)，在典型的计算集群中有两个部分
分布式文件系统(distributed file system)
在Hadoop中为HDFS(Hadoop Distributed File System)
files may be accessed from any node in the cluster
need special procedure, not accessible from normal Linux interface
not a shared file system, 如网络文件系统(network file system, NFS)
工作分配者(job scheduler)
在Hadoop中为YARN

building a program for YARN
必须是.jar文件，且包含程序运行需要的所有自定义java类(user-defined java class)
使用Maven(mvn)或ant build tool
需要设置环境变量CLASSPATH，可通过env命令查看

hadoop jar <parameter>: run a Hadoop MapReduce job
spark-submit --master yarn --deploy-mode cluster <parameter>
: run a Spark job in the cluster

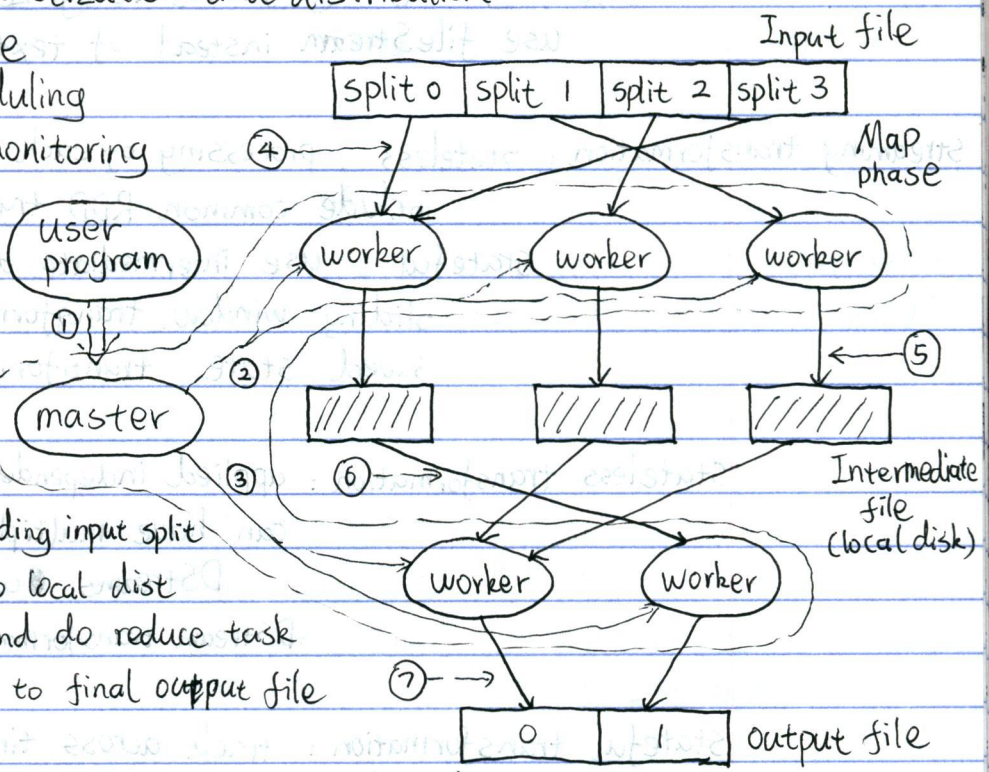
Parallel and Distributed Computing - P3

YARN 的文件处理

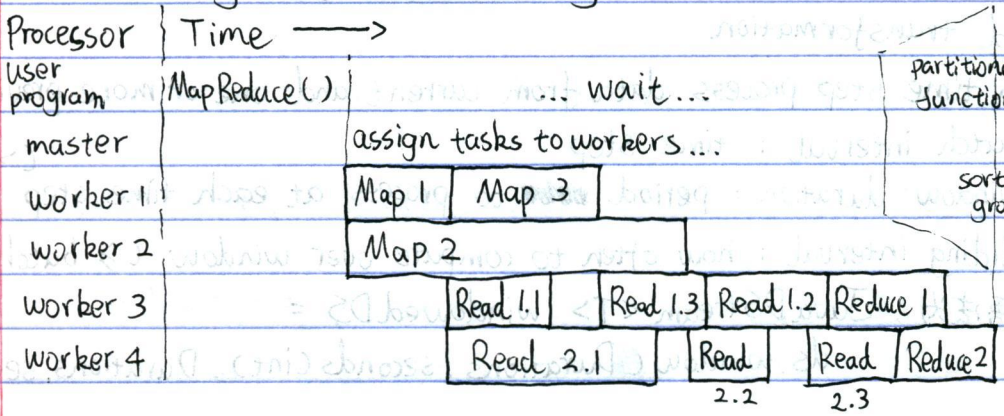
jobs running on the cluster may not have access to user Linux FS
data for jobs under YARN should be stored in HDFS
output under YARN should be generated in HDFS
can copy the result to Linux FS after job completion

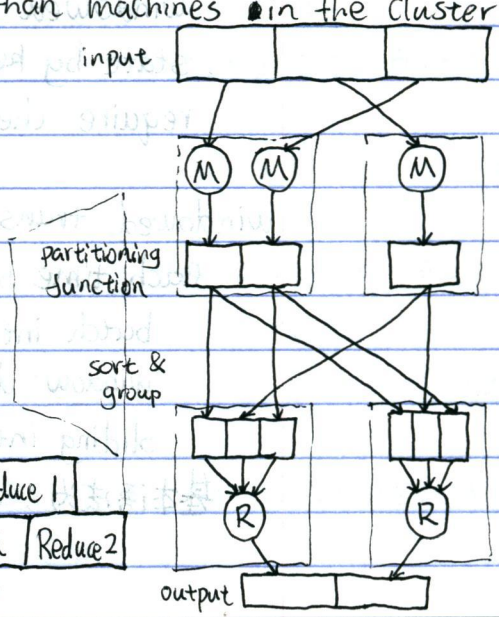
MapReduce

large scale data processing
automatic parallelization and distribution
fault-tolerance
I/O scheduling
status and monitoring



fine granularity task : many more map tasks than machines in the cluster
minimize time for fault recovery
can pipeline suffling with map execution
better dynamic load balancing





Parallel and Distributed Computing - P4

MapReduce fault tolerance : handle via re-execution
on worker failure (master failure unlikely)
detect failure via periodic heartbeats
re-execute completed and in-progress map tasks
re-execute in-progress reduce tasks
task competition commited through master

refinement redundant execution : slow workers significantly lengthen completion time
other jobs consuming resources on machine
bad disks with soft errors transfer data slowly
processor caches disabled (weird)
solution : spawn backup copies of tasks near end of phase
dramatically shorten job completion time

locality optimization : master scheduling policy
ask GFS (global file system) for locations of replicas of input file blocks
map tasks typically split into GFS block size
scheduled so GFS input block replica on same machine or same rack
thousands of machines read input at local disk speed
without this, rack switch limit read rate

skipping bad records : Map/Reduce function fail for particular input
best solution is to debug & fix, not always possible
on seg fault, send UDP (user datagram protocol) packet to master
from signal handler include sequence num of record being processed
if master see two failure on same record, next worker skip record
can work around bugs in third-party libraries

sorting guarantees within each reduce partition
compression of intermediate data
combiner : useful for saving network bandwidth
local execution for debugging / testing
user-defined counters

Parallel and Distributed Computing - P5

common RDD (resilient distributed dataset) types:
注意，这里的RDD type 均以 Java 中实现为准，且大小写敏感

JavaRDD<T> : an RDD made up of items of class T
如，JavaRDD<String> for data from a text file
each item is one line of text

JavaPairRDD<K,V> : an RDD made up of Key, Value pairs
key is class K, value is class V
注意：Java中的 key, value pair 表示为 Tuple2 类型
即有 Tuple2<K,V>，并且提供方法读取 key 或 value
如 K key = tuple2._1();
V value = tuple2._2();

transformation require specification of function to perform
Java function is always part of a class
pass a class object which has at least a method named "call"
as a function to Spark transformation
Spark defines Java interfaces for commonly-used transformation class

simple map : Function, (org.apache.spark.api.java.function.Function)
函数类型为 Function<T,R> :: T -> R
对应的 map 函数效果为
map :: JavaRDD<T> -> Function<T,R> -> JavaRDD<R>
其本语法结构为：对于 JavaRDD<T> t

```
JavaRDD<R> r = t.map(
    new Function<T, R>() {
        public R call(T arg) {
            ...
            return ret;
        }
    }
);
```

] 定义了新的 JavaRDD<R> 对象 r
] 传入 map 函数的 Function 实例
注意括号为空是因为其构造函数不需要传入参数
Function 实例中的 call 方法定义
返回值类型为 R，
传入参数为类型 T 的 arg
返回值 ret 也应为 类型 R

Parallel and Distributed Computing - P6

flatMap generate multiple outputs from each input

函数类型为 FlatMapFunction<T, R>

且有 FlatMapFunction<T, R> :: T -> Iterator<R>

flatMap 效果为 flatMap :: JavaRDD<T> -> FlatMapFunction<T, R> -> JavaRDD<R>

基本语法为，对于 JavaRDD<T> t

```
JavaRDD<R> r = t.flatMap(
    new FlatMapFunction<T, R>() {
        public Iterator<R> call(T arg){…
            ArrayList<R> ret; …
            return ret.iterator(); }});
```

mapToPair generate key, value pair from each input

函数类型为 PairFunction<T, K, V>

且有 PairFunction<T, K, V> :: T -> Tuple2<K, V>

于是有 mapToPair :: JavaRDD<T> -> PairFunction<T, K, V> -> JavaPairRDD<K, V>

基本语法为，对于 JavaRDD<T> t

```
JavaPairRDD<K,V> p = t.mapToPair(
    new PairFunction<T, K, V>() {
        public Tuple2<K,V> call(T arg){ …
            Tuple2<K,V> ret; …
            return ret; }});
```

flatMapToPair generate multiple pairs from each input

函数类型为 PairFlatMapFunction<T, K, V>

且有 PairFlatMapFunction<T, K, V> :: T -> Iterator<Tuple2<K, V>>

于是有 flatMapToPair :: JavaRDD<T> -> PairFlatMapFunction<T, K, V> -> JavaPairRDD<K,V>

基本语法为，对于 JavaRDD<T> t

```
JavaPairRDD<K,V> p = t.flatMapToPair(
    new PairFlatMapFunction<T, K, V>(){
        public Iterator<Tuple2<K,V>> call(T arg){…
            ArrayList<Tuple2<K,V>> ret; …
            return ret.iterator(); }});
```

Parallel and Distributed Computing - P7

reduceByKey combine all source items which have the same key

函数类型为 Function2<V, V, V> :: V -> V -> V

且函数必须满足结合律(associative)和交换律(commutative)

现有 reduceByKey :: JavaPairRDD<K,V> -> Function2<V, V, V> -> JavaPairRDD<K,V>

基本语法为，对于 JavaPairRDD<K, V> P1

```
JavaPairRDD<K, V> P2 = P1.reduceByKey(
    new Function2<V, V, V>() {
        public V call(V x, V y){ ...
            V ret; ...
            return ret; } });
```

serializable (可串行化的)，在 Java 中，对于 JavaRDD<T> 和 JavaRDD<K, V>

其中的数据类型必须是 serializable

class object can be stored in a byte representation
from which it can be later restored elsewhere

necessary : data will be stored in file
and/or transmitted over network

class contain only serializable data members
may be made serializable with no special handling other than declaration
links, recursion, nested class and reference
may require special handling
java.io.Serializable interface allow definition of method
to serialize and de-serialize class object

基本语法结构为

```
import java.io.Serializable;
class ClassName implements Serializable {
    int d;
    double f;
    String s; }
```

only data members matter
function method don't affect serialization

groupByKey not need to combine values for given key and just want to group
result in an iterator over values for each key, no user function needed

现有 groupByKey :: JavaPairRDD<K, V> -> JavaPairRDD<K, Iterable<V>>

基本语法为，对于 JavaPairRDD<K, V> P1

```
JavaPairRDD<K, Iterable<V>> P2 = P1.groupByKey();
```

Parallel and Distributed Computing - P8

partition (划分): RDD in Spark is parallelized in partitions
- each partition may be processed on separate worker
- number of partitions may be determined by default
  - depend on workers available, input shards, etc.
  - can override the default with specified value
- map operation typically result in the same partition
- group or aggregate operation often result in different number of partitions

combineByKey: general-purpose transformation by key
- used in implementation of other transformation, 如 reduceByKey, groupByKey
- may return values of different type from the source
- caller provide three functions:
  - createCombiner: initialize a combiner with the first value
    即对于处理的第1个(key, value), 用value初始化 combiner
    于是有 createCombiner :: V -> W
  - mergeValue: add single value to an existing combiner
    即描述如何将一个(key, value)中的 value 加入一个 combiner
    于是有 mergeValue :: W -> V -> W
  - mergeCombiners: combine two separate combiners into single combiner
    即描述如何合并两个现有的 combiner
    于是有 mergeCombiners :: W -> W -> W

则有 combineByKey :: JavaPairRDD<K, V> -> (V -> W) -> (W -> V -> W) -> (W -> W -> W) -> JavaPairRDD<K, W>

如 对于 JavaPairRDD<K, int> P1, 实现 groupByKey 和 reduceByKey (sum)

```
JavaPairRDD<K, ArrayList<int>> P2
  = P1.combineByKey(
    value -> {
      ArrayList<int> combiner = new ArrayList<int>();
      combiner.add(value);
      return combiner;},
    (list, value) -> {list.add(value);
      return list;},
    (list1, list2) -> {list1.addAll(list2);
      return list1;});
```

```
JavaPairRDD<K, int> P2
  = P1.combineByKey(
    value -> {
      return value;},
    (sum, value) -> {
      return (sum + value);},
    (sum1, sum2) -> {
      return (sum1 + sum2);}
  );
```

Parallel and Distributed Computing - Pg

data flow for combineByKey:
within each source partition:
combiner created for each key the first time the key occurs
subsequent occurrence of record with the same key add to combiner
after source partition processed:
hash of key determine partition in destination RDD
within each destination partition:
merge combiners if more than one occurence of ckey, (ckey, combiner) pair

Spark object file format: an easy way to store user-defined serializable Java class
use the objectFile method of the JavaSparkContext
to create an RDD from an object file
use the saveAsObjectFile method of an RDD
to save the RDD to an object file
the class definition data members
must not have changed between saving and reading the object file
Java serialization less efficient than Hadoop sequence file
but much easier to use

Hadoop writable object in Spark, Hadoop define interface Writable
custom serialization interface defined in Hadoop
can implement in own class
require implementing the method to serialize and de-serialize object
useful classes provided ending in "Writable". eg IntWritable, FloatWritable
non-transparent: method need to get/set value
Spark can read/write SequenceFile composed of object
reading: simple, using sequenceFile of JavaSparkContext
writing: complicated, require Spark interface to Hadoop routines

lambda expression in Java, easy way to implement anonymous class
基本语法为: arg | (arg1, arg2, ...) -> { ... return ret; }
eg func :: T -> U

等价于 class lambdaFunc {
public U call(T arg){ ...
U ret; ...
return ret; }

Parallel and Distributed Computing - P10

function specified for spark operation (map, reduceByKey, etc)
can be written to access variable at the outer class level
work fine when running locally
doesn't work correctly when running in distributed mode

outer class variable only get initialized in the master
worker only get a serialized version of the function
without outer class variable

lambda run in a closure which inherits function local variable

function object : pass needed variable to the constructor
store inside the function object class as class variable

Pregel algorithm designed to solve iterative, parallel, synchronous graph algorithm
input directed graph
each vertex and edge can have associated data
user program run at each vertex
initialization -> compute (superstep processing) -> output

message passing model : vertex can send message
typically (not necessarily) along edge
message type specified by programmer
message sent in superstep n are received in superstep n+1

termination condition : processing terminate when all vertices 'vote' to quit
when vertex has no immediate action to take without futher input
processing continue if any message sent by any vertices

define combiner : multiple incomming messages can be combined
aggregator : combine specified value from each vertex at end of superstep
reduction using programmer defined code
result is made available to all vertices at next superstep

Parallel and Distributed Computing - P11

Pregel

topology change : vertex can add/delete vertex/edge during superstep
become effective next superstep
removal first, then vertex addition, then edge addition
conflict handled by user-supplied handler

input / output : graph input/output class specified by user
can be predefined or user-defined class
any arbitrary format can be processed
data can be read from/write to GFS(HDFS) or Bigtable(HBase)

master task coordinate all workers
computation run on set of partitions (number specified by user)
each worker handle one or more partition
worker read part of the input in parallel
input contain vertices and edges
default assignment of vertices to partitions by hashing vertex ID
user can supply assignment to conform to graph topology
more efficient to put nearby vertices on same or nearby worker task
vertices not in worker's own partition are send by message to appropriate worker

superstep execution : at each superstep, the worker
call the compute method of the vertex for each its vertices
passing in any arriving messages
send outgoing messages asynchronously to destination vertex worker
for delivery next superstep
tell master when all vertices have completed superstep
tell master the number of vertices still active

algorithm terminate when all vertices have voted to halt
and all vertex input queues are empty
upon termination, the output method of all vertices and edges
are called to write the output