2013年度
実践的並列コンピューティング
第15回(最終回)

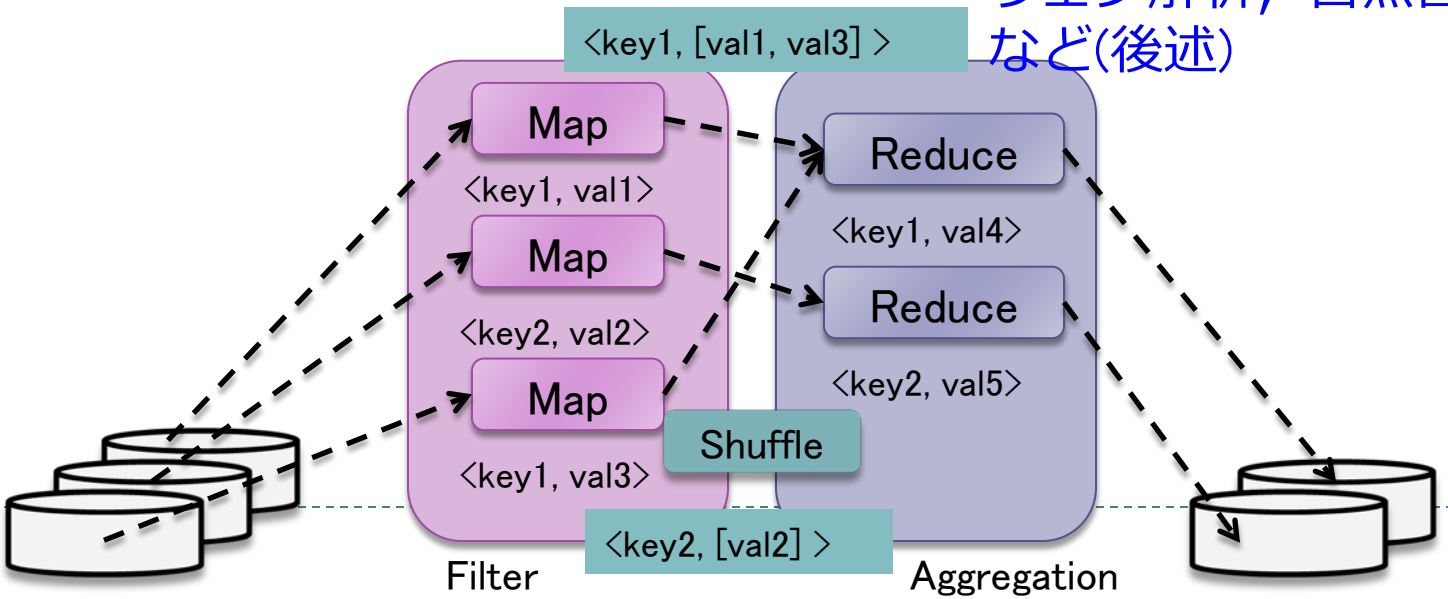
MapReduce Programming その２
2013年7月22日

遠藤敏夫 (endo@is.titech.ac.jp)
元スライド作成：佐藤仁　学術国際情報センター

MapReduce

- 2004年にGoogle社により提案された大量のデータを並列に処理するためのプログラミングモデル（とその実装）
  - Mapフェーズ
    - keyとvalueのペアから中間データとなるkeyとvalueのペアを生成する
  - Reduceフェーズ
    - 中間データから同じkeyに関連づけられたvalueを集めて処理し，結果を出力する

ウェブ解析，自然言語処理など(後述)

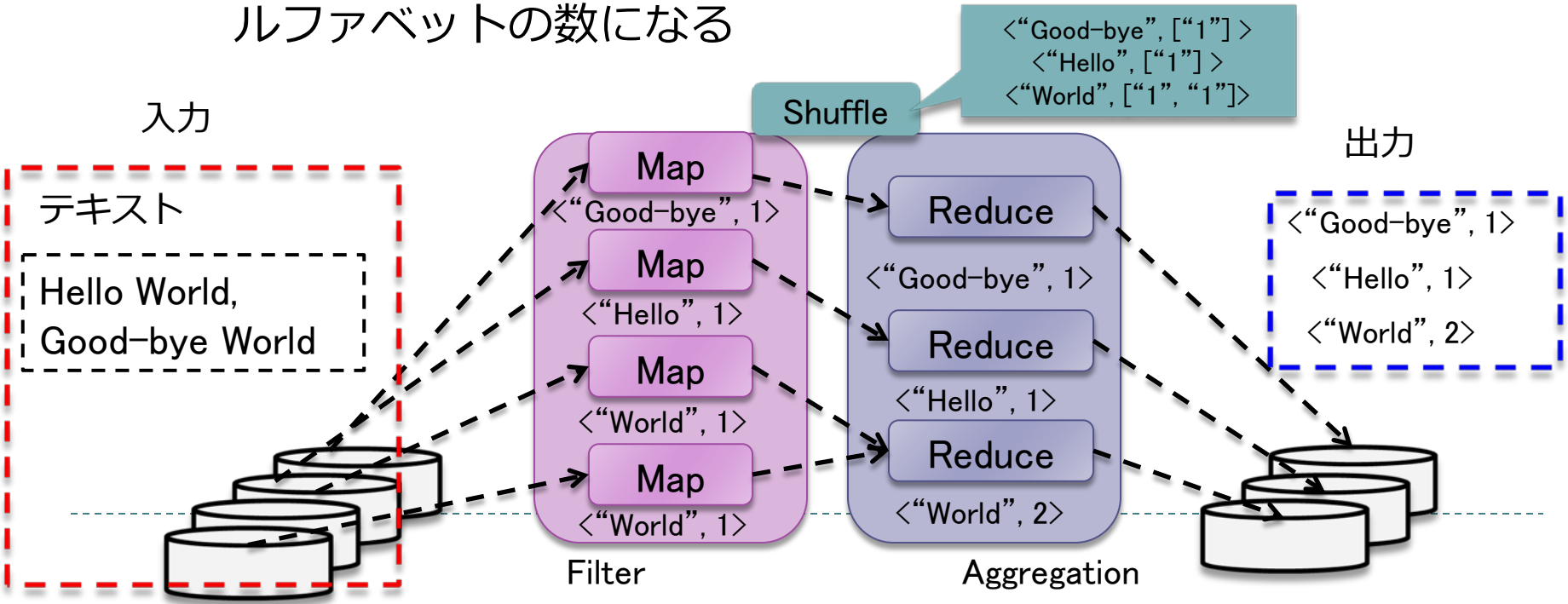


Hadoop

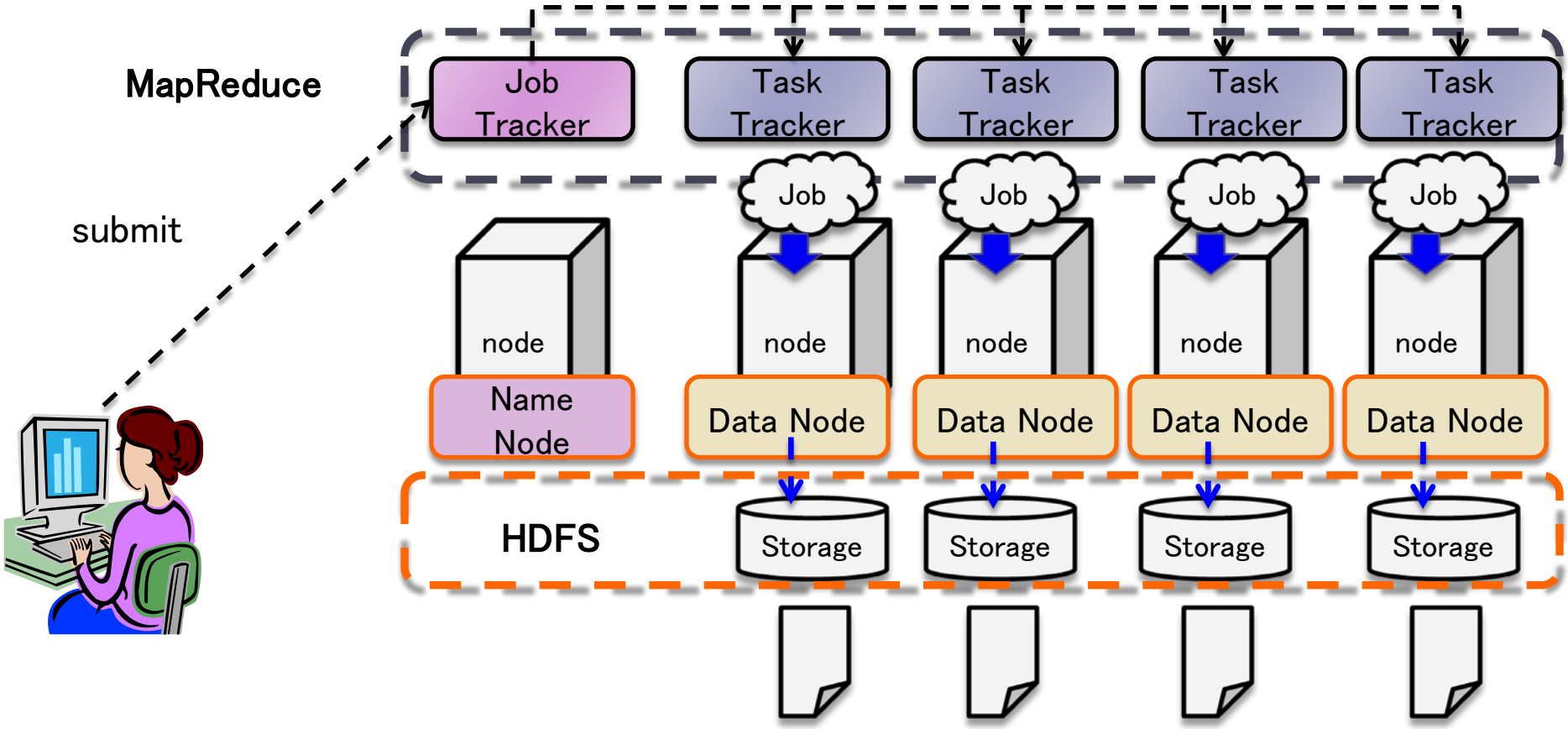
- Hadoop: MapReduceモデルの実装の一つ
  - より正確には、Hadoop MapReduce + Hadoop 分散ファイルシステム
- Tsudoop
  - Hadoop on TSUBAME
  - オリジナルのHadoopでは、利用マシン名の集合をユーザが指定する必要がある。TsudoopはHadoopとTSUBAMEのジョブスケジューラを接続する
  - GSIC佐藤仁による

HadoopでのMapReduceプログラミング
例. WordCount (単語数え)

- アルファベットの数を数えたい
  - Mapフェーズ
    - 文字列(Hello World)から，アルファベット(key)と存在を表す1(value)からなるkeyとvalueのペアを生成
  - Reduceフェーズ
    - 同じアルファベット(key)に対する1(value)の数を足し合わせるとアルファベットの数になる



Hadoopの構成


MapReduce
submit
Job Tracker
Task Tracker
Task Tracker
Task Tracker
Task Tracker
Job
Job
Job
Job
node
node
node
node
node
Name Node
Data Node
Data Node
Data Node
Data Node
HDFS
Storage
Storage
Storage
Storage

Hadoopでのプログラム実行の流れ

- プログラムを作成
  - 基本的にはJava（その他の言語でも可能）
  - Jar形式にする
    - xxxx.jar
- 実行
  - 入力データを準備する
    - FSに応じた入力場所を設定する（input）
  - ジョブのサブミッション
    - 入出力場所を引数に渡して実行

```
$ hadoop jar xxxx.jar input output
```

  - 出力データの解析
    - FSに応じた出力場所に出力される（output）

Example : WordCount.java

(>= version 0.20.0)

```
public class WordCount {

  static class WordCountMapper extends Mapper<IntWritable, Text, Text, IntWritable> {
    pubic void map(IntWritable key, Text value, Context context) {
       String line = value.toString();
       for (tokenizer.hasMoreTokens())
         context.write(new Text(line), new IntWritable(1));
    }
  }

  static class WordCountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text, Iterable<IntWritable> values, Context context) {
      int  sum = 0;
      for (IntWritable value : values)
        sum += value.get();
      context.write(key, new IntWritable(sum));
    }
  }
….
```

Example : WordCount.java (cont'd)

```
  …

  public static void main(String[] args) {
    Job job = new Job();
    job.setJarByClass(WordCount.class);

    FileInputFormat.setInputPaths(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.setMapperClass(WordCountMapper.class);
    job.setReducerClass(WordCountReducer.class);

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

プログラムの構成

- Map
  - TaskTrackerから呼び出され、1つのMapタスクとして動作
  - 渡されたKVに対して、Map処理を行い、中間KVを出力
- Reduce
  - TaskTrackerから呼び出され、1つReduceタスクとして動作
  - 渡されたKeyとValueのリストに対して、Reduce処理を行い、KVを出力
- main
  - MapReduceジョブの構成を設定し、実行

あとはシステムがやってくれる

Map

- Mapperクラスを継承
  - 実際のMap処理は、map関数を実装することで実現
    - startup() → map() → cleanup()
  - ContextにはMap処理のコンテキストが入る
    - ジョブの構成
    - どのKVを処理するか？
    - 中間KV（contextに対してwriteする）
  - 入出力のKVは, Writable Interfaceを介して行う

```
static class WordCountMapper extends Mapper<IntWritable, Text, Text, IntWritable> {
  pubic void map(IntWritable key, Text value, Context context) {
    String line = value.toString();
    for (tokenizer.hasMoreTokens())
      context.write(new Text(line), new IntWritable(1));   //← [line, 1]というKVペアを出力
  }
}
```

Writable Interface

- Writable InterfaceはHadoopでSerializationを行うためのフォーマット
- Serializationとはなにか？
  - 構造化されたオブジェクトをバイトストリームへ変換することと
  - プロセス間通信やストレージで利用
    - コンパクト、性能、拡張性、互換性などの利点
  - deserializationは、serializationの逆の操作
- Javaには既にSerializableというインタフェースがあるが、性能上の理由などによりHadoop独自のWritableを利用

```
package org.apache.hadoop.io;
import java.io.DataOutput;
import java.io.DataInput;
import java.io.IOException;

public interface Writable {
  void write(DataOutput out) throws IOException;
  void readFields(DataInput in) throws IOException;
}
```

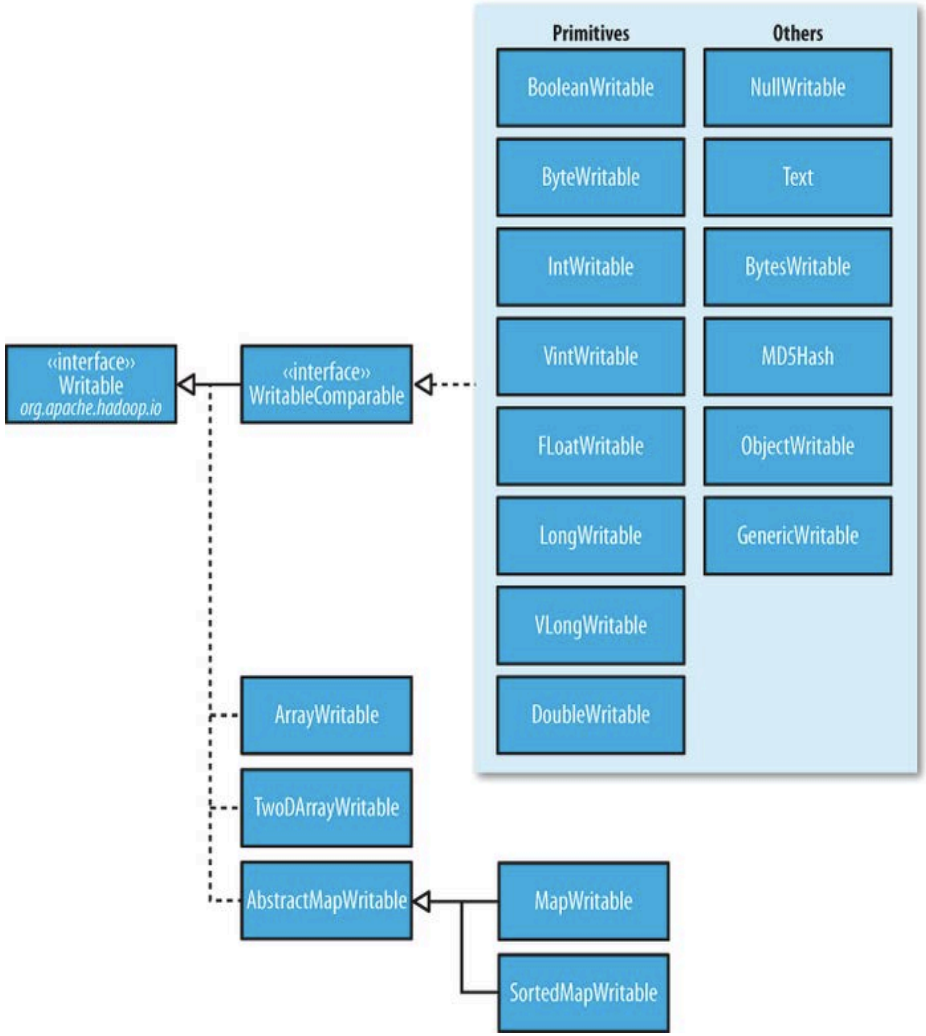
Writable Interfaceの例

- IntWritable
  - intに相当

```
IntWritable i = new IntWritable();
i.set(2010);
int j = i.get();
```

- Text
  - UTF8(String)に相当

```
Text p = new Text();
p.set("Hello World");
String q = p.getString();
```



Reduce

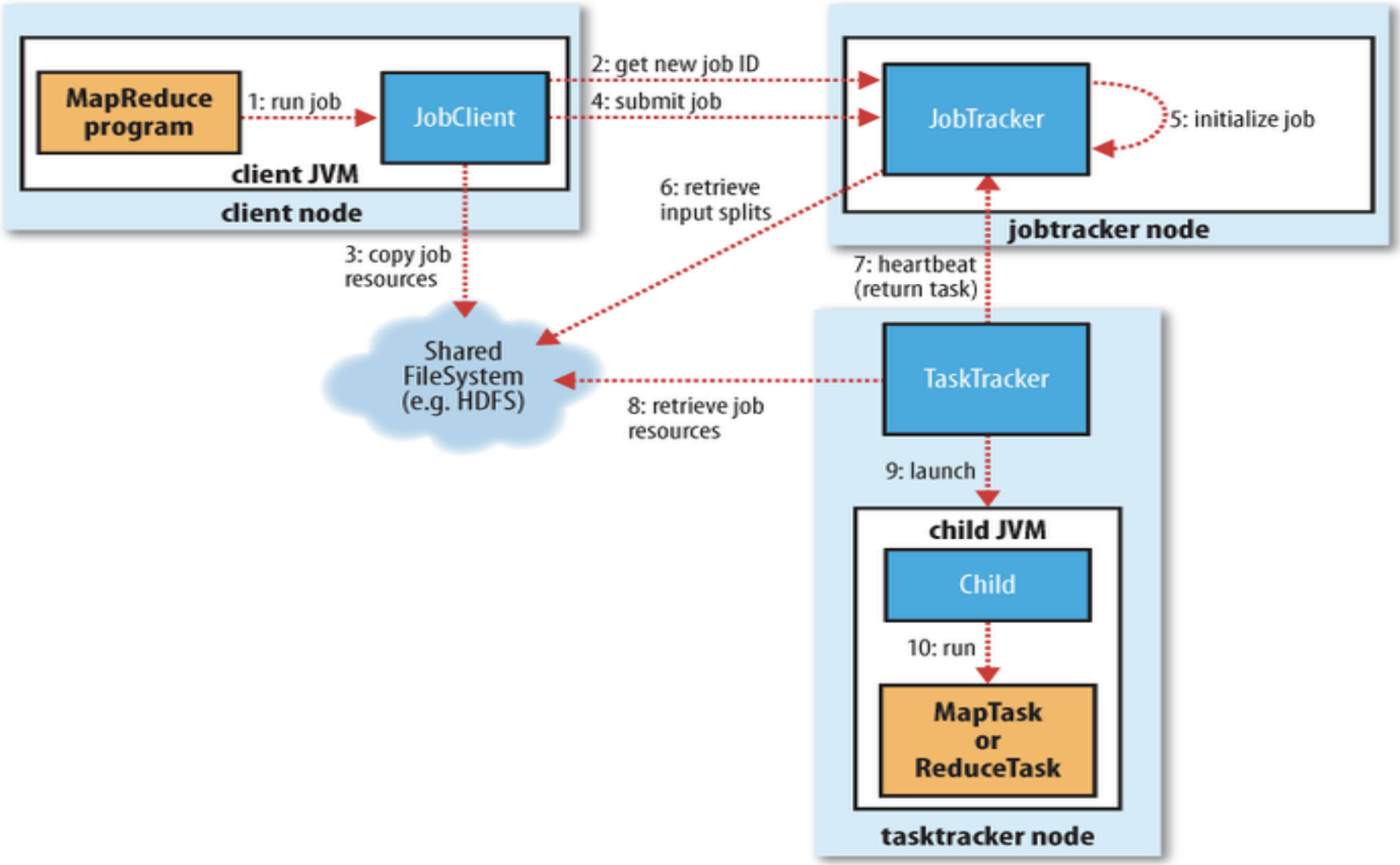
- Reducerクラスを継承
  - 実際のReduce処理は、reduce関数を実装することで実現
    - startup() → reduce() → cleanup()
    - 入力のvalueは、Iterableとして渡される
    - contextに出力KVをwriteする

```
static class WordCountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
   public void reduce(Text, Iterable<IntWritable> values, Context context) {
     int  sum = 0;
     for (IntWritable value : values)
       sum += value.get();
     context.write(key, new IntWritable(sum));
   }
 }
```

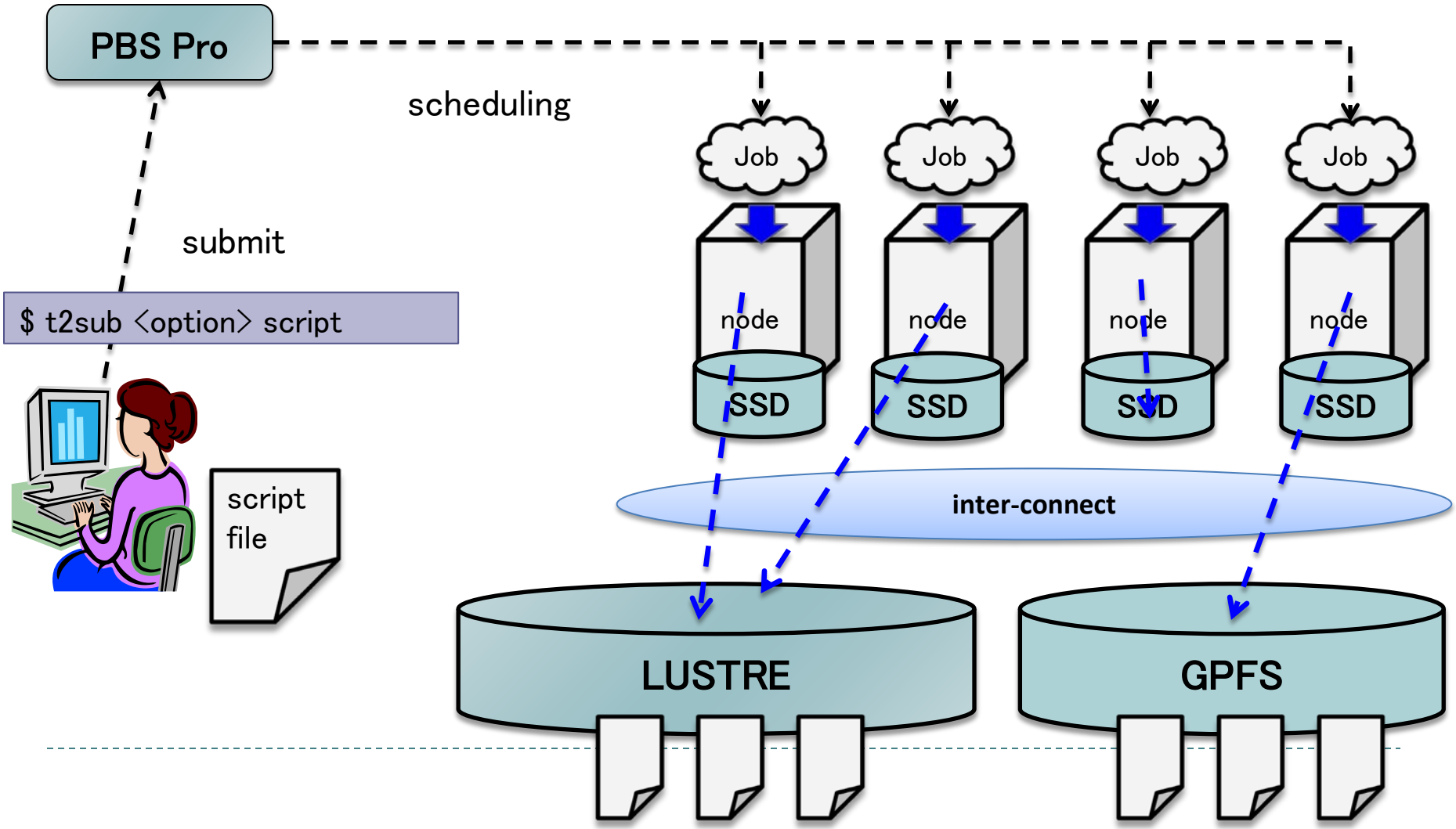
main

```
 public static void main(String[] args) {
    Job job = new Job(); // Jobクラスは、ジョブの構成、制御、Submitを担
    job.setJarByClass(WordCount.class); // Jarの中でどのクラスから開始するかを指定

    FileInputFormat.setInputPaths(job, new Path(args[0])); // データの入出力の位置
    FileOutputFormat.setOutputPath(job, new Path(args[1])); // を指定

    job.setMapperClass(WordCountMapper.class); // Mapperクラスを指定
    job.setReducerClass(WordCountReducer.class); // Reducerクラスを指定

    job.setOutputKeyClass(Text.class); // 最終出力となるkeyのクラスを指定
    job.setOutputValueClass(IntWritable.class); // 最終出力となるvalueのクラスを指定

    // ジョブをSubmitして、終了するまで待つ
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

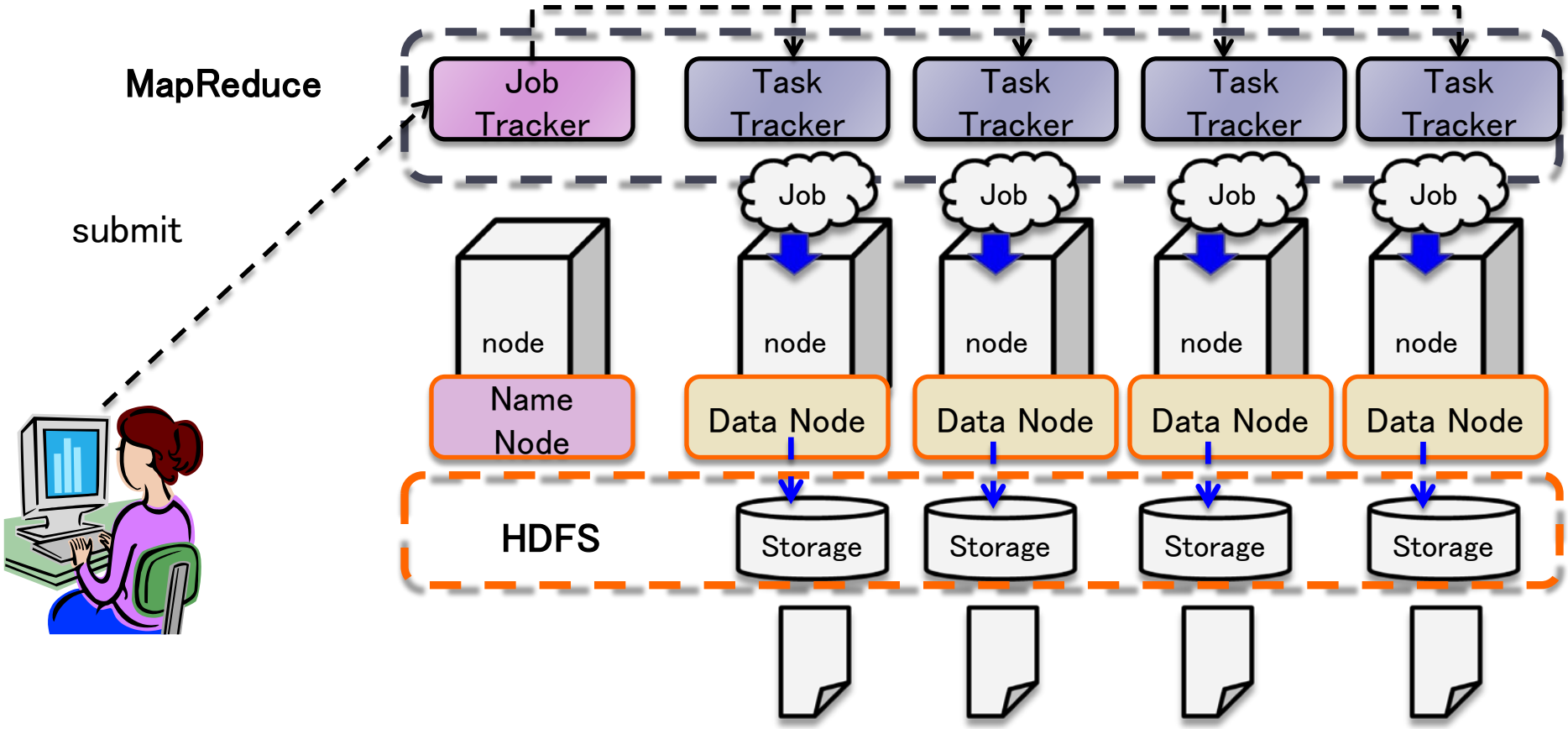
Hadoopプログラムの実行フロー



TSUBAME2.0でのジョブ実行



Hadoopの構成

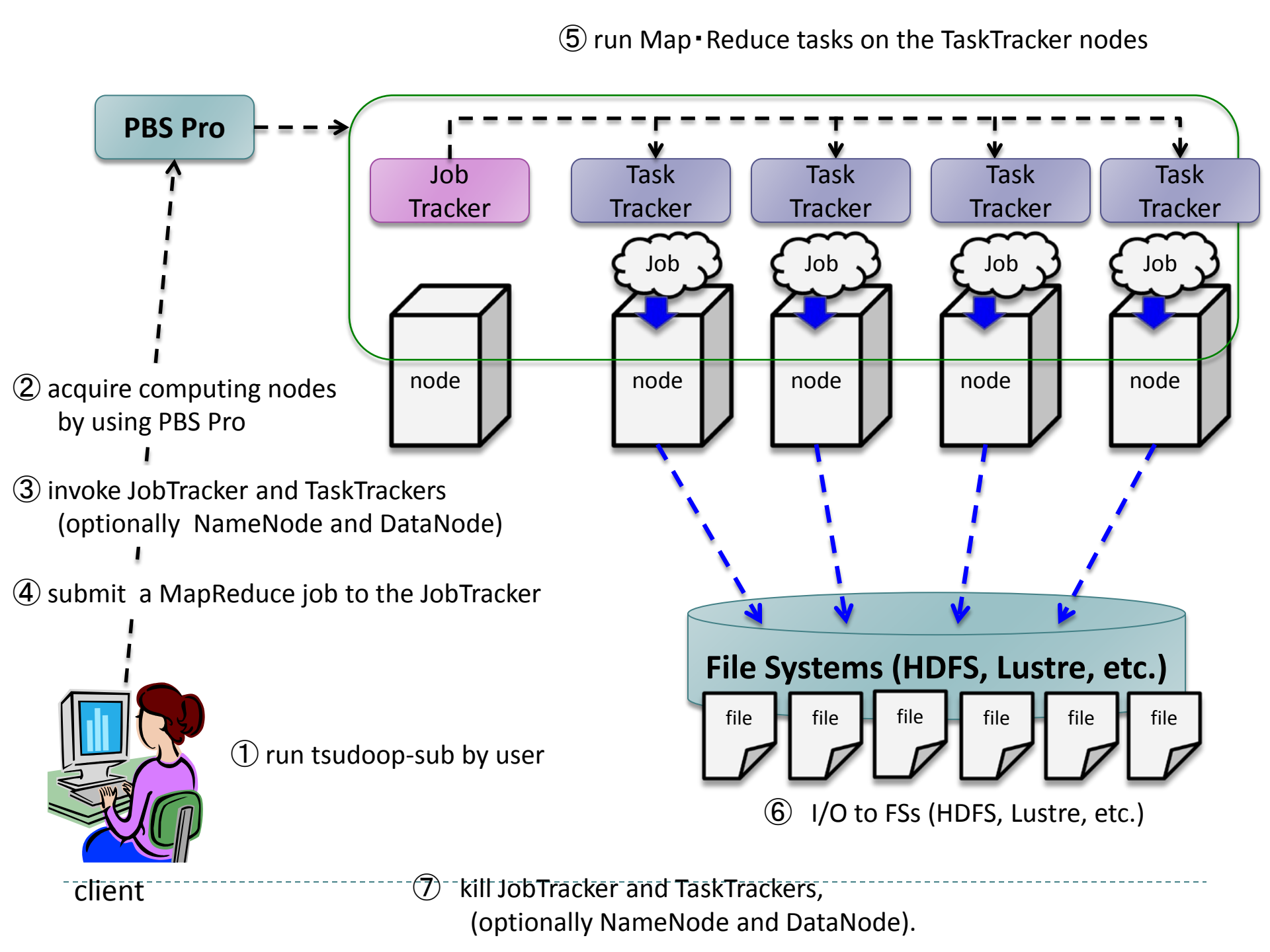

MapReduce
submit
Job Tracker
Task Tracker
Task Tracker
Task Tracker
Task Tracker
Job
Job
Job
Job
node
node
node
node
node
Name Node
Data Node
Data Node
Data Node
Data Node
HDFS
Storage
Storage
Storage
Storage

Tsudoop: Hadoop on TSUBAME

- 既存スケジューラへの対応
  - n1geからPBS Proへ変更
  - 複数ノードの取得
    - $PBS_NODELIST
  - 占有ノードにはsshログイン可
    - システムによりゾンビプロセスの削除
- ストレージは複数選択可
  - Lustre, GPFS
    - 計算ノードからクライアントアクセス
  - HDFS
    - 計算ノード以下のローカルSSDを集約して
      オンデマンドにFSを構築
    - FSに必要データをステージング

既存Hadoopの枠組みでを
そのまま対応

使い分けが必要


⑤ run Map・Reduce tasks on the TaskTracker nodes
PBS Pro
Job Tracker
Task Tracker
Task Tracker
Task Tracker
Task Tracker
Job
Job
Job
Job
node
node
node
node
node
② acquire computing nodes by using PBS Pro
③ invoke JobTracker and TaskTrackers (optionally NameNode and DataNode)
④ submit a MapReduce job to the JobTracker
File Systems (HDFS, Lustre, etc.)
file
file
file
file
file
file
① run tsudoop-sub by user
⑥ I/O to FSs (HDFS, Lustre, etc.)
client
⑦ kill JobTracker and TaskTrackers, (optionally NameNode and DataNode).

ユーザの視点

- t2subのラッパースクリプト

```
$ tsudoop-sub  <option>script
```

- PBS Proのオプション (キュー名, グループ名, ジョブ名 etc.)
- Hadoopのオプション (ノード数, ファイルシステムの選択, etc.)

```
#!/bin/bash

. $TSUDOOP_HOME/conf/tsudoop.sh

hadoop  jar hadoop-mapred-examples-*.jar sort  /input /output
```

Hadoopで書かれたサンプルプログラムを TSUBAME2.0上で実行 (1)

- TSUBAME2.0へのログイン
  - $ ssh -l taro login-t2.g.gsic.titech.ac.jp
- 作業用ディレクトリを作成、サンプルのコピー
  - $ mkdir -p ppcomp-mapred/sample （←このディレクトリ名は任意）
  - $ cd ppcomp-mapred/sample
  - $ cp -r /work0/GSIC/apps/tsudoop/sample/mapred/wc .
  - $ ls input （← 入力データを見てみる）
- 環境設定
  - $ export PATH=/work0/GSIC/apps/tsudoop/bin:$PATH
  - $. sample-settings.sh

Hadoopで書かれたサンプルプログラムをTSUBAME2.0上で実行 (2)

- コンパイル
  - $ mkdir wordcount_classes
  - $ javac -d wordcount_classes/ WordCount.java
  - $ jar cvf wordcount.jar -C wordcount_classes/ . （←最後にピリオド）
- ジョブ投入
  - $ tsudoop-sub -h
  - $ tsudoop-sub -n 4 -g t2g-ppcomp wc.sh
    - -nはTaskTrackerのノード数
    - -gは課金グループ
    - ノード数は最大8ぐらいまでにしておいてください。
- 出力データのディレクトリをみてみる
  - $ ls output
  - $ less output/part-r-00000 （←ファイル名はケースバイケース）

Hadoopで書かれたサンプルプログラムをTSUBAME2.0上で実行 (3)

- 結果確認
  - $ less OTHERS.o(Job ID)　(←標準出力)
  - $ less OTHERS.e(Job ID)　(←標準エラー出力)

```
        :
13/07/19 17:40:19 INFO input.FileInputFormat: Total input paths to process : 15
                ↑ジョブ開始時刻
13/07/19 17:40:20 INFO mapred.JobClient: Running job: job_201307191739_0001
13/07/19 17:40:21 INFO mapred.JobClient:  map 0% reduce 0%
13/07/19 17:40:36 INFO mapred.JobClient:  map 40% reduce 0%
13/07/19 17:40:39 INFO mapred.JobClient:  map 100% reduce 0%
13/07/19 17:40:48 INFO mapred.JobClient:  map 100% reduce 100%
13/07/19 17:40:53 INFO mapred.JobClient: Job complete: job_201307191739_0001
13/07/19 17:40:53 INFO mapred.JobClient: Counters: 23
                ↑ジョブ終了時刻
        :
```

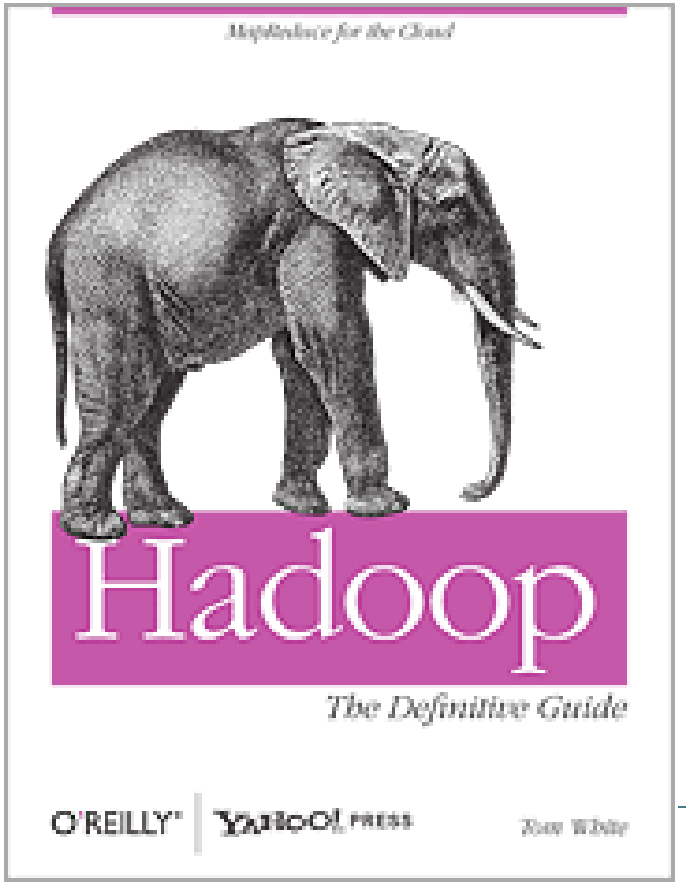
ジョブ開始時刻と終了時刻の差が、ジョブ実行にかかった時間

参考文献

- MapReduce: Simplified Data Processing on Large Clusters
  - Jeffrey Dean et al.
  - http://labs.google.com/papers/mapreduce.html
- The Google File System
  - Ghemawat et al.
  - http://labs.google.com/papers/gfs.html

参考文献 (cont’d)

- タイトル　Hadoop: The Definitive Guide
  著者 Tom Whilte
  出版元 O’Reilly Media



本授業のレポートについて

- 基礎編から一問＋応用編から一問、計二問のレポート提出を必須とします
- 基礎編
  - OpenMP+MPIの、1.～3.の中から一問以上
- 応用編
  - GPUプログラミング
  - Map-Reduce

  } この中から一問以上
- それぞれの編で二問以上を提出してもよい

応用編(MapReduce)課題説明 (1)

応用編レポートの〆切は8/8(木)

以下のM1, M2, M3、またはGPU編課題(G1, G2, G3)の、いずれかについてレポートを提出してください.

[M1] WordCountプログラムの性能を、入力データサイズや利用ノード数などを変化させながら性能評価してください。

- TSUDOOP (Hadoop on TSUBAME)を利用するか、他に各自用意したHadoopシステムでもよい
- プログラムを改良してもok
- 提出時には入力データはメールしないでください(巨大なので)

応用編(MapReduce)課題説明 (2)

[M2] 自由課題：任意のプログラムを、Hadoop を用いて並列化し、評価してください

- TSUDOOP (Hadoop on TSUBAME)を利用するか、他に各自用意したHadoopシステムでもよい
- たとえば，自分が研究している問題

応用編(MapReduce)課題説明 (3)

[M3] MapReduceプログラミングモデルと、MPIプログラミングモデルについて論じ、A4 1ページ以上にまとめてください

- 共通点・相違点について議論すること
  - たとえば、ユーザプログラマの記述する範囲、動作するアーキテクチャモデル、通信の方法、(システムレベルでの)マスタープロセスの有無…
- MPIの代わりに他のOpenMPなどと比べてもok
- PDFやWordをメールで提出

課題の注意

- いずれの課題の場合も，レポートに以下を含むこと([M3]をのぞく)
  - 計算・データの割り当て手法の説明
  - TSUBAME2などで実行したときの性能
    - スレッド数、スレッドブロック数、GPU数を様々に変化させたときの変化に触れているとなお良い
    - 問題サイズを様々に変化させたとき(可能な問題なら)
  - 高性能化のための工夫が含まれているとなお良い
    - 「XXXのためにXXXをしてみたが高速にならなかった」のような失敗でも可
  - プログラムについては，zipなどで圧縮して添付
    - 困難な場合，TSUBAME2の自分のホームディレクトリに置き，置き場所を連絡

最終回です。おつかれさまでした

- 応用編レポート〆切 8/8(木)
- 8/9より一週間TSUBAMEが止まりますので注意

- この秋、TSUBAME2.0 ⇒ TSUBAME2.5に進化します
  - 「TSUBAME2.5」で検索
  - GPUの性能が約3倍に。Tesla M2050⇒Tesla K20X
  - システム全体の理論演算性能は、
    - 倍精度で2.4PFlops ⇒ 5.7PFlops
    - 単精度で4.8PFlops ⇒ 17PFlops