Perspectives in parallel programming

MARCO DANELUTTO
PHD COURSE
MAY 2015
Exams

Teach a lesson on an argument agreed with the professor
- Just seen S-NET
- More today

Project
- Implemented with a structured parallel programming framework
- More in a while
Arguments for the lessons

Idea
◦ Take an argument
◦ Read available documentation/papers
◦ Summarize (with the viewpoint given by the course)

Lesson
◦ 45 mins with: Concepts, Experiments, Assessments & links/analogies/differences w.r.t. structured parallel programming
S-Net

http://snet-home.org/

S-Net is a declarative coordination language for describing streaming networks of asynchronous application components. Components are combined into larger streaming networks using an expression language. It features five network combinators as operators: serial composition, parallel composition, serial replication, parallel replication and feedback. With the exception of serial composition, the combinators come in two flavours each: the deterministic versions preserve the order of data on streams, whereas non-deterministic variants trade this property for improved throughput. Two additional primitive components serve housekeeping and synchronisation purposes. Streams are associated with record types: collections of data where each item is uniquely identified by its name. Structural subtyping on records directs the flow of data through the streaming network.

If you would like to know more about S-Net, we recommend you to read the latest edition of our technical report available for download under documents.
Storm (https://storm.apache.org/)

Apache Storm is a free and open source distributed realtime computation system. Storm makes it easy to reliably process unbounded streams of data, doing for realtime processing what Hadoop did for batch processing. Storm is simple, can be used with any programming language, and is a lot of fun to use!

Storm has many use cases: realtime analytics, online machine learning, continuous computation, distributed RPC, ETL, and more. Storm is fast: a benchmark clocked it at over a million tuples processed per second per node. It is scalable, fault-tolerant, guarantees your data will be processed, and is easy to set up and operate.

Storm integrates with the queueing and database technologies you already use. A Storm topology consumes streams of data and processes those streams in arbitrarily complex ways, repartitioning the streams between each stage of the computation however needed. Read more in the tutorial.
public static class ExclamationToken extends BaseRichBolt {
    OutputCollector _collector;

    public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
        _collector = collector;
    }

    public void execute(Tuple tuple) {
        _collector.emit(tuple, new Values(tuple.getString(0) + "!!!"));
        _collector.ack(tuple);
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}
Spark (https://spark.apache.org/)

Apache Spark™ is a fast and general engine for large-scale data processing.

Speed
Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.

Spark has an advanced DAG execution engine that supports cyclic data flow and in-memory computing.
Spark sample code

Word Count

In this example, we use a few more transformations to build a dataset of (String, Int) pairs called counts and then save it to a file.

Python Scala Java

```python
JavaRDD<String> textFile = spark.textFile("hdfs://...");
JavaRDD<String> words = textFile.flatMap(new FlatMapFunction<String, String>() {
    public Iterable<String> call(String s) { return Arrays.asList(s.split(" ")); }
});
JavaPairRDD<String, Integer> pairs = words.mapToPair(new PairFunction<String, String, Integer>() {
    public Tuple2<String, Integer> call(String s) { return new Tuple2<String, Integer>(s, 1); }
});
JavaPairRDD<String, Integer> counts = pairs.reduceByKey(new Function2<Integer, Integer, Integer>() {
    public Integer call(Integer a, Integer b) { return a + b; }
});
counts.saveAsTextFile("hdfs://...");
```

```scala

```
And more ...

Spark Streaming makes it easy to build scalable fault-tolerant streaming applications.

Ease of Use
Build applications through high-level operators.

Spark Streaming brings Spark’s language-integrated API to stream processing, letting you write streaming jobs the same way you write batch jobs. It supports Java, Scala and Python.

Spark Streaming Programming Guide

- Overview
- A Quick Example
- Basic Concepts
  - Linking
  - Initializing StreamingContext
  - Discretized Streams (DStreams)
  - Input DStreams and Receivers
  - Transformations on DStreams
  - Output Operations on DStreams
  - DataFrame and SQL Operations
  - MLlib Operations
  - Caching / Persistence
  - Checkpointing
  - Deploying Applications
  - Monitoring Applications
- Performance Tuning
  - Reducing the Batch Processing Times
  - Setting the Right Batch Interval
  - Memory Tuning
- Fault-tolerance Semantics
- Migration Guide from 0.9.1 or below to 1.x
- Where to Go from Here
**StreamIt** is a programming language and a compilation infrastructure, specifically engineered for modern streaming systems. It is designed to facilitate the programming of large streaming applications, as well as their efficient and effective mapping to a wide variety of target architectures, including commercial-off-the-shelf uniprocessors, multicore architectures, and clusters of workstations.

For more information about StreamIt, please visit the [research overview page](http://groups.csail.mit.edu/cag/streamit/index.shtml) and review the list of publications and related materials. The StreamIt compiler and benchmarks are available for download.
Filter Example: LowPassFilter

```c
float->float filter LowPassFilter (int N, float freq) {
    float[N] weights;
    init {
        weights = calcWeights(N, freq);
    }
    work peek N pop 1 push 1 {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * peek(i);
        }
        push(result);
        pop();
    }
}

Composing Filters: Structured Streams

- Hierarchical structures:
  - Pipeline
  - SplitJoin
  - Feedback Loop

- Basic programmable unit: Filter
Scala (http://www.scala-lang.org/)
Scala

Parallel and Concurrent Programming

- Futures and Promises  NEW IN 2.10
- Scala’s Parallel Collections Library
  - Overview
  - Concrete Parallel Collection Classes
  - Parallel Collection Conversions
  - Concurrent Tries
  - Architecture of the Parallel Collections Library
  - Creating Custom Parallel Collections
  - Configuring Parallel Collections
  - Measuring Performance
- The Scala Actors Migration Guide  NEW IN 2.10
- The Scala Actors API  DEPRECATED

trait

functions

| GOOD | def f(x: Int) = { x*x } |
| BAD | def f(x: Int) = { x*x } |
| GOOD | def f(x: Any) = println(x) |
| BAD | def f(x) = println(x) |

type R = Double

| define function | hidden error: without = it's a Unit |
| define function | syntax error: need types for every parameter |
| type alias | call-by-value |
| call-by-name (lazy parameters) | anonymous function |

Anonymous function: underscore

Anonymous function: to use and define in the same line

Anonymous function: bound in lambda

Anonymous function: block style

Anonymous functions: pipeline

Anonymous functions: to pass

Anonymous functions: currying, obvious syntax.

| val zscore = (mean:R, sd:R) => |
| (x:R) => (x-mean)/sd |
| val zscore(mean:R, sd:R) = (x:R) |
| => (x-mean)/sd |
Projects

Any application (your own choice)

With a structured parallel programming framework
  ◦ FastFlow

Applying the full methodology
Simple example: filtering images

Time = 288s

4-stage pipeline

Time = 112s

replicating Blur & Emboss functions by using task-farm

Time = 71s

farm with pipeline workers

Time = 75s

optimizing resources

Time = 34s

parallelizing I/O too

Time = 33s

2 Xeon E5-2695 @ 2.4GHz, 1 disk storage
Stream+Data-Parallel: Video de-noising, possible parallelization options using patterns

Sample applications

Bio*
- Sequence alignment

Video
- Stereo matching (real time)

Network
- Packet filtering

Numeric
- Matrix decomposition

Graphic
- PDF rendering

Parallel programming
- Exploration of skeleton rewriting space

Simulation
- Parallel agent system

Genetic algorithms
- Parallel Sudoku
Exams: when, how

Step 1: agree argument (lesson or application/project)
Step 2: prepare lesson (project)
Step 3: submit

Step 4: approval → presentation to the colleagues

◦ I’m available up to end of July, then from beginning of September