Implicit Parallelism in a Functional Subset of Scala

Scala Workshop, July 1, 2013

Shams Imam, Robert Cartwright Jr., Vivek Sarkar
Rice University

Introduction

- Advent of multi-core processors
- Renewed interest in parallelism
- Scala promotes functional programming
- Identifying and exploiting parallelism in functional Scala programs

Goal

Start from a

- Single-assignment (functional) subset of Scala

to

- Implicitly parallelize programs, and
- Achieve better performance

using

- Cooperative futures and accumulators

Outline

- Introduction
- Background: Sisal and Dr. Java
- Functional (subset of) Scala
- Parallel Constructs
- Parallelization Optimizations
- Preliminary Results

SISAL

- Single-assignment programming language with value-oriented semantics
- Compile time type checking with limited (intraprocedural) type inferences
- Sub-expressions may be evaluated in any order without effect on computed results
  - data dependences must be respected
- Sequential while expression with distinct names for “old” and “new” values of loop variables

Functional subset of Java

- Supported as the functional language level in DrJava – a pedagogical IDE
- Programs written in the functional subset of Scheme can be easily expressed
- Immutable fields and local variables
- Functional Level classes
  - Similar to Scala case classes
- Functional Java programs are strictly sequential

Outline

- Introduction
- Background: Sisal and Dr. java
- Functional (subset of) Scala
- Parallel Constructs
- Parallelization Optimizations
- Preliminary Results

Functional subset of Scala

- Disallow imperative statements that violate the single assignment property
- Allow immutable collections and operations on them like fold, map, filter, etc.
- Immutable arrays
  - no explicit element assignment
- Case classes
- Disallow explicit creation of threads

Quicksort

```
def quicksort[T](
  list: List[T]): List[T] = {
 if (list.isEmpty) {
  list
 } else {
   val x = list.head
   val xs = list.tail
   val (lP, rP) = xs partition(_ < x)
   val left = quicksort(lP)
   val right = quicksort(rP)
  left ++ (x :: right)
 }
}
```

Matrix Multiplication

```
def multiplySeq(
   a: Array2d, b: Array2d): Array2d = {
  val (T, R, C) =
   (a(0).length, a.length, b(0).length)
  val res = Array.tabulate(R, C) {
   (row, col) =>
    (0 until T).foldLeft(0.0) {
     (acc, i) =>
      acc + (a(row)(i) * b(i)(col))
    }
  }
  res
}
```

Functional subset of Scala

- Programs expose abundant opportunity for safe implicit fine-grain parallelism at the expression level
- Abundant opportunities for the compiler and runtime to exploit the parallelism on a given platform

Outline

- Introduction
- Background: Sisal and Dr. java
- Functional (subset of) Scala
- Parallel Constructs
- Parallelization Optimizations
- Preliminary Results

Parallel Constructs

- Lack of side-effects in functional programs makes it suitable for parallelism
- Immutable semantics guarantees the absence of data races
- Data dependences are explicit and can be computed using a simple dataflow analysis

Futures

- Represents a single-producer multiple-consumer pattern
- Computed asynchronously by a producer
- Consumers suspend until the value of the future is available
- Cooperative implementation ensures no blocking of threads
  - No explicit event-driven style code with callback registration!

Quicksort using Futures

```
def quicksort[T](
  list: List[T]): List[T] = {
 if (list.isEmpty) {
  list
 } else {
  val x = list.head
  val xs = list.tail
  val (lP, rP) = xs partition(_ < x)
  val leftFuture = async quicksort(lP)
  val rightFuture = async quicksort(rP)
  leftFuture.resolve() ++
    (x :: rightFuture.resolve())
 }
}
```

For Comprehensions

- Parallel forall loop that returns an array
- Elements initialized to the results of the individual iterations

Parallel Matrix Multiplication

```
def multiplySeq(
  a: Array2d, b: Array2d): Array2d = {
  ...
  val res: Array2d = forall(R, C) {
    (row, col) =>
      (0 until T).foldLeft(0.0) {
        ...
      }
  }
  ...
}
```

Scalar Reductions - Accumulators

- Parallelize associative and/or commutative operations
- Implemented using the finish accumulator construct for user-defined deterministic reductions
  - Results are always deterministic

Matrix Multiplication - Accumulator

```
def multiplySeq(
  a: Array2d, b: Array2d): Array2d = {
 ...
 val res: Array2d = forall(R, C) {
  (row, col) =>
   val acc = new Accumulator(0.0)(_+_)
   forall(T)(acc) {
    acc.put(a(row)(i) * b(i)(col))
   }
   acc.get() // return the result
 }
 ...
}
```

Outline

- Introduction
- Background: Sisal and Dr. java
- Functional (subset of) Scala
- Parallel Constructs
- Parallelization Optimizations
- Preliminary Results

Parallelization Optimizations

- Benefits depend on granularity
- Fine granularity
  - Overhead of managing the parallelism offsets any performance benefits
- Coarse granularity
  - Load balancing issues

Loop Chunking

- Each task in the forall loop consists of multiple iterations instead of a single iteration
- Increases computation performed in each task
- Works in the presence of synchronization constraints such as futures
- Number of chunks in a forall is divisible by number of workers

Parallelism Thresholds

- Use thresholds to determine when a fragment of work is too small to benefit from parallelization
- Implemented using ‘synchronous’ futures

Quicksort - Thresholds

```
def quicksortParallel[T](
  list: List[T]): List[T] = {
 if (list.isEmpty) {
  list
 } else {
   val x = list.head
   val xs = list.tail
   val (lP, rP) = xs partition(_ < x)
   val leftFuture = asyncSeq(
     someThresholdCond) {
    quicksortSerial(lP)
  } {
    quicksortParallel(lP)
  }
   val rightFuture = ...
  leftFuture.resolve() ++
    (x :: rightFuture.resolve())
 }
}
```

Redundant Futures

- ‘Redundant’ futures are evaluated almost immediately
  - E.g. rightFuture in Quicksort
- Reduces the number of tasks spawned
- Minimizes the synchronizations dynamically handled by the runtime

Outline

- Introduction
- Background: Sisal and Dr. java
- Functional (subset of) Scala
- Parallel Constructs
- Parallelization Optimizations
- Preliminary Results

Experimental Setup

- 12-core (two hex-cores) 2.8 GHz Intel Westmere SMP node
- 48 GB of RAM per node (4 GB per core)
- Red Hat Linux (RHEL 6.0)
- Each core has a 32 kB L1 cache and a 256 kB L2 cache.
- Java Hotspot JDK 1.7
- Scala 2.10.1
- Scala-Sisal 1.0-SNAPSHOT
- Hand-optimized Scala-Sisal programs

Matrix Multiplication

- Input matrix of size 1500 × 1500
- Execution times are reported in milliseconds

Sequential Execution Time:				78471.67
	Our Implementation		Parallel Collections	
Threads	Time	Speed-up	Time	Speed-up
1	79770.04	0.984	90856.94	0.864
2	41847.58	1.875	43510.11	1.804
4	22967.85	3.417	23172.42	3.386
6	16254.40	4.828	18122.74	4.330
8	13252.36	5.921	14493.94	5.414
10	11626.78	6.749	13599.06	5.770
12	11929.76	6.578	12534.55	6.260

Smith Waterman

- Tile size of 640
- Input strings of length 37120 × 38400
- Execution times are reported in milliseconds

Sequential Execution Time:		30020.76
Threads	**Execution Time**	**Speed-up**
1	30191.27	0.994
2	15186.69	1.977
4	7601.52	3.949
6	5096.49	5.890
8	3857.09	7.783
10	3119.40	9.624
12	2643.88	11.355

Quicksort

- Input array size of 4000000
- Recursive call depth threshold of 4
- Execution times are reported in milliseconds

Sequential Execution Time:		5908.45
Threads	**Execution Time**	**Speed-up**
1	7425.84	0.796
2	3979.64	1.485
4	3029.23	1.950
6	2622.73	2.253
8	2227.03	2.653
10	2915.70	2.026
12	2290.29	2.580

Summary

- Implicit parallelism of
  - A functional subset of Scala
  - Using insertion of futures and accumulators
  - Using a cooperative runtime for performance
  - Catalogued our Parallelization Optimizations
- Future work
  - Compiler plugin to Verify Functional nature of programs
  - Compiler plugin to Parallelize the program using mentioned Optimizations.

Acknowledgments

- Other members of the Habanero Group
    - Vincent Cavé
    - Dragoș Sbîrlea
    - Yunming Zhang

Thank you!

```
import audience.questions.*
```

[image source: http://www.jerryzeinfeld.com/tag/question-of-the-day/]

BACKUP SLIDES START HERE