Haskell vs. F# vs. Scala

High-level Language Features and Parallelism Support

Prabhat Totoo, Pantazis Deligiannis, Hans-Wolfgang Loidl

Denpendable Systems Group,
School of Mathematical and Computer Sciences,
Heriot-Watt University

EdLambda

8 May 2012

About this talk

- performance and programmability assessment of parallel programming support in Haskell, F# and Scala

- we will look at:
  - the languages - general features, parallel support, etc
  - some experiments - solving the n-body problem
  - some preliminary results
  - conclude - discussion of performance, programmability and pragmatic aspects

Motivation

- Functional languages offer many advantages which we can't ignore
  - referential transparency / side effects
  - expressive power
  - high-level abstractions - HOFs, compositionality
  - good match for parallelism - functions can usually be evaluated in any order; no races without mutable variables
- "specify **what** instead of **how** to compute something "
- SPJ: "The future of parallel is declarative "

Functional languages

- enables lots of things and makes a lot of things more difficult
- usual complaints
  - cannot do anything *useful* with pure functions only. side-effecting computation?
  - workaround in Haskell: handling of states through monads
- imperative/OO languages simply allow mutable states - not good for parallelism

Recent trends

- mainstream languages integrating functional features in their design
    - e.g. Generics (Java), lambda expression (C#, C++)
    - improves expressivity in the language by providing functional constructs
- Multi-paradigm languages - make func prg more approachable
    - F#
      - Microsoft .NET platform
      - functional-oriented; with imperative and OO constructs
    - Scala
      - JVM
      - emphasize on OO but combined with powerful functional features

THE LANGUAGES

Haskell[1]

- *pure* functional language, generally functions have no side-effects
- *lazy* by default
- typing: static, strong, type inference
- advanced type system supporting ADTs, typeclasses, type polymorphism
- monads used to:
    - chain computation (no default eval order)
    - IO monad separates pure from side-effecting computations
- main implementation: GHC
  - highly-optimised compiler and RTS

---

[1] `http://haskell.org/`

Haskell - parallel support

- includes support for *semi-explicit* parallelism through GpH extension for shared-memory via GHC-SMP (distributed-memory support via GUM not included)
- GpH:
  - provides a single primitive for parallelism: `par`
    `par :: a -> b -> b`
  - and a second primitive to enforce sequential ordering which is needed to arrange parallel computations: `pseq`
    `pseq :: a -> b -> b`
  - just annotate expr that could be usefully evaluated in parallel

Haskell - parallel support

- GpH example:
    - sequential : `s1 + s2`
    - in parallel : `s1 ‘par‘ s2 ‘pseq‘ (s1 + s2)`
- Evaluation strategies
    - building on top of the primitives
    - provides even higher-level abstraction by separating coordination aspects from main computation
    - e.g. `rpar` is used to specify evaluation order i.e. in parallel, and `rseq`, `rdeepseq` to specify degree (WHNF or NF)
- parallel map using strategies

```
map f xs ‘using‘ parList rdeepseq
```

Haskell - parallel support

- Other options:
  - `Par` monad
    - more explicit approach
    - uses `IVars`, `put` and `get` for communication
    - abstract some common functions e.g. parallel map
  - DPH
    - exploits data parallelism
  - Eden
    - for distributed-memory but good performance on shared-memory machines[*]
    - uses process abstraction

---

[*] Parallel Haskell implementations of the n-body problem.

F#[2]

- functional-oriented language
- combines imperative/OO features in the language
- syntax:
  - functional - influenced by Caml;
  - imperative/OO by C#, OCaml
- part of supported languages alongside C# and VB in the .NET framework
- can make use of arbitrary .NET libraries from F#
- strict by default; F# has lazy values as well
- advanced type system: discriminated union (=ADT), object types (for .NET interoperability)

---

[2] `http://research.microsoft.com/fsharp/`

F#

- *value* vs. *variable*
  - by default immutable but can use `mutable` keyword to denote variable whose value is allowed to change (using the left arrow operation `<-`)
- structures: list, seq, array, LazyList
- computation expressions *are* monads
- implementations: F#/.NET, F#/Mono

F# - parallel support

- benefits from the .NET Parallel Extensions infrastructure
  - high-level constructs to write/execute parallel programs
- Tasks Parallel Library - TPL
  - hide low-level thread creation, management, scheduling details
  - Tasks
    - main construct for task parallelism

    `Task`: provides high-level abstraction compared to working directly with threads; does not return a value

    `Task<TResult>`: represents an operation that calculates a value of type `TResult` eventually i.e. a future

F# - parallel support

- Tasks Parallel Library - TPL
  - Parallel Class - for data parallelism
    - basic loop parallelisation using `Parallel.For` and `Parallel.ForEach`
  - PLINQ - declarative model for data parallelism
    - uses tasks internally
- Async Workflows
  - use the `async {...}` keyword; doesn't block calling thread - provide basic parallelisation
  - intended mainly for operations involving I/O e.g. run multiple downloads in parallel

Scala[3]

- modern multi paradigm language: OOP + Functional + Imperative
- evaluation: strict; typing: static, strong and inferred
- full operability with Java; targeted for JVM
- designed towards extensibility: new features easily added without changing the core language syntax
- strong industrial following: Twitter, LinkedIn, the Guardian, FourSquare, Sony, etc.
- the Scala team recently won the 5 years Popular Parallel Programming European Research Grant

---

[3] `http://www.scala-lang.org/`

Scala - Parallel Collections Framework

- sophisticated hierarchy: traversable → iterable → seq/map/set → mutable/immutable
- uniform approach for interacting with multiple collections
- apply **par** on a sequential collection to invoke its corresponding parallel implementation
- apply **seq** on the parallel collection to return the sequential one
- backend: thread pool implementation that efficiently schedules fork/join tasks among available processors
- Example:
  - sequential: `xs.map( (x:  Int) => x + 1 )`
  - parallel: `xs.par.map( (x:  Int) => x + 1 ).seq`

Scala - Actors

- similar concurrency model to Erlang
- lightweight processes communicating with asynchronous messages
- messages stored in mailboxes, then iterated with pattern matching
- responses: create new actor; send new msg; change behaviour; etc.
- Example:
  - Send: `a!msg`
  - Receive: `case msg_pattern_1 => action_1`
- Akka Framework[4]
  - toolkit and runtime based on Scala actors
  - targets highly concurrent, distributed, and fault tolerant event-driven applications on the JVM

---

[4] http://akka.io/

Summary table

Table: Summary of language features.

	Key Features	Support for Parallelism
Haskell	pure functional; lazy; static, strong, inferred typing	par and pseq/ Evaluation strategies
F#	functional oriented + OO + imperative; strict, static, strong, inferred typing	Async Workflows/TPL/PLINQ
Scala	functional + object oriented + imperative; strict; static, strong, inferred typing	Parallel Collections/Actors

Syntax:

- Haskell
  ```
  sumOfSquares nums = sum $ map sqr nums
  ```
- F#
  ```
  let sumOfSquares nums = nums |> List.map sqr |> List.sum
  ```
- Scala
  ```
  def sumOfSquares(nums) = nums.map(sqr).sum
  ```

EXPERIMENTS

n-body problem

- problem of predicting and simulating the motion of a system of N bodies that interact with each other gravitationally
- Body = Position, Velocity and Mass
- simulation proceeds over a specified number of time steps
  each step: calc the accel of each body wrt the others; update position and velocity
- solving methods:
  - all-pairs: direct body-to-body comparison, not feasible for large N
  - **Barnes-Hut algorithm**: efficient approximation method, more advanced
    2 phases:
    - tree construction
    - force calculation (most compute-intensive)

Approach

- start off from highly-tuned Haskell implementation
- translate to F# and Scala
- general steps:
    - start with initial seq algo
    - profiling (heap) to identify any space leak
    - optimise seq algo
    - time profile to point out "big eaters"
    - implement parallel versions for multicores
    - performance tuning to improve runtime/speedup

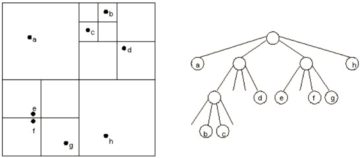
Haskell Implementation (1)

- Tree construction

```haskell
doSteps 0 bs = bs
doSteps s bs = doSteps (s-1) new_bs
  where
    bbox = findBounds bs
    tree = buildTree (bbox,bs)
    new_bs = map (updatePos . updateVel) bs

-- build the Barnes-Hut tree
buildTree :: (Bbox,[Body])->BHTree
buildTree (bb,bs) = BHT size cx cy cz cm subTrees
  where
    subTrees = if bs <= 1
       then []
       else map buildTree (splitPoints bb bs)
    Centroid cx cy cz cm = calcCentroid bs
    size = calcBoxSize bs

findBounds::[Body]->Bbox
-- split bodies into subregions
splitPoints::Bbox->[Body]->[(Bbox,[Body])]
-- calculate the centroid of points
calcCentroid::[Body]->Centroid
-- size of the region
calcBoxSize::Bbox->Double
```



Haskell Implementation (2)

- Force calculation

```
updatePos (Body x y z vx vy vz m) = ... -- same as allpairs

updateVel b@(Body x y z vx vy vz m) = Body x y z (vx-ax) (vy-ay) (vz-az) m
  where
    Accel ax ay az = calcAccel b tree

calcAccel::Body->BHTree->Accel
calcAccel b tree@(BHT _ _ _ _ subtrees)
  | null subtrees = accel tree b
  | isFar tree b  = accel tree b
  | otherwise = foldl addAccel (Accel 0 0 0) (map (calcAccel b) subtrees)
    where
      addAccel (Accel ax1 ay1 az1) (Accel ax2 ay2 az2) = Accel (ax1+ax2) (ay1+ay2) (az1
           +az2)

accel::BHTree->Body->Accel
isFar::BHTree->Body->Bool
```

Haskell Implementation (3)

- Optimisations
  - eliminates stack overflow by making functions tail recursive
  - add strictness annotations where required
  - use strict fields and `UNPACK` pragma in datatype definitions
  - fusion
- Introduce parallelism at the top-level map function
- Core parallel code for Haskell

```
chunksize = (length bs) `quot` (numCapabilities * 4)
new_bs = map f bs `using` parListChunk chunksize rdeepseq
```

- Parallel tuning
  - chunking to control granularity, not too many small tasks; not too few large tasks

F# Implementation

- translate Haskell code into F#
- keeping the code *functional*, so mostly syntax changes
- some general optimisations apply

```
(* Async Workflows *)
let pmap_async f xs =
  seq { for x in xs -> async { return f x } }
  |> Async.Parallel
  |> Async.RunSynchronously

(* TPL *)

(* Explicit tasks creation. Task<'T>: returns a result of type 'T *)
let pmap_tpl_tasks f (xs:array<_>) =
  Array.map (fun x ->
    Task<_>.Factory.StartNew(fun () -> f x).Result
  ) xs

(* Parallel.For *)
let pmap_tpl_parfor f (xs:array<_>) =
  Parallel.For(0, xs.Length, (fun i -> xs.[i] <- f (xs.[i]) ))
  |> ignore

(* PLINQ - uses TPL internally *)
let pmap_plinq f (xs:array<_>) =
  xs.AsParallel().Select(fun x -> f x).ToArray()
```

Scala Implementation

- translate Haskell and F# implementations into Scala
- keeping the code as much functional as possible
- some OO features

```
// Parallel Collections.
nbody.par.map((b: Body) => new Body(b.mass, updatePos(b), updateVel(b))).seq

// Parallel map using Futures.
def pmap[T](f: T => T) (xs: IndexedSeq[T]): IndexedSeq[T] = {
  val tasks = xs.map((x: T) => Futures.future { f(x) })
  tasks.map(future => future.apply())
}

// Parallel map with chunking using Futures.
def pmap_chunk[T](f : T => T, size : Int) (xs: IndexedSeq[T]): IndexedSeq[T] = {
  val chunks = chunk(xs, size)
  val task_chunks = chunks.map((c: IndexedSeq[T]) => Futures.future { c.map((x: T) => f(x
       )) })
  val tasks = task_chunks.map(future => future.apply())
  tasks.flatten
}
```

PRELIMINARY RESULTS

Experimental setup

- Linux machine: CentOS 5.8
  - Intel Xeon CPU E5410 @ 2.33GHz with 8 cores
- Windows machine: Win XP
  - Intel Core i7 860 @ 2.80Ghz with 4 cores and 8 hyper-threads
- Language implementations and platform versions:

	Linux CentOS 5.8	Windows XP
Haskell	GHC 7.0.1	GHC 7.0.4
F#	Mono 2.10.6 with F# 2.0 compiler	.NET platform 4.0 with F# 2.0
Scala	Typesafe Platform with Scala 2.9.1	Scala 2.9.1-1

- Input size: 80,000 bodies, 1 iteration

Results

(work in progress)

Table: Sequential and parallel results.

	Linux (8 cores)			Windows (4 cores, 8 hyperthreads)		
	Seq Runtime	Par Runtime	Speedup	Seq Runtime	Par Runtime	Speedup
Haskell	24.81	4.57	5.42	21.66	5.75	3.76
F#	187.17	109.26	1.71	49.13	22.70	2.16
Scala	59.15	19.74	2.99	—	—	—

Observations

- Performance
  - Haskell gives best performance on both platforms
    - much effort went into the optimisation of Haskell version
  - F# is 2 times slower than Haskell (on windows)
    - though direct translation from Haskell
    - need more optimisations specific to F# and .NET
    - difference in F#/Mono and F#/.NET runtime
  - Scala is approx. 2.5 times slower than Haskell (on linux)
    - various collections showcased similar performance
    - need more specific to Scala/JVM optimisations
    - need to experiment on Windows platform

Observations

- Programmability
  - we compare ease of introducing parallelism in the code, parallelism construct, skeletons, level of control/input from the programmer
  - adding parallelism to the nbody program amounts to using a parallel map
  - all 3 languages offer high-level constructs for data parallelism
  - initial parallelism easy to add in the 3 languages
  - with varying control though

Observations

- Programmability
  - chunking is a general parallel perf tuning but worked only for Haskell
  - Haskell allows initial parallelism to be easily specified
    + parallel options such as chunking and clustering
  - F# and Scala retain much control in the implementation
    + not easy to tune parallel program
  - implications of laziness in Haskell

Observations

- Pragmatics
  - Haskell
    - good tool support e.g. for space and time profiling
    - threadscope: visualisation tool to see work distribution
  - F#
    - Profiling and Analysis tools available in VS2011 Beta Pro - Concurrency Visualizer
  - Scala
    - benefits from free tools available for JVM

Work in Progress

- improve F# and Scala versions
  - optimisations
- take measurements using F# sequence and LazyList
- more sophisticated Scala Actors implementation (maybe use Akka)

Thank you for listening!

- Read more:
  - Paper (draft): `www.macs.hw.ac.uk/~pt114/papers/tfp12.pdf`
  - to be presented in TFP'12, University of St. Andrews
- Email: {pt114, pd85, H.W.Loidl}@hw.ac.uk