Biocaml: The OCaml Bioinformatics Library

Ashish Agarwal *(Solvuu)*

Sebastien Mondet *(Mount Sinai)*

Philippe Veber *(CNRS, Université Lyon 1)*

Christophe Troestler *(Université de Mons)*

Bioinformatics Open Source Conference (BOSC 2014)

Boston, USA

July 11, 2014

Multi-Paradigm

Imperative

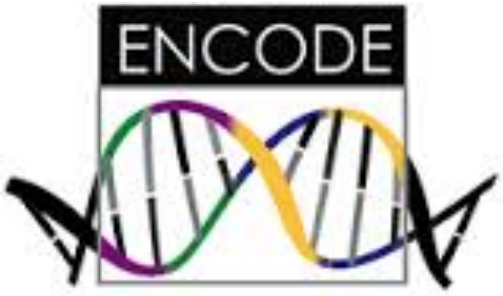
Object Oriented

Functional

Compilers

Compiler	Target
ocamlc	OCaml bytecode REPL
ocamlopt	native code all standard architectures supported
js_of_ocaml	Javascript
ocamljava (beta)	JVM
mirage (beta)	Xen VM

Notable Users



Example: Count Sam Alignments

count_sam_alignments.ml

```
#! /usr/bin/env ocamlscript
Ocaml.ocamlflags := ["-thread"];
Ocaml.packs := ["core"; "biocaml"; "biocaml.ez"]
--
open Core.Std
open Filename
open Biocaml
open Biocaml_ez;;

Sam.read_file Sys.argv.(1)
|> Sequence.fold ~init:0 ~f:(fun count item ->
     match item with
     | `Header_item _ -> count
     | `Alignment _ -> count+1
   )
|> printf "%s: %d\n" (basename Sys.argv.(1))
```

Call ocamlscript

Load Libraries

Open Common Modules

Main Code

```
$ ./count_sam_alignments.ml test.sam
test.sam: 4314041
```

Biocaml Modules

Data Formats

Bam
Bar
Bed
Bpmap
Cel
Fasta
Fastq
Gff
Jaspar
Lines
MzData
Psl
Sam
Sgr
Wig
Zip
Chr
Phred_score
Solexa_score
Roman_num

Data Structures and Analysis

Genome_map
Histogram
Interval_tree
Range (integer interval)
RSet (DIET sets)
Math (misc. stat functions)
Pwm (position weight matrix)

Data Clients

Entrez

Errors

With Exceptions

```
parse
  : string
-> item
```

Pros

- Easy API

Cons

- Exceptions are a *side effect*
- Not represented in types

Good for scripts.

Strongly Typed

```
parse
  : string
-> item Or_error.t
```

Pros

- Purely functional
- Types document error

Cons

- More complex API

Good for industrial strength code.

Example: exceptions are less safe

Version N

Biocaml_ez
string -> alignment

Biocaml

string -> alignment

Your code keeps compiling, but you *may* get runtime error.

Version N+1

Biocaml_ez

string -> alignment

Biocaml

string -> alignment Or_error.t

Compiler error. You must consider error case.

Concurrency

Blocking Calls

```
let a = Sam.read_file a.sam in
let b = Gff.read_file b.gff in
… do something with a and b
```

Pros

- Easy. Everyone can write this code.

Cons

- CPU, network, disk, etc. *may* sit idle unnecessarily.

Good for scripts. Sometimes faster.

Asynchronous Calls

```
Sam.read_file a.sam >>= fun a ->
Gff.read_file b.gff >>= fun b ->
… do something with a and b
```

Pros

- *Possibly* more efficient use of CPU, network, disk, etc.

Cons

- Writing monadic code is challenging.

Good for server side code. Sometimes faster.

Example: Concurrency

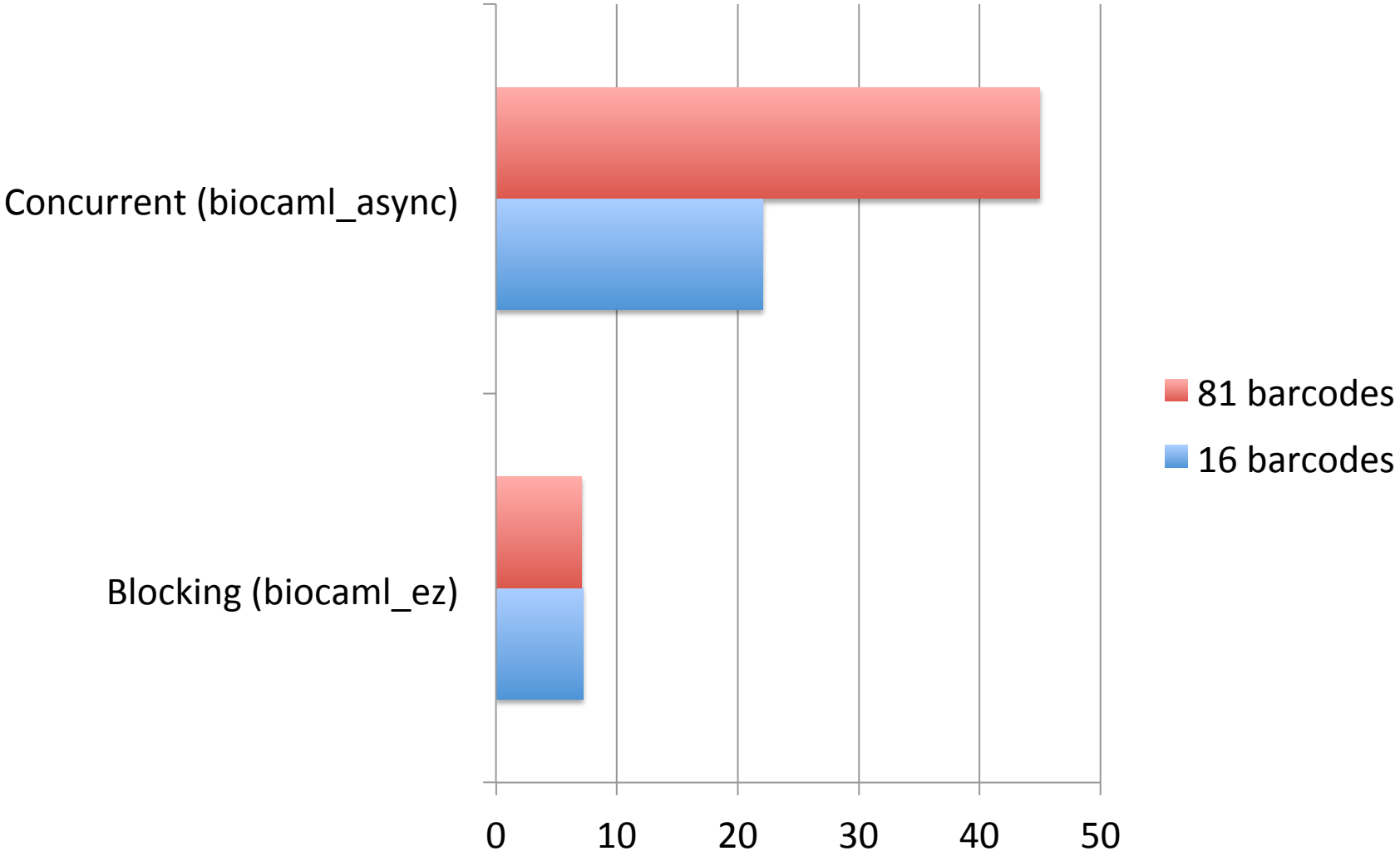
demux_ez

```
let demux_ez file =
  let open Biocaml_ez.Fastq in
  let outdir = temp_dir ~in_dir:(dirname file)
    "demux_" (basename file) in
  read_file file
  |> Sequence.fold ~init:String.Map.empty
      ~f:(fun out_chans item ->
        let barcode = String.slice item.sequence
          0 4 in
        let out_chans,out_chan =
          match Map.find out_chans barcode with
          | None ->
            let out_file = concat outdir
              (barcode ^ ".fastq") in
            let out_chan =
              Out_channel.create out_file in
            Map.add out_chans ~key:barcode
              ~data:out_chan,
            out_chan
          | Some out_chan ->
            out_chans, out_chan
        in
        Out_channel.output_string out_chan
          (item_to_string item);
        out_chans
      )
  |> Map.iter ~f:(fun ~key:_ ~data ->
    Out_channel.close data)
```

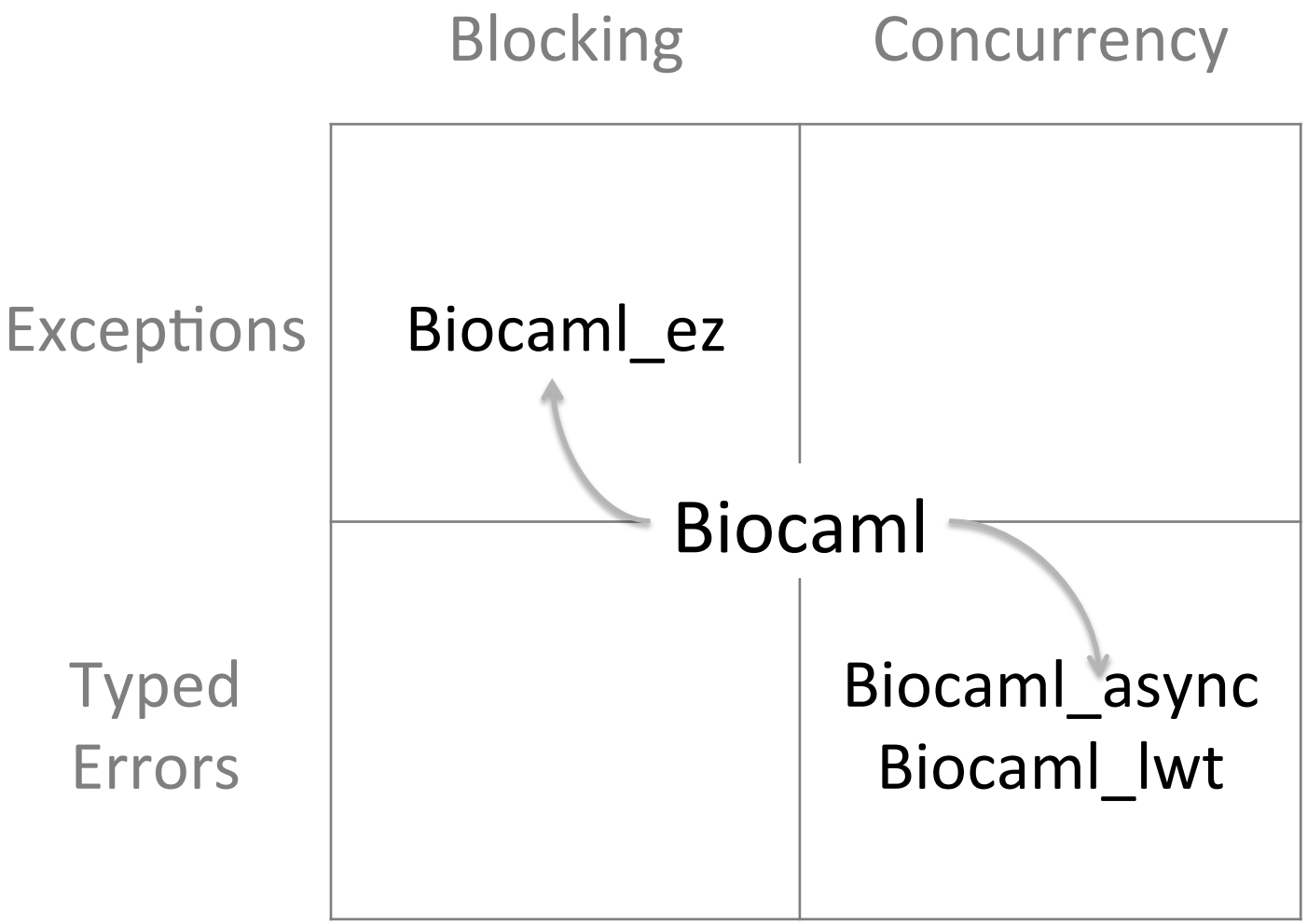
demux_async

```
let demux_async file =
  let open Async.Std in
  let open Biocaml_async.Fastq in
  let outdir = temp_dir ~in_dir:(dirname file)
    "demux_" (basename file) in
  read_file file >>=
  Pipe.fold ~init:String.Map.empty ~f:(
    fun out_chans item ->
      match item with
      | Error _ -> return out_chans
      | Ok item ->
        let barcode = String.slice
          item.sequence 0 4 in
        (match Map.find out_chans barcode with
        | None ->
          let out_file = concat outdir
            (barcode ^ ".fastq") in
          Writer.open_file out_file >>|
          fun out_chan ->
          (Map.add out_chans ~key:barcode
             ~data:out_chan,
           out_chan)
        | Some out_chan ->
          return (out_chans, out_chan)
        ) >>| fun (out_chans,out_chan) ->
        Writer.write out_chan (item_to_string
                                 item);
        out_chans
  ) >>=
  Deferred.Map.iter ~f:(fun ~key:_ ~data ->
    Writer.close data)
```

blocking calls can be faster



Biocaml Flavors



Parallelism

Parmap

- multi-core, single node
- arr – a large data array
- f – function to apply on every item of array

Serial code

- `map arr f`

Parallelize with parmap

- `parmap arr f`

Async_parallel

- distributed, multi-node
- hubs – a place where multiple clients can send/ read messages
- channel – communicate with a hub
- process – a job

- 10 minutes of coding – simple counting app with 4x speedup

Many more libraries and language extensions. No silver bullet.
Parallelism is an open research problem.

Conclusions

- Biocaml
  - easy to write easy code
    easier to write sophisticated code
  - many useful modules already exist
    in use for years, but
    undergoing complete re-write

- OCaml
  - strong theoretical foundations
  - increasingly large community and tools
  - covers wide spectrum of needs
    scripts ↔ enterprise level software architectures
    backend ↔ frontend

http://ocaml.org

http://biocaml.org