Haskell for the Cloud

Andrew P. Black

Joint work with Jeff Epstein & Simon Peyton Jones

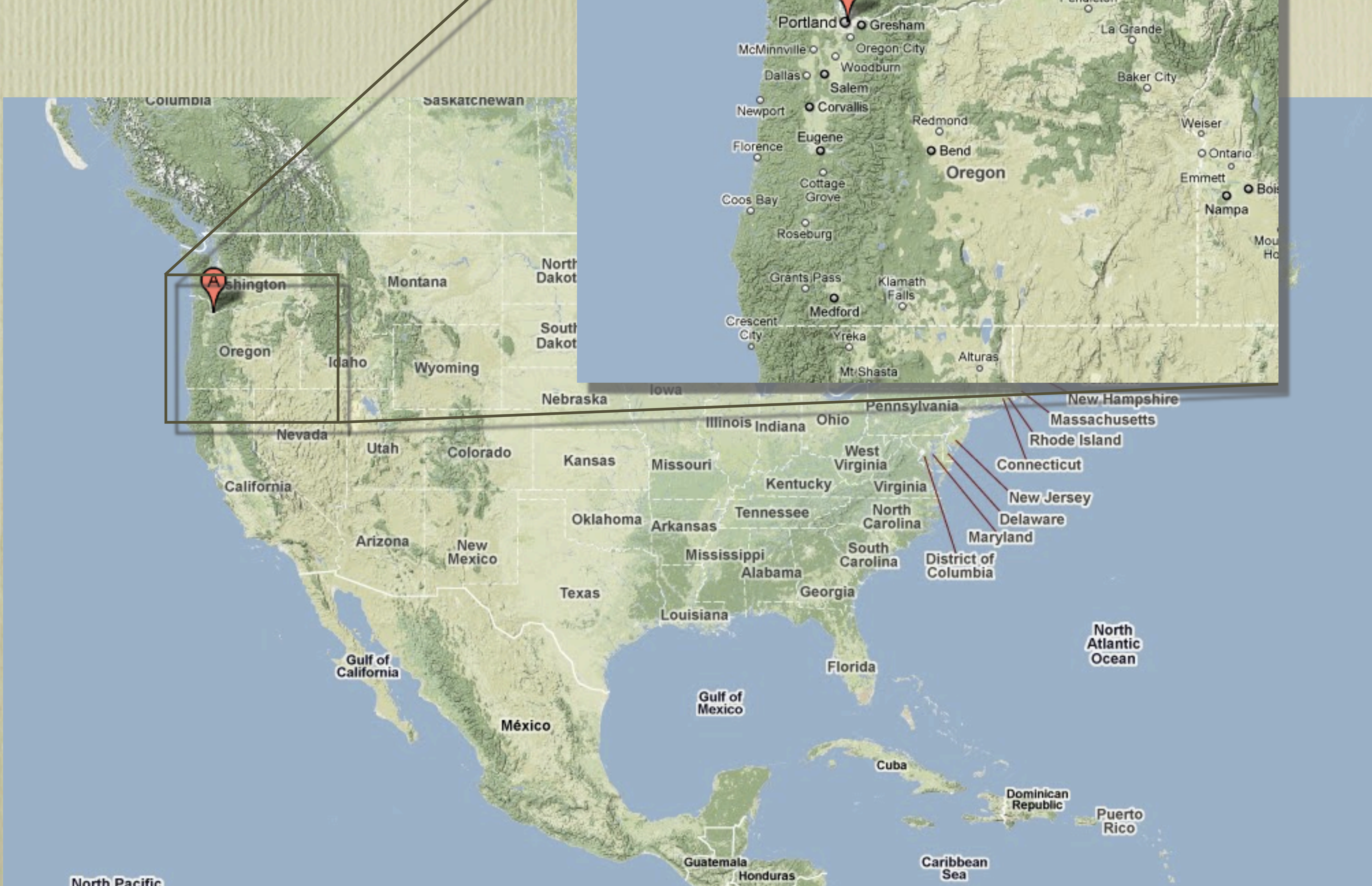
Portland State University

Coming right up,
after these messages ...

Andrew P. Black, Portland State University

Andrew P. Black, Portland State University

Andrew P. Black,

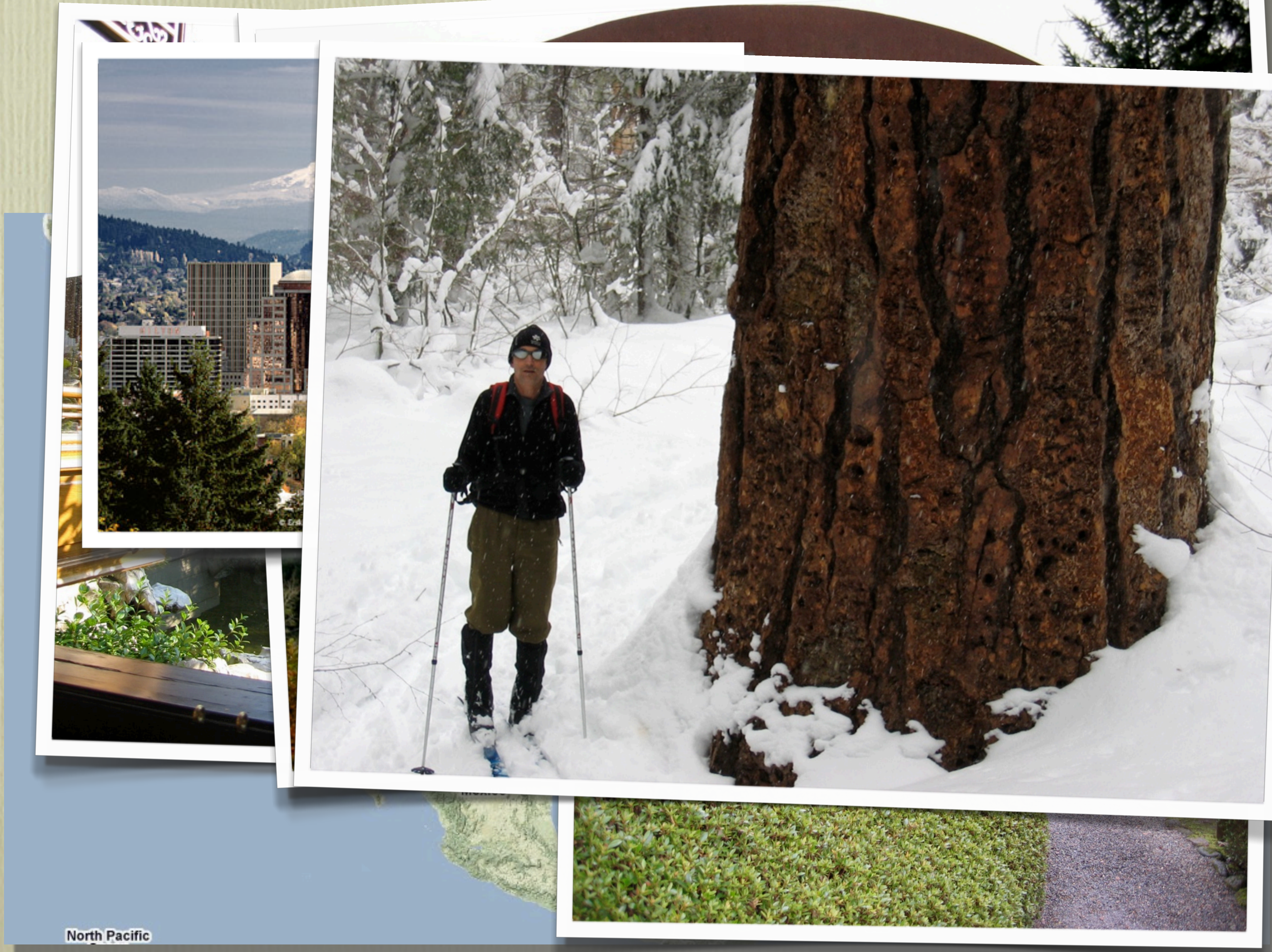


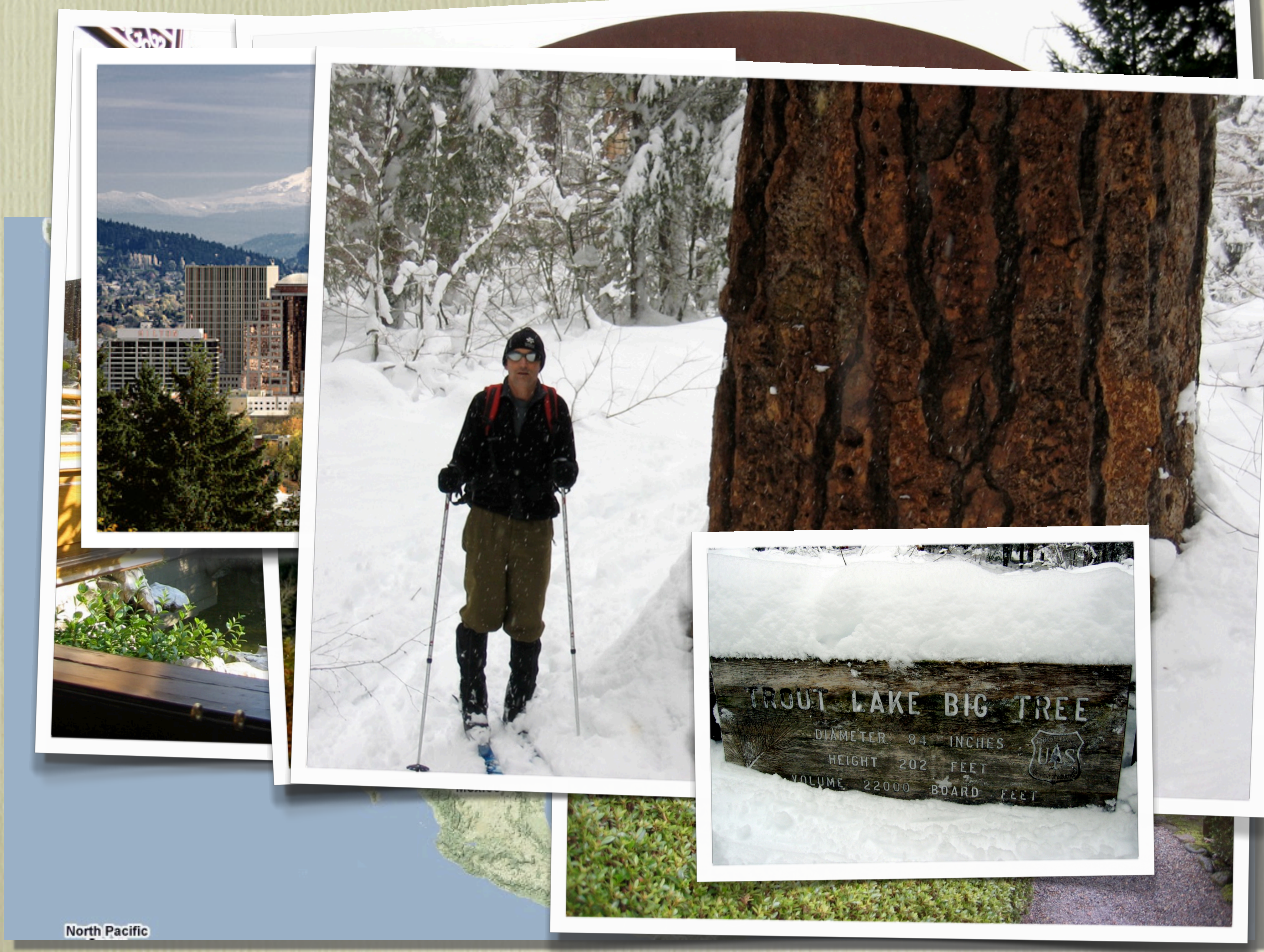
Yakima
Pullman
Lewiston
Longview
Kennewick
Walla Walla
Pendleton
A
Portland
Gresham
La Grande
McMinnville
Oregon City
Woodburn
Dallas
Salem
Baker City
Newport
Corvallis
Redmond
Weiser
Florence
Eugene
Bend
Ontario
Oregon
Emmett
Coos Bay
Cottage Grove
Nampa
Roseburg
Grants Pass
Klamath Falls
Medford
Crescent City
Yreka
Alturas
Mt Shasta
New Hampshire
Pennsylvania
Massachusetts
Illinois
Indiana
Ohio
Rhode Island
West Virginia
Connecticut
Kentucky
Virginia
New Jersey
Tennessee
North Carolina
Delaware
Maryland
Mississippi
South Carolina
District of Columbia
Alabama
Georgia
North Atlantic Ocean
Florida
Gulf of Mexico
México
Cuba
Dominican Republic
Puerto Rico
Caribbean Sea
Guatemala
Honduras
North Pacific

Yakima
Pullman
Lewiston
Longview
Kennewick
Walla Walla
Pendleton
A
Portland
Gresham
McMinnville
Oregon City
Woodburn
Dallas
Salem
La Grande
Newport
Corvallis
Baker City
Redmond
Florence
Eugene
Bend
Oregon
Weiser
Ontario
Emmett
Nampa
Coos Bay
Cottage Grove
Roseburg
Grants Pass
Klamath Falls
México
North Pacific

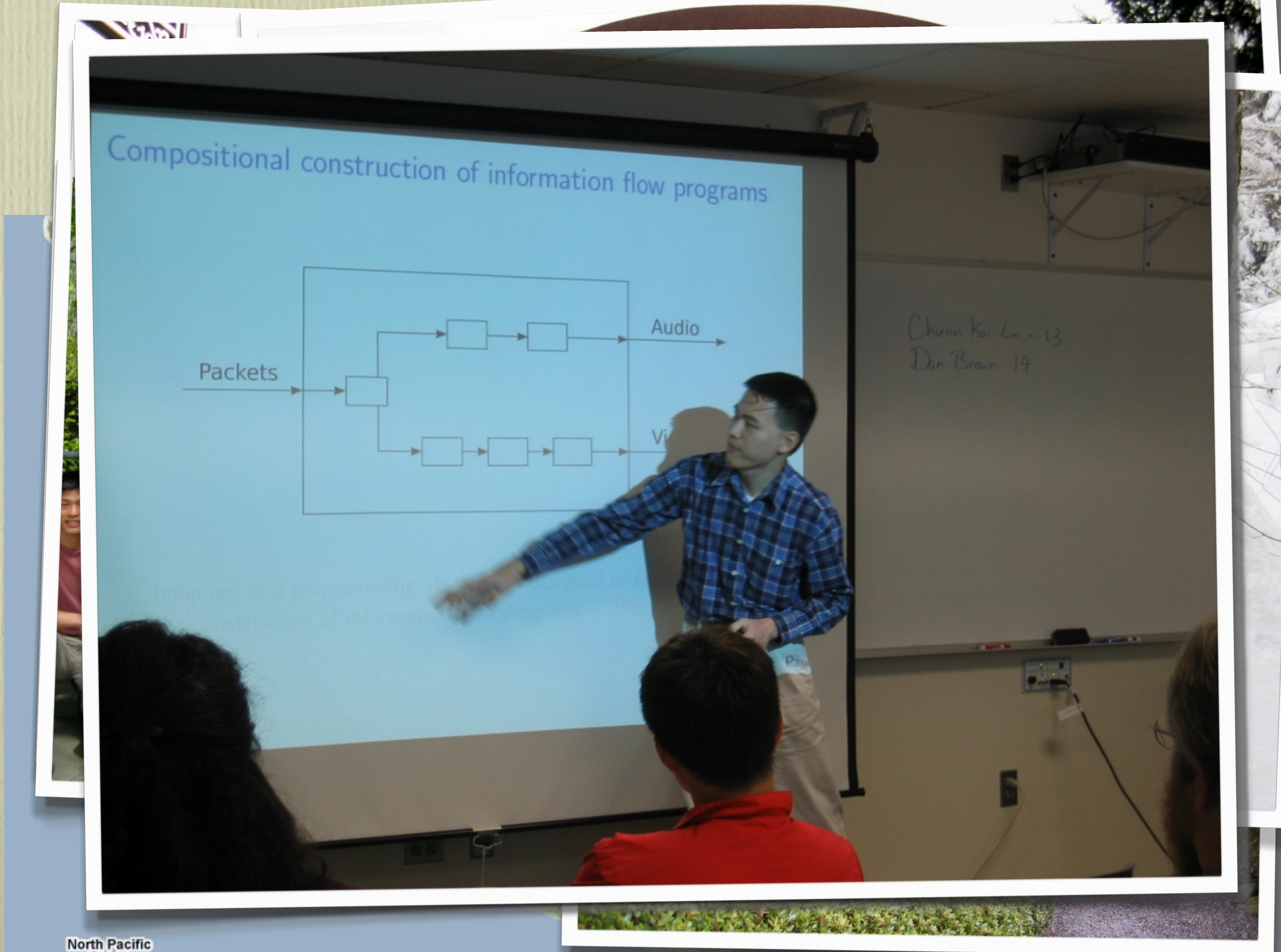

México
North Pacific


México
North Pacific


North Pacific


TROUT LAKE BIG TREE
DIAMETER 84 INCHES
HEIGHT 202 FEET
VOLUME 22000 BOARD FEET
North Pacific


North Pacific
AKE BIG TREE
TER 84 INCHES
HT 202 FEET
22000 BOARD FEET


Compositional construction of information flow programs
Packets
Audio
Chuan Kai Lin - 13
Don Brown 14
North Pacific

Andrew P. Black, Portland State University

Research interests:

- Programming language design
- Programming environments
- Distributed systems
- Concurrency
  - executable languages for expressing the concurrency inherent in algorithms

black@cs.pdx.edu

http://www.cs.pdx.edu/~black

Informatics Forum Rm 4.18B

Andrew P. Black, Portland State University

Research interests:

- Programming language design
- Programming environments
- Distributed systems
- Concurrency

    executable languages for expressing the concurrency inherent in algorithms

black@cs.pdx.edu

http://www.cs.pdx.edu/~black

Informatics Forum Rm 4.18B

Another Talk:

Thursday 28th April, 10:30am-12:00pm in Informatics Forum Room G.03

Grace: a New educational O-O programming language

Speaker: Professor Andrew P Black

Abstract:

We are engaged in the design of a new Object-oriented educational programming language called Grace.

Our motivation is frustration with available languages, none of which seems to be suited to our target audience: students in the first two programming courses.

What principles should we apply to help us design such a language? We started with a list of 17 “obviously good principles”, aware that some of them would conflict with each other. What we didn't expect was that some of them would conflict with good learning.

Haskell for the Cloud

Andrew P. Black

Joint work with Jeff Epstein & Simon Peyton Jones

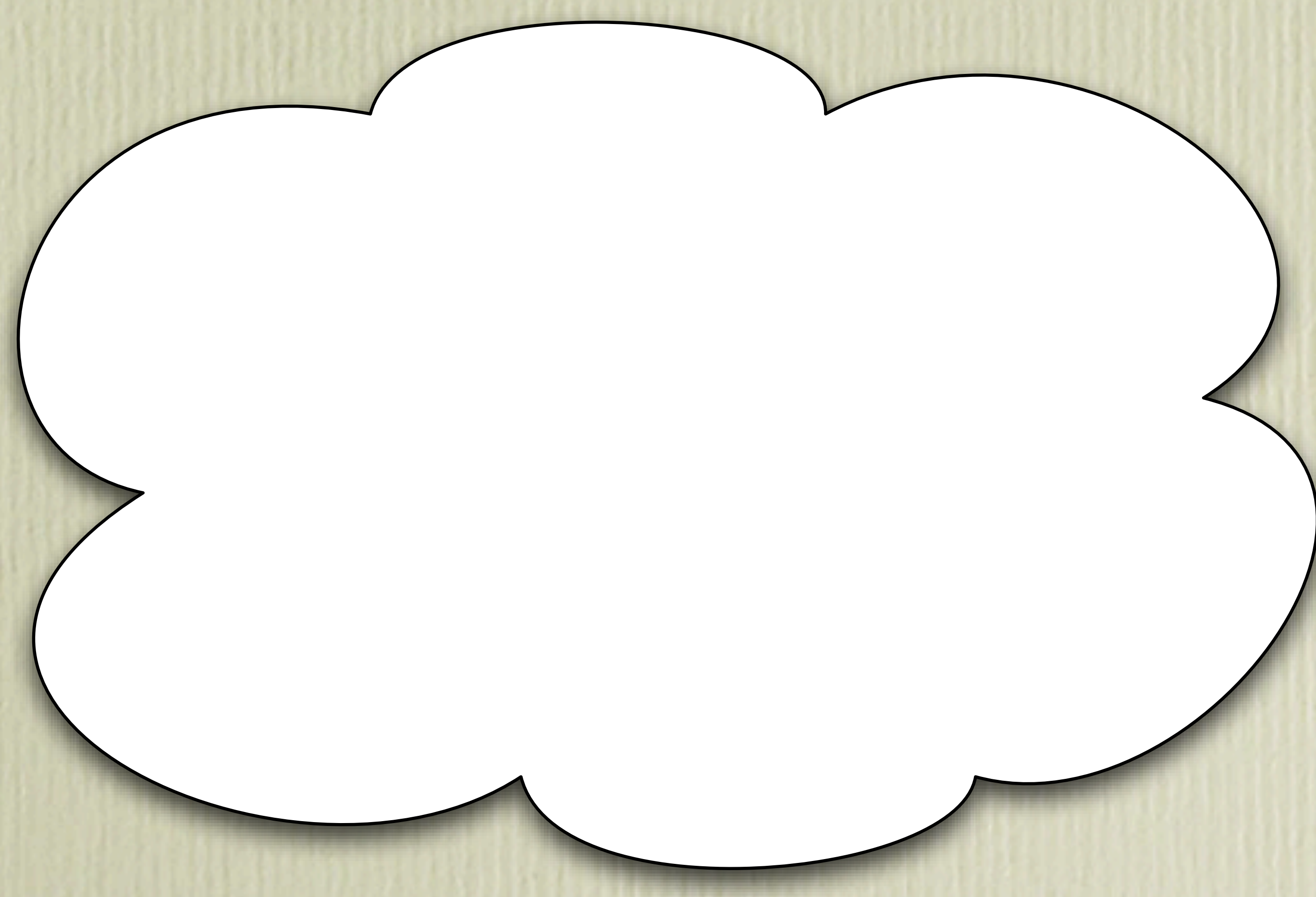
Portland State University

Cloud Haskell in a Nutshell

- A DSL for Cloud Computing implemented as a Haskell library
  - From Erlang:
    - Processes with message-passing parallelism
    - Failure and recovery model
  - From Haskell:
    - Types: purity and monads
    - Typed Channels
    - Shared-memory concurrency *within* a process

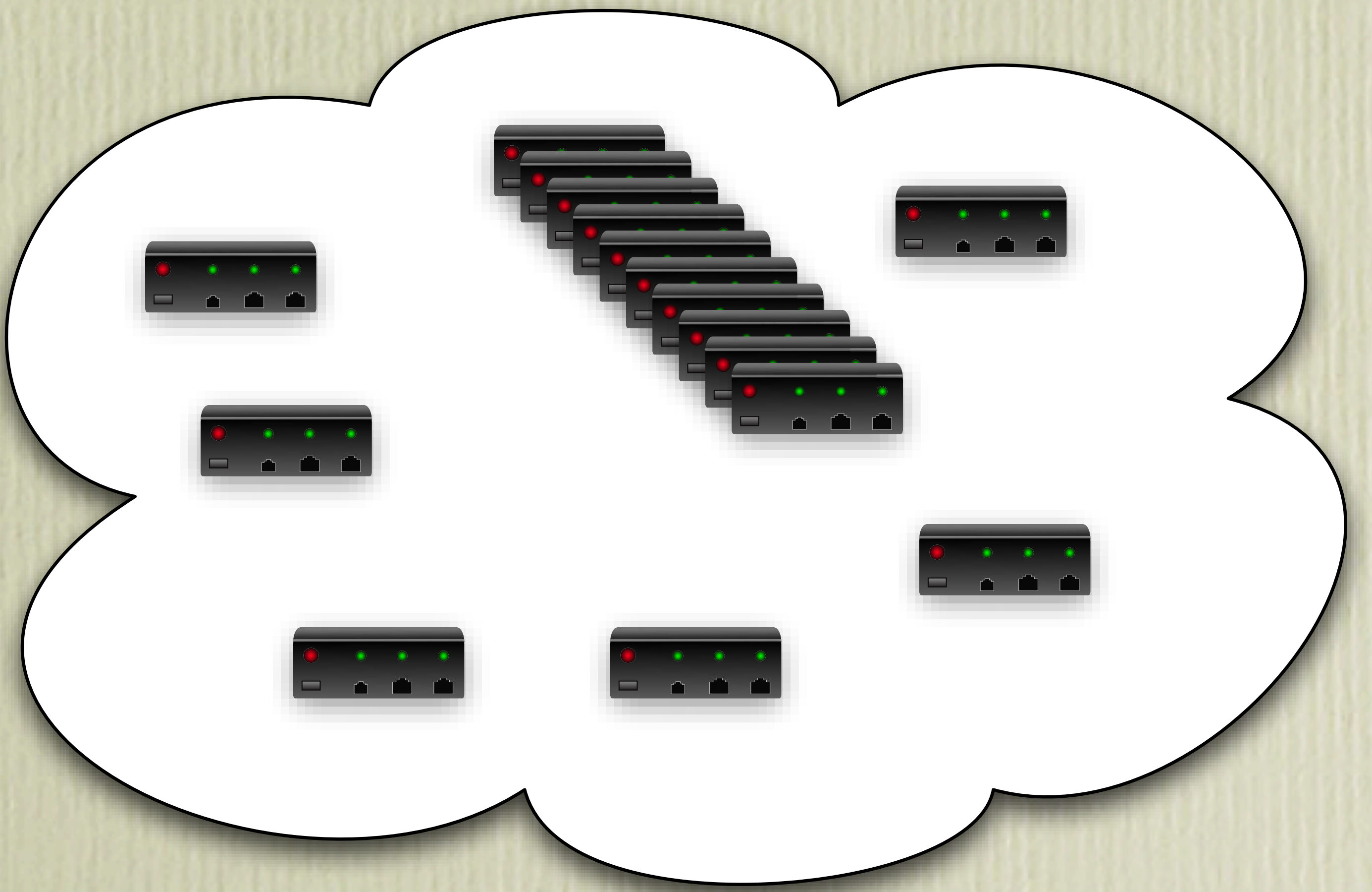
What's a Cloud?

What's a Cloud?



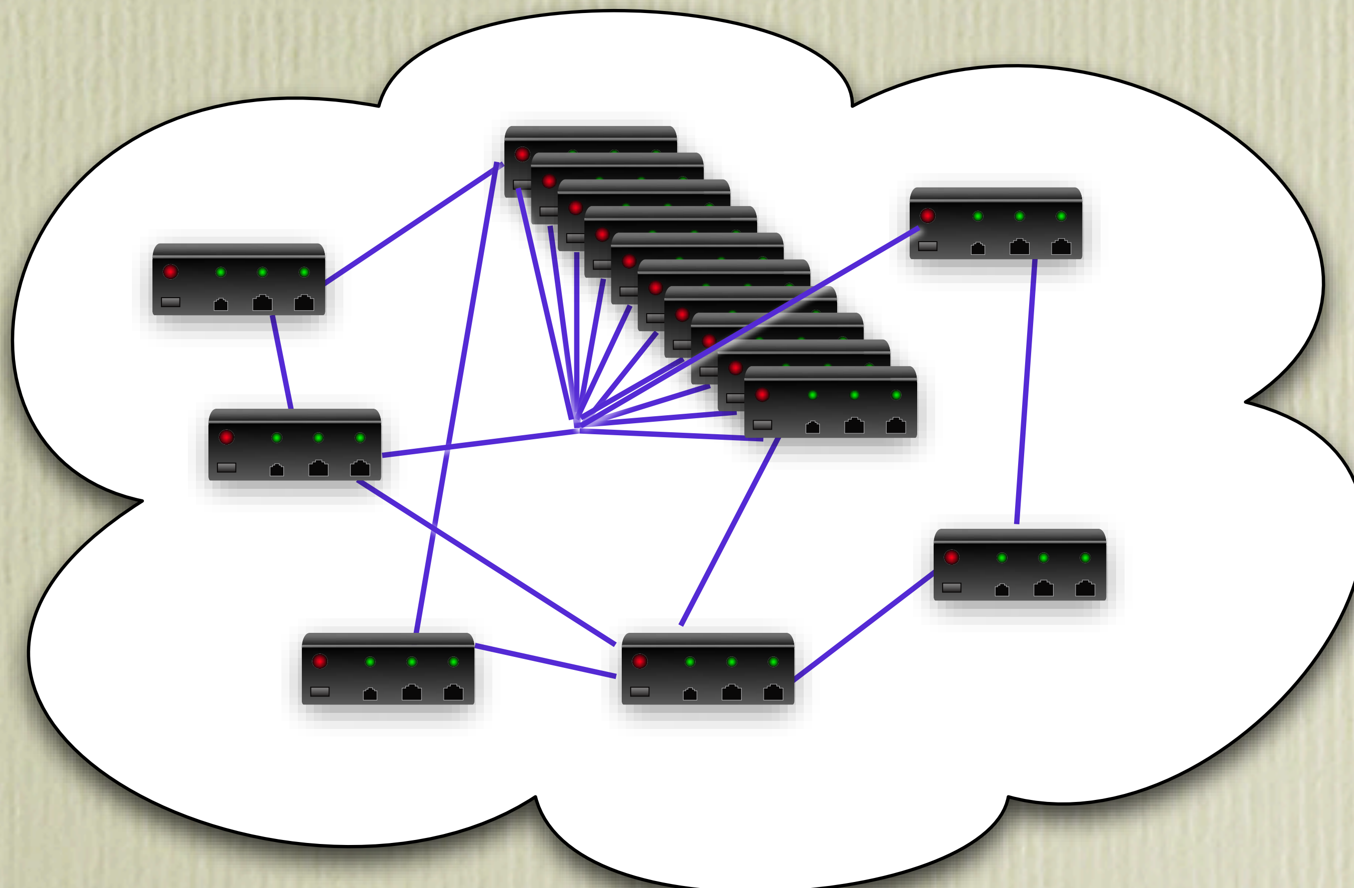
What's a Cloud?

many separate processors



What's a Cloud?

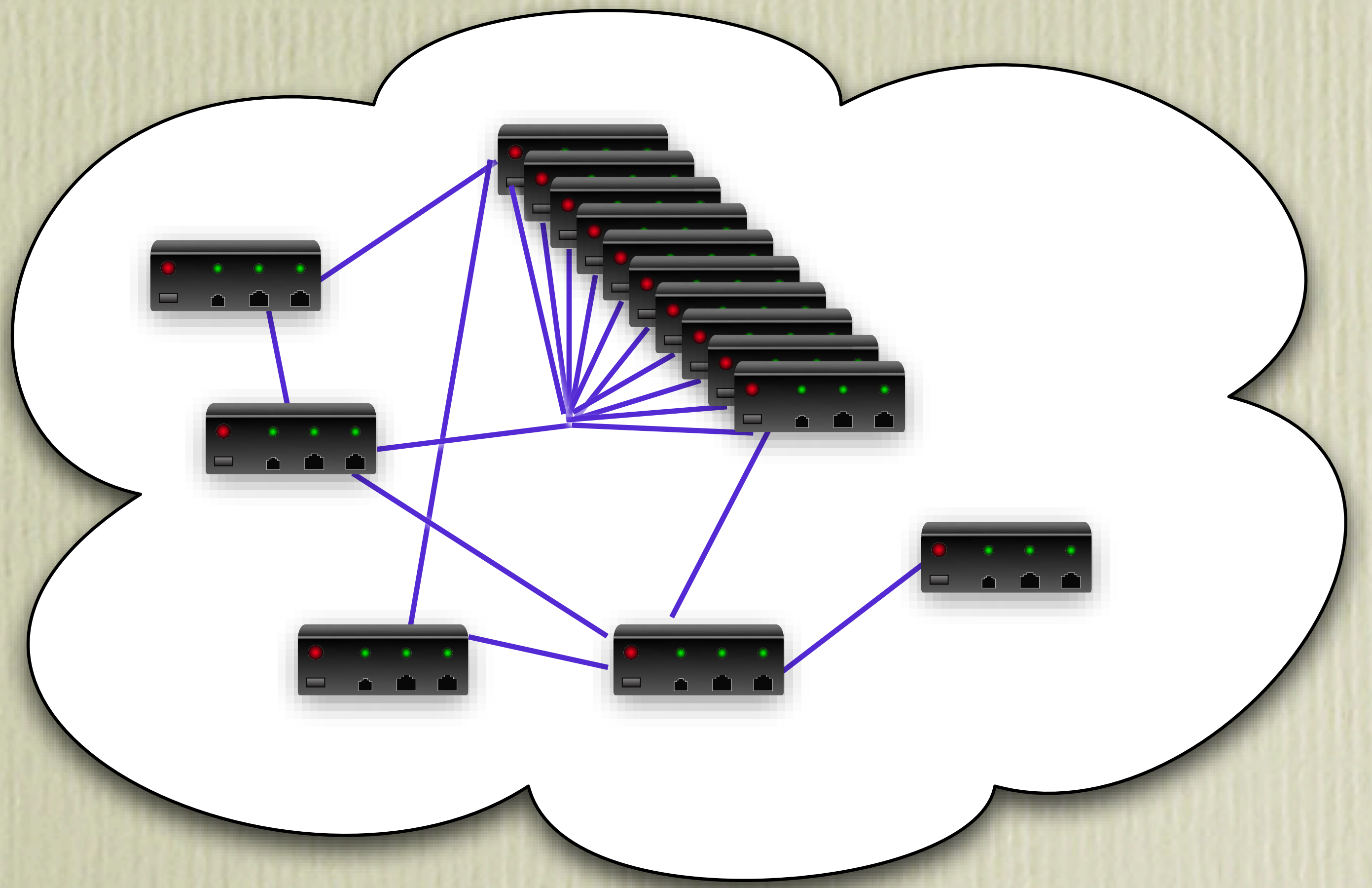
many separate processors connected by a network



What's a Cloud?

many separate processors connected by a network

independent failure modes



This Talk:

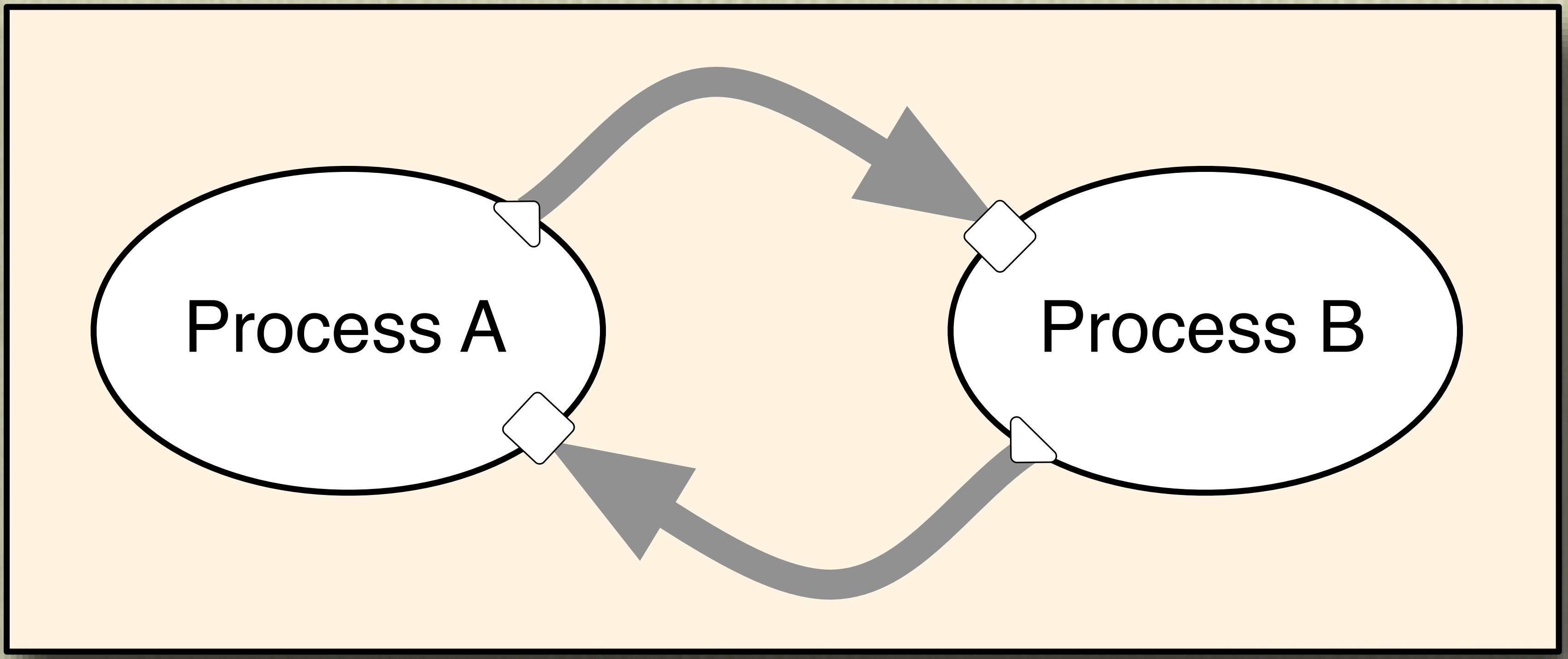
1. Erlang-style concurrency in Haskell
   - Processes, messages & failures
2. Typed Channels
3. Serialization of function closures
4. Assessment
   - Example applications

Erlang in Haskell

- Processes & Messages
- Linking Processes
- Selective Receive of Messages

Processes & Messages

- Process: a concurrent activity that has the ability to send and receive messages



- Processes *cannot* share memory

```
instance Monad ProcessM
instance MonadIO ProcessM
send  :: Serializable a ⇒ ProcessId → a →ProcessM ()
expect :: Serializable a ⇒ ProcessM a
```

```
instance Monad ProcessM
instance MonadIO ProcessM
send   :: Serializable a ⇒ ProcessId → a →ProcessM ()
expect    :: Serializable a ⇒ ProcessM a
```

- Ping pong:

```
data Ping = Ping ProcessId
data Pong = Pong ProcessId
— omitted: Serializable instance for Ping and Pong

ping :: ProcessM ()
ping = do { self ← getSelfPid
          ; Pong partner ← expect
          ; send partner (Ping self)
          ; ping }
```

```
instance Monad ProcessM
instance MonadIO ProcessM
send   :: Serializable a ⇒ ProcessId → a →ProcessM ()
expect    :: Serializable a ⇒ ProcessM a
```

- Compare with the Erlang version:

```
ping() → receive
            {pong, Partner} → Partner ! {ping, self()}
          end,
          ping().
```

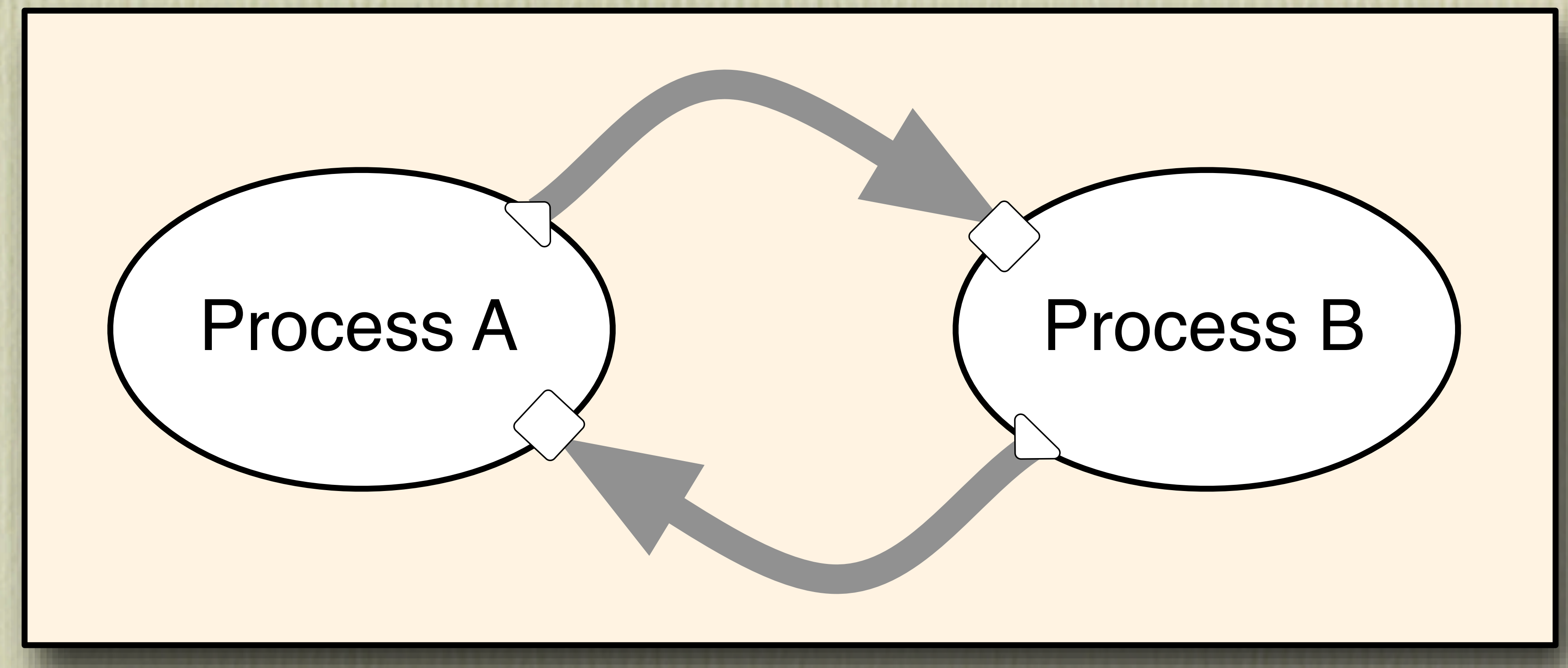
```
data Ping = Ping ProcessId
data Pong = Pong ProcessId
— omitted: Serializable instance for Ping and Pong

ping :: ProcessM ()
ping = do { self ← getSelfPid
          ; Pong partner ← expect
          ; send partner (Ping self)
          ; ping }
```

Portland State UNIVERSITY

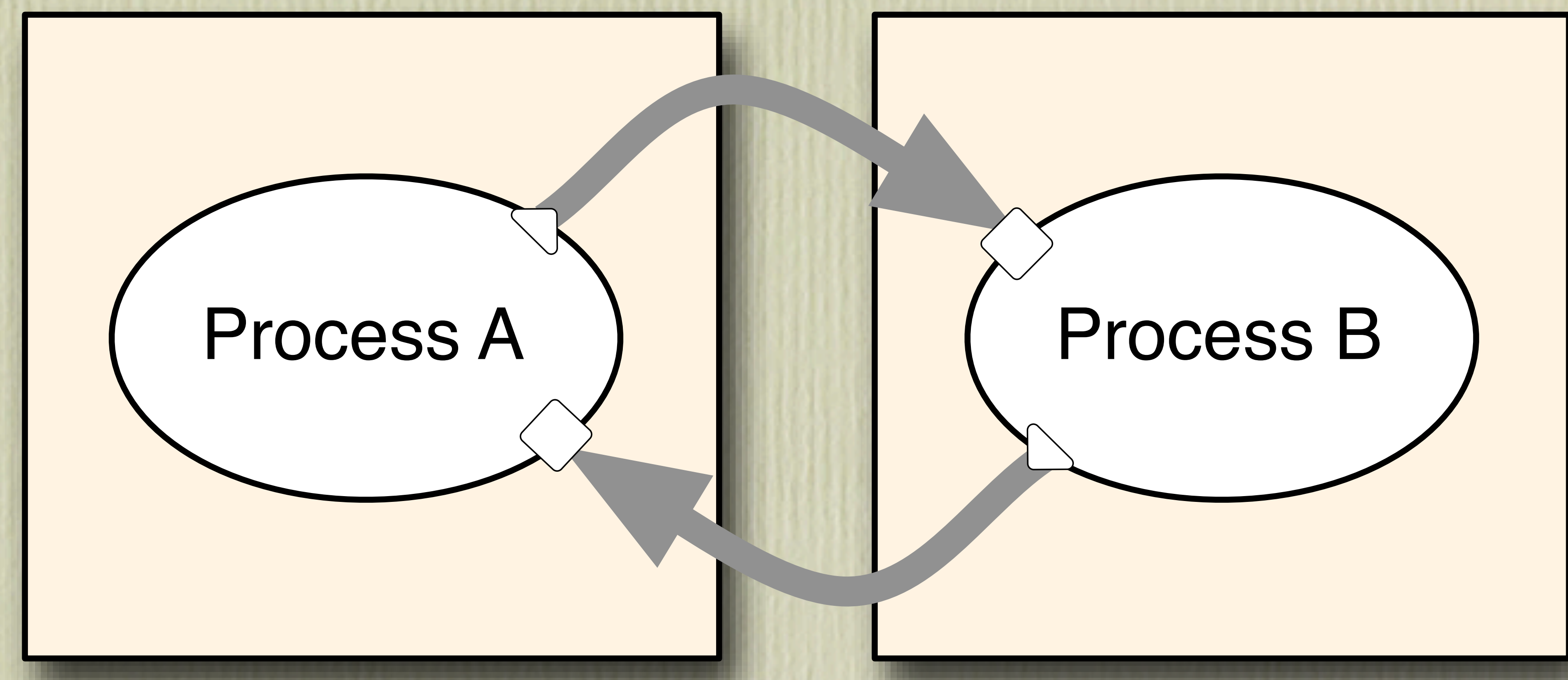
```
instance Monad ProcessM
instance MonadIO ProcessM
send  :: Serializable a ⇒ ProcessId → a →ProcessM ()
expect :: Serializable a ⇒ ProcessM a
```

- Key idea: only Serializable values can be sent in messages.
- Certain values are *deliberately* not serializable
  - MVars, IVars and TVars, in particular

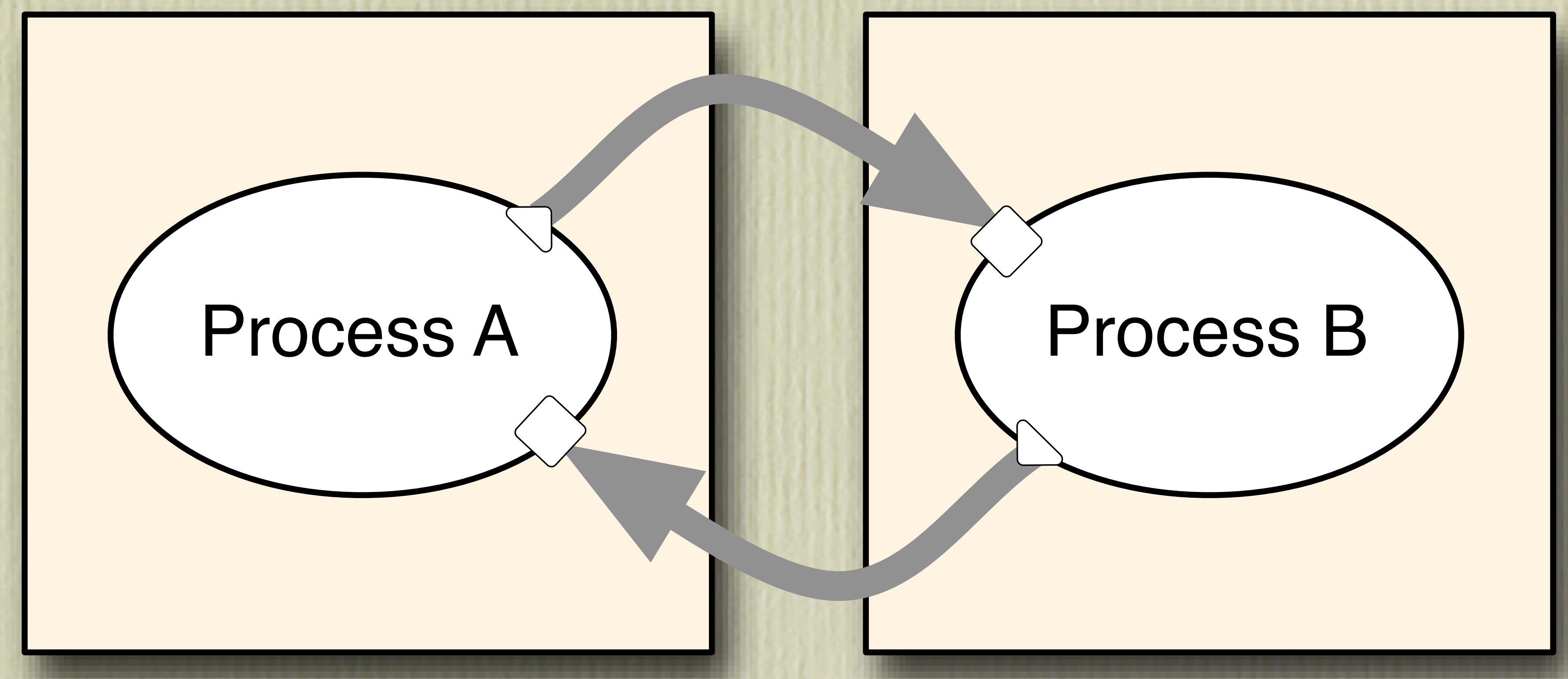
- Consequently:



Consequently:

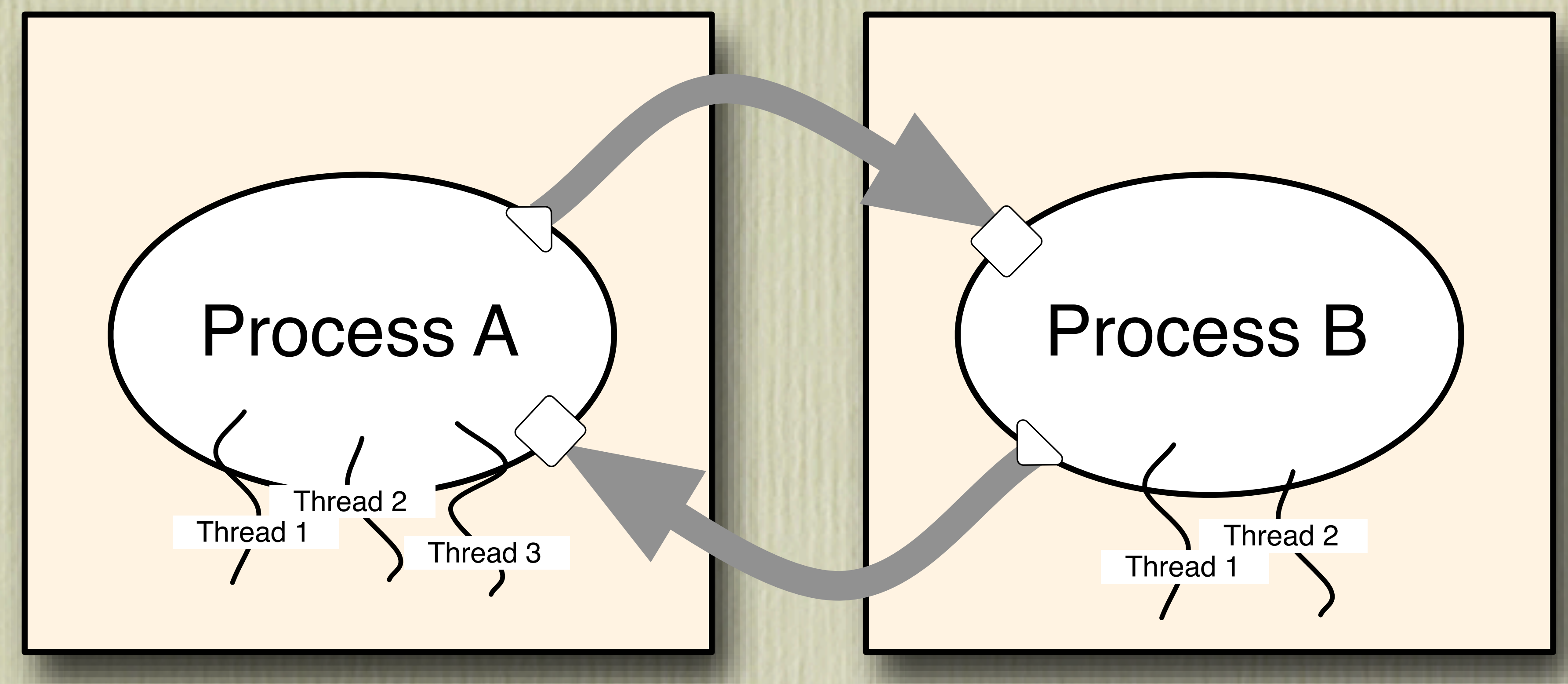


Consequently:

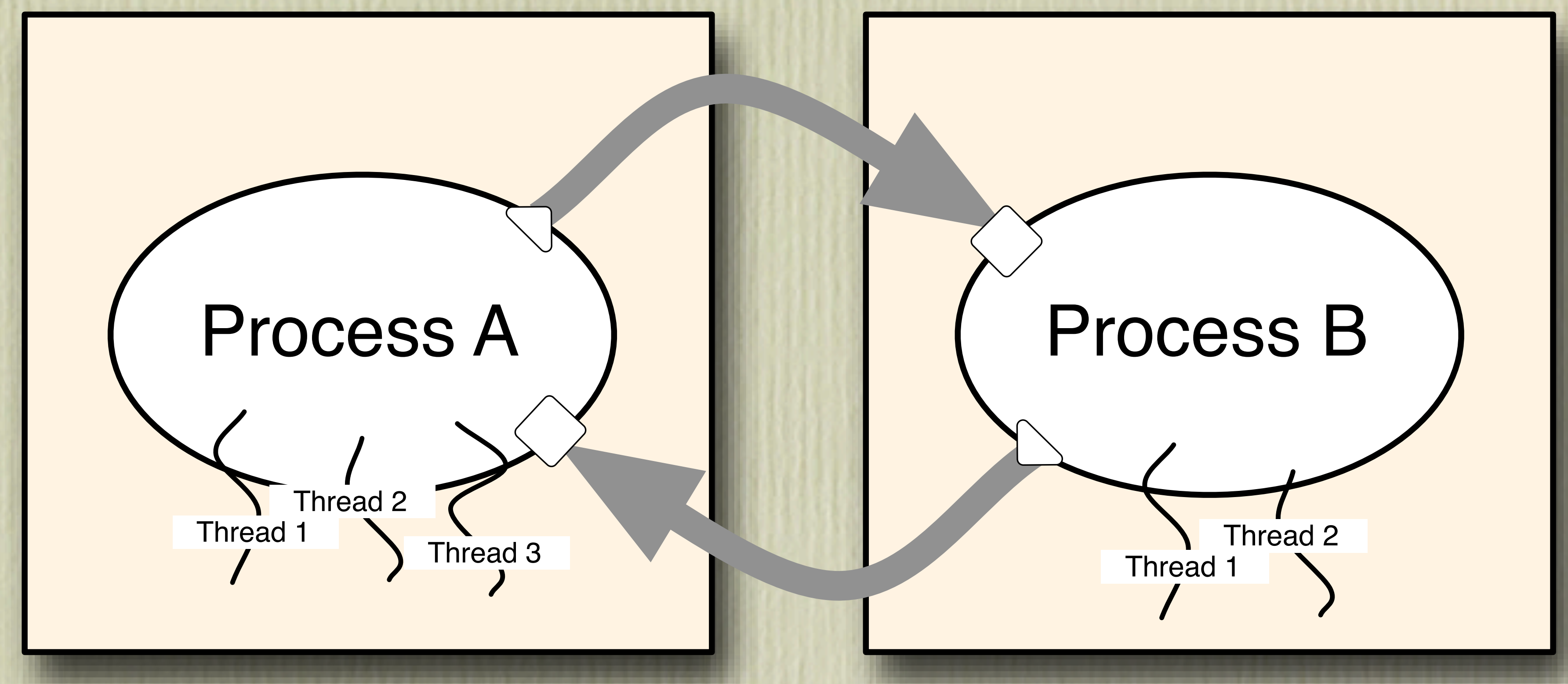


Processes can be moved from one computer to another without invalidating the programming model

Consequently:



Consequently:



Concurrent Haskell's threads, MVars, STM, etc., can all be used *inside* a single Process

Why?

Why?

- Would it be possible to serialize MVars?

Why?

- Would it be possible to serialize MVars?
  - = Is it possible to simulate shared memory in a distributed memory environment?

Why?

- Would it be possible to serialize MVars?
  - = Is it possible to simulate shared memory in a distributed memory environment?
  - Yes!

Why?

- Would it be possible to serialize MVars?
  - = Is it possible to simulate shared memory in a distributed memory environment?
  - Yes!
- Would it be a good idea to serialize MVars?

Why?

- Would it be possible to serialize MVars?
  = Is it possible to simulate shared memory in a distributed memory environment?
  - Yes!
- Would it be a good idea to serialize MVars?
  - We don't think so.

Why?

- Would it be possible to serialize MVars?
  = Is it possible to simulate shared memory in a distributed memory environment?
  - Yes!
- Would it be a good idea to serialize MVars?
  - We don't think so.
  - Glasgow Distributed Haskell disagrees!

Starting & Positioning Processes

- A Node (address space, or virtual computer) is identified by a `NodeId`
- Processes are created by `spawn`
  - First try:

```
— wrong
spawn :: NodeId → ProcessM () → ProcessM ProcessId
do   { pingProc ← spawn someNode ping
     ; pongProc ← spawn otherNode pong
     ; send pingProc (Pong pongProc) }
```

Actual type of Spawn

```
— wrong
spawn :: NodeId → ProcessM () → ProcessM ProcessId
do   { pingProc ← spawn someNode ping
     ; pongProc ← spawn otherNode pong
     ; send pingProc (Pong pongProc) }
```

```
— right
spawn :: NodeId → Closure (ProcessM ())
                              → ProcessM ProcessId
```

Portland State UNIVERSITY

Actual type of Spawn

```
— wrong
spawn :: NodeId → ProcessM () → ProcessM ProcessId
```

```
— right
spawn :: NodeId → Closure (ProcessM ())
                                → ProcessM ProcessId
```

Actual type of Spawn

```
— wrong
spawn :: NodeId → ProcessM () → ProcessM ProcessId
```

```
— right
spawn :: NodeId → Closure (ProcessM ())
                                    → ProcessM ProcessId
```

More about Closures later

Selective Receive

- Erlang provides selective receive by pattern-matching on atoms.

```
math() →
  receive
    {add, Pid, Num1, Num2} →
       Pid ! Num1 + Num2;
    {divide, Pid, Num1, Num2} when Num2 ≠ 0 →
       Pid ! Num1 / Num2;
    {divide, Pid, _, _} →
       Pid ! div_by_zero
  end,
  math().
```

- Haskell programmers would use type constructors instead of atoms:

```
data MathOp  =  Add ProcessId Double Double
             |  Divide ProcessId Double double
             |  Answer Double
             |  DivByZero
```

- However, this breaks modularity, e.g, it forces servers to respond to Answer and clients to respond to Add.

- It's better to use several independent types:

```
data Add        = Add ProcessId Double Double
data Divide     = Divide ProcessId Double Double
data DivByZero  = DivByZero
```

- However, now we need something more than expect, because we don't know which message will arrive first.

match & receiveWait

```
math :: ProcessM ()
math =
    receiveWait
    [     match   (λ(Add pid num1 num2) →
                        send pid (num1 + num2)),
          matchIf (λ(Divide _ _ num2) → num2 ≠ 0)
                  (λ(Divide pid num1 num2) →
                        send pid (num1 / num2)),
          match   (λ(Divide pid _ _) →
                        send pid DivByZero) ]
      ≫ math
```

match & receiveWait

```
math :: ProcessM ()
math =
    receiveWait
    [     match   (λ(Add pid num1 num2) →
                        send pid (num1 + num2)),
          matchIf (λ(Divide _ _ num2) → num2 ≠ 0)
                  (λ(Divide pid num1 num2) →
                        send pid (num1 / num2)),
          match   (λ(Divide pid _ _) →
                        send pid DivByZero) ]
      ≫ math
```

match :: Serializable a ⇒
(a → ProcessM q) → MatchM q ()

match & receiveWait

```
math :: ProcessM ()
math =
    receiveWait
    [     match   (λ(Add pid num1 num2) →
                          send pid (num1 + num2)),
          matchIf (λ(Divide _ _ num2) → num2 ≠ 0)
                  (λ(Divide pid num1 num2) →
                          send pid (num1 / num2)),
          match   (λ(Divide pid _ _) →
                          send pid DivByZero) ]
      ≫ math
```

```
receiveWait ::
    [MatchM q ()] → ProcessM q
```

```
match :: Serializable a ⇒
(a → ProcessM q) → MatchM q ()
```

match & receiveWait

```
math :: ProcessM ()
math =
  receiveWait
  [    match   (λ(Add pid num1 num2) →
                    send pid (num1 + num2)),
       matchIf (λ(Divide _ _ num2) → num2 ≠ 0)
               (λ(Divide pid num1 num2) →
                    send pid (num1 / num2)),
       match   (λ(Divide pid _ _) →
                    send pid DivByZe
    ≫ math
```

receiveWait ::
  [MatchM q ()] → ProcessM q

matchIf :: Serializable a ⇒
(a → Bool) → (a →
ProcessM q) → MatchM q ()

match :: Serializable a ⇒
(a → ProcessM q) → MatchM q ()

Also: receiveTimeout and matchUnkown

```
instance Monad MatchM
receiveWait    :: [MatchM q ()] → ProcessM q
receiveTimeout :: Int → [MatchM q ()]
                                  → ProcessM (Maybe q)
match   :: Serializable a ⇒ (a → ProcessM q)
                                  → MatchM q ()
matchIf :: Serializable a ⇒ (a → Bool)
                         → (a → ProcessM q) → MatchM q ()
matchUnknown :: ProcessM q → MatchM q ()
```

Typed Channels

- We can use types to ensure that processes are prepared to accept the messages that are sent to them
- Instead of sending a message to a process, we send it on a channel, specialized for a single type
  - A channel is a pair of ports: a send port and a receive port

Channel Interface

```
newChan  :: Serializable a ⇒
                ProcessM (SendPort a, ReceivePort a)
sendChan :: Serializable a ⇒
                SendPort a → a → ProcessM ()
receiveChan :: Serializable a ⇒
                ReceivePort a → ProcessM a
mergePortsBiased :: Serializable a ⇒
                [ReceivePort a]→ ProcessM (ReceivePort a)
mergePortsRR :: Serializable a ⇒
                [ReceivePort a] → ProcessM (ReceivePort a)
```

SendPort a is serializable; ReceivePort a is *not* serializable

Ping-Pong: once more, with Channels

```
ping2 :: SendPort Ping → ReceivePort Pong →
                                              ProcessM ()
ping2 pingout pongin =
  do  { (Pong partnersPort) ← receiveChan pongin
      ; sendChan partnersPort (Ping pongin)
      ; ping2 pingout pongin }
```

Combing Ports

- Suppose that we have several communication partners,
  - e.g., messages arrive from the hardware that we are monitoring, and from other control processes in the network.
- We want to receive from one of several ports.

MergePortsBiased	CombinePortsBiased
MergePortsRR	CombinePortsRR

Serializing function closures

- Sending a function to a remote address space involves serializing not only its code, but also its free variables:

```
— wrong
sendFunc :: SendPort (Int→Int) → Int → ProcessM ()
sendFunc p x = sendChan p (λy → x + y + 1)
```

- The function being sent is `(λy → x + y + 1)`, which captures the variable `x`.

Key insight

- Whether a function is serializable or not has nothing to do with its *type*.
  - It depends on whether it has free variables,
  - whether those free variables are serializable

  which are *not* extensional properties of the function

Prior Solutions

- Make the runtime responsible for serializing anything and everything
  - But some things should be serialized specially
  - And others should not be serialized at all
- Java does essentially this
- Yet: *de*-serialization must still be built-in
  - this requires runtime reflection

More modest magic

- Some functions are easy to serialize
  - those with no free variables
  - How? Serialize the code address
    - assuming the same code is running at both ends
- We need a way of charactering such definitions as a *type:*

```
instance Serializable (Static a)
```

- Intuition: values of type (Static a) are always serializable, regardless what a is!

Static and non-Static types

- Two new terms: static exp and unstatic exp
  - intuition: static exp is well-typed iff exp can be serialized.
- Top-level bindings are tagged S; all others are tagged D
- A term static exp has type τ iff exp :: τ and all the free variables in exp are S-bound

$$\begin{array}{rcl} \Gamma & ::= & \overline{x :_\delta \sigma} \\ \delta & ::= & \mathsf{S} \mid \mathsf{D} \end{array}$$

$$\Gamma\downarrow \;=\; \{x :_{\mathsf{s}} \sigma \mid x :_{\mathsf{s}} \sigma \in \Gamma\}$$

$$\frac{\Gamma\downarrow \;\vdash e : \tau}{\Gamma \vdash \mathsf{static}\ e : \mathsf{Static}\ \tau} \qquad \text{(Static intro)}$$

$$\frac{\Gamma \vdash e : \mathsf{Static}\ \tau}{\Gamma \vdash \mathsf{unstatic}\ e : \tau} \qquad \text{(Static elim)}$$

Examples:

Examples:

```
id :: a → a
id x = x
```

id is S-bound, but has a non-static type.

$id :_S a \to a$

Examples:

$id :: a \to a$ $id\ x = x$	id is S-bound, but has a non-static type. $id :_S a \to a$
$f :: Static\ a \to (Static\ a, Int)$ $f\ x = (x, 3)$	x is D-bound, but has a static type $x :_D Static\ a$

Examples:

id :: a → a id x = x	id is S-bound, but has a non-static type. id $:_S$ a → a
f :: Static a → (Static a, Int) f x = (x, 3)	x is D-bound, but has a static type x $:_D$ Static a
static (length ∘ filter id)	Free variables of a static term need not have static types

- So what? We need to serialize functions that *do* have free variables.
- Static values make it possible to do closure conversion
- Let's try:

```
— wrong
data Closure a where
    MkClosure :: Static (env → a) → env → Closure a
```

- This makes the environment explicit:
    - env is the (existentially quantified) type of the environment of our function

- Slight snag: `env` is not serializable
- OK: let's make it so!

```
— still wrong
data Closure a where
    MkClosure :: Serialzable env ⇒
                            Static (env → a) → env → Closure a
    deriving Typeable
```

- Now serialization is easy:

```
instance Binary (Closure a) where
    put (MkClosure f env) = put f ≫ put env
```

- But what about *de*-serialization?

- Deserialization is a problem because, at the receiving end, we don't know what `env` is.
  - Can we send a representation of its type?
  - And then what?
    - Do a run-time type-class lookup?
  - Send a representation of the de-serialization function?
    - This would require us to serialize closures ...
- Simple and (in hindsight!) obvious solution:
  - get rid of the existential!

The solution

```
— finally right
data Closure a where
   MkClosure :: Static (ByteString → a) → ByteString →
                                                   Closure a
```

- Isn't this awfully restrictive?

  No! *Any* env that is serializable is equipped with encode and decode functions that convert it to and from a ByteString!

- The (de)-serialization is now done at closure-construction time

Examples

```
sendFunc :: SendPort (Closure (Int → Int)) → Int → ProcessM ()
sendFunc p x = sendChan p clo
  where clo = MkClosure (static sfun) (encode x)

sfun :: ByteString → Int → Int
sfun = λbs → let x = decode bs
             in λy → x + y + 1
```

Add newWorker example

Examp

p is a SendPort that expects a (Closure (Int→Int))

```
sendFunc :: SendPort (Closure (Int → Int)) → Int → ProcessM ()
sendFunc p x = sendChan p clo
   where clo = MkClosure (static sfun) (encode x)

sfun :: ByteString → Int → Int
sfun = λbs → let x = decode bs
             in λy → x + y + 1
```

Add newWorker example

Examp

p is a SendPort that expects a (Closure (Int→Int))

```
sendFunc :: SendPort (Closure (Int → Int)) → Int → ProcessM ()
sendFunc p x = sendChan p clo
   where clo = MkClosure (static sfun) (encode x)
sfun :: ByteString → Int → Int
sfun = λbs → let x = decode bs
             in λy → x + y + 1
```

In the Closure we put a *pre-serialized* version of the free variable x

Add newWorker example

Examp

p is a SendPort that expects a (Closure (Int→Int))

```
sendFunc :: SendPort (Closure (Int → Int)) → Int → ProcessM ()
sendFunc p x = sendChan p clo
  where clo = MkClosure (static sfun) (encode x)

sfun :: ByteString → Int → Int
sfun = λbs → let x = decode bs
             in λy → x + y + 1
```

In the Closure we put a *pre-serialized* version of the free variable x

sfun *de-serializes* its own argument

Add newWorker example

Portland State UNIVERSITY

Summary

- New type constructor Static, with built-in serialization.
- A new term form (static e)
- A new primitive function unstatic :: Static a → a
- These primitives let us construct closures manually and control when and how they are serialized.
  - This looks tiresome, and programmers will probably want some syntactic support: future work

Faking it

- Static is not yet implemented in GHC
- We use Template Haskel workarounds

```
sendFunc :: SendPort (Closure (Int → Int)) → Int → ProcessM ()
sendFunc p x = sendChan p ($( mkClosure 'add1) x)
add1 :: Int → Int → Int
add1 x y = x + y + 1

$(remotable ['add1])
```

- Programmer is still doing closure-conversion
  - by defining `add1` as a top-level function whose first argument is an explicit environment (Int)
  - mkClosure operates on the *names* of functions:
    - `mkClosure :: Name → Q Exp`

Assessment

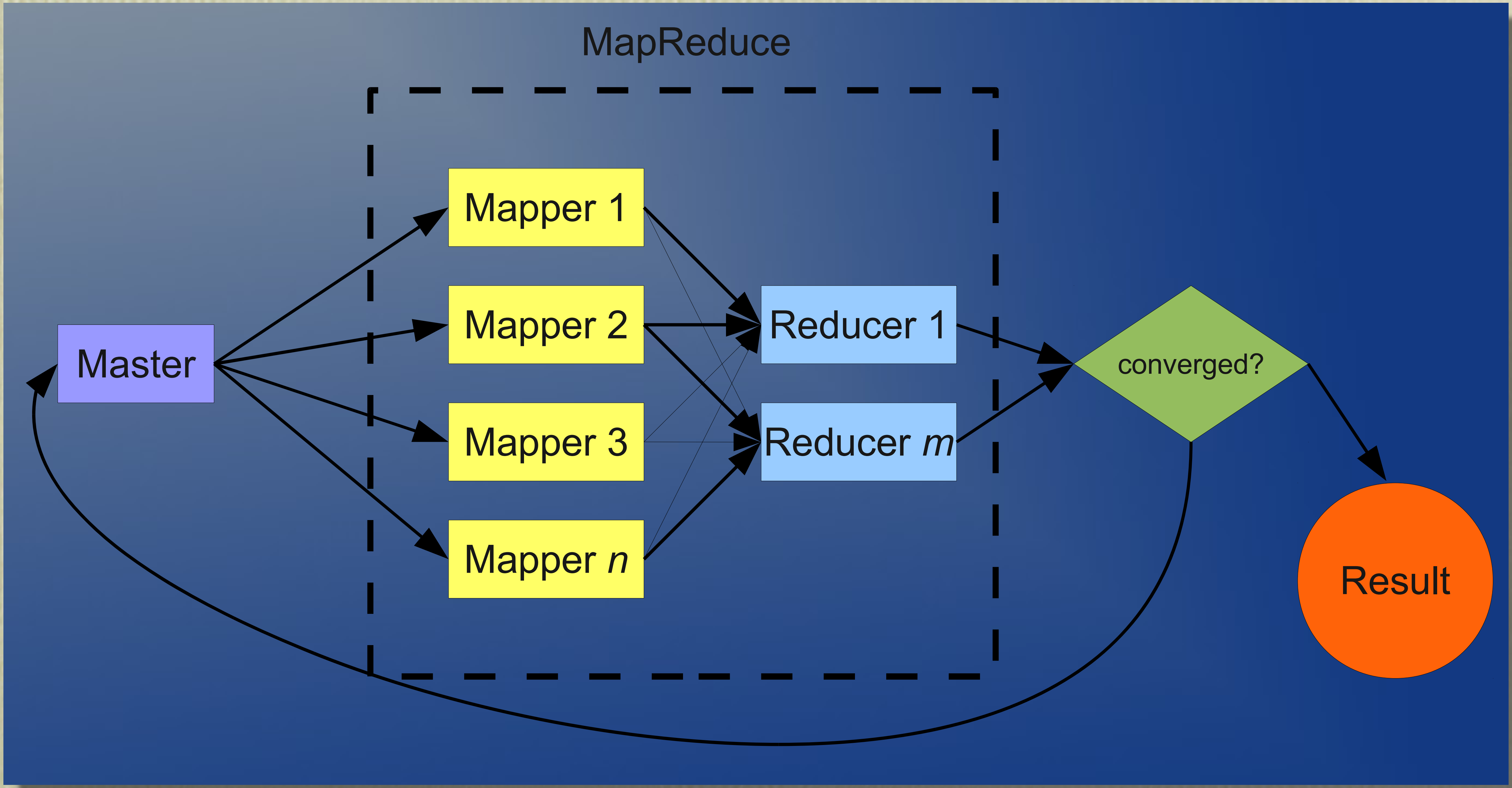
- Limited experience so far
- Small examples on local networks, and *k*-means on an Amazon EC2 cluster.

k-means

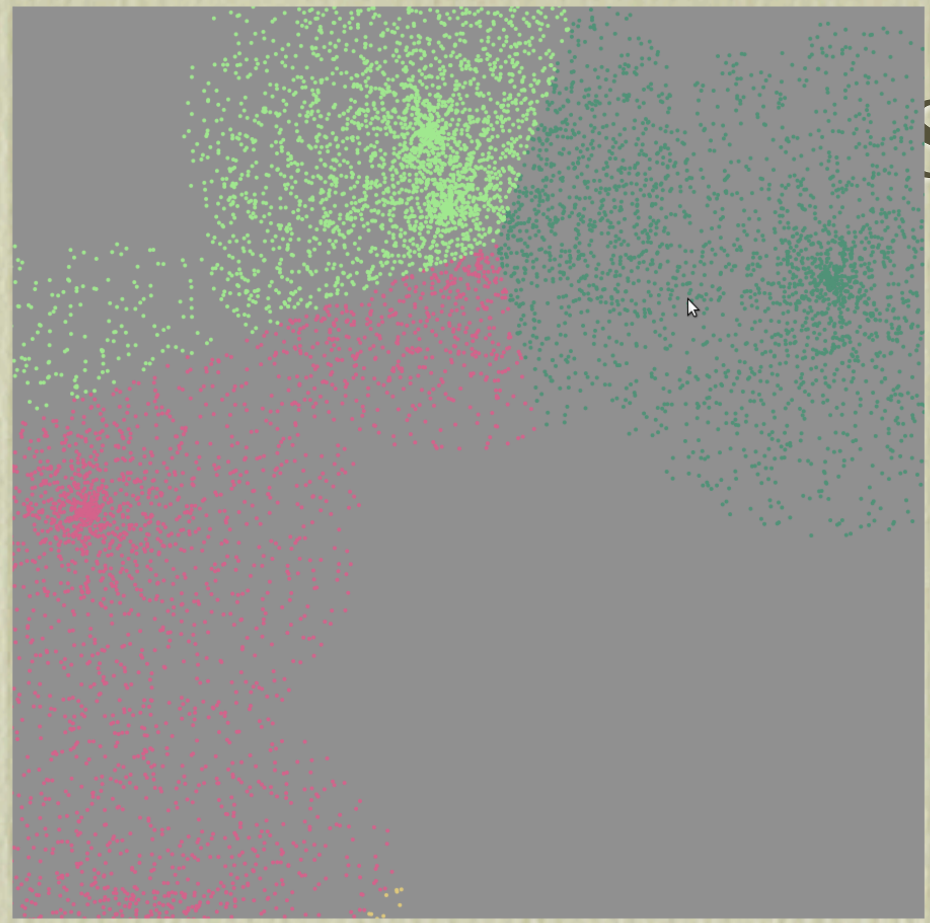
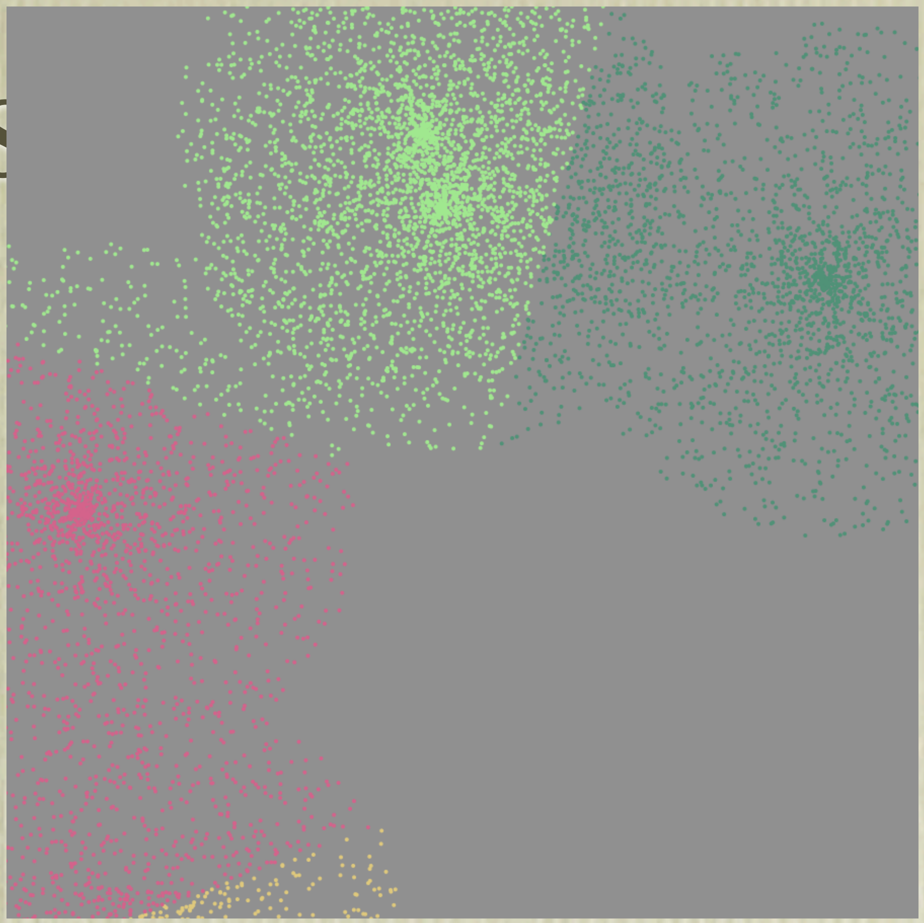
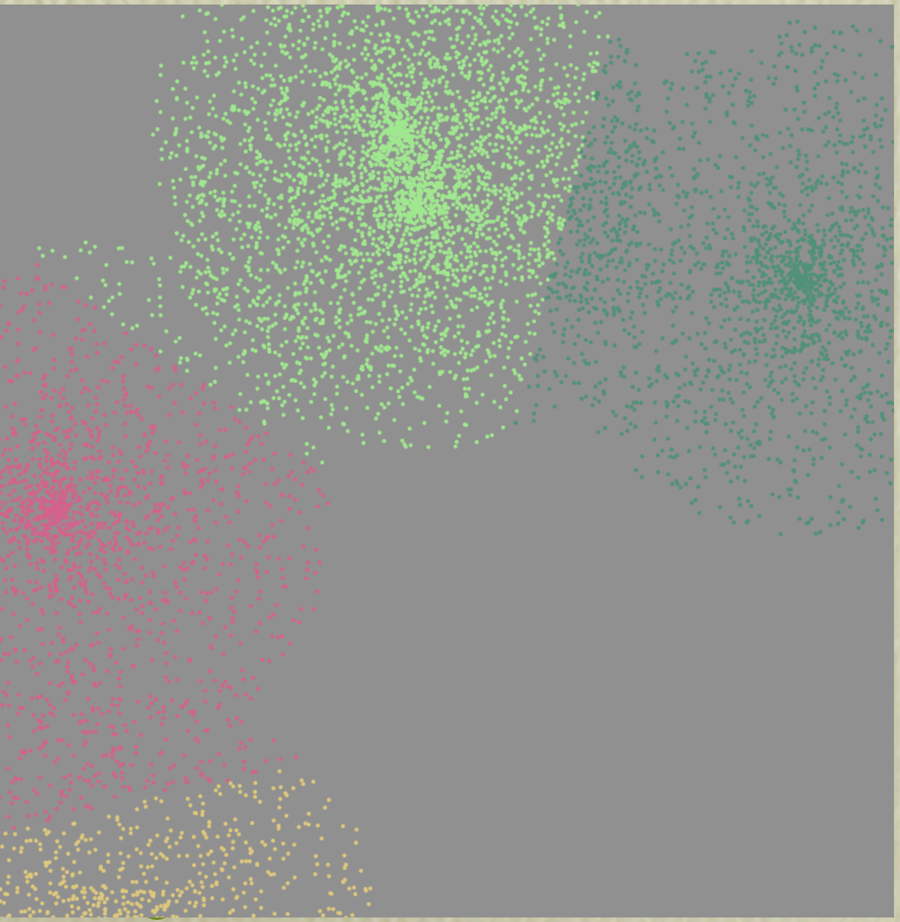
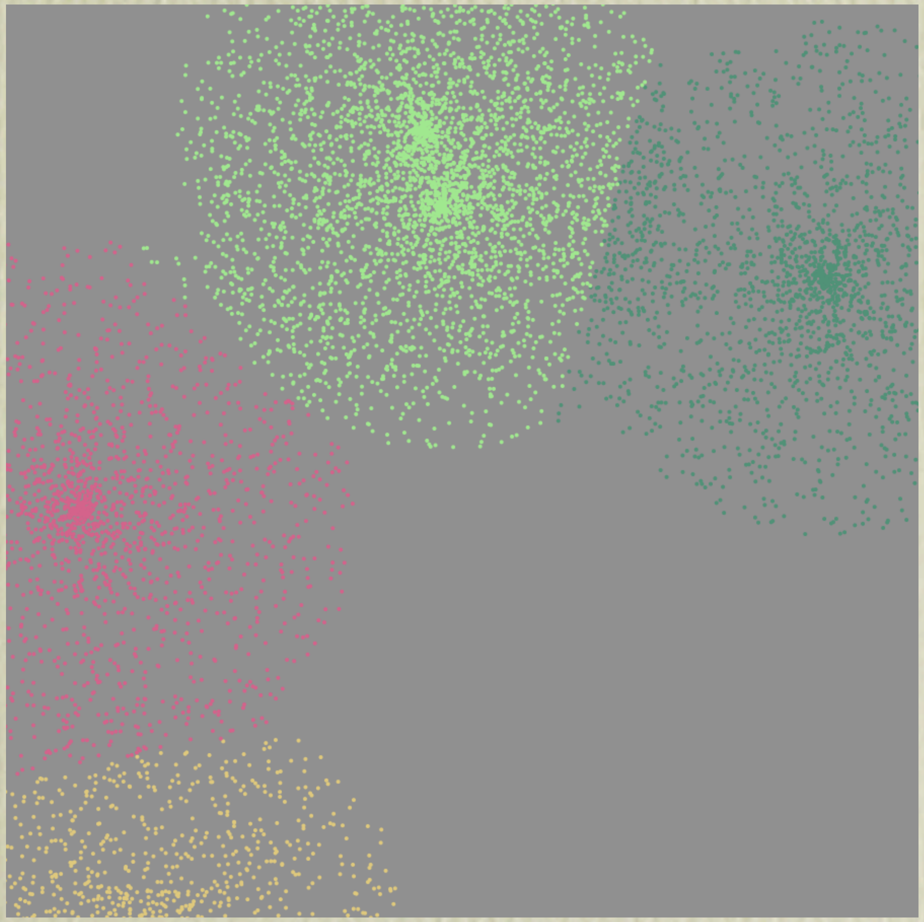
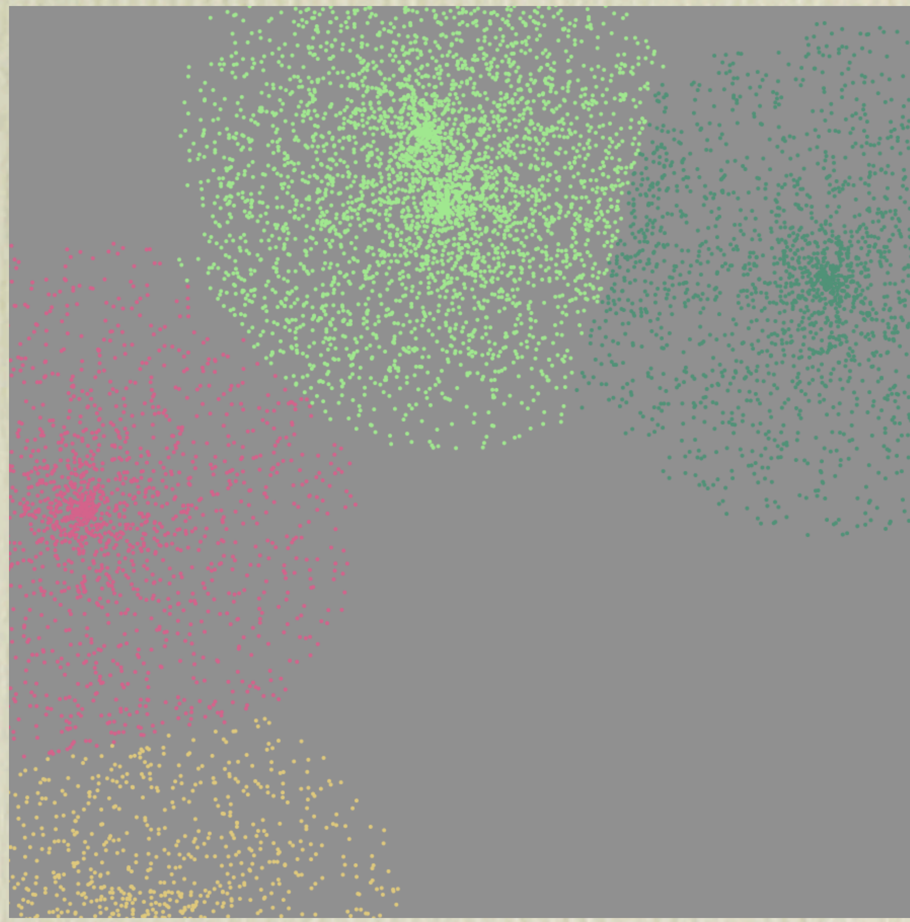
Data clustering algorithm:

1. Guess at centroids of *k* clusters
2. Put each point in nearest cluster
3. Compute the centroids of these cluster of points
4. Use the computed centroids as the next guess
5. Continue until convergence

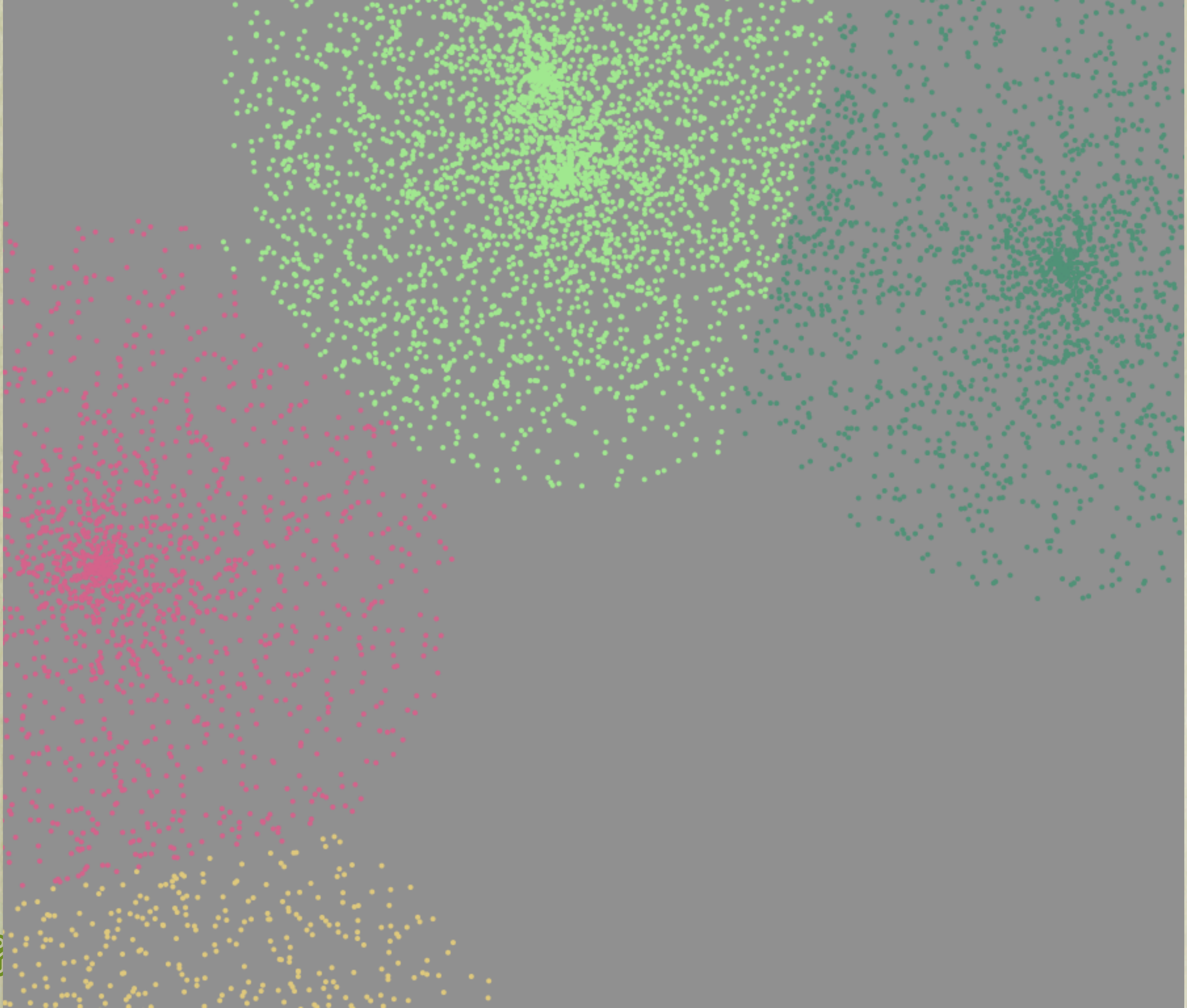
k-means

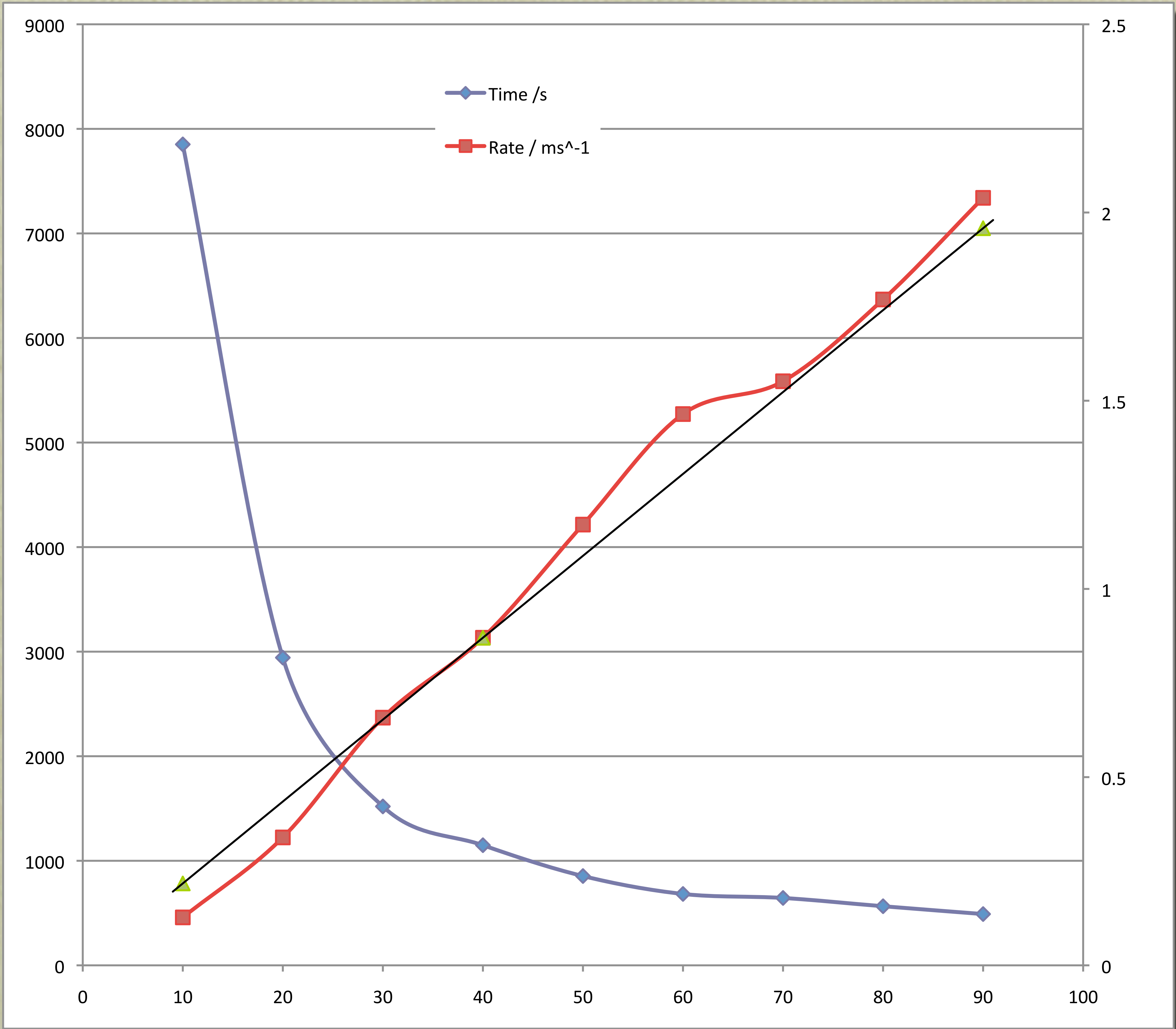

MapReduce
Master
Mapper 1
Mapper 2
Mapper 3
Mapper *n*
Reducer 1
Reducer *m*
converged?
Result

k-means results

k-means results





k-means results

nodes:

m1.small (1 core, 1.7 GB)

1 million 100-D points

one reducer

5 iterations

Related Work

- Inspired by Erlang
  - Also by Ciel execution engine and the Skywriting language [Murray *et al*]
- MPI from the HPC community
  - language independent
- RPC and RMI mechanisms
  - Birrell & Nelson, Emerald, CORBA, Java RMI, SOAP, …

- Distributed functional languages: GDH (distributed shared memory), Concurrent ML, paraML
  - Acute [Sewell *et al.*]: uses runtime representations of datatypes
  - HashCaml: does support serialization of function values, also with explicit type-passing
  - Alice [Rossberg's Thesis]
  - Clean: type-safe pickling, including function closures
- Our design point: serialization of closures is *not* built-in

Future Work

- Low level: implement Static in GHC
- Restartable task level
  - inspired by Skywriting project
  - tasks: idempotent, restartable computations
  - system tracks data dependencies between tasks
  - allocates tasks to processors
  - recovers from failure

Summary

- Cloud Haskell: a *starting point* for building distributed applications
- Contributions:
  - Typed version of Erlang's process & messaging interfaces
  - Typed channels; receive port is not Serializable
  - Serialization of function closures
  - It works (on 90 Amazon EC2 nodes)