Parallel and Concurrent Haskell
Part I
Simon Marlow
(Microsoft Research, Cambridge, UK)

All you need is X

• Where X is actors, threads, transactional memory, futures...
• Often true, but for a given application, some Xs will be much more suitable than others.
• In Haskell, our approach is to give you lots of different Xs
  – “Embrace diversity (but control side effects)” (Simon Peyton Jones)

Parallelism vs. Concurrency

• Primary distinguishing feature of Parallel Haskell: determinism
  – The program does “the same thing” regardless of how many cores are used to run it.
  – No race conditions or deadlocks
  – Add parallelism without sacrificing correctness
  – Parallelism is used to speed up pure (non-IO monad) Haskell code
Parallelism vs. Concurrency

- Primary distinguishing feature of Concurrent Haskell: threads of control
  - Concurrent programming is done in the IO monad
    - because threads have effects
    - effects from multiple threads are interleaved nondeterministically at runtime.
  - Concurrent programming allows programs that interact with multiple external agents to be modular
    - the interaction with each agent is programmed separately
    - Allows programs to be structured as a collection of interacting agents (actors)

I. Parallel Haskell

- In this part of the course, you will learn how to:
  - Do basic parallelism:
    - compile and run a Haskell program, and measure its performance
    - parallelise a simple Haskell program (a Sudoku solver)
    - use ThreadScope to profile parallel execution
    - do dynamic rather than static partitioning
    - measure parallel speedup
      - use Amdahl's law to calculate possible speedup
  - Work with Evaluation Strategies
    - build simple Strategies
    - parallelise a data-mining problem: K-Means
  - Work with the Par Monad
    - Use the Par monad for expressing dataflow parallelism
    - Parallelise a type-inference engine

Running example: solving Sudoku

- code from the Haskell wiki (brute force search with some intelligent pruning)
- can solve all 49,000 problems in 2 mins
- input: a line of text representing a problem

Solving Sudoku problems

- Sequentially:
  - divide the file into lines
  - call the solver for each line

Compile the program...

```
$ ghc -O2 sudoku1.hs -rtsopts
[1 of 2] Compiling Sudoku           ( Sudoku.hs, Sudoku.o )
[2 of 2] Compiling Main             ( sudoku1.hs, sudoku1.o )
Linking sudoku1 ...
```

Run the program...

```
-- solving a single Sudoku problem: 300 sec -> 7 min
2,392,127 bytes allocated in the heap
36,629,592 bytes copied during GC
191,168 bytes maximum residency (11 sample(s))
82,256 bytes maximum slop
2 MB total memory in use (0 MB lost due to fragmentation)
Generation 0:  4570 collections,  0 parallel,  0.14s,  0.13s elapsed
Generation 1:    11 collections,  0 parallel,  0.00s,  0.00s elapsed
...
MUT   time    2.92s  (  2.92s elapsed)
EXIT  time    0.00s  (  0.00s elapsed)
Total time    3.06s  (  3.06s elapsed)
```
Now to parallelise it...

- Doing parallel computation entails specifying coordination in some way – compute A in parallel with B
- This is a constraint on evaluation order
- But by design, Haskell does not have a specified evaluation order
- So we need to add something to the language to express constraints on evaluation order

The Eval monad

- Eval is pure
- Just for expressing sequencing between rpar/rseq – nothing more
- Compositional – larger Eval sequences can be built by composing smaller ones using monad combinators
- Internal workings of Eval are very simple (see Haskell Symposium 2010 paper)

What does rpar actually do?

- rpar creates a spark by writing an entry in the spark pool
  - rpar is very cheap (not a thread)
- the spark pool is a circular buffer
- when a processor has nothing to do, it tries to remove an entry from its own spark pool, or steal an entry from another spark pool (work stealing)
- when a spark is found, it is evaluated
- The spark pool can be full – watch out for spark overflow!

Basic Eval patterns

- To compute a in parallel with b, and return a pair of the results:
  - do a' <- rpar a
    b' <- rseq b
    return (a', b')

- alternatively:
  - do a' <- rpar a
    b' <- rseq b
    rseq a'
    return (a', b')

- what is the difference between the two?

Parallelising Sudoku

- Let's divide the work in two, so we can solve each half in parallel:
  - \texttt{let (as, bs) = splitAt (length grids `div` 2) grids}
  - \texttt{runEval $ do}
    as' <- rpar (map solve as)
    bs' <- rpar (map solve bs)
    rseq as'
    rseq bs'
    return ()

- Now we need something like

But this won’t work...

- rpar evaluates its argument to Weak Head Normal Form (WHNF)
- WTF is WHNF?
  - evaluates as far as the first constructor
    - e.g. for a list, we get either [] or (x:xs)
    - e.g. WHNF of "map solve (a:as)" would be "solve a : map solve as"
- But we want to evaluate the whole list, and the elements

import Control.Parallel.Strategies

\texttt{data Eval a}

\texttt{instance Monad Eval}

\texttt{runEval :: Eval a -> a}

\texttt{rpar :: a -> Eval a}

\texttt{rseq :: a -> Eval a}
We need 'deep'

```haskell
import Control.DeepSeq

deep :: NFData a => a -> a
deep a = deepseq a a
```

- deep fully evaluates a nested data structure and returns it
  - e.g. a list: the list is fully evaluated, including the elements
- uses overloading: the argument must be an instance of NFData
  - instances for most common types are provided by the library

Ok, adding deep

```haskell
runEval $ do
  as' <- rpar (deep (map solve as))
  bs' <- rpar (deep (map solve bs))
  rseq as'
  rseq bs'
  return ()
```

- (normally using the result would be enough to force evaluation, but we're not using the result here)

Let's try it...

- Compile sudoku2
  - (add-threaded -rtsopts)
  - run with `sudoku17.1000.txt +RTS -N2`
- Take note of the Elapsed Time

Runtime results...

```
$ ./sudoku2 sudoku17.1000.txt +RTS -N2
2,400,125,664 bytes allocated in the heap
48,845,008 bytes copied during GC
2,617,120 bytes maximum residency (7 sample(s))
313,496 bytes maximum slop
9 MB total memory in use (0 MB lost due to fragmentation)
Generation 0:  2975 collections,  2974 parallel,  1.04s,  0.15s elapsed
Generation 1:     7 collections,     7 parallel,  0.05s,  0.02s elapsed
Parallel GC work balance: 1.52 (6087267 / 3999565, ideal 2)
SPARKS: 2 (1 converted, 0 pruned)
INIT  time    0.00s  (  0.00s elapsed)
MUT   time    2.21s  (  1.80s elapsed)
GC    time    1.08s  (  0.17s elapsed)
EXIT  time    0.00s  (  0.00s elapsed)
Total time    3.29s  (  1.97s elapsed)
```

Calculating Speedup

- Calculating speedup with 2 processors:
  - Elapsed time (1 proc) / Elapsed Time (2 procs)
  - NB. not CPU time (2 procs) / Elapsed (2 procs)!
  - NB. compare against sequential program, not parallel program running on 1 proc

  - Speedup for sudoku2: 3.06/1.97 = 1.55
    - not great...

Why not 2?

- there are two reasons for lack of parallel speedup:
  - less than 100% utilisation (some processors idle for part of the time)
  - extra overhead in the parallel version
- Each of these has many possible causes...
A menu of ways to screw up

- less than 100% utilisation
  - parallelism was not created, or was discarded
  - algorithm not fully parallelised – residual sequential computation
  - uneven work loads
  - poor scheduling
  - communication latency
- extra overhead in the parallel version
  - overheads from rpar, work-stealing, deep, ...
  - larger memory requirements leads to GC overhead
  - GC synchronisation
  - duplicating work

So we need tools

- to tell us why the program isn’t performing as well as it could be
- For Parallel Haskell we have ThreadScope

Uneven workloads...

- So one of the tasks took longer than the other, leading to less than 100% utilisation

Partitioning

- Dividing up the work along fixed pre-defined boundaries, as we did here, is called static partitioning
  - static partitioning is simple, but can lead to under-utilisation if the tasks can vary in size
  - static partitioning does not adapt to varying availability of processors – our solution here can use only 2 processors

Dynamic Partitioning

- Dynamic partitioning involves
  - dividing the work into smaller units
  - assigning work units to processors dynamically at runtime using a scheduler
- Benefits:
  - copes with problems that have unknown or varying distributions of work
  - adapts to different number of processors: the same program scales over a wide range of cores
- GHC’s runtime system provides spark pools to track the work units, and a work-stealing scheduler to assign them to processors
- So all we need to do is use smaller tasks and more rpars, and we get dynamic partitioning
Revisiting Sudoku...

- So previously we had this:

```haskell
runEval $ do
  a <- rpar (deep (map solve as))
  b <- rpar (deep (map solve bs))
```

- We want to push `rpar` down into the map – each call to `solve` will be a separate spark

A parallel map

- Provided by `Control.Parallel.Strategies`
- Also: `parMap :: (a -> b) -> [a] -> Eval [b]`

```haskell
parMap f [] = return []
parMap f (a:as) = do
  b <- rpar (f a)
  bs <- parMap f as
  return (b:bs)
```

Create a spark to evaluate \( f(a) \) for each element `a`

Return the new list

Putting it together...

```haskell
evaluate $ deep $ runEval $ parMap solve grids
```

- NB. `evaluate $ deep` to fully evaluate the result list
- Code is simpler than the static partitioning version!

Results

```haskell
evaluate $ deep $ runEval $ parMap solve grids
```

- 1.7 speedup
- 5.2 speedup

Now 1.7 speedup

5.2 speedup
• Lots of GC
• One core doing all the GC work
  — indicates one core generating lots of data

Are there any sequential parts of this program?
• Reading the file, dividing it into lines, and
  traversing the list in parMap are all sequential
• but readFile, lines are lazy: some parallel work
  will be overlapped with the file parsing

Calculating possible speedup

• When part of the program is sequential, Amdahl’s law tells us what the maximum speedup is.

\[
\frac{1}{(1 - P) + \frac{P}{N}}
\]

• P = parallel portion of runtime
• N = number of processors
Applying Amdahl’s law

- In our case:
  - runtime = 3.06s (NB. sequential runtime!)
  - non-parallel portion = 0.038s (P = 0.9876)
  - N = 2, max speedup = 1 / ((1 – 0.9876) + 0.9876/2)
    - ~= 1.98
  - on 2 processors, maximum speedup is not affected much by this sequential portion
  - N = 64, max speedup = 35.93
  - on 64 processors, 38ms of sequential execution has a dramatic effect on speedup

Amdahl’s or Gustafson’s law?

- Amdahl’s law paints a bleak picture
  - speedup gets increasingly hard to achieve as we add more cores
  - returns diminish quickly when more cores are added
  - small amounts of sequential execution have a dramatic effect
  - proposed solutions include heterogeneity in the cores (e.g. one big core and several smaller ones), which is likely to create bigger problems for programmers
- See also Gustafson’s law — the situation might not be as bleak as Amdahl’s law suggests:
  - with more processors, you can solve a bigger problem
  - the sequential portion is often fixed or grows slowly with problem size
- Note: in Haskell it is hard to identify the sequential parts anyway, due to lazy evaluation

Evaluation Strategies

- So far we have used Eval/rpar/rseq
  - these are quite low-level tools
  - but it’s important to understand how the underlying mechanisms work
- Now, we will raise the level of abstraction
- Goal: encapsulate parallel idioms as re-usable components that can be composed together.

The Strategy type

- A Strategy is...
  - a function that,
  - when applied to a value ‘a’,
  - evaluates ‘a’ to some degree
  - (possibly sparking evaluation of sub-components of ‘a’ in parallel),
  - and returns an equivalent ‘a’ in the Eval monad
- NB. the return value should be observably equivalent to the original
  - (why not the same? we’ll come back to that...)

Example...

- A Strategy on lists that sparks each element of the list
- This is usually not sufficient – suppose we want to evaluate the elements fully (e.g. with deep), or do parList on nested lists.
- So we parameterise parList over the Strategy to apply to the elements:
Defining `parList`

```haskell
type Strategy a = a -> Eval a
parList :: Strategy a -> Strategy [a]
• We have the building blocks:
  rpar :: a -> Eval a
  :: Strategy a
parList (a -> Eval a) -> [a] -> Eval [a]
parList f []     = return ()
parList f (x:xs) = do
  x' <- rpar (runEval (f x))
  xs' <- parList f xs
  return (x':xs')
```

By why do Strategies return a value?

```haskell
parList (a -> Eval a) -> [a] -> Eval [a]
parList f []     = return ()
parList f (x:xs) = do
  x' <- rpar (runEval (f x))
  xs' <- parList f xs
  return (x':xs')
```

• Spark pool points to `(runEval (f x))`
• If nothing else points to this expression, the runtime will discard the spark, on the grounds that it is not required
• Always keep hold of the return value of `rpar`
• (see the notes for more details on this)

Let’s generalise...

• Instead of `parList` which has the sparking behaviour built-in, start with a basic traversal in the Eval monad:

```haskell
evalList :: Strategy a -> Strategy [a]
evalList f []     = return ()
evalList f (x:xs) = do
  x' <- f x
  xs' <- evalList f xs
  return (x':xs')
```

• and now:

```haskell
parList f = evalList (rpar `dot` f)
  where s1 `dot` s2 = s1 . runEval . s2
```

Generalise further...

• In fact, `evalList` already exists for arbitrary data types in the form of `traverse`.

```haskell
evalTraversable :: Traversable t => Strategy a -> Strategy (t a)
evalTraversable = traverse
```

• So, building Strategies for arbitrary data structures is easy, given an instance of Traversable.
• (not necessary to understand Traversable here, just be aware that many Strategies are just generic traversals in the Eval monad).

How do we use a Strategy?

```haskell
• We could just use `runEval`
  x `using` s = runEval (s x)
• But this is better:
  x `using` s = x
```

• Why better? Because we have a “law”:
  x `using` s = x
  We can insert or delete “using”s without changing the semantics of the program

Is that really true?

• Well, not entirely.

1. It relies on Strategies returning “the same value” (identity-safety)
   • Built-in Strategies obey this property
   • Be careful when writing your own Strategies

2. x ‘using’ s might do more evaluation than just x.
   • So the program with x ‘using’ s might be __, but the program with just x might have a value
   • if identity-safety holds, adding using cannot make the program produce a different result (other than __)
But we wanted ‘parMap’

- Earlier we used parMap to parallelise Sudoku
- But parMap is a combination of two concepts:
  - The algorithm, ‘map’
  - The parallelism, ‘parList’
- With Strategies, the algorithm can be separated from the parallelism.
  - The algorithm produces a (lazy) result
  - A Strategy filters the result, but does not do any computation – it returns the same result.

```
parMap f x = map f xs `using` parList
```

K-Means

- A data-mining algorithm, to identify clusters in a data set.

K-Means

- We use a heuristic technique (Lloyd’s algorithm), based on iterative refinement.
  1. Input: an initial guess at each cluster location
  2. Assign each data point to the cluster to which it is closest
  3. Find the centroid of each cluster (the average of all points)
  4. repeat 2-3 until clusters stabilise
- Making the initial guess:
  1. Input: number of clusters to find
  2. Assign each data point to a random cluster
  3. Find the centroid of each cluster
- Careful: sometimes a cluster ends up with no points!

K-Means: basics

```
data Vector = Vector Double Double
addVector :: Vector -> Vector -> Vector
  addVector (Vector a b) (Vector c d) = Vector (a+c) (b+d)
data Cluster = Cluster
  { clId    :: !Int,
    clCount :: !Int,
    clSum   :: !Vector,
    clCent  :: !Vector
  }
sqDistance :: Vector -> Vector -> Double
  -- square of distance between vectors
makeCluster :: Int -> [Vector] -> Cluster
  -- builds Cluster from a set of points
```

K-Means:

```
assign :: Int -- number of clusters
    -> [Cluster] -- clusters
    -> [Vector] -- points
    -> Array Int [Vector] -- points assigned to clusters
makeNewClusters :: Array Int [Vector] -> [Cluster]
  -- takes result of assign, produces new clusters
step :: Int -> [Cluster] -> [Vector] -> [Cluster]
  step nclusters clusters points = makeNewClusters (assign nclusters clusters points)
```

Putting it together.. sequentially

```
means_seq :: Int -> [Vector] -> [Cluster] -> IO [Cluster]
means_seq nclusters points clusters = do
  let
    loop :: Int -> [Cluster] -> IO [Cluster]
      loop n clusters | n > tooMany = return clusters
      loop n clusters = do
        hPrintf stderr "iteration %d\n" n
        hPutStr stderr (unlines (map show clusters))
        clusters' = step nclusters clusters points
        if clusters' == clusters
          then return clusters
          else loop (n+1) clusters'
  loop 0 clusters
```

- assign is step 2
- makeNewClusters is step 3
- step is (2,3) – one iteration
**Parallelise makeNewClusters?**

- essentially a map over the clusters
- number of clusters is small
- not enough parallelism here – grains are too large, fan-out is too small

**How to parallelise?**

- Parallelise assign?
  - essentially map/reduce: map nearest + accumArray
  - the map parallelises, but accumArray doesn’t
  - could divide into chunks... but is there a better way?

**Sub-divide the data**

- Suppose we divided the data set in two, and called step on each half
- We need a way to combine the results:
  
  \[
  \text{step } n \cdot \text{as} \cdot (n \cdot \text{bs}) = \text{step } n \cdot \text{as} \cdot \text{combine} \cdot \text{step } n \cdot \text{bs}
  \]
- but what is combine?
  
  \[
  \text{combine} :: [\text{Cluster}] \rightarrow [\text{Cluster}] \rightarrow [\text{Cluster}]
  \]
- assuming we can match up cluster pairs, we just need a way to combine two clusters

**Combining clusters**

- A cluster is notionally a set of points
- Its centroid is the average of the points
- A Cluster is represented by its centroid:

\[
\text{data Cluster} = \text{Cluster}
\]

  \[
  \begin{array}{c}
  \text{clId :: Int,} \\
  \text{clCount :: Int,} & \text{num of points} \\
  \text{clSum :: Vector,} & \text{sum of points} \\
  \text{clCent :: Vector,} & \text{clSum / clCount}
  \end{array}
\]

- but note that we cached clCount and clSum
- these let us merge two clusters and recompute the centroid in \(O(1)\)

**Combining clusters**

- So using

\[
\text{combineClusters :: Cluster \rightarrow Cluster \rightarrow Cluster}
\]

- we can define

\[
\text{reduce :: Int \rightarrow [[Cluster]] \rightarrow [Cluster]}
\]

- (see notes for the code; straightforward)
- now we can express K-Means as a map/reduce

**Final parallel implementation**

\[
\text{final_par :: Int \rightarrow Int \rightarrow [Vector] \rightarrow [Cluster] \rightarrow IO [Cluster]}
\]

- we can define

\[
\begin{array}{c}
\text{data Cluster} = \text{Cluster}
\end{array}
\]

  \[
  \begin{array}{c}
  \text{clId :: Int,} \\
  \text{clCount :: Int,} & \text{num of points} \\
  \text{clSum :: Vector,} & \text{sum of points} \\
  \text{clCent :: Vector,} & \text{clSum / clCount}
  \end{array}
\]

- but note that we cached clCount and clSum
- these let us merge two clusters and recompute the centroid in \(O(1)\)

\[
\text{combineClusters :: Cluster \rightarrow Cluster \rightarrow Cluster}
\]

- we can define

\[
\text{reduce :: Int \rightarrow [[Cluster]] \rightarrow [Cluster]}
\]

- (see notes for the code; straightforward)
- now we can express K-Means as a map/reduce
What chunk size?

- Divide data by number of processors?
  - No! Static partitioning could lead to poor utilisation (see earlier)
  - there’s no need to have such large chunks, the RTS will schedule smaller work items across the available cores

Further thoughts

- We had to restructure the algorithm to make the maximum amount of parallelism available
  - map/reduce
  - move the branching point to the top
  - make reduce as cheap as possible
  - a tree of reducers is also possible
- Note that the parallel algorithm is data-local – this makes it particularly suitable for distributed parallelism (indeed K-Means is commonly used as an example of distributed parallelism).
- But be careful of static partitioning

An alternative programming model

- Strategies, in theory:
  - Algorithm + Strategy = Parallelism
- Strategies, in practice (sometimes):
  - Algorithm + Strategy = No Parallelism
- Laziness is the magic ingredient that bestows modularity, but laziness can be tricky to deal with.
- The Par monad:
  - abandon modularity via laziness
  - get a more direct programming model
  - avoid some common pitfalls
  - modularity via higher-order skeletons

A menu of ways to screw up

- less than 100% utilisation
  - parallelism was not created, or was discarded
  - algorithm not fully parallelised – residual sequential computation
  - uneven work loads
  - poor scheduling
  - communication latency
- extra overhead in the parallel version
  - overheads from rpar, work-stealing, deep, ...
  - lack of locality, cache effects...
  - larger memory requirements leads to GC overhead
  - GC synchronisation
  - duplicating work

Par expresses dynamic dataflow

- Results for 170000 2-D points, 4 clusters, 1000 chunks
The \texttt{Par Monad}

- \texttt{Monad} instance
- \texttt{fork} and \texttt{runPar}
- \texttt{IVar} with \texttt{new}, \texttt{get}, and \texttt{put}

Parallel computations are pure (and hence deterministic), \texttt{forking} is explicit.

\texttt{Par} is a monad for parallel computation.

Examples

- \texttt{Par} can express regular parallelism, like \texttt{parMap}.
- First expand our vocabulary a bit:

\begin{verbatim}
spawn :: Par a -> Par (IVar a)
spawn p = do r <- new
          fork $ p >>= put r
          return r
\end{verbatim}

- Define \texttt{parMap} (actually \texttt{parMapM})

\begin{verbatim}
parMapM :: NFData b => (a -> Par b) -> [a] -> Par [b]
parMapM f as = do
    ibs <- mapM (spawn . f) as
    mapM get ibs
\end{verbatim}

- \texttt{Divide and conquer parallelism}:

\begin{verbatim}
parfib :: Int -> Int -> Par Int
parfib n = case n of
  | n <= 2    = return 1
  | otherwise = do
    x <- spawn $ parfib (n-1)
    y <- spawn $ parfib (n-2)
    return (x + y)
\end{verbatim}

Dataflow problems

- \texttt{Par} really shines when the problem is easily expressed as a dataflow graph, particularly an irregular or dynamic graph (e.g. shape depends on the program input).

Example

- Consider typechecking (or inferring types for) a set of non-recursive bindings.
- Each binding is of the form \( x = e \) for variable \( x \), expression \( e \).
- To typecheck a binding:
  - input: the types of the identifiers mentioned in \( e \)
  - output: the type of \( x \)
- So this is a dataflow graph
  - a node represents the typechecking of a binding
  - the types of identifiers flow down the edges

Example

- Consider typechecking (or inferring types for) a set of non-recursive bindings.
- Each binding is of the form \( x = e \) for variable \( x \), expression \( e \).
- To typecheck a binding:
  - input: the types of the identifiers mentioned in \( e \)
  - output: the type of \( x \)
- So this is a dataflow graph
  - a node represents the typechecking of a binding
  - the types of identifiers flow down the edges
Implementation

• We parallelised an existing type checker (nofib/infer).

• Algorithm works on a single term:

\[
\text{data Term} = \text{Let VarId Term Term} | \ldots
\]

• So we parallelise checking of the top-level Let bindings.

The parallel type inferencer

• Given:

\[
\text{inferTop} :: \text{TopEnv} \to \text{Term} \to \text{Par MonoType}
\]

• We need a type environment:

\[
\text{type TopEnv} = \text{Map VarId (IVar PolyType)}
\]

• The top-level inferencer has the following type:

\[
\text{inferTop} :: \text{TopEnv} \to \text{Term} \to \text{Par MonoType}
\]

Parallel type inference

\[
\text{inferTop} :: \text{TopEnv} \to \text{Term} \to \text{Par MonoType}
\]

\[
\text{inferTop} \text{topenv} (\text{Let x u v}) = \text{do}
\]

\[
\text{vu} \leftarrow \text{new}
\]

\[
\text{fork \& do}
\]

\[
\text{let fu} = \text{Set.toList (freeVars u)}
\]

\[
\text{tfu} \leftarrow \text{mapM (get \circ \text{fromJust \circ \text{flip \text{Map.lookup topenv}}) fu}
\]

\[
\text{let aa} = \text{makeEnv (zip fu tfu)}
\]

\[
\text{put vu (\text{inferTopRhs aa u})}
\]

\[
\text{inferTop} (\text{Map.insert x vu topenv}) v
\]

\[
\text{inferTop} \text{topenv} t = \text{do}
\]

• the boring case: invoke the normal sequential type inference engine

Results

• -N1: 1.12x

• -N2: 0.60s (1.87x speedup)

• available parallelism depends on the input: these bindings only have two branches

Thoughts to take away...

• Parallelism is not the goal

  – Making your program faster is the goal

  – (Unlike Concurrency, which is a goal in itself)

  – If you can make your program fast enough without parallelism, all well and good

  – However, designing your code with parallelism in mind should ensure that it can ride Moore’s law a bit longer

  – maps and trees, not folds

Open research problems?

• How to do safe nondeterminism

• Par monad:

  – implement and compare scheduling algorithms

  – better raw performance (integrate more deeply with the RTS)

• Strategies:

  – ways to ensure identity safety

  – generic clustering