Parallel Haskell on MultiCores and Clusters

(part of Advanced Development Techniques)

Hans-Wolfgang Loidl

School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh
(presented at the University of Milan)

Part 1.
Haskell Introduction

Characteristics of Functional Languages

GpH (Glasgow Parallel Haskell) is a conservative extension to the purely-functional, non-strict language Haskell.
Thus, GpH provides all the of the advanced features inherited from Haskell:

- Sophisticated polymorphic type system, with type inference
- Pattern matching
- Higher-order functions
- Data abstraction
- Garbage-collected storage management

Most relevant for parallel execution is *referential transparency*:

> *The only thing that matters about an expression is its value, and any subexpression can be replaced by any other equal in value. [Stoy, 1977]*

Consequences of Referential Transparency

Equational reasoning:

- Proofs of correctness are much easier than reasoning about state as in procedural languages.
- Used to *transform* programs, e.g. to transform simple specifications into efficient programs.

Freedom from execution order:

- Meaning of program is not dependent on execution order.
- *Lazy evaluation:* an expression is only evaluated when, and if, it is needed.
- *Parallel/distributed evaluation*. Often there are many expressions that can be evaluated at a time, because we know that the order of evaluation doesn't change the meaning, the sub-expressions can be evaluated in parallel (Wegner 1978)

Elimination of *side effects* (unexpected actions on a global state).

Characteristics of Functional Languages

Like other modern functional languages e.g. F# or Racket, Haskell includes *advanced features:*

- Sophisticated polymorphic type system, with type inference.

  $length :: [a] \to Int$

- Pattern matching.

  $length :: [a] \to Int$
  $length\ [] = 0$
  $length\ (x : xs) = 1 + length\ xs$

- Higher-order functions.

  $map\ (*2)\ [1, 2, 3, 4]$

- Data abstraction.

  $\textbf{data}\ MyList\ a = Nil \mid Cons\ a\ (MyList\ a)$

- Garbage-collected storage management.

Preliminaries

Basic types in Haskell:

- *Bool*: boolean values: *True* und *False*
- *Char*: characters
- *String*: strings (as list of characters)
- *Int*: fixed precision integers
- *Integer*: arbitrary precision integers
- *Float*: single-precision floating point numbers

Preliminaries

Compound types:

- *Lists*: $[\cdot]$, e.g. $[Int]$ list of (fixed precision) integers;
- *Tupels*: $(\cdot, \cdots)$, e.g. $(Bool, Int)$ tupel of boolean values and integers;
- *Records*: $\cdot\ \{\cdot, \cdots\}$, e.g. $BI\ \{\ b :: Bool,\ i :: Int\ \}$ a record of boolean values and integers;
- *Functions*: $a \to b$, e.g. $Int \to Bool$

Typesynonyms can be defined like this:

$\textbf{type}\ IntList = [Int]$

Haskell Types & Values

Note that all Haskell values are *first-class*: they may be passed as arguments to functions, returned as results, placed in data structures.

Example

Example Haskell values and types:

$$
\begin{array}{ll}
5 & :: \mathit{Integer} \\
'a' & :: \mathit{Char} \\
\mathit{True} & :: \mathit{Bool} \\
\mathit{inc} & :: \mathit{Integer} \to \mathit{Integer} \\
[1,2,3] & :: [\mathit{Integer}] \\
('b',4) & :: (\mathit{Char}, \mathit{Integer})
\end{array}
$$

N.B: The "::" can be read "has type."

Function Definitions

Functions are normally defined by a series of equations. Giving type signatures for functions is optional, but highly recommended.

$$
\begin{array}{l}
\mathit{inc} :: \mathit{Integer} \to \mathit{Integer} \\
\mathit{inc}\ n = n + 1
\end{array}
$$

To indicate that an expression e1 evaluates to another expression e2, we write

Evaluation

e1 $\Rightarrow$e2

Evaluation

inc (inc 3) $\Rightarrow$*5*

User-defined Types

Data constructors are introduced with the keyword **data**.
Nullary data constructors, or enumerated types:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Bool} = \mathit{False} \mid \mathit{True} \\
\textbf{data}\ \mathit{Color} = \mathit{Red} \mid \mathit{Green} \mid \mathit{Blue} \\
\textbf{data}\ \mathit{Day} = \mathit{Mon} \mid \mathit{Tue} \mid \mathit{Wed} \mid \mathit{Thu} \mid \mathit{Fri} \mid \mathit{Sat} \mid \mathit{Sun}
\end{array}
$$

Pattern matching over such data types is frequently used to make a case distinction over a user-defined (algebraic) data-type:

$$
\begin{array}{ll}
\mathit{next} & :: \mathit{Day} \to \mathit{Day} \\
\mathit{next}\ \mathit{Mon} & = \mathit{Tue} \\
\mathit{next}\ \mathit{Tue} & = \mathit{Wed} \\
\mathit{next}\ \mathit{Wed} & = \mathit{Thu} \\
\mathit{next}\ \mathit{Thu} & = \mathit{Fri} \\
\mathit{next}\ \mathit{Fri} & = \mathit{Sat} \\
\mathit{next}\ \mathit{Sat} & = \mathit{Sun} \\
\mathit{next}\ \mathit{Sun} & = \mathit{Mon}
\end{array}
$$

User-defined Types

A recursive data constructor:

$$
\textbf{data}\ \mathit{Tree}\ a = \mathit{Leaf}\ a \mid \mathit{Branch}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)
$$

$$
\begin{array}{ll}
\mathit{fringe} & :: \mathit{Tree}\ a \to [a] \\
\mathit{fringe}\ (\mathit{Leaf}\ x) & = [x] \\
\mathit{fringe}\ (\mathit{Branch}\ \mathit{left}\ \mathit{right}) & = \mathit{fringe}\ \mathit{left}\ +\!+\ \mathit{fringe}\ \mathit{right}
\end{array}
$$

Here ++ is the infix operator that concatenates two lists.
N.B: type constructor names must be capitalised.
N.B: This is the same declaration in *SML:*

$$
\begin{array}{l}
\textbf{datatype}\ 'a\ \mathit{binTree} = \mathit{leaf} \\
\qquad\qquad\qquad\qquad \mid \mathit{node}\ \textbf{of}\ 'a * 'a\ \mathit{binTree} * 'a\ \mathit{binTree};
\end{array}
$$

Type Synonyms

Type synonyms are names for commonly used types, rather than new types, and defined with the keyword **type**:

$$\textbf{type}\ String = [Char]$$
$$\textbf{type}\ Person = (Name, Address)$$
$$\textbf{type}\ Name = String$$

$$\textbf{data}\ Address = None \mid Addr\ String$$

Syntactic support is provided for strings, e.g. ``bye'' ⇒`['b','y','e']`, and list operations operations can be applied to them, e.g. `length ``bye''` ⇒3.

Pattern Matching

A pattern may contain a **wildcard**, e.g. to chose just the first `n` elements of a list,

Evaluation

take 2 [1,2,3] ⇒*[1,2]*

$$take\ 0\ \_ = []$$
$$take\ \_\ [] = []$$
$$take\ n\ (x : xs) = x\ :\ take\ (n-1)\ xs$$

Guarded Patterns

A pattern may contain a **guard**: a condition that must be true for the pattern to match, e.g.

$$\begin{array}{ll} sign\ x & \mid x > 0 \quad = 1 \\ & \mid x == 0 = 0 \\ & \mid x < 0 \quad = -1 \end{array}$$

Lists

Constructed from cons (`:`) and nil (`[]`), e.g.

$$1 : []$$
$$1 : 2 : 3 : []$$
$$'b' : 'y' : 'e' : []$$

having types [*Integer*], [Integer] and [Char]. Lists are commonly abbreviated:

$$[1]$$
$$[1, 2, 3]$$
$$['b', 'y', 'e']$$

Lists

A list can be indexed with the !! operator:

Evaluation

$[1,2,3]\ !!\ 0 \Rightarrow 1$
$['b','y','e']\ !!\ 2 \Rightarrow 'e'$

A list can be enumerated:

Evaluation

$[1\ ..\ 5] \Rightarrow [1,2,3,4,5]$

List Comprehensions

"*List comprehensions*" are a short-hand notation for defining lists, with notation similar to set comprehension in mathematics. They have been introduced as ZF-comprehensions in Miranda, a pre-cursor of Haskell, and are also supported in Python.
Example: List of square values of all even numbers in a given list of integers *xs*:

$$sq\_even\ xs\ =\ [x*x\ |\ x\ \leftarrow\ xs,\ even\ x]\ sq\_even\ xs\ =\ [\underline{x*x}\ |\ x\ \leftarrow\ xs,\ even\ x$$

The expression $\underline{x*x}$ is the body of the list comprehension. It defines the value of each list element.
The expression $\underline{x \leftarrow xs}$ is the generator and binds the elements in *xs* to the new variable *x*, one at a time.
The condition $\underline{even\ x}$ determines, whether the value of *x* should be used in computing the next list element.

List comprehension example

List comprehensions enable list functions to be expressed concisely:

$$\begin{array}{l} quicksort\ []\ =\ [] \\ quicksort\ (x:xs)\ = \\ \qquad quicksort\ [y\ |\ y\ \leftarrow\ xs,\ y<x\ ]\ ++ \\ \qquad [x]\ ++ \\ \qquad quicksort\ [y\ |\ y\ \leftarrow\ xs,\ y>=x] \end{array}$$

Polymorphic Functions

A *polymorphic function* (generic method in Java or C#) can operate on values of many types, e.g.

$$\begin{array}{lll} length & :: [a] \rightarrow Integer \\ length\ [] & = 0 \\ length\ (x:xs) & = 1\ +\ length\ xs \end{array}$$

Evaluation

$length\ [1,2,3] \Rightarrow 3$
$length\ ['b','y','e'] \Rightarrow 3$
$length\ [[1],[2]] \Rightarrow 2$

N.B: a is a type variable, that can stand for any type.

Local Definitions

Haskell, like SML has a mutually recursive **let** binding, e.g.

$$\begin{array}{lll} \textbf{let} & y & = a * b \\ & f\ x & = (x + y)/y \\ \textbf{in} & f\ c\ +\ f\ d & \end{array}$$

The Haskell **where** binding scopes over several guarded equations:

$$\begin{array}{ll} f\ x\ y & \mid y > z \ \ = \ldots \\ & \mid y == z = \ldots \\ & \mid y < z \ \ = \ldots \\ & \quad\quad \textbf{where}\ z\ =\ x * x \end{array}$$

Layout

Haskell lays out equations in columns to disambiguate between multiple equations, e.g. could previous definition be:

$$\begin{array}{ll} \textbf{let} & y\ =\ a * b\ f \\ & x\ =\ (x + y)/y \\ \textbf{in} & f\ c\ +\ f\ d \end{array}$$

Key rule: declarations start to the right of **where** or **let**.

Curried & Infix Functions

Currying: a function requiring n arguments can be applied to fewer arguments to get another function, e.g.

$$\begin{array}{l} add\ x\ y\ =\ x\ +\ y \\ inc\ ::\ Integer\ \rightarrow\ Integer \\ inc\ =\ add\ 1 \end{array}$$

Example

Define the sum over list of squares over all even numbers:

$$sqs\_even\ ::\ [Integer]\ \rightarrow\ Integer$$

$$\begin{array}{lll} sqs\_even\ [] & & =\ 0 \\ sqs\_even\ (x : xs) & \mid even\ x & =\ x^2\ +\ sqs\_even\ xs \\ & \mid otherwise & =\ sqs\_even\ xs \end{array}$$

Higher Order Functions

Functions are first class values and can be *passed as arguments to other functions and returned as the result* of a function.
Many useful higher-order functions are defined in the Prelude and libraries, including most of those from your SML course, e.g.
filter takes a list and a boolean function and produces a list containing only those elements that return `True`

$$filter\ ::\ (a\ \rightarrow\ Bool)\ \rightarrow\ [a]\ \rightarrow\ [a]$$

$$\begin{array}{lll} filter\ p\ [] & & =\ [] \\ filter\ p\ (x : xs) & \mid p\ x & =\ x\ :\ filter\ p\ xs \\ & \mid otherwise & =\ filter\ p\ xs \end{array}$$

Evaluation

filter even [1,2,3]⇒[2]

The higher-order function map

map applies a function f to every element of a list

$$\begin{aligned} &map :: (a \to b) \to [a] \to [b] \\ &map\ f\ [] \quad\ = [] \\ &map\ f\ (x : xs) = (f\ x) : map\ f\ xs \end{aligned}$$

Evaluation

$\underline{map\ inc\ [2,3]}$
$\Rightarrow (inc\ 2) : \underline{map\ inc\ [3]}$
$\Rightarrow (inc\ 2) : (inc\ 3) : \underline{map\ inc\ []}$
$\Rightarrow (inc\ 2) : (inc\ 3) : \underline{[]}$
...
$\Rightarrow [3,4]$

In general:

Evaluation

$$map\ f\ [x_0, x_1, \ldots, x_n] \Longrightarrow [f\ x_0, \ldots, f\ x_n]$$

map example

Example: sum over list of squares over all even numbers:

$$\begin{aligned} &sqs\_even :: [Integer] \to Integer \\ &sqs\_even\ xs = sum\ (map\ (\lambda x \to x * x)\ (filter\ even\ xs)) \end{aligned}$$

The higher-order function *foldr*

foldr applies a binary function to every element of a list:

$$\begin{aligned} &foldr :: (a \to b \to b) \to b \to [a] \to b \\ &foldr\ f\ z\ [] \quad\ = z \\ &foldr\ f\ z\ (x : xs) = f\ x\ (foldr\ f\ z\ xs) \end{aligned}$$

Example:

Evaluation

$\underline{foldr\ add\ 0\ [2,3]}$
$\Rightarrow add\ 2\ (\underline{foldr\ add\ 0\ [3]})$
$\Rightarrow add\ 2\ (add\ 3\ (\underline{foldr\ add\ 0\ []}))$
...
$\Rightarrow 5$

In general: *foldr* replaces every : in the list by an f, and the [] by an v:

Evaluation

$$foldr \oplus v (x_0 : \ldots : x_n : []) \Longrightarrow x_0 \oplus \ldots \oplus (x_n \oplus v)$$

zip converts a pair of lists into a list of pairs:

$$zip :: [a] \to [b] \to [(a, b)]$$

Evaluation

$zip\ [1,2,3]\ [9,8,7] \Rightarrow [(1,9),(2,8),(3,7)]$

`zipWith` takes a pair of lists and a binary function and produces a list containing the result of applying the function to each 'matching' pair:

Evaluation

$$\begin{aligned} &zipWith \oplus (x_0 : \ldots : x_n : [])\ (y_0 : \ldots : y_n : []) \\ &\Longrightarrow (x_0 \oplus y_0) : \ldots : (x_n \oplus y_n) \end{aligned}$$

Example

```
dotProduct xs ys = sum (zipWith (*) xs ys)
```

Lambda Abstractions

Functions may be defined "anonymously" via a lambda abstraction (`fn` in SML). For example definitions like

$$inc\ x = x + 1$$
$$add\ x\ y = x + y$$

are really shorthand for:

$$inc = \backslash x \rightarrow x + 1$$
$$add = \backslash x\ y \rightarrow x + y$$

Lambda Expressions	
SML:	*Haskell:*
`fn x => ...`	`\ x -> ...`

Infix Operators

Infix operators are really just functions, and can also be defined using equations, e.g. list concatenation:

$$(++) :: [a] \rightarrow [a] \rightarrow [a]$$
$$[] ++ ys = ys$$
$$(x : xs) ++ ys = x : (xs ++ ys)$$

and function composition:

$$(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$$
$$f\ .\ g = \backslash x \rightarrow f\ (g\ x)$$

Lexically, infix operators consist entirely of "symbols," as opposed to normal identifiers which are alphanumeric.

Function composition	
SML:	*Haskell:*
`f o g`	`f . g`

Sections

Since operators are just functions, they can be curried, e.g. (parentheses mandatory)

$$(x+) = \backslash y \rightarrow x + y$$
$$(+y) = \backslash x \rightarrow x + y$$
$$(+) = \backslash x\ y \rightarrow x + y$$

Example

The sum over list of squares of all even numbers:

$$sqs\_even :: [Integer] \rightarrow Integer$$
$$sqs\_even = foldr\ (+)\ 0\ .\ map\ (\lambda x \rightarrow x * x)\ .\ filter\ even$$

Lazy Functions

Most programming languages have *strict* semantics: the arguments to a function are evaluated before the evaluating the function body. This sometimes wastes work, e.g.

$$f\ True\ y = 1$$
$$f\ False\ y = y$$

It may even cause a program to fail when it could complete, e.g.

Evaluation

$$f\ True\ (1/0) \Rightarrow ?$$

It may even cause a program to fail when it could complete, e.g.

Evaluation

$$f\ True\ (1/0) \Rightarrow 1$$

Haskell functions are *lazy*: the evaluation of the arguments is delayed, and

"Infinite" Data Structures

As an example of lazy evaluation, consider the following function:

$$\begin{aligned} foo\ x\ y\ z\ &= \textbf{if}\ \underline{x < 0}\ \textbf{then}\ abs\ x \\ &\quad \textbf{else}\ x + y foo\ x\ y\ z\ = \textbf{if}\ x < 0\ \textbf{then}\ \underline{abs\ x} \\ &\quad \textbf{else}\ x + y foo\ x\ y\ z\ = \textbf{if}\ x < 0\ \textbf{then}\ abs\ x \\ &\quad \textbf{else}\ \underline{x + y} \end{aligned}$$

Evaluation order:

- Evaluating the conditional requires the evaluation of $\underline{x < 0}$ which in turn requires the evaluation of the argument x.
- If $x < 0$ is True, the value of $\underline{abs\ x}$ will be returned; in this case *neither* y *nor* z will be evaluated.
- If $x < 0$ id False, the value of $\underline{x + y}$ will be returned; this requires the evaluation of y.
- z won't be evaluated in either of these two cases.
- In particular, the expression $foo\ 1\ 2\ (1\ `div`\ 0)$ is well-defined.

"Infinite" Data Structures

Data constructors are also lazy (they're just a special kind of function), e.g. list construction (:)

Non-strict constructors permit the definition of (conceptually) infinite data structures. Here is an infinite list of ones:

$$ones\ =\ 1\ :\ ones$$

More interesting examples, successive integers, and all squares (using infix exponentiation):

$$\begin{aligned} numsFrom\ n &= n\ :\ numsFrom\ (n+1) \\ squares\ \ \ \ &= map\ (\hat{}2)\ (numsFrom\ 0) \end{aligned}$$

Any program only constructs a finite part of an infinite sequence, e.g.

Evaluation

take 5 squares $\Rightarrow$*[0,1,4,9,16]*

Infinite Data-structures

The prelude function for selecting the n-the element of a list:

$$\begin{aligned} [] !!\ \_\ &= error\ "Empty\ list" \\ (x : \_) !!\ 0\ &= x \\ (\_ : xs) !!\ n\ &= xs !!\ (n-1) \end{aligned}$$

Here is the evaluation order of [0..]!!2:

Evaluation

[0..]!!2	$\Longrightarrow$	*is the list empty?*
(0 : [1..])!!2	$\Longrightarrow$	*is the index 0?*
[1..]!!1	$\Longrightarrow$	*is the list empty?*
(1 : [2..])!!1	$\Longrightarrow$	*is the index 0?*
[2..]!!0	$\Longrightarrow$	*is the list empty?*
(2 : [3..])!!0	$\Longrightarrow$	*is the index 0?*
2		

Example Sieve of Erathostenes

Compute a list of all prime numbers by,

1. starting with a list of all natural numbers,
2. take the first number in the list and cancel out all its multiples,
3. repeat Step 2 forever.

```
-- iteratively remove all multiples of identified prime numbers
sieve :: [Integer] -> [Integer]
sieve (p:xs) = p : sieve (removeMults p xs)

-- remove all multiples of a given number
removeMults :: Integer -> [Integer] -> [Integer]
removeMults p xs = [ x | x <- xs, not (x `rem` p == 0) ]

-- define an infinite list of prime numbers
primes :: [Integer]
primes = sieve [2..]
```

An example of an infinite data structure

The goal is to create a list of *Hamming numbers*, i.e. numbers of the form $2^i 3^j 5^k \quad \text{for} \quad i, j, k \in \mathbb{N}$

```
hamming = 1 : map (2*) hamming `union`
              map (3*) hamming `union`
              map (5*) hamming

union a@(x:xs) b@(y:ys) = case compare x y of
            LT -> x : union  xs  b
            EQ -> x : union  xs  ys
            GT -> y : union  a   ys
```

Note, that `hamming` is an infinite list, defined as a cyclic data structure, where the computation of one element depends on prior elements in the list.

[0] Solution from Rosetta code

Normal Forms

Normal forms are defined in terms of reducable expressions, or **redexes**, i.e. expressions that can be simplified e.g. `(+) 3 4` is reducable, but `7` is not. Strict languages like SML reduce expressions to *Normal Form (NF)*, i.e. until no redexes exist (they are "fully evaluated"). Example NF expressions:

5
$[4, 5, 6]$
$\backslash \mathtt{x} \rightarrow \mathtt{x} + 1$

Lazy languages like Haskell reduce expressions to *Weak Head Normal Form (WHNF)*, i.e. until no top level redexes exist. Example WHNF expressions:

$(:)\, 2\, [2+1]$ *usually written as* $2 : [2+1]$

Examples

Example *non*-WHNF expressions:

$(+)\, 4\, 1$
$(\backslash x \rightarrow x + 1)\, 3$

Example WHNF expressions that are *not in NF*:

$(*)\, ((+)\, 4\, 1)$
$\backslash x \rightarrow 5 + 1$
$(3+1) : [4, 5]$
$(2^2) : (map\, (^2)\, [4, 6])$

N.B: In a parallel non-strict language threads, by default, reduce expressions to WHNF.

In addition to the *parametric* polymorphism already discussed, e.g.

$length :: [a] \rightarrow Int$

Haskell also supports *ad hoc* polymorphism or overloading, e.g.

- 1, 2, etc. represent both fixed and arbitrary precision integers.
- Operators such as + are defined to work on many different kinds of numbers.
- Equality operator (==) works on numbers and other types.

Note that these overloaded behaviors are different for each type and may be an error, whereas in parametric polymorphism the type truly does not matter, e.g. `length` works for lists of any type.

Declaring Classes and Instances

It is useful to define equality for many types, e.g. `String`, `Char`, `Int`, etc, but not all, e.g. functions.
A Haskell class declaration, with a single method:

$$\begin{aligned}&\textit{class Eq a where}\\&(==) :: a \to a \to \textit{Bool}\end{aligned}$$

Example instance declarations, `integerEq` and `floatEq` are primitive functions:

$$\begin{aligned}&\textit{instance Eq Integer where}\\&\quad x == y = x \,`\textit{integerEq}`\, y\\&\textit{instance Eq Float where}\\&\quad x == y = x \,`\textit{floatEq}`\, y\\&\textit{instance } (\textit{Eq a}) \Rightarrow \textit{Eq } (\textit{Tree a}) \textit{ where}\\&\quad \textit{Leaf } a \quad\quad == \textit{Leaf } b \quad\quad = a == b\\&\quad (\textit{Branch } l1\ r1) == (\textit{Branch } l2\ r2) = (l1 == l2)\ \&\&\ (r1 == r2)\\&\quad \_ \quad\quad == \_ \quad\quad = \textit{False}\end{aligned}$$

Number Classes

Haskell has a rich set of numeric types and classes that inherit in the obvious ways. The root of the numeric class hierarchy is `Num`.

- `Integer` in class `Integral`: Arbitrary-precision integers
- `Int` in class `Integral`: Fixed-precision integers
- `Float` in class `RealFloat`: Real floating-point, single precision

Input/Output

To preserve referential transparency, stateful interaction in Haskell (e.g. I/O) is performed in a *Monad*.
Input/Output actions occur in the IO Monad, e.g.

$$\begin{aligned}&\textit{getChar} :: \textit{IO Char}\\&\textit{putChar} :: \textit{Char} \to \textit{IO}\ ()\\&\textit{getArgs} :: \textit{IO}\ [\textit{String}]\\&\textit{putStr}, \textit{putStrLn} :: \textit{String} \to \textit{IO}\ ()\\&\textit{print} :: \textit{Show a} \Rightarrow a \to \textit{IO}\ ()\end{aligned}$$

Every Haskell program has a `main :: IO ()` function, e.g.

$$\textit{main} = \textit{putStr}\ \text{"}\textit{Hello}\text{"}$$

A `do` statement performs a sequence of actions, e.g.

$$\begin{aligned}&\textit{main} :: \textit{IO}\ ()\\&\textit{main} = \textit{do}\ c \leftarrow \textit{getChar}\\&\qquad\qquad \textit{putChar}\ c\end{aligned}$$

Useful I/O

Many useful IO functions are in the system module and must be imported, see below.

$$\textit{show} :: (\textit{Show a}) \Rightarrow a \to \textit{String}$$

converts values of most types into a `String`, e.g.

Evaluation

$\textit{show } 3 \Rightarrow \text{“}3\text{”}$

$$\texttt{read} :: (\texttt{Read a}) \Rightarrow \texttt{String} \to \texttt{a}$$

parses a value of most types from a `String`.
A program returning the sum of its command line arguments:

$$\begin{aligned}\textit{main} = \textit{do}\ &\textit{args} \leftarrow \textit{getArgs}\\&\textit{let } x = \textit{read}\ (\textit{args}!!0)\\&\textit{let } y = \textit{read}\ (\textit{args}!!1)\\&\textit{putStrLn}\ (\textit{show}\ (x + y))\end{aligned}$$

How to read monadic code

Monadic code enforces a step-by-step execution of commands, operating on a hidden state that is specific to the monad
$\Rightarrow$ this is exactly the programming model you are used to from other languages.

In functional languages, monadic code is a special case, and typically used when interacting with the outside world. We need to distinguish between monadic and purely functional code. This distinction is made in the type, e.g.

$$readFile :: FilePath \rightarrow IO\ String$$

Read this as: "the `readFile` function takes a file-name, as a full file-path, as argument and *performs an action in the IO monad* which returns the file contents as a string."

NB: Calling `readFile` doesn't give you the string contents, rather it performs an *action*

Another example of monadic code

Read a file that contains one number at each line, and compute the sum over all numbers.

```
myAction :: String -> IO Int  -- define an IO-action, that takes a string as input
myAction fn =
 do                           -- this starts a block of monadic actions
   str <- readFile fn         -- perform an action, reading from file
   let lns = lines str        -- split the file contents by lines
   let nums = map read lns    -- turn each line into an integer value
   let res = sum nums         -- compute the sum over all integer values
   print res                  -- print the sum
   return res                 -- return the sum
```

NB: the $\leftarrow$ operator (written `<-`) *binds* the result from executing monadic code to a variable.
The `let` constructs *assigns* the value of a (purely functional) computation to a variable.

Example: Caesar Cipher

To summarise:

> *to encrypt a plaintext message M, take every letter in M and shift it by e elements to the right to obtain the encrypted letter; to decrypt a ciphertext, take every letter and shift it by $d = -e$ elements to the left*

As an example, using $e = 3$ as key, the letter A is encrypted as a D, B as an E etc.

```
Plain:    ABCDEFGHIJKLMNOPQRSTUVWXYZ
Cipher:   DEFGHIJKLMNOPQRSTUVWXYZABC
```

Encrypting a concrete text, works as follows:

```
Plaintext:  the quick brown fox jumps over the lazy dog
Ciphertext: WKH TXLFN EURZQ IRA MXPSV RYHU WKH ODCB GRJ
```

More formally we have the following functions for en-/de-cryption:

$$E_e(x) = x + e \mod 26$$

$$D_e(x) = x - e \mod 26$$

Characteristics of Caesar's Cipher

Note the following:

- The sets of plain- and cipher-text are only latin characters. We cannot encrypt punctuation symbols etc.
- The en- and de-cryption algorithms are the same. They only differ in the choice of the key.
- The key strength is not tunable: shifting by 4 letters is no more safe than shifting by 3.
- This is an example of a *symmetric or shared-key cryptosystem*.

Exercise

Implement an en-/de-cryption function based on the Caesar cipher. Implement a function that tries to crack a Caesar cipher, ie. that retrieves plaintext from ciphertext for an unknown key.

Program header and import statements

```
module Caesar where

import Data.Char
import Math.Algebra.Field.Base
import Data.List
import Test.QuickCheck
```

The `import` statement makes all definitions from a different module available in this file.

Helper functions

```
-- convert a character to an integer, starting with 0 for 'a' etc
let2int :: Char -> Int
let2int c = ord c - ord 'a'

-- convert an index, starting with 0, to a character, e.g 'a'
int2let :: Int -> Char
int2let n = chr (ord 'a' + n)

-- shift a character c by n slots to the right
shift :: Int -> Char -> Char
shift n c | isLower c = int2let (((let2int c) + n) `mod` 26)
          | otherwise = c
```

The `shift` function is the basic operation that we need in the Caesar Cipher: it shifts a character by given number of steps through the alphabet.

The encoding function

```
-- top-level string encoding function
encode :: Int -> String -> String
encode n cs = [ shift n c | c <- cs ]

percent :: Int -> Int -> Float
percent n m = (fromIntegral n / fromIntegral m)*100

-- compute frequencies of letters 'a'..'z' in a given string
freqs :: String -> [Float]
freqs cs = [percent (count c cs) n | c <- ['a'..'z'] ]
           where n = lowers cs
```

The function `freqs` determines the frequencies of all letters of the alphabet in the given text `cs`.

The decoding function

```
-- chi-square function for computing distance between 2 frequency lists
chisqr :: [Float] -> [Float] -> Float
chisqr os es = sum [((o-e)^2)/e | (o,e) <- zip os es]

-- table of frequencies of letters 'a'..'z'
table :: [Float]
table = [8.2, 1.5, 2.8, 4.3, 12.7, 2.2, 2.0,
         6.1, 7.0, 0.2, 0.8, 4.0, 2.4,
         6.7, 7.5, 1.9, 0.1,  6.0, 6.3, 9.1,
         2.8, 1.0, 2.4, 0.2, 2.0, 0.1]
```

The `chisqr` function formalises the intuition of matching two curves and returning a value that represents a "distance" between the two curves.

The decoding function

```
-- top-level decoding function
crack :: String -> String
crack cs = encode (-factor) cs
           where factor = head (positions (minimum chitab) chitab)
                 chitab = [ chisqr (rotate n table') table
                          | n <- [0..25] ]
                 table' = freqs cs
```

In the `crack` function, we try all possible shift values, and match the curve for each value with the known frequency of letters, taken from an English dictionary.

More helper functions

```
-- rotate a list by n slots to the left; take, drop are Prelude functions
rotate :: Int -> [a] -> [a]
rotate n xs = drop n xs ++ take n xs

-- the number of lower case letters in a given string
lowers :: String -> Int
lowers cs = length [ c | c <- cs, isLower c]

-- count the number of occurrences of c in cs
count :: Char -> String -> Int
count c cs = length [ c' | c' <- cs, c==c']

-- find list of positions of x in the list xs
positions :: Eq a => a -> [a] -> [Int]
positions x xs = [ i' | (x', i') <- zip xs [0..n], x==x' ]
                 where n = length xs - 1
```

These are the helper functions, needed by `crack`.

Case Study

To test the program, start the Haskell interpreter `ghci`:

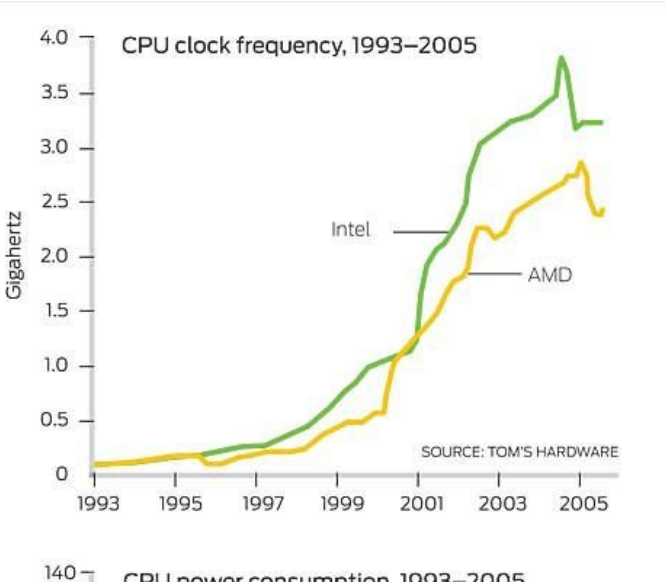
```
# ghci -package HaskellForMaths -package QuickCheck caesar.hs
```

Now, inside the interpreter try these commands:

```
#> let s1 = "a completely random text string"
#> let c1 = encode 3 s1
#> c1
#> let d1 = crack c1
#> d1
#> let s2 = "unusal random string"
#> let c2 = encode 7 s2
#> c2
#> let d2 = crack c2
#> d2
```


Part 2.
Parallel Programming Overview

Performance: The Free Lunch is over!

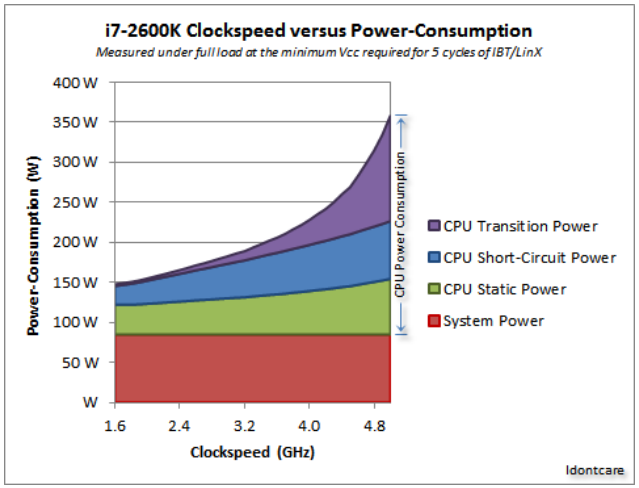


The Free Lunch is over!

- Don't expect your sequential program to run faster on new processors (Moore's law: CPU/memory speed doubles every 18 months)
- Still, processor technology advances
- BUT the focus now is on *multiple cores per chip*
- Today's desktops typically have 8 cores.
- Today's servers have up to 64 cores.
- Expect 100s of cores in the near future.
- Additionally, there is specialised hardware such as multi-byte vector processors (e.g. Intel MMX - 128 bit) or high-end graphics cards (GPGPUs)
- Together, this is a *heterogeneous, high-performance* architecture.

Power usage is the show stopper!

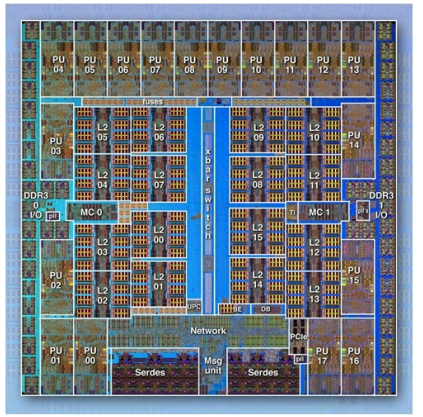
Power consumption of an Intel i7 processor:



NB: Exponential increase in transistor power!

[0]Source: http://forums.anandtech.com/showthread.php?t=2195927

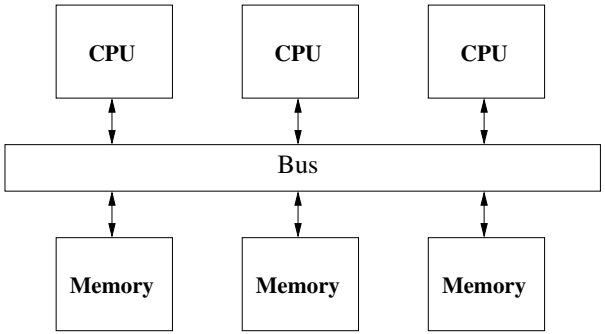
Typical multi-core architecture today



- 18 cores (PU) on one chip
- several levels of caches
- some of them are *shared* between cores
- shared memory, BUT
- non-uniform memory access (NUMA) to this shared memory

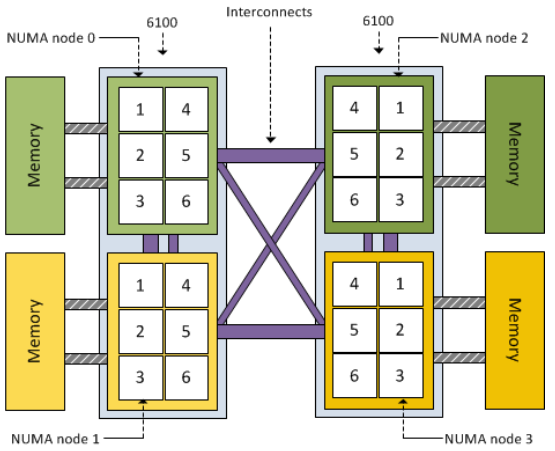
Shared-memory architectures

- Multi-core, shared-memory servers are now common-place.
- Shared-memory is easier to program than distributed memory.
- Shared-memory architecture:



NUMA architectures

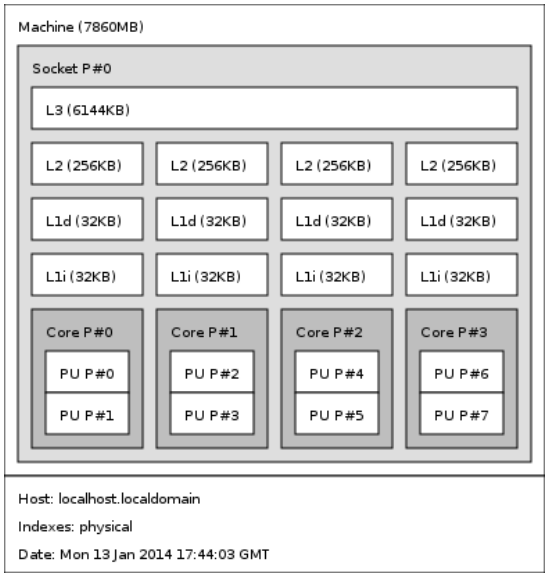
Memory access costs depend very much on which memory bank ("NUMA region") is accessed.



Here: 24-cores, BUT in 4 NUMA regions with 6 cores each.

State-of-the-art Servers are multi-cores

Even high-end laptops are multi-cores



NUMA costs

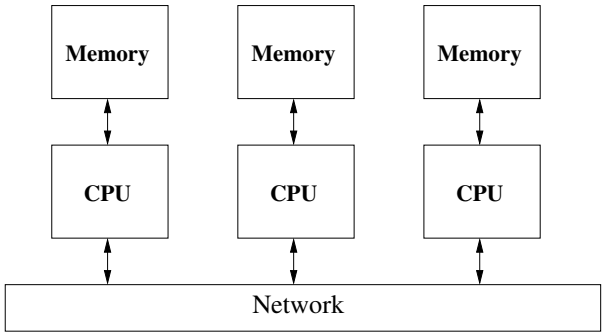
- NUMA architectures pose a challenge to parallel applications.
  - Asymmetric memory latencies
  - Asymmetric memory bandwidth between different memory regions.

Memory access times between different NUMA regions[1]

node	0:	1:	2:	3:	4:	5:	6:	7:
0:	10	16	16	**22**	16	**22**	16	22
1:	16	10	**22**	16	**22**	16	**22**	16
2:	16	**22**	10	16	16	**22**	16	22
3:	**22**	16	16	10	**22**	16	**22**	16
4:	16	**22**	16	**22**	10	16	16	**22**
5:	**22**	16	**22**	16	16	10	**22**	16
6:	16	**22**	16	**22**	16	**22**	10	16
7:	**22**	16	**22**	16	**22**	16	16	10

[1] Measured using `numactl -H`

Distributed memory architectures



- advantage: highly scalable
- disadvantage: explicit data communication is relatively slow

Example: "Beowulf" clusters of commodity desktop hardware with an GB Ethernet network.

Hector supercomputer

Supercomputers

The Hector supercomputer at the Edinburgh Parallel Computing Center (2011):

- total of 464 compute blades;
- each blade contains four compute nodes,
- each with two 12-core AMD Opteron 2.1GHz Magny Cours processors.
- Total: *44,544 cores*
- Upgraded in 2011 to 24-core chips with a total of *90,112 cores*

See the TOP500 list of fastest supercomputers for the most recent picture.

Supercomputers

Hector is *out-dated* and will be turned off in March 2014. The new supercomputer at EPCC is Archer:

- Cray XC30 architecture
- uses Intel Xeon Ivy Bridge processors
- total of 3008 compute nodes
- each node comprises two 12-core 2.7 GHz Ivy Bridge multi-core processors,
- Total: *72,192 cores*
- Peak performance: *1.56 Petaflops*
- each node has at least 64 GB of DDR3-1833 MHz main memory,
- scratch disk storage: Cray Sonexion Scalable Storage Units (*4.4PB* at 100GB/s)
- all compute nodes are interconnected via an Aries Network Interface Card.

[1] For details on Archer see:
`https://www.epcc.ed.ac.uk/media/publications/newsletters/epcc-news-74`
and `www.archer.ac.uk`

The fastest super-computer today: TianHe-2 (MilkyWay-2)

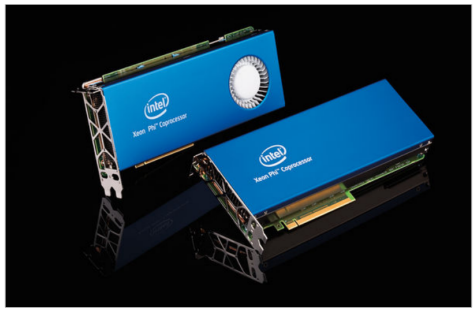
- located at the National Super Computer Center in Guangzhou, China
- 16,000 nodes, each with 2 Ivy Bridge multi-cores and *3 Xeon Phis*
- *3,120,000 cores* in total!
- Linpack performance: 33.86 PFlop/s
- Theoretical peak perf: 54.90 PFlop/s (ca 733k× my laptop)
- Power: 17.8 MW

Trends in Parallel Hardware

- hundreds of (light-weight) cores (today's servers have 48–64 cores)
- NUMA-structured shared-memory with partially shared caches
- probably not all of the being used at the same time (dark silicon)
- attached many-core co-processors (Xeon-Phi, Parallela)
- attached general purpose GPUs (for data-parallel floating point performance)
- possibly soft cores (in the form of FPGAs)
- *highly heterogeneous*

Such architectures are challenging to program
⇒ we need *high-level programming models* to simplify that task

Xeon Phi



- Example of a many-core co-processor
- Acts similar to a graphics card, plugged into the main board
- Provides *60 cores* on one board
- Can be programmed like a network of 60 machines
- Similar, but cheaper products come onto the market: Parallela

Parallel Programming: implicit parallelism

Implicit parallelism:

- compiler/run time system exploits parallelism latent in program
  e.g. High Performance Fortran
- avoids need for programmer expertise in architecture/communication
- identifying parallelism in programs is hard and undecidable in general
- requires complex usage analysis to ensure independence of potentially parallel components
- typically look for parallelism in loops
- check that each iteration is independent of next
- advantages
  - no programmer effort
- disadvantages
  - parallelising compilers very complex
  - beyond common patterns of parallelism, often human can do better

Example Code

A *good* example:

```
for (i=0; i<n; i++)
 a[i] = b[i]*c[i]
```

- no dependency between stages
- could execute a[i] = b[i]*c[i] on separate processors

A *bad* example:

```
for (i=1; i<n; i++)
 a[i] = a[i-1]*b[i]
```

- each stage depends on previous stage so no parallelism

Parallel Programming: explicit parallelism

Explicit parallelism:

- programmer nominates program components for parallel execution
- three approaches
  - extend existing language
  - design new language
  - libraries

Parallel Programming: extend existing languages

Extend existing languages

- add primitives for parallelism to an existing language
- advantage:
  - can build on current suite of language support e.g. compilers, IDEs etc
  - user doesn't have to learn whole new language
  - can migrate existing code
- disadvantage
  - no principled way to add new constructs
  - tends to be ad-hoc,
  - i.e. the parallelism is language dependent
- e.g. many parallel Cs in 1990s
- none have become standard, yet
- an emerging standard is Unified Parallel C (UPC)

Parallel Programming: language independent extensions

Use language independent extensions

- Example: *OpenMP*
- for shared memory programming, e.g. on multi-core
- Host language can be Fortran, C or C++
- programmer marks code as
  - `parallel`: execute code block as multiple parallel threads
  - `for`: for loop with independent iterations
  - `critical`: critical section with single access

Parallel Programming: compiler directives

Compiler directives

- advantage
  - directives are transparent so can run program in normal sequential environment
  - concepts cross languages
- disadvantage
  - up to implementor to decide how to realise constructs
  - no guarantee of cross-language consistency
  - i.e. the parallelism is platform dependent

Parallel Programming: Develop new languages

Develop new languages

- advantage:
  - clean slate
- disadvantage
  - huge start up costs (define/implement language)
  - hard to persuade people to change from mature language
- Case study: sad tale of INMOS occam (late 1980's)
  - developed for transputer RISC CPU with CSP formal model
  - explicit channels + wiring for point to point CPU communication
  - multi-process and multi-processor
  - great British design: unified CPU, language & formal methodology
  - great British failure
  - INMOS never licensed/developed occam for CPUs other than transputer
  - T9000 transputers delivered late & expensive compared with other CPUs libraries

Parallel Programming: Language independent libraries

Language independent libraries:

- most successful approach so far
  - language independent
  - platform independent
- Examples:
  - Posix thread library for multi-core
  - Parallel Virtual Machines (PVM) for multi-processor
  - Message Passing Interface (*MPI*) for multi-processor
- widely available for different languages under different operating systems on different CPUs
  e.g. MPICH-G: MPI for GRID enabled systems with Globus
- we will use *C with MPI* on Beowulf

Parallel Programming Examples

As an example of *low-level vs high-level parallel programming*, we want to compute a Taylor series in parallel:

$$e^z = \lim_{n\to\infty} \Sigma_{n=1}^{\infty} \frac{z^n}{n!}$$

The basic idea is to compute the components of the sum in parallel.

Sequential C version

```
/* exponential function */
double power_e(long z) {
  long n;
  double sum, old_sum, eps;
  for (n=0, sum=0.0, eps=1.0; eps > EPSILON; n++) {
    old_sum = sum;
    sum += ((double)pow_int(z,n)) / ((double) fact(n));
    eps = sum - old_sum;
  }
  return sum;
}
```

Simple code, but soon overflows ⇒ we need arbitrary precision integers.

Seq. C with GMP for arbitrary precision integers

We use the GNU multi-precision library (GMP) to get arbitrary precision.

```
/* exponential function, result will be stored in res */
void power_e(mpq_t res, ui z, ui d) {
  mpq_t sum, old_sum, eps;
  mpq_t tmp_num, tmp_den, tmp, tmpe, epsilon;
  ui n;
  mpq_init(epsilon);  mpq_init(eps);  mpq_init(sum); \ldots
  pow_int(tmpe, 10l, d);  mpq_inv(epsilon, tmpe); // 10^-d
  mpq_set_ui(sum, 0l, 1l);  mpq_set_ui(eps, 1l, 1l);
  for (n=0; mpq_cmp(eps, epsilon)>0; n++) {
    mpq_set(old_sum, sum);
    pow_int(tmp_num, z, n);
    fact(tmp_den, n);
    mpq_div(tmp, tmp_num, tmp_den);
    mpq_add(sum, sum, tmp);
    mpq_set(eps, tmp);
  }
  mpq_clear(tmp_num); mpq_clear(tmp_den); \ldots
  mpq_set(res,sum);
}
```

Same structure as before, but harder to read due to lack of syntactic sugar.

Basic Point to Point Communication in MPI

MPI (the Message Passing Interface) offers two basic point to point communication functions:

- `MPI_Send(message, count, datatype, dest, tag, comm)`
  - Blocks until `count` items of type `datatype` are sent from the `message` buffer to processor `dest` in communicator `comm`.
    - ★ `message` buffer may be reused on return, but message may still be in transit!
- `MPI_Recv(message, count, datatype, source, tag, comm, status)`
  - Blocks until receiving a `tag`-labelled message from processor `source` in communicator `comm`.
  - Places the message in `message` buffer.
    - ★ `datatype` must match datatype used by sender!
    - ★ Receiving fewer than `count` items is OK, but receiving more is an error!

Aside: These are the two most important MPI primitives you have to know.

Send and Receive in more Detail

```
int MPI_Send(
  void * message,
  int count,
  MPI_Datatype datatype,
  int dest,
  int tag,
  MPI_Comm comm)
```

```
int MPI_Recv(
  void * message,
  int count,
  MPI_Datatype datatype,
  int source,
  int tag,
  MPI_Comm comm,
  MPI_Status * status)
```

- message — pointer to send/receive buffer
- count — number of data items to be sent/received
- datatype — type of data items
- comm — communicator of destination/source processor
  - For now, use default communicator MPI_COMM_WORLD
- dest/source — rank (in comm) of destination/source processor
  - Pass MPI_ANY_SOURCE to MPI_Recv() if source is irrelevant
- tag — user defined message label
  - Pass MPI_ANY_TAG to MPI_Recv() if tag is irrelevant
- status — pointer to struct with info about transmission
  - Info about source, tag and #items in message received

Parallel C+MPI version: Master

The master distributes the work and collects results.

```
// start the timer
MPI_Barrier(MPI_COMM_WORLD);
elapsed_time = - MPI_Wtime();

/* General: use p-1 workers, each computing every (p-1)-th element of the series */

long from, step, last_n, max_n;  int len, l; double *times; MPI_Status status;

max_n = (long)p-1;
step=(long)(p-1);
for (i=1; i<p; i++) {
  from=(long)(i-1);
  MPI_Send(&z, 1, MPI_LONG, i, 0, MPI_COMM_WORLD);   /* send input to worker i */
  MPI_Send(&d, 1, MPI_LONG, i, 0, MPI_COMM_WORLD);   /* send input to worker i */
  MPI_Send(&from, 1, MPI_LONG, i, 0, MPI_COMM_WORLD);/* send input to worker i */
  MPI_Send(&step, 1, MPI_LONG, i, 0, MPI_COMM_WORLD);/* send input to worker i */
}

times = (double *)malloc(p*sizeof(double));
for (i=1; i<p; i++) {
  MPI_Recv(&len, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &status); /* recv result from
       worker i */
  MPI_Recv(&res_str, len, MPI_CHAR, i, 0, MPI_COMM_WORLD, &status);
  MPI_Recv(&last_n, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &status);
  MPI_Recv(&time, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &status);
  res_str[len] = '\0';
  /* unmarshall the GMP data */
  if (gmp_sscanf(res_str,"%Qd",&res)==0) {
    fprintf(stderr,"[%d] Error in gmp_sscanf", id); MPI_Abort(MPI_COMM_WORLD, 2);
  }
```


Parallel C+MPI version: Master (cont'd)

```
    times[i] = time;
    max_n = (last_n>max_n) ? last_n : max_n;
    mpq_add(result, result, res);
  }

  /* Q: how can we determine in general that this is close enough to the solution? */

  // Maybe use an epsilon returned by the last PE to decide whether to compute more
  // mpq_set_ui(eps2, 1l, 1l); // eps2 = 1/1

  // stop the timer
  elapsed_time += MPI_Wtime();
```


Parallel C+MPI version: Worker

The workers do the actual computation.

```
mpq_init(res);   mpq_init(eps);

// start the timer
MPI_Barrier(MPI_COMM_WORLD);
// elapsed_time = - MPI_Wtime();

MPI_Recv(&z, 1, MPI_LONG, 0, 0, MPI_COMM_WORLD, &status);  /* receive input */
MPI_Recv(&d, 1, MPI_LONG, 0, 0, MPI_COMM_WORLD, &status);  /*   from master */
MPI_Recv(&from, 1, MPI_LONG, 0, 0, MPI_COMM_WORLD, &status);
MPI_Recv(&step, 1, MPI_LONG, 0, 0, MPI_COMM_WORLD, &status);

// start the timer
elapsed_time = - MPI_Wtime();

power_e_step(res, eps, &max_n, z, d, from, step, id);        // result of our
     interval
if ((len = gmp_sprintf(buf,"%Qd", res))==0 || len>=GMP_STR_SIZE) { // marshall to
     string
  fprintf(stderr,"[%d] Error in gmp_sprintf", id);
  MPI_Abort(MPI_COMM_WORLD, 2);
}

// stop the timer
elapsed_time += MPI_Wtime();

MPI_Send(&len, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);    /* send result */
MPI_Send(buf, len, MPI_CHAR, 0, 0, MPI_COMM_WORLD);  /*  to master  */
MPI_Send(&max_n, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
MPI_Send(&elapsed_time, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
```

Parallel C+MPI version: Worker

The core compute function is a step-wise loop, similar to the sequential version.

```
/* exponential function, result will be stored in res */
static inline
void power_e_step(mpq_t res, mpq_t eps, int *max_n, ui z, ui d, int from, int step, int
     id) {
  mpq_t sum, old_sum; \ldots
  mpq_init(epsilon); \dots
  pow_int(tmpe, 10l, d);              // 10^d
  mpq_inv(epsilon, tmpe);             // 10^-d
  mpq_set_ui(sum, 0l, 1l);
  mpq_set_ui(eps, 1l, 1l);
  for (n=from; mpq_cmp(eps, epsilon)>0; n+=step) { // step-wise loop
    mpq_set(old_sum, sum);
    pow_int(tmp_num, z, n);
    fact(tmp_den, n);
    mpq_div(tmp, tmp_num, tmp_den);
    mpq_add(sum, sum, tmp);
    mpq_sub(eps, sum, old_sum);
  }
  *max_n = n-step;
  mpq_clear(tmp_num); \ldots
}
```


Sequential Haskell version

Compare the previous version with this Haskell version.

```
-- compute e^z up to d digits
power_e :: Integer -> Integer -> Rational
power_e z d = sum (takeWhile (>(1 % (10^d))) taylor)
              where -- infinite list of entire Taylor series
                    taylor =  [ (pow_ints_n % 1) / (factorials_n % 1) | n<-[0,1..] ]
                    -- 2 circular lists, to memoise earlier computations
                    factorials = 1:1:[ (toInteger i)*(factorials(i-1)) | i<-[2..] ]
                    pow_ints = 1:[ z*(pow_ints(i-1)) | i<-[1..] ]
```

NB:

- we use list comprehension notation to define the 3 main lists we need
- `Integer` is a data-type of arbitrary precision integers
- the definitions of `factorials` and `pow_ints` are *circular*, i.e. the definition of the i-th element refers to the $i-1$-st element in the same list (this only works in a lazy language)

Parallel Haskell version

We observe that: `sum = foldr (+) 0`

```
-- compute e^z up to d digits
power_e :: Integer -> Integer -> Integer -> Rational
power_e z d v = parfoldr (+) 0  (takeWhile (>(1 % (10^d))) taylor)
                where -- infinite list of entire Taylor series
                      taylor =  [ (pow_intsn % 1) / (factorialsn % 1) | n<-[0,1..] ]
                      -- 2 circular lists, to memoise earlier computations
                      factorials = 1:1:[ (toInteger i)*(factorials(i-1)) | i<-[2..] ]
                      pow_ints = 1:[ z*(pow_ints(i-1)) | i<-[1..] ]

parfold fold f z [] = z
parfold fold f z xs = res
  where
        parts = chunk (chunksize xs) xs
        partsRs = map (fold f z) parts `using` parList rdeepseq
        res = fold f z partsRs
```

NB: The original code is almost identical, only replacing a `fold` (in `sum`) by a `parfold`.

Part 3.
GpH — Parallelism in a side-effect free language

The Challenge of Parallel Programming

Engineering a parallel program entails specifying

- *computation*: a correct, efficient algorithm
  in GpH the semantics of the program is unchanged
- *coordination*: arranging the computations to achieve "good" parallel behaviour.
  in GpH coordination and computation are cleanly separated

Coordination Aspects

Coordinating parallel behaviour entails, *inter alia*:

- partitioning
  - what threads to create
  - how much work should each thread perform
- thread synchronisation
- load management
- communication
- storage management

Specifying full coordination details is a significant burden on the programmer

High Level Parallel Programming

High level parallel programming aims to reduce the programmer's coordination management burden.

This can be achieved by using

- specific execution models (array languages such as *SAC*),
- skeletons or parallel patterns (*MapReduce*, *Eden*),
- data-oriented parallelism (*PGAS* languages),
- dataflow languages such as *Swan*),
- parallelising compilers (*pH* for Haskell).

GpH (Glasgow parallel Haskell) uses a model of *semi-explicit* parallelism: only a few key aspects of coordination need to be specified by the programmer.

GpH Coordination Primitives

GpH provides parallel composition to *hint* that an expression may usefully be evaluated by a parallel thread.
We say `x` is *"sparked"*: if there is an idle processor a thread may be created to evaluate it.

Evaluation

```
x `par` y ⇒y
```

GpH provides sequential composition to sequence computations and specify how much evaluation a thread should perform. `x` is evaluated to Weak Head Normal Form (WHNF) before returning `y`.

Evaluation

```
x `pseq` y ⇒y
```

Introducing Parallelism: a GpH Factorial

Factorial is a classic *divide and conquer* algorithm.

Example (Parallel factorial)

```
pfact n = pfact' 1 n

pfact' :: Integer -> Integer -> Integer
pfact' m n
  | m == n    = m
  | otherwise = left `par` right `pseq` (left * right)
      where mid   = (m + n) `div` 2
            left  = pfact' m mid
            right = pfact' (mid+1) n
```


Controlling Evaluation Order

Notice that we must *control evaluation order:* If we wrote the function as follows, then the addition may evaluate `left` on this core/processor before any other has a chance to evaluate it

```
  | otherwise = left `par` (left * right)
```

The `right `pseq` ensures that `left` and `right` are evaluated before we multiply them.

Controlling Evaluation Degree

In a non strict language we must specify *how much* of a value should be computed.

For example the obvious quicksort produces almost no parallelism because the threads reach WHNF very soon: once the first cons cell of the sublist exists!

Example (Quicksort)

```
quicksortN :: (Ord a) => [a] -> [a]
quicksortN []     = []
quicksortN [x]    = [x]
quicksortN (x:xs) =
  losort `par`
  hisort `par`
  losort ++ (x:hisort)
    where
      losort = quicksortN [y|y <- xs, y < x]
      hisort = quicksortN [y|y <- xs, y >= x]
```


Controlling Evaluation Degree (cont'd)

Forcing the evaluation of the sublists gives the desired behaviour:

Example (Quicksort with forcing functions)

```
forceList :: [a] -> ()
forceList [] = ()
forceList (x:xs) = x `pseq` forceList xs

quicksortF []     = []
quicksortF [x]    = [x]
quicksortF (x:xs) =
  (forceList losort) `par`
  (forceList hisort) `par`
  losort ++ (x:hisort)
    where
      losort = quicksortF [y|y <- xs, y < x]
      hisort = quicksortF [y|y <- xs, y >= x]
```

Problem: we need a different forcing function for each datatype.

GpH Coordination Aspects

To specify parallel coordination in Haskell we must

1. Introduce parallelism
2. Specify Evaluation Order
3. Specify Evaluation Degree

This is much less than most parallel paradigms, e.g. no communication, synchronisation etc.

It is important that we do so *without cluttering the program*. In many parallel languages, e.g. C with MPI, coordination so dominates the program text that it obscures the computation.

Evaluation Strategies: Separating Computation and Coordination

Evaluation Strategies abstract over `par` and `pseq`,

- raising the level of abstraction, and
- separating coordination and computation concerns

It should be possible to understand the semantics of a function without considering its coordination behaviour.

Evaluation Strategies

An *evaluation strategy* is a function that specifies the coordination required when computing a value of a given type, and preserves the value i.e. it is an identity function.

```
type Strategy a = a -> Eval a

data Eval a = Done a
```

We provide a simple function to extract a value from `Eval`:

```
runEval :: Eval a -> a
runEval (Done a) = a
```

The `return` operator from the `Eval` monad will introduce a value into the monad:

```
return :: a -> Eval a
return x = Done x
```


Applying Strategies

`using` applies a strategy to a value, e.g.

```
using :: a -> Strategy a -> a
using x s = runEval (s x)
```

Example

A typical GpH function looks like this:

```
somefun x y = someexpr `using` somestrat
```

Simple Strategies

Simple strategies can now be defined.

`r0` performs no reduction at all. Used, for example, to evaluate only the first element but not the second of a pair.

`rseq` reduces its argument to Weak Head Normal Form (WHNF).

`rpar` sparks its argument.

```
r0 :: Strategy a
r0 x = Done x

rseq :: Strategy a
rseq x = x `pseq` Done x

rpar :: Strategy a
rpar x = x `par` Done x
```


Controlling Evaluation Order

We control evaluation order by using a monad to sequence the application of strategies.

So our parallel factorial can be written as:

Example (Parallel factorial)

```
pfact' :: Integer -> Integer -> Integer
pfact' m n
  | m == n    = m
  | otherwise = (left * right) `using` strategy
      where mid   = (m + n) `div` 2
            left  = pfact' m mid
            right = pfact' (mid+1) n
            strategy result = do
                                rpar left
                                rseq right
                                return result
```


Controlling Evaluation Degree - The DeepSeq Module

Both `r0` and `rseq` control the evaluation degree of an expression.

It is also often useful to reduce an expression to *normal form* (NF), i.e. a form that contains *no* redexes. We do this using the `rnf` strategy in a type class.

As NF and WHNF coincide for many simple types such as `Integer` and `Bool`, the default method for `rnf` is `rwhnf`.

```
class NFData a where
  rnf ::  a -> ()
  rnf x = x `pseq` ()
```

We define `NFData` instances for many types, e.g.

```
instance NFData Int
instance NFData Char
instance NFData Bool
```


Evaluation Degree Strategies

We can define `NFData` for type constructors, e.g.

```
instance NFData a => NFData [a] where
  rnf [] = ()
  rnf (x:xs) = rnf x `seq` rnf xs
```

We can define a `deepseq` operator that fully evaluates its first argument:

```
deepseq :: NFData a => a -> b -> b
deepseq a b = rnf a `seq` b
```

Reducing all of an expression with `rdeepseq` is by far the most common evaluation degree strategy:

```
rdeepseq :: NFData a => Strategy a
rdeepseq x = x `deepseq` Done x
```

Combining Strategies

As strategies are simply functions they can be combined using the full power of the language, e.g. passed as parameters or composed.

`dot` composes two strategies on the same type:

```
dot :: Strategy a -> Strategy a -> Strategy a
s2 `dot` s1 = s2 . runEval . s1
```

`evalList` sequentially applies strategy `s` to every element of a list:

Example (Parametric list strategy)

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


Data Parallel Strategies

Often coordination follows the data structure, e.g. a thread is created for each element of a data structure.

For example `parList` applies a strategy to every element of a list in parallel using `evalList`

```
parList :: Strategy a -> Strategy [a]
parList s = evalList (rpar `dot` s)
```

`parMap` is a higher order function using a strategy to specify data-oriented parallelism over a list.

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


Control-oriented Parallelism

Example (Strategic quicksort)

```
quicksortS []     = []
quicksortS [x]    = [x]
quicksortS (x:xs) =
  losort ++ (x:hisort) `using` strategy
    where
      losort = quicksortS [y|y <- xs, y < x]
      hisort = quicksortS [y|y <- xs, y >= x]
      strategy res = do
                     (rpar `dot` rdeepseq) losort
                     (rpar `dot` rdeepseq) hisort
                      rdeepseq res
```

Note how the coordination code is cleanly separated from the computation.

Thread Granularity

Using semi-explicit parallelism, programs often have massive, fine-grain parallelism, and several techniques are used to increase thread granularity.

It is only worth creating a thread if the *cost of the computation will outweigh the overheads* of the thread, including

- communicating the computation
- thread creation
- memory allocation
- scheduling

It may be necessary to transform the program to achieve good parallel performance, e.g. to improve thread granularity.
Thresholding: in divide and conquer programs, generate parallelism only up to a certain threshold, and when it is reached, solve the small problem sequentially.

Threshold Factorial

Example (Strategic factorial with threshold)

```
pfactThresh :: Integer -> Integer -> Integer
pfactThresh n t = pfactThresh' 1 n t

-- thresholding version
pfactThresh' :: Integer -> Integer -> Integer -> Integer
pfactThresh' m n t
  | (n-m) <= t = product [m..n]   -- seq solve
  | otherwise = (left * right) `using` strategy
      where mid   = (m + n) `div` 2
            left  = pfactThresh' m mid t
            right = pfactThresh' (mid+1) n t
            strategy result = do
                              rpar left
                              rseq right
                              return result
```


Chunking Data Parallelism

Evaluating individual elements of a data structure may give too fine thread granularity, whereas evaluating many elements in a single thread give appropriate granularity. The number of elements (the size of the chunk) can be tuned to give good performance.

It's possible to do this by changing the computational part of the program, e.g. replacing

```
parMap rdeepseq fact [12 .. 30]
```

with

```
concat (parMap rdeepseq
          (map fact) (chunk 5 [12 .. 30]))

chunk :: Int -> [a] -> [[a]]
chunk _ [] = [[]]
chunk n xs = y1 : chunk n y2
  where
    (y1, y2) = splitAt n xs
```


Strategic Chunking

Rather than change the computational part of the program, it's better to change only the strategy.

We can do so using the `parListChunk` strategy which applies a strategy `s` sequentially to sublists of length `n`:

```
map fact [12 .. 30] `using` parListChunk 5 rdeepseq
```

Uses Strategy library functions:

```
parListChunk :: Int -> Strategy [a] -> Strategy [a]
parListChunk n s =
  parListSplitAt n s (parListChunk n s)

parListSplitAt :: Int -> Strategy [a]
                  Strategy [a] -> Strategy [a]
parListSplitAt n stratPref stratSuff =
 evalListSplitAt n (rpar `dot` stratPref)
                   (rpar `dot` stratSuff)
```



```
evalListSplitAt :: Int -> Strategy [a] ->
                   Strategy [a] -> Strategy [a]
 evalListSplitAt n stratPref stratSuff [] = return []
 evalListSplitAt n stratPref stratSuff xs
  = do
      ys' <- stratPref ys
      zs' <- stratSuff zs
      return (ys' ++ zs')
      where
        (ys,zs) = splitAt n xs
```

Systematic Clustering

Sometimes we require to aggregate collections in a way that cannot be expressed using only strategies. We can do so systematically using the `Cluster` class:

- `cluster n` maps the collection into a collection of collections each of size `n`
- `decluster` retrieves the original collection
  `decluster . cluster == id`
- `lift` applies a function on the original collection to the clustered collection

```
class (Traversable c, Monoid a) => Cluster a c where
  cluster    :: Int -> a -> c a
  decluster  :: c a -> a
  lift       :: (a -> b) -> c a -> c b

  lift = fmap       -- c is a Functor, via Traversable
  decluster = fold  -- c is Foldable,  via Traversable
```


An instance for lists requires us only to define `cluster`

```
instance Cluster [a] [] where
  cluster = chunk
```


A Strategic Div&Conq Skeleton

```
divConq :: (a -> b)           -- compute the result
        -> a                  -- the value
        -> (a -> Bool)        -- threshold reached?
        -> (b -> b -> b)      -- combine results
        -> (a -> Maybe (a,a)) -- divide
        -> b
divConq f arg threshold conquer divide = go arg
 where
  go arg =
    case divide arg of
     Nothing      -> f arg
     Just (l0,r0) -> conquer l1 r1 `using` strat
    where
     l1 = go l0
     r1 = go r0
     strat x = do { r l1; r r1; return x }
               where r | threshold arg = rseq
                       | otherwise     = rpar
data Maybe a = Nothing | Just a
```


A Strategic Div&Conq Skeleton: Discussion

- The skeleton is a *higher-order function*, with arguments for the divide-, combine-, and base-phase.
- The strategy `strat` specified that parallelism should be used up to the threshold.
- The strategy is applyed to the result of the `conquer` phase.
- Again, the coordination code is cleanly separated from the compute code.

Summary

Evaluation strategies in GpH

- use laziness to *separate computation from coordination*
- use the `Eval` monad to specify evaluation order
- use overloaded functions (`NFData`) to specify the evaluation-degree
- provide high level abstractions, e.g. `parList, parSqMatrix`
- are functions in algorithmic language ⇒
  - comprehensible,
  - can be combined, passed as parameters etc,
  - extensible: write application-specific strategies, and
  - can be defined over (almost) any type
- general: pipeline, d&c, data parallel etc.
- Capable of expressing complex coordination, e.g. embedded parallelism, `Clustering`, skeletons

For a list of (parallel) Haskell exercises with usage instructions see:
`http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/tutorial0.html#gph`

Further Reading & Deeper Hacking

- P.W. Trinder, K. Hammond, H.-W. Loidl, S.L. Peyton Jones *Algorithm + Strategy = Parallelism*. In Journal of Functional Programming 8(1), Jan 1998. DOI: 10.1017/S0956796897002967 `https://www.macs.hw.ac.uk/~dsg/gph/papers/abstracts/strategies.html`
- S. Marlow and P. Maier and H-W. Loidl and M.K. Aswad and P. Trinder, *"Seq no more: Better Strategies for Parallel Haskell"*. In *Haskell'10 — Haskell Symposium*, Baltimore MD, U.S.A., September 2010. ACM Press. `http://www.macs.hw.ac.uk/~dsg/projects/gph/papers/abstracts/new-strategies.html`
- *"Parallel and concurrent programming in Haskell"*, by Simon Marlow. O'Reilly, 2013. ISBN: 9781449335946.

Further Reading & Deeper Hacking

- An excellent site for learning (sequential) Haskell is: `https://www.fpcomplete.com/school`
- Glasgow parallel Haskell web page: `http://www.macs.hw.ac.uk/~dsg/gph`
- Our course on parallel technologies covers GpH in more detail and has more exercises: `http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP`
- Specifically, for a list of (parallel) Haskell exercises with usage instructions see: `http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/tutorial0.html#gph`

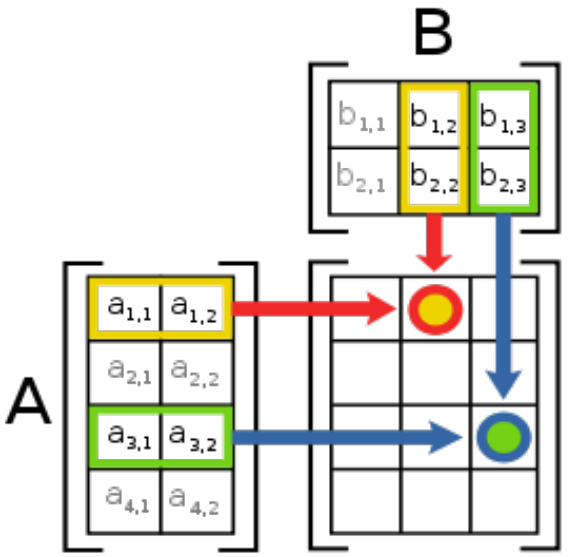
Part 4.
A Case Study of Using GpH

Case study: Parallel Matrix Multiplication

As an example of a parallel program lets consider: matrix multiplication.

Problem If matrix A is an $m \times n$ matrix $[a_{ij}]$ and B is an $n \times p$ matrix $[b_{ij}]$, then the product is an $m \times p$ matrix C where $C_{ik} = \Sigma_{j=1}^{n} a_{ij} b_{jk}$

Matrix Multiplication



[1]Picture from http://en.wikipedia.org/wiki/Matrix_multiplication

Sequential Implementation in Haskell

```
-- Type synonyms
type Vec a = [a]
type Mat a = Vec (Vec a)

-- vector multiplication ('dot-product')
mulVec :: Num a => Vec a -> Vec a -> a
u `mulVec` v = sum (zipWith (*) u v)

-- matrix multiplication, in terms of vector multiplications
mulMat :: Num a => Mat a -> Mat a -> Mat a
a `mulMat` b =
  [[u `mulVec` v | v <- bt ] | u <- a]
     where bt = transpose b
```

NB: the top-level matrix multiplication function boils down to one list comprehension

Time Profile

See GHC profiling documentation `http://www.haskell.org/ghc/docs/latest/html/users_guide/profiling.html`

Compile for sequential profiling `-prof -auto-all`. Note naming convention for profiling binary.

Run for a 200 by 200 matrix with time `-pT` and space `-hC` profiling turned on

```
> ghc -prof -auto-all --make -cpp -i/home/hwloidl/packages/random-1.0.1.1
  -o MatMultSeq0_prof MatMultSeq0.hs
> ./MatMultSeq0_prof 100 1701   +RTS -pT -hC
```

Inspect profiles:

```
> less MatMultSeq0_prof.prof
       Fri Jun 26 23:01 2015 Time and Allocation Profiling Report  (Final)

          MatMultSeq0_prof +RTS -pT -hC -RTS 100 1701

       total time  =        0.35 secs   (348 ticks @ 1000 us, 1 processor)
       total alloc = 329,719,848 bytes  (excludes profiling overheads)
```

Time profile

```
COST CENTRE              MODULE        %time %alloc

mulVec                   Main           29.6   53.7
stdNext.s2'              System.Random  10.3    6.1
...
main.rands               Main            1.7    1.0
chunk.(...)              Main            1.4    1.6
```


Time profile

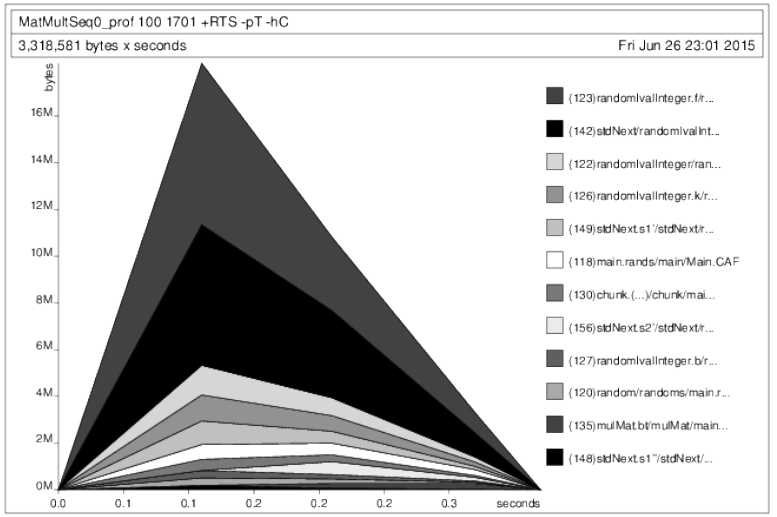
```
                                                                individual     inherited
COST CENTRE                MODULE                no.    entries  %time %alloc  %time %alloc

MAIN                       MAIN                   55          0    0.3   0.0   100.0 100.0
 CAF                       System.Random         109          0    0.0   0.0     0.0   0.0
  next                     System.Random         141          1    0.0   0.0     0.0   0.0
 CAF                       Main                  108          0    0.0   0.0    99.7 100.0
  main                     Main                  110          1    0.3   0.0    99.7 100.0
   main.seed               Main                  153          1    0.0   0.0     0.0   0.0
   main.ub                 Main                  138          1    0.0   0.0     0.0   0.0
   main.b                  Main                  136          1    0.0   0.0     0.0   0.0
   main.(...)              Main                  132          1    0.0   0.0     0.0   0.0
   main.k                  Main                  131          1    0.0   0.0     0.0   0.0
   main.rands              Main                  118          1    1.7   1.0    66.7  44.0
    mkStdGen               System.Random         144          1    0.0   0.0     0.0   0.0
     ...
    randoms                System.Random         119      20000    1.4   0.3    64.9  43.0
     ...
   main.(...)              Main                  116          1    0.0   0.0     1.7   1.6
    chunk                  Main                  117        204    0.3   0.0     1.7   1.6
     chunk.ys              Main                  133        202    0.0   0.0     0.0   0.0
     chunk.(...)           Main                  130        202    1.4   1.6     1.4   1.6
     chunk.zs              Main                  129        201    0.0   0.0     0.0   0.0
   main.a                  Main                  115          1    0.0   0.0     0.0   0.0
   main.c                  Main                  113          1    0.0   0.0    30.5  54.2
    mulMat                 Main                  114          1    0.6   0.3    30.5  54.2
     mulVec                Main                  137      10000   29.6  53.7    29.6  53.7
     mulMat.bt             Main                  135          1    0.3   0.3     0.3   0.3
   chksum                  Main                  111          1    0.0   0.0     0.6   0.2
    chksum.chksum'         Main                  112        101    0.3   0.1     0.6   0.1
     chksum.chksum'.\      Main                  134      10100    0.3   0.0     0.3   0.0
 CAF                       GHC.IO.Encoding        99          0    0.0   0.0     0.0   0.0
 ...
```


Space Profile

Improving space consumption is important for sequential tuning: minimising space usage saves time and reduces garbage collection.

```
> hp2ps MatMultSeq0_prof.hp
> gv -orientation=seascape MatMultSeq0_prof.ps
```


Space Profile



Parallel Implementation

1st attempt: *naive version*: parallelise every element of the result matrix, or both 'maps'

```
mulMatPar :: (NFData a, Num a) =>
                Mat a -> Mat a -> Mat a
mulMatPar a b = (a `mulMat` b) `using` strat
 where
  strat m = parList (parList rdeepseq) m
```

- Easy to get a first parallel version.
- Unlikely to give good performance straight away.
- Some performance tuning is necessary (as with all parallel programming activities).

Shared-Memory Results

600 x 600 matrices on an 8-core shared memory machine (Dell PowerEdge).

Compile with profiling; run on 4 cores; view results

```
> ghc --make -O2 -rtsopts -threaded -eventlog \
>   -o MatMultPM MatMultPM.hs
> ./MatMultPM 3 600 60 60 60 1701 +RTS -N7 -ls
```

No. Cores	Runtime (s)	Relative Speedup	Absolute Speedup
Seq	56.1		1.0
1	62.6	1.0	*0.89*
2	56.9	*1.10*	0.99
4	59.7	1.04	0.95
7	60.2	1.04	0.96

Improving Granularity

Currently parallelise both maps (outer over columns, inner over rows)

Parallelising *only the outer*, and performing the inner sequentially will *increase thread granularity*
$\Longrightarrow$ *row-based parallelism*

```
mulMatParRow :: (NFData a, Num a) =>
                Mat a -> Mat a -> Mat a
mulMatParRow a b =
  (a `mulMat` b) `using` strat
    where
      strat m = parList rdeepseq m
```


Row-clustering

Granularity can be further increased by *'row clustering'*, i.e. evaluating c rows in a single thread, e.g.

```
mulMatParRows :: (NFData a, Num a) =>
  Int -> Mat a -> Mat a -> Mat a
mulMatParRows m a b =
  (a `mulMat` b) `using` strat
    where
      strat m = parListChunk c rdeepseq m
```

Clustering (or chunking) is a common technique for increase the performance of data parallel programs.

Results from the row-clustering version

600 x 600 matrices with clusters of 90 rows:

No. Cores	Runtime (s)	Relative Speedup	Absolute Speedup
Seq	56.1		1.0
1	60.4	1.0	0.93
2	31.4	1.9	1.8
4	18.0	3.4	3.4
7	9.2	6.6	*6.6*

Algorithmic Improvements

Using *blockwise clustering* (a.k.a. Gentleman's algorithm) reduces communication as only part of matrix B needs to be communicated.

N.B. Prior to this point we have preserved the computational part of the program and simply added strategies. Now additional computational components are added to cluster the matrix into blocks size `m` times `n`.

```
mulMatParBlocks :: (NFData a, Num a) =>
  Int -> Int -> Mat a -> Mat a -> Mat a
mulMatParBlocks m n a b =
 (a `mulMat` b) `using` strat
  where
   strat x = return (unblock (block m n x
               `using` parList rdeepseq))
```

Algorithmic changes can drastically improve parallel performance, e.g. by reducing communication or by improving data locality.

Block clustering

`block` clusters a matrix into a matrix of matrices, and `unblock` does the reverse.

```
block :: Int -> Int -> Mat a -> Mat (Mat a)
block m n = map f . chunk m where
  f :: Mat a -> Vec (Mat a)
  f = map transpose . chunk n . transpose

-- Left inverse of @block m n@.
unblock :: Mat (Mat a) -> Mat a
unblock = unchunk . map g where
  g :: Vec (Mat a) -> Mat a
  g = transpose . unchunk . map transpose
```


Results from the Tuned Parallel Version

600 x 600 matrices with block clusters: 20 x 20

No. Cores	Runtime (s)	Relative Speedup	Absolute Speedup
Seq	56.1		1.0
1	60.4	1.0	0.93
2	26.9	*2.2*	2.1
4	14.1	*4.2*	3.9
7	8.4	*7.2*	*6.7*

Speedup graphs

```
> # start a batch of measurements in the background
> sh runItBaldur.sh &
> # once finished, extract the data like this
> cat LOG | sed -ne '/PEs/H;/rows time/H;${G;p}' | sed -e 's/^.*PEs \([0-9]*\).
> # download a gnuplot script for plotting the data
> wget http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/speedups.gp
> # edit speedups, setting seq runtime and x-/y-ranges
> gnuplot speedups.gp
> gv speedups.pdf
```


Parallel Threadscope Profiles

For parallelism profiles compile with option `-eventlog`

```
> ghc -O2 -rtsopts -threaded -eventlog \
      -o parsum_thr_l parsum.hs
```

then run with runtime-system option `-ls`

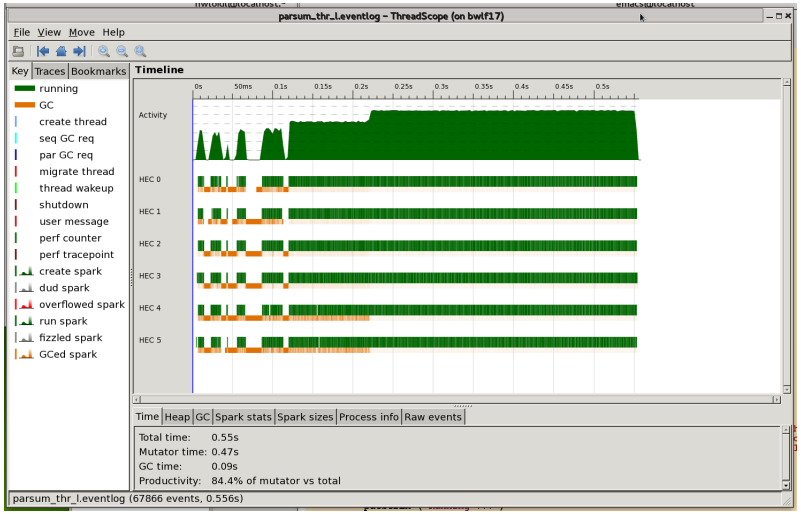
```
> ./parsum_thr_l 90M 100 +RTS -N6 -ls
```

and visualise the generated eventlog profile like this:

```
> threadscope parsum_thr_l.eventlog
```

You probably want to do this on small inputs, otherwise the eventlog file becomes huge!

Parallel Threadscope Profiles

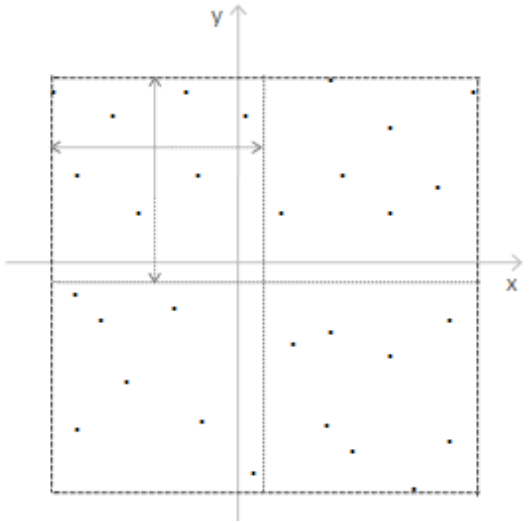


Part 5.
Advanced GpH Programming

Data-oriented Parallelism

- Traversals over large data structures are time consuming and candidates for parallelisation.
- Large, complex data structures are of increasing importance: "BigData" hype
- Examples:
  - quad-trees representing particles in 3D space
  - graphs representing social networks
- These algorithms are increasingly used as parallelism benchmarks: www.graph500.com
- We call parallel algorithms, that are driven by such data-structures, *data-oriented parallelism*

Example data-structure: quad-tree



Example data-structure: quad-tree

As a Haskell data-structure we define a more general *k-ary tree*:

```
data Tree k tl tn = E | L tl
                  | N tn (k (Tree k tl tn))
type QTree tl tn  = Tree Quad tl tn
```

NB:

- Each node `N` can have any number of children
- The `k` constructor-argument fixes how many (`Quad`)
- The type arguments `tl,tn` represent the type of the *leaf*- and *node*-elements

Performance tuning of tree traversals

A common performance problem in large-scale traversal is the *throttling* of the parallelism: we want to limit the total amount of parallelism to avoid excessive overhead, but we need to be flexible in the way we generate parallelism to avoid idle time.

We have already seen some techniques to achieve this. The most commonly used technique is: *depth-based thresholding*

Depth-based Thresholding

Listing 1: Depth-based thresholding

```
parTreeDepth::Int->Strategy (QTree tl)
                 ->Strategy (QTree tl)
parTreeDepth 0 _     t = return t
parTreeDepth d strat (N (Q nw ne sw se))  =
    (N <$> (Q <$> parTreeDepth (d-1) strat nw
              <*> parTreeDepth (d-1) strat ne
              <*> parTreeDepth (d-1) strat sw
              <*> parTreeDepth (d-1) strat se))
    >>= rparWith strat
parTreeDepth _ _     t = return t
```

NB:

- The argument *d* represents the depth (counting down)
- When it drops to 0 no more parallelism is generated
- We use applicative combinators <$> and <*> to compose the overall strategy

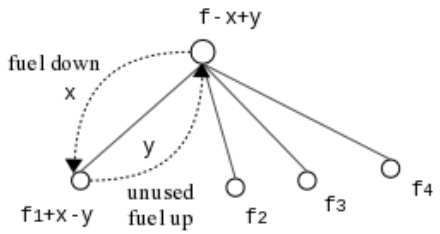
Discussion

- Such depth-based thresholding works well for balanced trees
- For imbalanced trees, however, we want to go deeper down in some sub-trees
- We don't want to hard-code depth-constants into the strategy

We introduce the notion of *fuel*:

- *Fuel* is a limited, explicit resource, needed to generate parallelism
- On traversal, we split the fuel among the children
- This allows for different depths in the traversals
- We use the notion of *fuel give-back* to be even more flexible

Fuel-based parallelism with give-back using circularity



- The resource of "fuel" is used to limit the amount of parallelism when traversing a data structure
- It is passed down from the root of the tree
- It is given back if the tree is empty or fuel is unused
- The give-back mechanism is implemented via *circularity*

Advanced Mechanisms

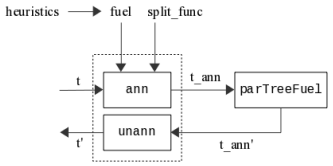
Fuel-based control

- fuel
  - limited resources distributed among nodes
  - similar to "potential" in amortised cost
  - and the concept of "engines" to control computation in Scheme
- parallelism generation (sparks) created until fuel runs out
- more flexible to throttle parallelism

Advanced Mechanisms

Fuel-based control

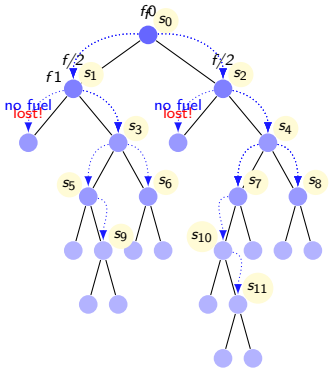
- fuel split function
  - flexibility of defining custom function specifying how fuel is distributed among sub-nodes
  - e.g. *pure*, *lookahead*, *perfectsplit*
  - split function influences which path in the tree will benefit most of parallel evaluation



annotate tree with fuel info based on `split_func`

Fuel-based Control Mechanism

pure, lookahead, perfectsplit



pure
- Info flow: down
- Context: local
- Parameter: *f*

lookahead
- Info flow: down/limited
- Context: local (*N*)
- Parameter: *f*

perfectsplit
- Info flow: down
- Context: global
- Parameter: *f*

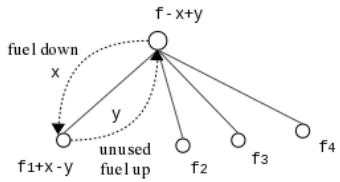
- Characteristics of pure version
  - splits fuel equally among sub-nodes
  - fuel lost on outer nodes
- Characteristics of lookahead version
  - looks ahead *N* level down before distributing unneeded fuel

- Characteristics of perfectsplit version
  - perfect fuel splitting

Advanced Mechanisms

Fuel-based control

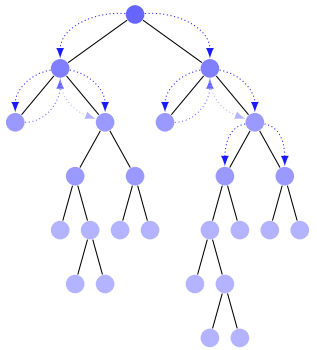
- bi-directional fuel transfer – *giveback* version
  - fuel is passed down from root
  - fuel is given back if tree is empty or fuel is unused
  - *giveback* mechanism is implemented via circularity



- fuel represented using list of values instead of an (atomic) integer
- giveback mechanism is effective in enabling additional parallelism for irregular tree
  - distribution carries deeper inside the tree

Fuel-based Control Mechanism

giveback fuel flow



giveback
- Info flow: down/up
- Context: local
- Parameter: *f*

- f_in: fuel down
- f_out: fuel up
- f_in': fuel reallocated

Advanced Mechanisms

Fuel-based control with giveback using circularity

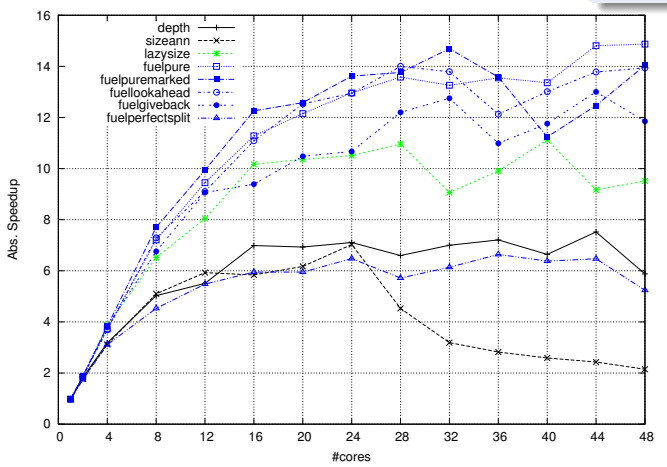
```
-- | Fuel with giveback annotation
annFuel_giveback::Fuel -> QTree tl -> AnnQTree Fuel tl
annFuel_giveback f t = fst \$ ann (fuelL f) t
 where
  ann::FuelL -> QTree tl -> (AnnQTree Fuel tl,FuelL)
  ann f_in E                = (E,f_in)
  ann f_in (L x)            = (L x,f_in)
  ann f_in (N (Q a b c d)) = (N (AQ (A (length f_in)) a' b' c' d'),
      emptyFuelL)
   where
    (f1_in:f2_in:f3_in:f4_in:_) = fuelsplit numnodes f_in
    (a',f1_out) = ann (f1_in ++ f4_out) a
    (b',f2_out) = ann (f2_in ++ f1_out) b
    (c',f3_out) = ann (f3_in ++ f2_out) c
    (d',f4_out) = ann (f4_in ++ f3_out) d
```

- fuel flows back in a circular way

Performance Evaluation

– multiple clusters distr.
– parallel force comp.
– no restructuring of seq code necessary

Barnes-Hut speedups on 1-48 cores. 2 million bodies. 1 iteration.



- pure fuel gives best perf. – simple but cheap fuel distr.; lookahead/giveback within 6/20%
- fuel ann/unann overheads: 11/4% for 2m bodies
- more instances of giveback due to highly irregular input (7682 for 100k bodies, $f = 2000$)

Further Reading & Deeper Hacking

- Prabhat Totoo, Hans-Wolfgang Loidl. *"Lazy Data-Oriented Evaluation Strategies"*. In *FHPC 2014: The 3rd ACM SIGPLAN Workshop on Functional High-Performance Computing*, Gothenburg, Sweden, September, 2014. `http://www.macs.hw.ac.uk/~dsg/gph/papers/abstracts/fhpc14.html`
- S. Marlow and P. Maier and H-W. Loidl and M.K. Aswad and P. Trinder, *"Seq no more: Better Strategies for Parallel Haskell"*. In *Haskell'10 — Haskell Symposium*, Baltimore MD, U.S.A., September 2010. ACM Press. `http://www.macs.hw.ac.uk/~dsg/projects/gph/papers/abstracts/new-strategies.html`
- *"Parallel and concurrent programming in Haskell"*, by Simon Marlow. O'Reilly, 2013. ISBN: 9781449335946.

Further Reading & Deeper Hacking

- An excellent site for learning (sequential) Haskell is: `https://www.fpcomplete.com/school`
- Glasgow parallel Haskell web page: `http://www.macs.hw.ac.uk/~dsg/gph`
- Our course on parallel technologies covers GpH in more detail and has more exercises: `http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP`
- Specifically, for a list of (parallel) Haskell exercises with usage instructions see: `http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/tutorial0.html#gph`

Part 6.
Dataflow Parallelism: The Par Monad

[1] Based on Chapter 4 of "Parallel and concurrent programming in Haskell", by Simon Marlow. O'Reilly, 2013

Dataflow Parallelism: The Par Monad

The Par Monad is a different way to express parallelism in Haskell, which

- gives the programmer *more control* over the parallel execution;
- requires to express parallelism in a monadic style;
- retains the benefit of deterministic parallelism;
- is entirely implemented as a library

This leads to a programming style that is *more explicit* about granularity and data dependencies.

Basic interface of the Par Monad

```
newtype Par a
instance Applicative Par
instance Monad Par
-- execute the monad
runPar :: Par a -> a
-- create a parallel task
fork :: Par () -> Par ()
```

NB:

- `runPar` executes a computation (similar to `runEval`)
- `forkPar` creates a parallel task inside the Par monad

Communication Mechanisms

We also need explicit mechanisms for synchronisation and data exchange:

```
data IVar a  -- instance Eq

new :: Par (IVar a)
put :: NFData a => IVar a -> a -> Par ()
get :: IVar a -> Par a
```

- An `IVar`, or *future*, is a variable with automatic synchronisation.
- It is initially empty.
- If a task `get`s from an empty `IVar`, the task automatically blocks and waits.
- It a task `put`s into an empty `IVar`, that value becomes available to other tasks, and all waiting tasks are awoken.
- If a task `put`s into a non-empty `IVar` an *error* is raised, i.e. they are *single-write*.
- `MVar`s behave like `IVar`s but they are *multi-write*, i.e. without the last restriction (mainly for Concurrent Haskell).

Introductory Example

Here a simple example of running 2 computations (fib) in parallel and adding the results:

```
runPar $ do
      i <- new                  -- create an IVar
      j <- new                  -- create an IVar
      fork (put i (fib n))      -- start a parallel task
      fork (put j (fib m))      -- start a parallel task
      a <- get i                -- get the result (when available)
      b <- get j                -- get the result (when available)
      return (a+b)              -- return the result
```

NB:

- We need two `IVar`s, `i` and `j`, to capture the results from the two recursive calls.
- We use two `fork`s to (asynchronously) launch the recursive calls.
- The forked code must take care to return the value in the expected `IVar`
- The main thread blocks on the `IVar`s until their values become available.
- Finally, the main thread returns the sum of both values.

A parMap pattern

First we generate a helper function that combines a `fork` with a custom `IVar` for the result:

```
spawn :: NFData a => Par a -> Par (IVar a)
spawn p = do
  i <- new
  fork (do x <- p; put i x)
  return i
```

Now we can define a ParMonad version of our favourite `parMap` pattern:

```
parMapM :: NFData b => (a -> Par b) -> [a] -> Par [b]
parMapM f as = do
  ibs <- mapM (spawn . f) as
  mapM get ibs
```

- `parMapM` uses a monadic version of map, `mapM`, to perfrom a computation over every element of a list.
- It then extracts the results out of the resulting list of `IVar`s
- `f` itself is a computation in the Par monad, so it can generate more, nested parallelism
- Note that this version of `parMapM` waits for all its results before returning
- Note that `put` will fully evaluate its argument, by internally calling `deepseq`
- For a lazy version of `put`, use `put_`

Parallel Fibonacci

Here our favourite Fibonacci example using ParMonad:

```
pfib :: Int -> Par Int
pfib n | n<=1 = return 1      -- base case
pfib n | otherwise =
 do
   nv1 <- spawn (pfib (n-1)) -- start a parallel task & return IVar
   nv2 <- spawn (pfib (n-2)) -- start a parallel task & return IVar
   nf1 <- get nv1            -- get the result (when available)
   nf2 <- get nv2            -- get the result (when available)
   return (nf1+nf2+1)
```

NB:

- we use `spawn` to automatically generate an `IVar` in each recursive call
- in this naive version, we generate parallelism in all levels of the tree
- $\Longrightarrow$ poor granularity
- to improve performance, introduce thresholding to the program

Example: Shortest Paths (Idea)

We want to implement a parallel version of the Floyd-Warshall all-pairs shortest-path algorithm.
This is a *naive version* of the algorithm, capturing its basic idea:

```
shortestPath :: Graph -> Vertex -> Vertex -> Vertex -> Weight
shortestPath g i j 0 = weight g i j
shortestPath g i j k = min (shortestPath g i j (k-1))
                           (shortestPath g i k (k-1) + shortestPath
                                g k j (k-1))
```

- `shortestPath g i j k` ... length of the shortest path from `i` to `j`, passing through vertices up to `k` only
- $k == 0$ ... the paths between each pair of vertices consists of the direct edges only
- For a non-zero `k`, there are two cases:
  - if the shortest path from `i` to `j` passes through `k`: the length is sum of the shortest path from `i` to `k` and from `k` to `j`
  - otherwise: the length is the same as the one only using nodes up to `k-1`
- the overall result is the *minimum* of both cases

Example: Shortest Path (Floyd-Warshall)

Pseudo-code algorithm:

```
ShortestPaths(W):
n := rows(W)
D[0] := W
for k in 1 to n do
  for i in 1 to n do
    for j in 1 to n do
      D[k][i,j] := min(D[k-1][i,j], D[k-1][i,k]+D[k-1][k,j]

return D[n]
```

For a visualisation of the Floyd-Warshall algorithm, see this web page: https://www.cs.usfca.edu/ galles/visualization/Floyd.html.

Example: Shortest Paths (code)

Below is a sequential implementation of the Floyd-Warshall all-pairs shortest-path algorithm:

```
shortestPaths :: [Vertex] -> Graph -> Graph
shortestPaths vs g = foldl' update g vs              -- <1>
 where
  update g k = Map.mapWithKey shortmap g             -- <2>
   where
     shortmap :: Vertex -> IntMap Weight -> IntMap Weight
     shortmap i jmap = foldr shortest Map.empty vs -- <3>
        where shortest j m =
                case (old,new) of                    -- <6>
                   (Nothing, Nothing) -> m
                   (Nothing, Just w ) -> Map.insert j w m
                   (Just w,  Nothing) -> Map.insert j w m
                   (Just w1, Just w2) -> Map.insert j (min w1 w2) m
                where
                  old = Map.lookup j jmap            -- <4>
                  new = do w1 <- weight g i k        -- <5>
                           w2 <- weight g k j
                           return (w1+w2)
```


Example: Shortest Paths (explanation)

- <1> the left-fold over the vertices corresponds to iterating over `k` in the naive version
- <3> `shortmap` takes `i`, the current vertex, and `jmap`, the mapping of shortest paths from `i`.
- <4> shortest path from `i` to `j`
- <5> shortest path from i to j via k (if one exists)
- <6> the result is the minimum

Example: Shortest Paths (Parallel Version)

- The key idea in parallelising this algorithm is to parallelise the `update` function, which is a (slightly unusual) map: `Map.mapWithKey`
- To parallelise this function we use a library function `traverseWithKey`, which provides a monadic version of a traversal over a `Map`
- `traverseWithKey`
  - maps a monadic function over `IntMap`;
  - the monadic function takes an element (of type `a`) and a key as arguments;
  - this matches the interface of `shortmap`, which needs a `Vertex` (source) and an `IntMap` (the map from destination vertices to weights) as arguments

```
update g k = runPar $ do
    m <- Map.traverseWithKey (\i jmap -> spawn (return (shortmap i
        jmap))) g
    traverse get m
```

Example: Shortest Paths (Parallel Version)

- `Map.traverseWithKey` returns an `IntMap (IVar (IntMap Weight))`; that is, there's an `IVar` in place of each element.
- To get the new `Graph`, we need to call `get` on each of these `IVars` and produce a new `Graph` with all the elements, which is what the final call to traverse does.
- The `traverse` function is from the Traversable class; for our purposes here, it is like `traverseWithKey` but doesn't pass the Key to the function.

NB: the rest of the algorithm is unchanged!

Example: Running Shortest Path

Running the sequential algorithm gives us the following baseline:

```
$ ./fwsparse 1000 800 +RTS -s
...
  Total   time    4.16s  (  4.17s elapsed)
```

Running this algorithm on 4 cores gives a speedup of approx. 3.02

```
$ ./fwsparse1 1000 800 +RTS -s -N4
...
  Total   time    5.27s  (  1.38s elapsed)
```


Par Monad Compared to Strategies

Trade-offs in the choice between Evaluation Strategies and the Par monad:

- If your algorithm naturally produces a *lazy data structure*, then writing a Strategy to evaluate it in parallel will probably work well.
- `runPar` is relatively expensive, whereas `runEval` is free. Therefore, use *coarser grained parallelism with the Par monad*, and be careful about nested parallelism.
- Strategies allow a *separation between computation and coordination*, which can allow more reuse and a cleaner specification of parallelism.
- Parallel skeletons can be defined on top of both approaches.
- The Par monad is implemented entirely in a Haskell *library* (the monad-par package), and is thus easily modified. There is a choice of scheduling strategies
- The Eval monad has more diagnostics in *ThreadScope*, showing creation rate, conversion rate of sparks, etc.
- The Par monad does not support speculative parallelism in the sense that `rpar` does

Further Reading & Deeper Hacking

- *"Parallel and concurrent programming in Haskell"*, by Simon Marlow. O'Reilly, 2013. ISBN: 9781449335946.
  Full sources are available on Hackage.

Exercises

- run the naive version of parfib and observe the performance
- implement thresholding in this version to improve performance
- pick-up the `parsum` example from the GpH part, and rewrite it in ParMonad notation
- implement a parallel version of the Euler totient function, using the parallel map in ParMonad notation

Part 7.
Skeletons
Implementations of Parallel Patterns

Overview

- Higher-order functions cover common patterns of computation
- Higher-order functions are a natural construct in functional languages like Haskell (link to previous examples)
- If implemented in parallel, these provide parallelism for free
- Examples of parallel patterns: map-reduce, task-farm etc

Algorithmic Skeletons — What?

A *skeleton* is

- a useful pattern of parallel computation and interaction,
- packaged as a *framework/second order/template* construct (i.e. parametrised by other pieces of code).
- *Slogan:* Skeletons have *structure* (coordination) but lack *detail* (computation).

Each skeleton has

- one interface (e.g. generic type), and
- one or more (architecture-specific) implementations.
  - Each implementations comes with its own *cost model*.

A skeleton *instance* is

- the code for computation together with
- an implementation of the skeleton.
  - The implementation may be shared across several instances.

Note: Skeletons are more than design patterns.

Algorithmic Skeletons — How and Why?

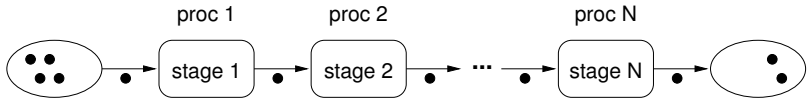
Programming methodology:

1. Write sequential code, identifying where to introduce parallelism through skeletons.
2. Estimate/measure sequential processing cost of potentially parallel components.
3. Estimate/measure communication costs.
4. Evaluate cost model (using estimates/measurements).
5. Replace sequential code at sites of useful parallelism with appropriate skeleton instances.

Pros/Cons of skeletal parallelism:

+ simpler to program than unstructured parallelism
+ code re-use (of skeleton implementations)
+ structure may enable optimisations
- *not* universal

Common Skeletons — Pipeline



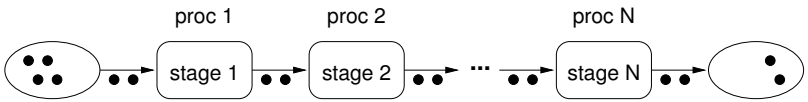
- Data flow skeleton
  - Data items pass from stage to stage.
  - All stages compute in parallel.
  - Ideally, pipeline processes many data items (e.g. sits inside loop).

Pipeline — Load Balancing

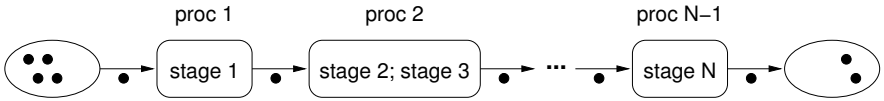
Typical problems:

1. Ratio communication/computation too high.
2. Computation cost not uniform over stages.

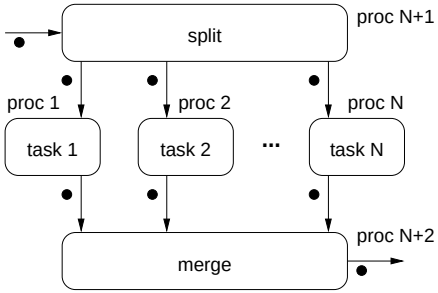
Ad (1) Pass chunks instead of single items



Ad (1,2) Merge adjacent stages

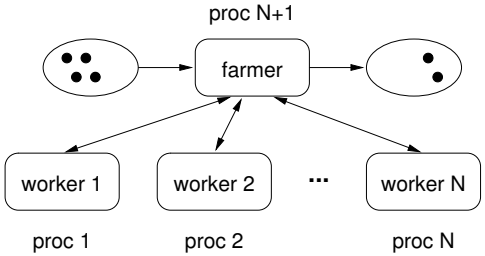


Common Skeletons — Parallel Tasks



- Data flow skeleton
  - Input split on to fixed set of (different) tasks.
  - Tasks compute in parallel.
  - Output gathered and merged together.
    - ⋆ Split and merge often trivial; often executed on proc 1.
- Dual (in a sense) to pipeline skeleton.
- **Beware:** Skeleton name non-standard.

Common Skeletons — Task Farm

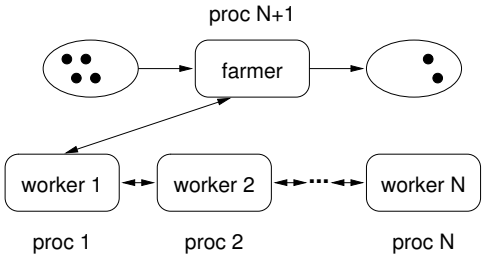


- Data parallel skeleton (e.g. parallel sort scatter phase)
  - Farmer distributes input to a pool of N identical workers.
  - Workers compute in parallel.
  - Farmer gathers and merges output.
- Static vs. dynamic task farm:
  - *Static:* Farmer splits input once into N chunks.
    - Farmer may be executed on proc 1.
  - *Dynamic:* Farmer continually assigns input to free workers.

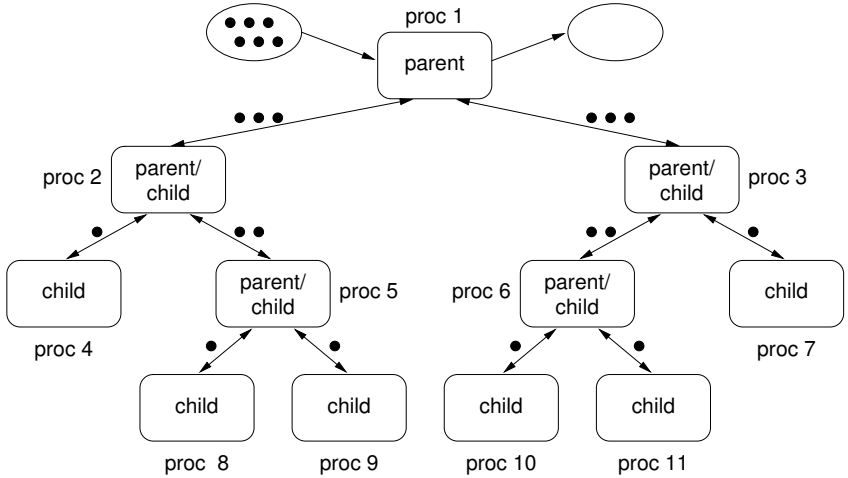
Task Farm — Load Balancing

Typical problems:

1. Irregular computation cost (worker).
   - Use dynamic rather than static task farm.
   - Decrease chunk size: Balance granularity vs. comm overhead.
2. Farmer is bottleneck.
   - Use self-balancing *chain gang* dynamic task farm.
     - Workers organised in linear chain.
     - Farmer keeps track of # free workers, sends input to first in chain.
     - If worker busy, sends data to next in chain.



Common Skeletons — Divide & Conquer



- Recursive algorithm skeleton (e.g. parallel sort merge phase)

Common Skeletons — Divide & Conquer II

- Recursive algorithm skeleton
  - Recursive call tree structure
    - Parent nodes *divide* input and pass parts to children.
    - All leaves compute the same sequential algorithm.
    - Parents gather output from children and *conquer*, i.e. combine and post-process output.
- To achieve good load balance:
  1. Balance call tree.
  2. Process data in parent nodes as well as at leaves.

Skeletons in the Real World

Skeletal Programming

- can be done in many programming languages,
  - skeleton libraries for C/C++
  - skeletons for functional languages (GpH, OCaml, ...)
  - skeletons for embedded systems
- is still not mainstream,
  - Murray Cole. *Bringing Skeletons out of the Closet*, Parallel Computing 30(3) pages 389–406, 2004.
  - González-Vélez, Horacio and Leyton, Mario. *A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers*, Software: Practice and Experience 40(12) pages 1135–1160, 2010.
- but an active area of research.
  - $>$ 30 groups/projects listed on skeleton homepage
- and it is slowly becoming mainstream
  - TPL library of Parallel Patterns in C# (blessed by Microsoft)

Skeletons Are Parallel Higher-Order Functions

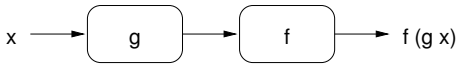
Observations:

- A *skeleton* (or any other template) is essentially a higher-order function (HOF), ie. a function taking functions as arguments.
  - Sequential code parameters are functional arguments.
- Skeleton implementation is parallelisation of HOF.
- Many well-known HOFs have parallel implementations.
  - Thinking in terms of higher-order functions (rather than explicit recursion) helps in discovering parallelism.

Consequences:

- Skeletons can be combined (by function composition).
- Skeletons can be nested (by passing skeletons as arguments).

Skeletons Are PHOFs — Pipeline



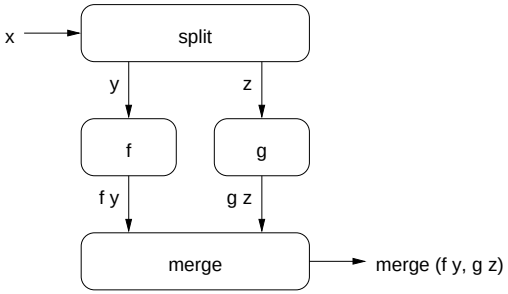
Code (parallel implementation in red)

```
pipe2 :: (b -> c) -> (a -> b) -> a -> c
pipe2 f g x = let y = g x in
              y `par` f y
```

Notes:

- `pipe2` is also known as *function composition*.
- In Haskell, sequential function composition is written as `.` (read "dot").

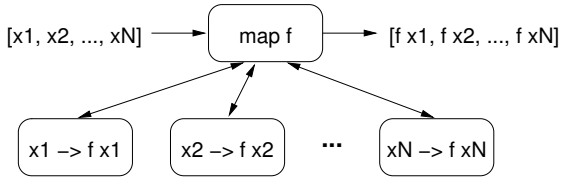
Skeletons Are PHOFs — Parallel Tasks



Code (parallel implementation in red)

```
task2 :: (a -> (b,c)) -> (d -> e -> f) -> (b -> d) -> (c -> e) -> a -> f
task2 split merge f g x = let (y,z) = split x
                              fy = f y
                              gz = g z in
                          fy `par` gz `pseq` merge fy gz
```

Skeletons Are PHOFs — Task Farm



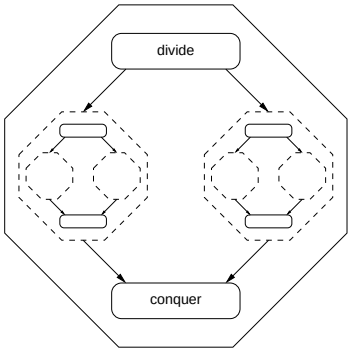
Code (parallel implementation in red)

```
farm :: (a -> b) -> [a] -> [b]
farm f []     = []
farm f (x:xs) = let fx = f x in
                fx `par` fx : (farm f xs)
```

Notes:

- `farm` is also known as *parallel map*.
  - Map functions exist for many data types (not just lists).
- Missing in implementation: strategy to force eval of lazy list.
- Strategies also useful to increase granularity (by chunking).

Skeletons Are PHOFs — Divide & Conquer



Code (parallel implementation in red)

```
dnc :: (a -> (a,a)) -> (b -> b -> b) -> (a -> Bool) -> (a -> b) -> a -> b
dnc div conq atomic f x | atomic x  = f x
                        | otherwise = let (l0,r0) = div x
                                          l = dnc div conq atomic f l0
                                          r = dnc div conq atomic f r0 in
                                      l `par` r `pseq` conq l r
```


Skeletons Are PHOFs — Divide & Conquer

Notes:

- Divide & Conquer is a generalised *parallel fold*.
  - Folds exist for many data types (not just lists).
- Missing in impl: strategies to force eval and improve granularity.

Aside: folding/reducing lists

```
fold :: (a -> a -> a) -> a -> [a] -> a
-- fold f e [x1,x2,...,xn] == e `f` x1 `f` x2 ... `f` xn, provided that
-- (1) f is associative, and
-- (2) e is an identity for f.

-- Tail-recursive sequential implementation:
fold f e []     = e
fold f e (x:xs) = fold f (e `f` x) xs

-- Parallel implementation as instance of divide & conquer:
fold f e = dnc split f atomic evalAtom where
    split xs = splitAt (length xs `div` 2) xs
    atomic []  = True
    atomic [_] = True
    atomic _   = False
    evalAtom []  = e
    evalAtom [x] = x
```

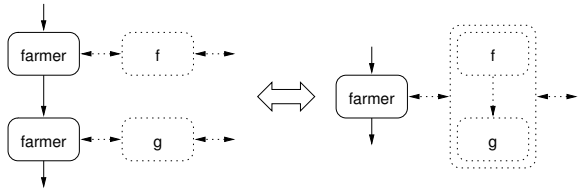

Program Transformations

Observation:

- HOFs can be transformed into other HOFs with provably equivalent (sequential) semantics.

Example: Pipeline of farms vs. farm of pipelines

- `map g . map f == map (g . f)`



- use `map g . map f` (pipe of farms) if ratio comp/comm high
- use `map (g . f)` (farm of pipes) if ratio comp/comm low
- More transformations in
  - G. Michaelson, N. Scaife. *Skeleton Realisations from Functional Prototypes*, Chap. 5 in S. Gorlatch and F. Rabhi (Eds), *Patterns and Skeletons for Parallel and Distributed Computing*, Springer, 2002

Program Development with Functional Skeletons

Programming Methodology:

1. Write seq code using HOFs with known equivalent skeleton.
2. Measure sequential processing cost of functions passed to HOFs.
3. Evaluate skeleton cost model.
4. If no useful parallelism, transform program and go back to 3.
5. Replace HOFs that display useful parallelism with their skeletons.

Tool support:

- Compilers can automate some steps (see Michaelson/Scaife)
  - Only for small, pre-selected set of skeletons
- Example: PMLS (developed by Greg Michaelson et al.)
  - Skeletons: map/fold (arbitrarily nested)
  - Automates steps 2-5.
    - ⋆ Step 2: automatic profiling
    - ⋆ Step 4: rule-driven program transformation + synthesis of HOFs
    - ⋆ Step 5: map/fold skeletons implemented in C+MPI

Further Reading

- Ian Foster. *"Designing & Building Parallel Programs: Concepts & Tools for Parallel Software Engineering"*, Addison-Wesley, 1995
  Online: `http://www.mcs.anl.gov/~itf/dbpp/`
- J. Dean, S. Ghemawat. *"MapReduce: Simplified Data Processing on Large Clusters"*. Commun. ACM 51(1):107–113, 2008.
  Online: `http://dx.doi.org/10.1145/1327452.1327492`
- G. Michaelson, N. Scaife. *"Skeleton Realisations from Functional Prototypes"*, Chap. 5 in S. Gorlatch and F. Rabhi (Eds), *Patterns and Skeletons for Parallel and Distributed Computing*, Springer, 2002
- Michael McCool, James Reinders, Arch Robison. *"Structured Parallel Programming"*. Morgan Kaufmann Publishers, Jul 2012. ISBN10: 0124159931 (paperback)

A simple example: parallel fold

```
parfold :: (a -> a -> a) -> a -> [a] -> a
parfold f z l = parfold' (length l) f z l
where
  parfold' _ f z [] = z
  parfold' _ f z [x] = x
  parfold' n f z xs =
    let n2 = n `div` 2 in!
    let (l,r) = splitAt n2 xs in!
    let lt = parfold' n2 f z l;
        rt = parfold' (n2 + n `rem` 2) f z r in
        rt `par` (lt `pseq` f lt rt)!
```


MapReduce — For Functional Programmers

What func programmers think when they hear "map/reduce"

```
-- map followed by reduce (= fold of associative m with identity e)
fp_map_reduce ::    (a -> b)
                 -> (b -> b -> b) -> b
                 -> [a] -> b
fp_map_reduce f m e = foldr m e . map f

-- map followed by group followed by groupwise reduce
fp_map_group_reduce ::    (a -> b)
                       -> ([b] -> [[b]])
                       -> ([b] -> b)
                       -> [a] -> [b]
fp_map_group_reduce f g r = map r . g . map f

-- list-valued map then group then groupwise list-valued reduce
fp_map_group_reduce' ::    (a -> [b])
                        -> ([b] -> [[b]])
                        -> ([b] -> [b])
                        -> [a] -> [[b]]
fp_map_group_reduce' f g r = map r . g . (concat . map f)
```

MapReduce — For Functional Programmers

Google's MapReduce (sequential semantics)

```
-- list-valued map then fixed group stage then list-valued reduce
map_reduce :: Ord c => ((a,b) -> [(c,d)])
                    -> (c -> [d] -> [d])
                    -> [(a,b)] -> [(c,[d])]
map_reduce f r = map (\ (k,vs) -> (k, r k vs)) .
                 (group . sort) .
                 (concat . map f)
  where sort :: Ord c => [(c,d)] -> [(c,d)]
        sort = sortBy (\ (k1,_) (k2,_) -> compare k1 k2)
        group :: Eq c => [(c,d)] -> [(c,[d])]
        group = map (\ ((k,v):kvs) -> (k, v : map snd kvs)) .
                groupBy (\ (k1,_) (k2,_) -> k1 == k2)
```

- Specialised for processing key/value pairs.
  - Group by keys
  - Reduction may depend on key and values
- Not restricted to lists — applicable to any container data type
  - Reduction should be associative+commutative in 2nd argument

MapReduce — Applications

TotientRange

```
-- Euler phi function
euler :: Int -> Int
euler n = length (filter (relprime n) [1 .. n-1])
  where relprime x y = hcf x y == 1
        hcf x 0 = x
        hcf x y = hcf y (rem x y)

-- Summing over the phi functions in the interval [lower .. upper]
sumTotient :: Int -> Int -> Int
sumTotient lower upper = head (snd (head (map_reduce f r input)))
    where input :: [((),Int)]
          input = zip (repeat ()) [lower, lower+1 .. upper]
          f :: ((),Int) -> [((),Int)]
          f (k,v) = [(k, euler v)]
          r :: () -> [Int] -> [Int]
          r _ vs = [sum vs]   -- reduction assoc+comm in 2nd arg
```

- Degenerate example: only single key
- Still exhibits useful parallelism
  - but would not perform well on Google's implementation

MapReduce — Applications

URL count

```
isURL :: String -> Bool
isURL word = "http://" `isPrefixOf` word

-- input:  lines of log file
-- output: frequency of URLs in input
countURL :: [String] -> [(String,[Int])]
countURL lines = map_reduce f r input
  where input :: [((),String)]
        input = zip (repeat ()) lines
        f :: ((),String) -> [(String,Int)]
        f (_,line) = zip (filter isURL (words line)) (repeat 1)
        r :: String -> [Int] -> [Int]
        r url ones = [length ones]
```

- Map phase
  1. breaks line into words
  2. filters words that are URLs
  3. zips URLs (which become keys) with value 1
- Group phase groups URLs with values (which = 1)
- Reduction phase counts #values

MapReduce — How To Parallelise

Sequential code

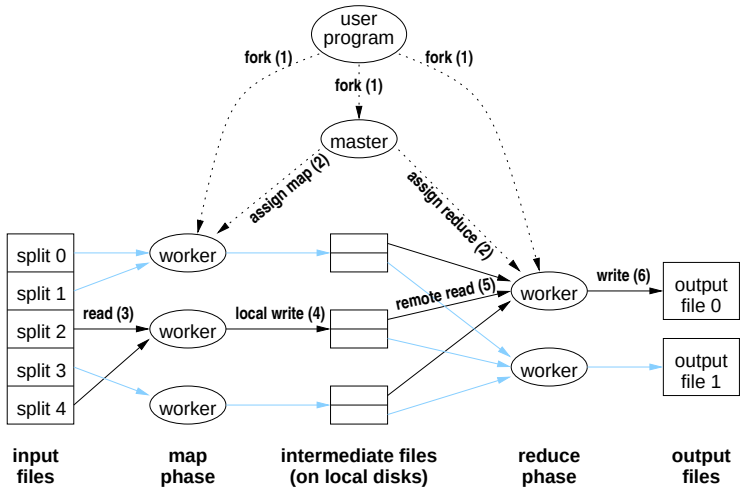
```
map_reduce f r = map (\ (k,vs) -> (k, r k vs)) .
                 (group . sort) .
                 (concat . map f)
```

suggests 3-stage pipeline

1. map phase
   - data parallel task farm
2. parallel sorting and grouping
   - parallel mergesort
3. groupwise reduce phase
   - data parallel task farm

Note: This is not how Google do it.

Google MapReduce — Execution Overview



- J. Dean, S. Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*, Commun. ACM 51(1):107–113, 2008

Google MapReduce — Execution Overview

Execution steps:

1. User program forks master, M map workers, R reduce workers.
2. Master assigns map/reduce tasks to map/reduce workers.
   - Map task = 16–64 MB chunk of input
   - Reduce task = range of keys + names of M intermediate files
3. Map worker reads input from GFS and processes it.
4. Map worker writes output to local disk.
   - Output partitioned into R files (grouped by key)
5. Reduce worker gathers files from map workers and reduces them.
   1. Merge M intermediate files together, grouping by key.
   2. Reduce values groupwise.
6. Reduce worker writes output to GFS.
7. Master returns control to user program after all task completed.

Main Selling Points of MapReduce

- Easy to use for non-experts in parallel programming (details are hidden in the MapReduce implementation)
- Fault tolerance is integrated in the implementation
- Good modularity: many problems can be implemented as sequences of MapReduce
- Flexibility: many problems are instances of MapReduce
- Good scalability: using 1000s of machines at the moment
- Tuned for large data volumes: several TB of data
- Highly tuned parallel implementation to achieve eg. good load balance

Eden: a skeleton programming language

See separate slides and examples

Further Reading & Deeper Hacking

- S. Marlow and P. Maier and H-W. Loidl and M.K. Aswad and P. Trinder, *"Seq no more: Better Strategies for Parallel Haskell"*. In *Haskell'10 — Haskell Symposium*, Baltimore MD, U.S.A., September 2010. ACM Press. `http://www.macs.hw.ac.uk/~dsg/projects/gph/papers/abstracts/new-strategies.html`
- Prabhat Totoo, Hans-Wolfgang Loidl. *"Lazy Data-Oriented Evaluation Strategies"*. In *FHPC 2014: The 3rd ACM SIGPLAN Workshop on Functional High-Performance Computing*, Gothenburg, Sweden, September, 2014. `http://www.macs.hw.ac.uk/~dsg/projects/gph/papers/abstracts/fhpc14.html`
- *"Parallel and concurrent programming in Haskell"*, by Simon Marlow. O'Reilly, 2013. ISBN: 9781449335946.
- Slides on the Eden parallel Haskell dialect: `http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/Eden-slides.pdf`

Further Reading & Deeper Hacking

- An excellent site for learning (sequential) Haskell is: `https://www.fpcomplete.com/school`
- Glasgow parallel Haskell web page: `http://www.macs.hw.ac.uk/~dsg/gph`
- Our course on parallel technologies covers GpH in more detail and has more exercises: `http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP`
- Specifically, for a list of (parallel) Haskell exercises with usage instructions see: `http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/tutorial0.html#gph`