Parallel Functional Programming

Yolanda Ortega Mallén
Universidad Complutense de Madrid (Spain)

Thanks: Rita Loogen - Philipps-Universität Marburg (Germany)
Overview

- Introduction
- From Implicit to Controlled Parallelism
- Process-control and Coordination Languages
- Data Parallelism
- Algorithmic Skeletons
Kevin Hammond and Greg Michaelson (Editors): *Research Directions in Parallel Functional Programming*  
Springer 1999  
20 chapters by 27 authors  
> 600 references

This presentation uses mainly material from the following chapters:

- **Foreword** by Simon Peyton Jones
- **Chapter 1**: *Introduction* by Kevin Hammond and Greg Michaelson
- **Chapter 3**: *Programming Language Constructs* by Rita Loogen
- **Chapter 7**: *Data Parallelism* by John O’Donnell
- **Chapter 13**: *Algorithmic Skeletons* by Murray Cole
- **Chapter 14**: *Coordination Languages* by Paul Kelly
Excerpts from the Foreword by S. Peyton Jones

Programming is hard. ... But parallel programming is much, much harder. ...
Functional programming is a radical, elegant, high-level attack on the programming problem. ...
Parallel functional programming is the same, only more so. The rewards are even greater. ...

Parallelism without tears, perhaps? Definitely not. ... Two things have become clear over the last 15 years or so.
First, it is a very substantial task to engineer a parallel functional language implementation. ...
Second, ... Quite a bit of work needs to go into designing and expressing a parallel algorithm for the application. ...

Is parallel functional programming any good? If I am honest, I have to say that the jury is still out. ....

Is it worth bothering, then? Emphatically, yes!
Why Functional Programming Matters

- Hughes 1989: promotes modular programming
  - Ease of program construction
  - Ease of function/module reuse
  - Simplicity
  - Generality through higher-order functions
    (“functional glue”)
- Additional points suggested by experience
  - Ease of reasoning / proof
  - Ease of program transformation
  - Scope for optimisation
Why **Parallel** Functional Programming **Matters**

- Advantages of functional programming
- Hammond 1999: additional reasons for the parallel programmer:
  - Ease of partitioning a parallel program
  - Simple communication model
  - Absence of deadlock
  - Straightforward semantic debugging
  - Easy exploitation of pipelining and other parallel control constructs

Specify **what** has to be evaluated in parallel and **not how** the parallel evaluation has to be organised.
Inherent Parallelism

- **Church Rosser property** (confluence) of reduction semantics
  independent subexpressions can be evaluated in **parallel**

- **Data dependencies** introduce the need for **communication**:

  ```
  let f x = e1
  g x = e2
  in (f 10) + (g 20)
  ```

  ```
  let f x = e1
  g x = e2
  in g (f 10)
  ```
Semantic Properties

- **Determinacy:** Purely functional programs have the same semantic value when evaluated in parallel as when evaluated sequentially. The value is independent of the evaluation order.
  - no race conditions
  - system issues as variations in communication latencies, or the intricacies of scheduling of parallel tasks, do not affect the result.
  - Testing and debugging can be done on a sequential machine.
  - Performance monitoring tools necessary on the parallel machine.

- **Absence of Deadlock:** Any program that delivers a value when run sequentially will deliver the same value when run in parallel.
Parallelism vs. Concurrency

- cooperation of inter-dependent tasks on a single activity
- transformational systems, parallel algorithms
  - improve performance (speed, throughput or response time)
  - by creating subtasks to deal with units of work

- independent, but collaborating, processes
- reactive systems, e.g. graphical user interfaces, operating systems
  - support abstraction and improve security
  - by separating activities in logically independent processes
<table>
<thead>
<tr>
<th>Parallelism</th>
<th>control</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>implicit</td>
<td>automatic parallelisation</td>
<td>data parallel languages</td>
</tr>
<tr>
<td></td>
<td>annotation-based languages</td>
<td></td>
</tr>
<tr>
<td>controlled</td>
<td>para-functional programming evaluation strategies</td>
<td>high-level data parallelism</td>
</tr>
<tr>
<td></td>
<td>skeletons</td>
<td></td>
</tr>
<tr>
<td>explicit</td>
<td>process control languages</td>
<td></td>
</tr>
<tr>
<td></td>
<td>message passing languages</td>
<td></td>
</tr>
<tr>
<td></td>
<td>concurrent languages</td>
<td></td>
</tr>
</tbody>
</table>
Examples

- **binomial coefficients:**
  ```haskell
data Binomial Coefficients

binom :: Int -> Int -> Int
binom n k |
| k == 0 && n >= 0 = 1
| n < k && n >= 0 = 0
| n >= k && k >= 0 = binom (n-1) k + binom (n-1) (k-1)
| otherwise = error "negative params"
```

- **multiplication of sparse matrices with dense vectors:**
  ```haskell
data Sparse Matrix

type SparseMatrix a = [[[Int,a]]] -- rows with (col,nz-val) pairs

type Vector a = [a]

matvec :: Num a => SparseMatrix a -> Vector a -> Vector a
matvec m v = map (sum.map (\ (i,x) -> x * v!!i)) m
```
**From Implicit to Controlled Parallelism**

**Implicit Parallelism:** exploit parallelism that is inherent in the reduction semantics
- Automatic Parallelisation, Strictness Analysis
- Indicating Parallelism: parallel let, annotations, parallel combinators

*semantically transparent parallelism introduced through low-level language constructs*

**Controlled Parallelism**
- Para-functional programming
- Evaluation strategies

*still semantically transparent parallelism programmer is aware of parallelism higher-level language constructs*
**Automatic Parallelisation**

(Lazy) Functional Language

Parallel Intermediate Language

Parallel Computer

Parallelising Compiler:
- Strictness analysis
- Granularity / Cost analysis

Low level parallel language constructs

Parallel runtime system
Strictness Analysis

- Lazy evaluation is inherently sequential because demand for the evaluation of sub-expressions is propagated step-by-step outside-in.

- Strictness analysis is necessary in order to avoid speculative parallelism and preserve conservative parallelism.

A function \( f : D_1 \rightarrow D_2 \rightarrow \ldots \rightarrow D_n \rightarrow D_{n+1} \) is strict in its i-th argument, if
\[
f a_1 \ldots a_{i-1} \perp a_{i+1} \ldots a_n = \perp
\]
for all \( a_i \in D_i \) (\( 1 \leq i \leq n \)).

- Demandedness \( \Rightarrow \) Strictness, but not vice versa

It is safe to evaluate strict function arguments in parallel.
Indicating Parallelism

- parallel let
- annotations
- predefined combinators

Very difficult to detect parallelism automatically.

It is common for programmers to indicate parallelism manually.

- semantically transparent
- only advice for the compiler
- do not enforce parallel evaluation
Parallel let

- allow local definitions within expressions
- can be used to express parallelism explicitly

```
letpar var_1 = expr_1
... 
var_p = expr_p
in expr
```

expressions that may be evaluated in parallel

- **advantages:**
  - special compilation scheme independent from the evaluation order of the sequential compiler can be used
  - additional decoration possible: information on how to allocate processes to processors or about the degree of safe evaluation

- **disadvantages:**
  - parallelisation requires program transformation
  - small indirection overhead on sequential evaluation
Examples for Parallel let

**binomial coefficients:**

```haskell
class :: Int -> Int -> Int
binom n k | k == 0 && n >= 0 = 1
           | n < k && n >= 0 = 0
           | n >= k && k >= 0 = letpar v = binom (n-1) (k-1)
in binom (n-1) k + v
           | otherwise = error "negative params"
```

**parallel map:**

```haskell
parmap :: (a -> b) -> [a] -> [b]
parmap f [] = []
parmap f (x:xs) = letpar y = (f x) -- only safe for NF evaluation
in y : parmap f xs
```
Annotations

- easiest way to indicate parallelism
- sub-expressions that can be evaluated in parallel are annotated

**Concurrent Clean annotations:**

```
{| P |} and {| P AT location |}:: creation of parallel process which evaluates the annotated expression to WHNF
{| I |} initiates an interleaved process on the same processor element
```

- **advantages:**
  - easy to use, only runtime behaviour is affected
  - sequential compilation can simply ignore the annotations

- **disadvantages:**
  - runtime behavior depends on evaluation order (left-to-right, right-to-left), additional annotations to influence the evaluation order
Examples for Annotations

**binomial coefficients:**

binom :: Int -> Int -> Int

\[
\text{binom } n \ k \ | \ k == 0 \ &\ & n >= 0 = 1 \\
| n < k \ &\ & n >= 0 = 0 \\
| n >= k \ &\ & k >= 0 = \text{binom} (n-1) \ k + \{\{P\}\} \ \text{binom} (n-1) (k-1) \\
| \text{otherwise} = \text{error} \ "\text{negative params}" \\
\]

**parallel map:**

parmap :: (a-> b) -> [a] -> [b]

\[
\text{parmap } f \ [ ] = [ ] \\
\text{parmap } f \ (x:xs) = \{\{P\}\} (f x) : \{\{I\}\} \ \text{parmap } f \ xs \\
\]

Forces evaluation of complete list
Parallel Combinators

- special projection functions which provide control over the evaluation of their arguments

in Glasgow parallel Haskell (GpH):  \( \text{par, seq :: a -> b -> b} \)

\( \text{par e1 e2} \) creates a spark for \( e1 \) and returns \( e2 \).
A spark is a marker that an expression can be evaluated in parallel.

\( \text{seq e1 e2} \) evaluates \( e1 \) to WHNF and returns \( e2 \)
(sequential composition).

- advantages:
  - \textbf{simple}, annotations as functions

- disadvantages:
  - explicit control of evaluation order by use of seq necessary
  - programs must be restructured
Examples with Parallel Combinators

- **binomial coefficients:**

  \[
  \text{binom} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}
  \]

  \[
  \text{binom} \ n \ k \mid k == 0 \&\& n >= 0 = 1 \\
  \mid n < k \&\& n >= 0 = 0 \\
  \mid n >= k \&\& k >= 0 = \text{let } b1 = \text{binom} (n-1) k \\
  b2 = \text{binom} (n-1) (k-1) \\
  \text{in } b2 \ 'par' \ b1 \ 'seq' \ (b1 + b2) \\
  \mid \text{otherwise } = \text{error "negative params"}
  \]

- **parallel map:**

  \[
  \text{parmap} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]
  \]

  \[
  \text{parmap} \ f \ [ ] = [ ] \\
  \text{parmap} \ f \ (x:xs) = \text{let } fx = (f x) \\
  fxs = \text{parmap} \ f \ xs \\
  \text{in } fx \ 'par' \ fxs \ 'seq' \ (fx : fxs)
  \]

explicit control of evaluation order
Controlled Parallelism

- parallelism under the control of the programmer
- semi-explicit
  - explicit in the form of special constructs or operations
- more powerful constructs
  - details are hidden within the implementation of these constructs/operations
- no explicit notion of a parallel process
- denotational semantics remains unchanged, parallelism is only a matter of the implementation

para-functional programming [Hudak 1986]
evaluation strategies [Trinder, Hammond, Loidl, Peyton Jones 1998]
Para-functional Programming

provides **annotations** to express

- the **scheduling** of a computation, i.e. a specific partial ordering on program execution:

  \[ \text{exp sched sexpr} \]

  the evaluation of \text{exp} is determined by the schedule expression \text{sexpr}.

  A **schedule expression** composes events by sequential (\(\cdot\)) and parallel (\(|\)) composition.

  **Events** are e.g. Dlab (demand for evaluation of lab) and the start and the end of expression evaluations.

- the **mapping** of a program onto a machine, i.e. on which processor a particular part of the program is to be executed:

  \[ \text{exp on pid} \]

  exp is to be evaluated on the processor \text{pid}.

  Processor identifiers are integers which can be manipulated by conventional arithmetic operations:

  \[
  \begin{align*}
  \text{left pid} &= 2 \times \text{pid} \\
  \text{right pid} &= 2 \times \text{pid} + 1
  \end{align*}
  \]

  model neighbor connections in an infinite binary tree.
Example for Para-functional Programming

binomial coefficients:

```haskell
binom :: Int -> Int -> Int
binom n k | k == 0 && n >= 0 = 1
| n < k && n >= 0 = 0
| n >= k && k >= 0 = b1 + b2
| otherwise = error "negative params"
where
  b1 = binom (n-1) k
  b2 = binom (n-1) (k-1)
```

- clear separation of
  - the functional specification of what has to be computed and
  - the annotations to control the dynamic behaviour and the mapping
- but extremely fine control which may lead to cumbersome programs
Evaluation Strategies

- high-level control of dynamic behavior, i.e. the evaluation degree of an expression and parallelism
- defined on top of parallel combinators par and seq

Evaluation strategy: function taking as argument the value to be computed. It is executed purely for effect. Its result is simply ():

\[
\text{type Strategy } a = a \rightarrow ()
\]

The using function allows strategies to be attached to functions:

\[
\text{using } :: a \rightarrow \text{Strategy } a \rightarrow a
\]

\[
x \ `\text{using} ` s = (s x) \ `\text{seq}` x
\]

- clear separation of the algorithm specified by a functional program and the specification of its dynamic behavior
Example for Evaluation Strategies

binomial coefficients:

\[
\binom{n}{k} = \begin{cases} 
1 & \text{if } k = 0 \text{ and } n \geq 0 \\
0 & \text{if } n < k \text{ and } n \geq 0 \\
\binom{n-1}{k} + \binom{n-1}{k-1} & \text{if } n \geq k \text{ and } k \geq 0 \\
\text{error "negative params"} & \text{otherwise}
\end{cases}
\]

where

\[
\begin{align*}
b_1 &= \binom{n-1}{k} \\
b_2 &= \binom{n-1}{k-1} \\
\text{strat} &= b_2 \text{ `par` } b_1 \text{ `seq` } ()
\end{align*}
\]
Evaluation Degrees

- **Strategies which specify the degree of evaluation**
  - no reduction: \( r_0 :: \text{Strategy}\ a \text{ with } r_0 \_ = () \)
  - reduce to weak head normal form:
    \( r\text{whn}\_ :: \text{Strategy}\ a \text{ with } r\text{whn}\_ x = x \ `\text{seq}` () \)
  - reduce to full normal form:
    
    \[
    \text{class NFDData}\ a\ \text{where}\\
    \text{rnf} :: \text{Strategy}\ a\\
    \text{rnf} = \text{rwhn}\_ \quad -- \text{default definition}
    \]

- **Instance Declarations provide special definitions for data structures:**

  \[
  \text{instance NFDData}\ a \Rightarrow [a]\ \text{where}\\
  \text{rnf}\ [\ ] = ()\\
  \text{rnf}\ (x;xs) = \text{rnf}\ x \ `\text{seq}` \text{rnf}\ xs
  \]

  \[
  \text{instance (NFDData}\ a,\ NFDData\ b) \Rightarrow (a,b)\ \text{where}\\
  \text{rnf}\ (a,b) = \text{rnf}\ a \ `\text{seq}` \text{rnf}\ b \ `\text{seq}` ()
  \]
Composing Strategies

Strategies are normal higher-order functions:

- can be passed as parameters
- composed with other strategies (using function composition etc.)

seqList is a strategy on lists that is parameterised by a strategy on list elements

```haskell
seqList :: Strategy a -> Strategy [a]
seqList strat [] = ( )
seqList strat (x:xs) = strat x `seq` (seqList strat xs)
```

e.g. seqList r0 evaluate spine of list
     seqList rwhnf evaluate every element to WHNF
Data-Oriented Parallelism

parList :: Strategy a -> Strategy [a]

parList strat [] = ()
parList strat (x:xs) = strat x `par` (parList strat xs)

E.g. parList rwhnf evaluate each $x_i$ in parallel
Parallel Map

Algorithm: map
Behaviour: parList strat

\[ \text{parMap} :: \text{Strategy } b \to (a \to b) \to [a] \to [b] \]
\[ \text{parMap} \; \text{strat} \; f \; \text{xs} \; = \; \text{map} \; f \; \text{xs} \; \text{`using`} \; \text{parList} \; \text{strat} \]
Process-control and Coordination Languages

- Higher-order functions and laziness are powerful abstraction mechanisms which can also be exploited for parallelism:
  - lazy lists can be used to model communication streams
  - higher-order functions can be used to define general process structures or skeletons
- Dynamically evolving process networks can simply be described in a functional framework [Kahn, MacQueen 1977]

```
let outp2 = p2 inp
(outp3, out) = p3 outp1 outp2
outp1 = p1 outp3
```
Caliban

- declarative annotations scheme to provide explicit control over coordination: process placements, communications, and resource allocation
  - partitioning expressed declaratively
  - full power of functional language available to specify parallelism
  - operators used in applications can be reused for coordination
  - coordination separated from configuration

- static functional process networks: can be configured at compile-time

- Description of tasks graphs

  ![Diagram]

  Bundle [a,b] And Bundle [c,d,e] And Bundle [f] And (Arc a d) And (Arc a f)

  a, b .... f are list expressions
Binomial Coefficients in Caliban

\[ \text{binom} :: \text{Int} \to \text{Int} \to \text{Int} \]

as before

\[ \text{parbinom} :: \text{Int} \to \text{Int} \to \text{Int} \]

\[ \text{parbinom} \ n \ k | n \geq k \land k \geq 0 \ = \ \text{head } \text{parsum} \]
\[ \text{parbinom} \ n \ k | \text{otherwise} \ = \ \text{binom } n \ k \]

moreover Bundle \([\text{parsum}, \text{b1}]\) And Bundle \([\text{b2}]\)

And (Arc \text{parsum} \text{b2})

where \text{parsum} = [\text{head } \text{b1} + \text{head } \text{b2}]

\text{b1} = [\text{binom } (n-1) \ k]

\text{b2} = [\text{binom } (n-1) \ (k-1)]
Another Example: Ray Tracing

\[
\text{rayTrace} :: \text{CamPos} \rightarrow \text{[Object]} \rightarrow \text{[Impact]}
\]

\[
\text{rayTrace} \ \text{viewpoint} \ \text{scene} = \text{map} \ \text{impact} \ \text{rays}
\]

where

\[
\text{rays} = \text{generateRays} \ \text{viewpoint}
\]

\[
\text{impact ray} = \text{fold earlier impacts}
\]

where

\[
\text{impacts} = \text{map} \ (\text{hit ray}) \ \text{scene}
\]

parallelize
Processor Farm in Caliban

farmed = unpertition slaves

slave0 =
map impact (partition 3 (generateRays viewpoint))!0

slave1 =
map impact (partition 3 (generateRays viewpoint))!1

slave2 =
map impact (partition 3 (generateRays viewpoint))!2
Processor Farm in Caliban

network forming operators, e.g. a fan of arcs:

- fan :: Stream -> [Stream] -> Placement
  - fan s [] = NoPlace -- null assertion
  - fan s (a:as) = (Bundle [a]) And (Arc a s) And (fan s as)

a farm skeleton:

- farm :: Int -> (a -> a) -> [a] -> [a]
  - farm n func input = farmed moreover (fan farmed slaves)
  - where
    - farmed = unpartition slaves
    - slaves = map (map func) (partition n input)

must be fixed at compile time
Ray Tracing

Ray Tracing

**Ray Trace** :: CamPos -> [Object] -> [Impact]

rayTrace viewpoint scene = farm N impact rays

where rays = generateRays viewpoint

impact ray = fold earlier impacts

where

impacts = map (hit ray) scene
Eden

Parallel programming at a high level of abstraction

- inherent parallelism
- automatic parallelisation or annotations
- functional language (e.g. Haskell)
  => concise programs
  => high programming efficiency

=> concise programs
=> high programming efficiency
Eden

Parallel programming at a high level of abstraction

- parallelism control
  - explicit processes
  - implicit communication
  - ...

+ functional language (e.g. Haskell)
  => concise programs
  => high programming efficiency

Eden = parallel functional programming language
Eden = Haskell + Coordination

- computation language: Haskell
- coordination language:
  - process abstraction
    
    ```
    process :: (Trans a, Trans b) => (a -> b) -> Process a b
    pabs = process (i₁,...,iₙ) -> (o₁,...,oₘ)
    where eqn₁ ... eqnₖ
    ```
  - process instantiation
    
    ```
    ( # ) :: (Trans a, Trans b) => Process a b -> a -> b
    pabs # (inp₁,...,inpₙ)
    ```
Eden = Haskell + Coordination

- computation language: Haskell
- coordination language:
  - process abstraction
  ```
  process :: (Trans a, Trans b) => (a -> b) -> Process a b
  pabs = process (i1,...,in) -> (o1,...,om)
  where eqn1 ... eqnk
  ```
  - process instantiation
  ```
  ( # ) :: (Trans a, Trans b) => Process a b -> a -> b
  pabs # (inp1,...,inp_n)
  ```

process outputs computed by concurrent threads, lists sent as streams
Lazy Evaluation vs. Parallelism

- **Problem:** demand driven evaluation ==> distributed sequentiality

- Eden’s approach:
  - **eager communication:**
    - normal form evaluation of process outputs (by independent threads)
    - push messaging, values are sent as soon as they are available
  - **speculative evaluation of process instantiations in let-exps:**
    - `let outps = pabs # inps in expr[outps]`
      => eager process creation on evaluation of let-expression
    - `expr[pabs # inps]`
      => demand driven process creation
  - **explicit demand control using strategies**
Parallel Map

Haskell Definition:
map :: (a -> b) -> [a] -> [b]
map f xs = [ f x | x <- xs ]

Eden-Version:
parMap :: (Trans a, Trans b) => (a -> b) -> [a] -> [b]
parMap f xs = [ (process f) # x | x <- xs ]

parMap f [ i₁, i₂, ..., iₙ ] => ((process f) # i₁) : parMap f [i₂, ..., iₙ ]

WHNF
no process creation
Parallel Map

Haskell Definition:
map :: (a -> b) -> [a] -> [b]
map f xs = [ f x | x <- xs ]

Eden-Version:
parMap :: (Trans a, Trans b) => (a -> b) -> [a] -> [b]
parMap f xs = [ (process f) # x | x <- xs ] ‘using’ spine

1 process per list element
Parallel Map

Haskell Definition:
map :: (a -> b) -> [a] -> [b]
map f xs = [ f x | x <- xs ]

Eden-Version:
parMap :: (Trans a, Trans b) => (a -> b) -> [a] -> [b]
parMap f xs = [(process f) # x | x <- xs] ‘using’ spine

farm :: (Trans a, Trans b) => (a->b) -> [a] -> [b]
farm f xs = shuffle (parMap (map f) (unshuffle noPe xs))

1 process per processor with static task distribution
Reactive systems

- m:1 communication (nondeterminism)

merge :: Trans a => Process [ [a] ] [a]

- dynamic reply channels

new (ch_name :: ChanName t, chan :: t) expr
creates a new channel ch_name with contents chan

ch_name !* expr1 par expr2
creates new concurrent thread for expr1 with outport ch_name
Master-Worker-Systems

Functionality:
- workers solve tasks and return results
- master distributes tasks to worker processes
- master collects and sorts results, and distributes new tasks to idle workers

dynamic load balancing
Functionality:
- workers solve tasks and return results
- master distributes tasks to worker processes
- master collects and sorts results, and distributes new tasks to idle workers

```
workpool :: (Trans a, Trans b) => Int -> Int -> (a -> b) -> [a] -> [b]
workpool np prefetch f tasks = tag_sort results

where fromWs = zipWith (#) wProcs toWs
wProcs = [process (zip [n,n..].map f) | n <-[1..np]]
toWs = distribute (tag tasks) reqs
(newReqs, results) = (unzip.merge) fromWs
reqs = initReqs ++ newReqs
```
Explicit Message Passing

- In process control languages like Caliban and Eden:
  - explicit processes and communication topology
  - but implicit communication, no send/receive commands
- message passing languages

  explicit processes/programs and communication

  - Scampi [Sérot 1999]: Simple Caml Interface to MPI
    - library that adds MPI features to Objective Caml
    - simple static SPMD model, no dynamic process creation
    - communication by MPI routines
  - Clean [Serrarens 1998]: message passing primitives for distributed applications
    - inter-program communication in a type-safe manner
    - send and receive channels and operations
Global operations on large data structures are done in parallel by performing the individual operations on singleton elements simultaneously.

The parallelism is determined by the organisation of data structures rather than the organisation of processes.

\[ ys = \text{map} \ (2 \times) \ xs \]

→ explicit control of parallelism, inherently parallel operations
→ naturally scaling with the problem size
Data-parallel Languages

- **main application area:** scientific computing
- **requirements:** efficient matrix and vector operations
  - distributed arrays
  - parallel transformation and reduction operations
- **implicit parallelism:** the compiler extracts the parallelism depending on the target architecture. The programmer just chooses the constructs for the specification of the algorithm.
  - **SISAL (Streams and Iterations in a Single Assignment Language):** applicative-order evaluation, forall-expressions, stream-/pipeline parallelism, function parallelism
  - **Id, pH (parallel Haskell):** concurrent evaluation, I- and M-structures (write-once and updatable storage locations), expression, loop and function parallelism.
  - **SAC (Single Assignment C):** With-loops (dimension-invariant form of array comprehensions)
Finite Sequences

- simplest parallel data structure
- vector, array, list distributed across processors of a distributed-memory multiprocessor
  A finite sequence $xs$ of length $k$ is written as
  $$[x_0, x_1, \ldots, x_{k-1}]$$
  For simplicity, we assume that $k = N$, where $N$ is the number of processor elements. The element $x_i$ is placed in the memory of processor $P_i$.
- Lists can be used to represent finite sequences.
  - must have finite length,
  - do not allow sharing of sublists, and
  - will be computed strictly.
Data Parallel Combinators

- **Higher-order functions** are good at expressing data parallel operations:
  - flexible and general, may be user-defined
  - normal reasoning tools applicable, special methods are not required

- **Sequence transformation**:

  \[
  \text{map} \quad :: \quad (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
  \text{map } f \ [\ ] \quad = \quad [\ ] \\
  \text{map } f \ (x:xs) \quad = \quad (f x) : \text{map } f \ x : \text{map } f \ xs
  \]

  \[
  \begin{align*}
  \text{map } f \ x : \text{map } f \ x \quad &\quad [\ ] \\
  f \quad f \quad f \quad f \quad f \quad &\quad [\ ] \\
  \end{align*}
  \]

  only seen as specification of the semantics, not as an implementation
Communication Combinators

Nearest Neighbour Network

- unidirectional communication:

  \[
  \text{shiftr} :: a \rightarrow [a] \rightarrow ([a], a) \\
  \text{shiftr} \ a \ [ ] = ([ ], a) \\
  \text{shiftr} \ a \ (x:xs) = (a:xs', x') \\
  \]

  where \((xs', x') = \text{shiftr} \ x \ xs\)

\[ \begin{array}{c}
  a \\
  \Rightarrow x \\
  \Rightarrow \square \\
  \Rightarrow \square \\
  \Rightarrow \square \\
  \Rightarrow \square \\
  \Rightarrow \ldots \\
  \Rightarrow \square \\
\end{array} \]
Communication Combinators

Nearest Neighbour Network

- **unidirectional communication:**
  
  \[ \text{shiftr} :: a \to [a] \to ([a], a) \]
  
  \[ \text{shiftr} a \quad [] = ([], a) \]
  
  \[ \text{shiftr} a \quad (x:x) = (a:x’, x’) \]
  
  where \((x’, x’) = \text{shiftr} x \quad x)\]

- **bidirectional communication:**
  
  \[ \text{shift} :: a \to b \to [(a, b)] \to (a, b, [(a, b)]) \]
  
  \[ \text{shift} a \quad b \quad [] = (a, b, []) \]
  
  \[ \text{shift} a \quad b \quad (x : x) = (a’, x b, (a, b’):x) \]
  
  where \((a’, b’, x’) = \text{shift} x a \quad b)\]

![Diagram of communication combinator network](image)
Example: The Heat Equation

Numerical Solution of the one-dimensional heat equation

\[
\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}, \text{ for } x \in (0,1) \text{ and } t > 0
\]

The continuous interval is represented as a linear sequence of \( n \) discrete gridpoints \( u_i \), for \( 1 \leq i \leq n \), and the solution proceeds in discrete timesteps:

\[
u_i' = u_i + \frac{k}{h^2} [u_{i-1} - 2u_i + u_{i+1}]
\]
Example: The Heat Equation

The following function computes the vector at the next timestep:

```haskell
step :: Float -> Float -> [Float] -> [Float]
step u0 u_{n+1} us = map g (zip us zs)
where
  g (x, (a,b)) = (k / h*h) * (a - 2*x + b)
  (a’,b’,zs) = shift u0 u_{n+1} (map (\ u -> (u,u)) us)
```

```
  u_i' = u_i + k/h^2 [u_{i-1} -2u_i + u_{i+1}]
```
Reduction Combinators

- combine **computation with communication**

- folding:

  \[
  \text{foldl} :: (a -> b -> a) -> a -> [b] -> a \\
  \text{foldl } f a \; [ ] \; = \; a \\
  \text{foldl } f a \; (x:x) \; = \; \text{foldl } f \; (f \; a \; x) \; x
  \]

  \[
  a \xrightarrow{\oplus} x_0 \xrightarrow{\oplus} x_1 \xrightarrow{\oplus} x_2 \xrightarrow{\oplus} \cdots \xrightarrow{\oplus} x_{n-1} \xrightarrow{\oplus} \text{foldl } \oplus \; a \; x \\
  y_0 \xrightarrow{\oplus} y_1 \xrightarrow{\oplus} y_2 \xrightarrow{\oplus} \cdots \xrightarrow{\oplus} y_{n-1} \xrightarrow{\oplus} \text{ys} = \text{scanl } \oplus \; a \; x
  \]

- scanning:

  \[
  \text{scanl} :: (a -> b -> a) -> a -> [b] -> [a] \\
  \text{scanl } f \; a \; x \; = \; [\text{foldl } f \; a \; (\text{take } i \; x) \mid i \leftarrow [0..\text{length } x]-1]\]
Bidirectional Map-Scan

\[
\text{mscan} :: (\text{a} \rightarrow \text{b} \rightarrow \text{c} \rightarrow (\text{a}, \text{b}, \text{d})) \rightarrow \text{a} \rightarrow \text{b} \rightarrow [\text{c}] \rightarrow (\text{a}, \text{b}, [\text{d}])
\]

\[
\text{mscan} \ f \ \text{a} \ \text{b} \ [\ ] = (\text{a}, \ \text{b}, \ [\ ])
\]

\[
\text{mscan} \ f \ \text{a} \ \text{b} \ (\text{x}:\text{xs}) = (\text{a}''', \ \text{b}''', \ \text{x}' : \text{xs}')
\]

where \((\text{a}''', \ \text{b}''', \ \text{xs}') = \text{mscan} \ f \ \text{a}' \ \text{b} \ \text{xs}\)

\((\text{a}', \ \text{b}''', \ \text{x}') = f \ \text{a} \ \text{b}' \ \text{x} \)
Example: Maximum Segment Sum

- **Problem:** Take a list of numbers, and find the largest possible sum over any segment of contiguous numbers within the list.

Example: \([-500, 3, 4, 5, 6, -9, -8, 10, 20, 30, -9, 1, 2]\)

- **Solution:** For each \(i\), where \(0 \leq i < n\), let \(p_i\) be the maximum segment sum which is constrained to contain \(x_i\), and let \(p_s\) be the list of all the \(p_i\).

\[\Rightarrow \text{the maximum segment sum for the entire list is just}\]
\[\text{fold max } p_s.\]

How can be compute the maximum segment sum which is constrained to contain \(x_i\)?
Example: Maximum Segment Sum

**Problem**: Take a list of numbers, and find the largest possible sum over any segment of contiguous numbers within the list.

Example: 

\[-500, 3, 4, 5, 6, -9, -8, 10, 20, 30, -9, 1, 2\]

```haskell
mss :: [Int] -> (Int, [Int])
mss xs = (fold max ps, ps)
where
  (a', b', ps) = mscan g 0 0 xs
  g a b x = (max 0 (a+x), max 0 (b+x), a + b + x)
```

**mss** returns the list of maximum segment sums for each element as well as the overall result.

mss [-500, 1, 2, 3, -500, 4, 5, 6, -500] => (15, [-494, 6, 6, 6, -479, 15, 15, 15, -485])
Skeletons

A number of patterns recur frequently in the body of parallel algorithms. These patterns are composed of

- computations and
- the interactions between them.

The patterns can be conceptually abstracted from the details of the activities they control.

This leads to (algorithmic) skeletons which are

- higher-order functions
- with an associated parallel evaluation strategy and
- a cost (performance) model to estimate the execution time

Campbell’s classification:

- recursively partitioned (divide and conquer): tree-like structured
- task queue (farm, master worker): jobs solved independently
- systolic (pipeline): stages with data flowing
Divide and Conquer

**higher order function:**

\[
d&c :: (a \to \text{Bool}) \to (a \to b) \to (a \to [a]) \to ([b] \to b) \to a \to b
\]

\[
d&c \text{ trivial solve divide conquer } p
\]

\[
= \text{ if (trivial p) then solve p }
\]

\[
\text{ else conquer (map}
\]

\[
(d&c \text{ trivial solve divide conquer) (divide p))}
\]

**parallel implementation:**

- idealised implementation on tree of processors
- Implementation of binary d&c on a grid:
Task Queue - Farm - Master/Worker

higher order function:

\[
\text{farm} :: (a -> b -> c) -> a -> [b] -> [c]
\]

\[
\text{farm f env} = \text{map (f env)}
\]

implementation:

– static task distribution:

– dynamic task distribution:
Systolic Scheme - Pipelining

higher order function:

\[
\text{pipe} :: [ [a] -> [a] ] \rightarrow [a] -> [a]
\]

\[
\text{pipe} = \text{foldr} \ (\cdot) \ \text{id}
\]

implementation:

linear pipeline of processes

\[
\begin{array}{c}
\rightarrow f_1 \rightarrow f_2 \rightarrow f_3 \rightarrow \cdots \rightarrow f_p \\
\end{array}
\]
Skeletal Programming

- fixed number of higher-order skeletons *(algorithmic skeletons)*
- different highly optimized implementations for different target architectures *(architectural skeletons)*

**Programming Methodology:**
- choose suitable skeletons
- compose them in a program
- estimate its expected performance => cost model
- make changes in design, if necessary => transformation rules

**most well-known systems:**
- P³L Pisa Parallel Programming Language
  based on imperative computation language
- SCL Structured Coordination Language
  based on functional computation language
SCL - Structured Coordination Language

layered skeletal approach:

- skeletons operate upon explicitly distributed arrays ParArray that are constructed by partitioning conventional sequential arrays:
  - partition :: Partition_pattern \rightarrow SeqArray \rightarrow ParArray
    \rightarrow ParArray (SeqArray index a)
  - gather :: Partition_pattern \rightarrow ParArray index (SeqArray index a)
    \rightarrow SeqArray index a
  - align :: ParArray index a \rightarrow ParArray index b \rightarrow ParArray index (a,b)

- higher level skeletons:
  - elementary skeletons (data parallel operations over distributed arrays)
  - computational skeletons (parallel control flows)
  - communication skeletons

program's parallel behavior, including data partitioning, placement, movement

skeletons cannot be called from sequential code
SCL Skeletons

- **elementary**
  - `map :: (a -> b) -> ParArray index a -> ParArray index b`
  - `imap :: (index -> a -> b) -> ParArray index a -> ParArray index b`
  - `fold :: (a -> a -> a) -> ParArray index a -> a`
  - `scan :: (a -> a -> a) -> ParArray index a -> ParArray index a`

- **computational**
  - `farm :: (a -> b -> c) -> a -> ParArray index b -> ParArray index c`
    
    \[
    farm f e = map (f e)
    \]

  - `spmd :: [ (ParArray index a -> ParArray index a, index -> a -> a) ] -> ParArray index a -> ParArray index a`
    
    \[
    spmd [ ] = id
    spmd ((gf, lf) : fs) = (spmd fs) . gf . (imap lf)
    \]

- **communication**
  - `rotate :: Int -> ParArray index a -> ParArray index a`
Homomorphisms

- Functions are defined on non-empty finite lists, with list concatenation ++ as constructor.
- Function h on lists is a homomorphism iff there exists a binary operator \( \otimes \) such that, for all lists \( xs \) and \( ys \):
  \[
h (xs ++ ys) = h xs \otimes h ys
  \]
- \( \otimes \) is necessarily associative, because ++ is associative.

Examples:
- Mapping:  \( \text{map } f \quad [x_1, x_2, \ldots, x_n] = [f x_1, f x_2, \ldots, f x_n] \)
- Reduction: \( \text{red } (\oplus) \quad [x_1, x_2, \ldots, x_n] = x_1 \oplus x_2 \ldots \oplus x_n \)
- Scanning: \( \text{scan } (\oplus) \quad [x_1, x_2, \ldots, x_n] = [x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \ldots \oplus x_n] \)

  with associative operator \( \oplus \)
Scan-Reduce Composition

An **almost-homomorphism** is a function that becomes a homomorphism when tupled with one or more auxiliary functions.

Let $\text{scanred} (\otimes, \oplus) := \text{red} (\oplus) \circ \text{scan} (\otimes)$.

Assume that $\otimes$ distributes over $\oplus$: $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$.

When tupled with function red, $\text{scanred}$ becomes the homomorphism $\text{scanred}'$:

$$\text{scanred'} (\otimes, \oplus) x := (\text{scanred} (\otimes, \oplus) x, \text{red} (\otimes) x)$$

For arbitrary binary, associative operators $\otimes$ and $\oplus$, such that $\otimes$ distributes over $\oplus$:

$$\text{red} (\oplus) \circ \text{scan} (\otimes) = \pi_1 \circ \text{red} (<\oplus, \otimes>) \circ \text{map pair}$$

where $\text{pair } x := (x, x)$

and $(s1,r1) <\oplus, \otimes> (s2,r2) := (s1 \oplus (r1 \otimes s2), r1 \otimes r2)$.
For arbitrary binary, associative operators \( \otimes \) and \( \oplus \), where \( \otimes \) distributes over \( \oplus \),

\[
\text{scan (} \oplus \text{) } \circ \text{ scan (} \otimes \text{) } = \text{ map } \pi_1 \circ \text{ scan (} \langle \oplus, \otimes \rangle \text{) } \circ \text{ map pair}
\]

This result can be derived using the auxiliary function

\[
\text{inits } [x_1, x_2, \ldots, x_n] = [ [x_1], [x_1, x_2], \ldots, [x_1, x_2, \ldots, x_n] ]
\]

and the following identities

\[
\text{map (} f \circ g \text{) } = \text{ map } f \circ \text{ map } g
\]

\[
\text{scan (} \otimes \text{) } = \text{ map (red (} \otimes \text{)) } \circ \text{ inits}
\]

\[
\text{inits } \circ \text{ map } f = \text{ map (} \text{map } f \circ \text{) } \circ \text{ inits}
\]

\[
\text{inits } \circ \text{ scan (} \otimes \text{) } = \text{ map (} \text{scan (} \otimes \text{)) } \circ \text{ inits}
\]
Transformation of Skeletal Programs

- Problem: composition of skeletons
  - try to predict impact on performance -> cost model required
  - develop transformation rules

- Questions:
  - Expressiveness: How much ground does the class of (almost-)homomorphisms cover?
  - Implementation: How can homomorphic skeletons be implemented efficiently on parallel computers?
  - Composition: Are certain compositions of standard homomorphisms good candidates for new homomorphic skeletons? How can these be optimized further?
  - Decomposition: Can a more complex homomorphism be decomposed into simpler homomorphisms, with the result of improved performance?
Large Scale Applications

- **Numerical applications**: even exceed the performance by fastest Fortran compilers!
  - SISAL: Australian weather prediction model, Fourier transforms
  - Id: hydrodynamics (*Simple*)
  - SCLSCL: Gauss Jordan solver, Conjugate Gradient solver

- **Symbolic applications**:
  - Id, ParLisp: Boyer-Moore theorem prover
  - SkelML, Concurrent Clean: Ray tracer
  - GpH: Natural language processor (*Lolita*)
  - Concurrent Haskell: Formal development environment (*UniForM Concurrency Toolkit*)

- **Data intensive applications**:
  - Erlang (Ericsson): Distributed database management system (*Mnesia*), Mobility server
  - GpH: Transaction-processing, Complex database queries
Present...

- **Programming model:** find the right level of abstraction for both computation and coordination
- **Semantics, analysis and transformation:** strictness, granularity, types and effects, cost analysis,…
- **Memory consumption:** efficient dynamic allocation; use of *impure* features
- **Foreign language interfacing:** components from legacy code, shared libraries, better written in some other paradigm
- **Architecture independence:** systems are often portable to different classes of architecture (shared-memory, distributed-memory multiprocessors, networks of workstations)
… and Future

- **Metrics for parallelism + Benchmarks**: standard set of program and parallel platform characteristics / performance metrics (no restricted to parallel functional programs).
- **Programming environments**: improved profiling and visualisation tools, proof systems, program transformations,…
- **New applications areas**: safety-critical systems, mobility, multi-media programming, distributed operating systems,…
- **Others**: resource-certified software, formal semantics…