Parallel Functional Programming Languages: a guided tour

Françoise Baude, INRIA-I3S OASIS team

MCF Univ. Nice Sophia-Antipolis

Master STIC, RSD & PMLT

Slide 1

1 Motivation

A common point of view for studying and comparing various parallel programming languages is to consider that

a good parallel programming languages (in whatever style of programming) succeeds in

filling the gap between

- the level at which the programmer would like to reason about its program
- and the level at which one must program the target machine in order to yield efficient performances

Remark: this fact is also true regarding sequential programming languages !

Slide 2

There have been several tentatives in many different programming language styles (imperative, functional, logic, object-oriented, ...). They are more or less satisfactory:

- either they provide a too much abstract view of the computation → not efficient enough at runtime
- or they provide a too much concrete view of the computation → too close to the target machine and as such, not portable.

→ Everybody is seeking for a compromise in between !

Target machines are MIMD architectures: widespread use (i.e. parallel **and** distributed computing) & the most scalable.
Points to consider are:

- How to break the work into tasks among the processors
- Mapping and load balancing of data and tasks

Slide 3

- How to manage comms. & synchros. between tasks

People of the functional programming community claim that the functional programming style gives **much more** opportunity than other styles in finding a good compromise

→ The following sections will present their arguments

Slide 4

Plan of the course

1. Why a functional programming style is a good candidate for parallel programming
2. How to automatically extract and exploit parallelism (at compile and run time)
3. Why and how the programmer may help in the exploitation of parallelism
   (a) Using new syntaxic constructs explicitely
   (b) Using only available[a] program skeletons
4. What about distributed functional computing

[a] the ones that yield an efficient implementation !

Slide 5

2 Why a functional programming style is a good candidate for parallel programming

Properties of Functional langages

- They are *naturally* parallel = *implicit* parallelism
- Unnecessary dependencies (i.e. as in imperative languages) are eliminated: only required dependencies from (1) function composition, (2) the fact that arguments required by a function must be evaluated before the function itself.
- Amount of effective parallelism depends on the implicit control governing the evaluation process (strict or non-strict : demand or data driven)

Slide 6

Slides 9–26 from Guy Tremblay, UQAM

Slide 7

As a conclusion of this section

- it is embarassingly easy to partition a functional program into (many) fine-grained parallel tasks
- one can foresee to be very difficult to exploit parallelism efficiently, as it is required to *minimize the overhead of fine-grained parallelism*.

Slide 8

3 How to automatically extract and exploit parallelism (at compile and run time) i.e., somehow without any language or program restrictions

3.1 Principles

After the compilation process, the underlying computation is considered to be a *reduction graph*.
Each step of the reduction is to evaluate a node of this graph: a *redex* (a step of the reduction process).

As evaluating this graph in any possible way yields the same result (the computation is deterministic) there is opportunity to execute all (or a subset of all) redexes in parallel.

Slide 9

3.2 The problems

- In a non-strict language, not all redexes are worth executing. Which ones must be executed ? The ones that are effectively needed → this can only be known at compile time by a strictness analysis (difficult because higher-order functions, polymorphism,...).
- A redex might be a (very !) fine-grained operation.

→ it may be inefficient to create and schedule a parallel task.

1. This depends of the *expected* granularity of the redex.
   → a – complex – granularity/cost analysis would be required at compile time. This analysis may be coupled at runtime with a dynamic decision depending on the overall runtime load (e.g. see below)
2. create a *potential* task and execute it in a lenient or in a lazy

Slide 10

way ("lazy future").

Slide 11

3.3 Example of Implementation of Graph Reduction in Parallel

For instance in GRIP (Graph Reduction In Parallel: compiling Haskell for the MIMD local & shared-memory machine called GRIP)

Parallel task creation Create a *spark* containing the redex to evaluate, and add it to the *local* spark pool, even if the spark might be latter exported to the *global* spark pool (-> potential dynamic mapping of spark evaluation onto an idle remote CPU).

Slide 12

Parallel task synchronisation Assuming the sparked node is needed by the parent thread, one of the 3 situations may arise when its value is demanded :

- It has been evaluated by another thread. In this case the parent thread demands the value of the node and continues execution.
- It has not yet been evaluated. In this case, the parent thread simply evaluates the node *without creating and scheduling a new thread*. It is discarded later by the spark scheduler.
- It is currently being evaluated by another thread. In this case, the parent thread must block until the child thread has finished evaluation.

Slide 13

Early experiments (e.g. the Alice machine) of purely implicit approaches for reducing in parallel a graph representing a functional program proved to be disapointing.

For this reason, in the GRIP experiment,

- the Haskell language had to be (just slightly !) modified with 2 new keywords: *par, seq* (GpH, i.e. Glasgow Parallel Haskell [9]) so as to indicate *potential parallelism*.

  E.g.:

```
parmap :: (a -> b) -> [a] -> [b]
parmap f [] = []
parmap f (x:xs) = `par` fx (`seq` fxs (fx : fxs))
      where fx = f x
            fxs = parmap f xs
```

or, in order to spark nodes in parallel (assuming that traversing the

Slide 14

list sequentially is a time consuming operation):

```
parmap f l = parmap' l
   where parmap' [] = []
         parmap' (x:xs) = fx `par` fxs `par` (fx : fxs)
            where fx = f x
                  fxs = parmap' xs
```

- GpH is a semi-implicit parallel extension of Haskell realising a thread-based approach to parallelism: it allows threads to be created, but do not provide mechanisms to control those threads. They are thus managed entirely under runtime-system control.
- (Gp)Haskell programs compile, by transformation, into code for the abstract G-machine (plus the GUM [11] runtime system calls regarding par/seq constructs). The abstract G-machine model [S.Peyton Jones 1992] is a stack based machine. G-machine code

Slide 15

is then compiled for the GRIP machine or into C.

- GUM (Graph reduction for a Unified Machine model) also targets NOWs [11]: using C+PVM (or C+MPI), each workstation runs a PVM process hosting several GUM threads and a local spark pool. Threads increase their granularity by executing subsequent spark nodes if available. Only when all threads are blocked, some local spark node may be turned into a new thread. If they are no local sparks, then a spark node is seek by *fishing* through the other processes which yields to the communication of the node plus some neighbour nodes in the reduction graph and the update of the spark node location (for subsequent comm & synchro.): work stealing load-balancing. Performances are quite good, even on irregular parallel problems: 18 on a 32 CPU Beowulf cluster [10]
- and more recently, on Grids: Grid-GUM [14]: a slight

Slide 16

improvement on the way to fish, trying to take into account heterogeneity of the grid machines: use stamps on each message, indicating the current number of sparks on this node, the static information regarding CPU power and communication latency towards each node in the Grid. Each node thus maitain tables, and uses them to fish: fish towards the "best" nodes

Slide 17

4 Why and how the programmer may help in the exploitation of parallelism

Motivation, i.e. "why" Obvious, given the disapointing early experiments of parallel graph reduction ! But, as mentioned in slide 14, delimiting where are the parallel tasks (delimiting task) is just the answer of the question *What to execute in parallel*.

It remains to know *Where* to map those tasks onto the parallel and distributed architecture, and also *When* to execute those tasks.

In fact, research around runtime systems for solving the *where* and *when* questions are still done. Remark: comm. & synchro. between tasks should still be transparent to the programmer.

In the meanwhile, people have investigated other ways to efficiently exploit parallism in functional languages.

Slide 18

Good surveys: [4, 12, 1]

4.1 Using new syntaxic constructs explicitely

4.1.1 Lighter implication of the programmer: semi-explicit approaches

Annotate the program such as the compiler and runtime system know where lies a good potential of parallelism.

- Refer to GpH (slide 14)
- MultiLisp (concurrent and strict functional programming using Future [6]).

Example of the sorted insertion of an element in a binary tree written in Multilisp:

```
def insert (elt, tree)
```

Slide 19

```
if (empty-tree? tree) elt
(if (leaf? tree)
   (if (< tree elt)
     (make-node tree tree elt)
     (make-node elt elt tree))
   (if (< (discriminant tree) elt) // discr.: max value left
     (make-node (left-child tree) (discriminant tree)
         (future (insert elt (right-child tree))))
     (make-node (future (insert elt (left-child tree)))
                (discriminant tree) (right-child tree))))
```

Given a tree T storing elements 3, 5, 17, 19, run the function `(insert 29 (insert 4 T))` and see that the effective insertion of the element 29 can start as soon as the insertion of the element 4 has started and may not be yet finished. Indeed, 4 and 29 will not be stored in the same sub-tree.

Slide 20

→ One side-effect of those approaches is that it may not yield massive parallelism as the programmer may not indicate sufficient amount of parallelism. Historically not well adapted to massively parallel machines.

Slide 21

4.1.2 Tentative solutions to solve the *where* and *when* questions explicitely

- *Para-functional* programming [2]

1. Tasks are delimited using { }
2. mapping is introduced using the $on operator
3. Lazy evaluation by default, and immediate evaluation of an expression if preceded by #
4. the evaluation order within a task is close to the one of concurrent Prolog (i.e. declarative rules)

```
{f1 (x1, x2, ...xk) == exp1 $on proc ;
                    x == exp2 ;
              result exp;
 fn (x1,...     xn) == expj; }
```

Slide 22

Computing the factorial function in parallel onto a binary tree of virtual processes (or CPUs)

```
{result pfac (1,k) $on root ;
 pfac (lo,hi) == if lo=hi then lo
                 else if lo=(hi-1) then lo*hi
         else
                 { result (pfac (lo,mid) $on left($self))
                    *
                   (pfac (mid+1,hi) $on right($self));
                   mid == (lo+hi)/2; } ;
 left(pe) == if 2*pe > n then pe else 2*pe;
 right(pe) == if 2*pe > n then pe else 2*pe+1 ;
 root == 1;
}
```

Slide 23

- Actors (refer also to section 5)

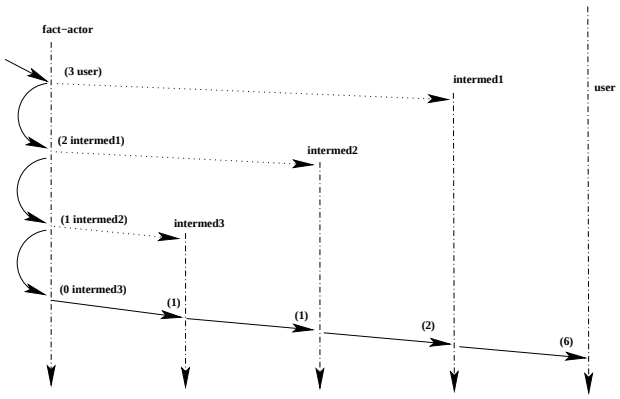
An actor is an entity with a mailbox; its behaviour is to accept a message, execute something, send some new messages and so on. It has been the basis of multi-agent distributed systems (no shared memory assumption) and also, but more seldom, the basis of parallel computations (i.e. as a low-level, implementation language for MIMD architectures).
Definition of the factorial function that may be triggered by sending the n value (and as continuation: user) to an actor running *fact*

```
(define behavior
  (serialized fact()
     (accept (n cont)
            (if (= n 0)
                 (send cont 1)
                 (create ( (intermed intermfunc (n cont)) )
```

Slide 24

```
                          (send self (- n 1) intermed)))) ) )
(define behavior
  (serialized intermfunc (m cont)
  (accept (result)
          (send cont (* m result))) ) )
```

Slide 25



Slide 26

Criticism of explicit approaches Based on

- Explicit task creation by the programmer not by the compiler or runtime system
- Sometimes, explicit mapping of tasks yielding the topology of the inter-task communications.

in other words, *a high-level description of a network of tasks*, which then requires at runtime, to:

- effectively embedd the topology onto the effective distributed architecture
  → static *graph embedding* were popular at the time parallel machines were built around an interconnexion network such as mesh, torus, hypercube, etc.
- or use arbitrary mapping strategies for tasks such as round-robin, random, etc.

Slide 27

4.2 Automatic but restricted extraction of parallelism

4.2.1 Main idea

Limitation to some specific language constructs or program constructs. Only syntaxic constructs or functions that may yield efficient parallelism are considered for parallel task creations.

Targetting parallelism needs massive amount of work to be executed in parallel !, so the focus on large data structures, i.e. *parallel evaluation of every element in a collection* also called **data-parallelism**:

- Collections of interest are arrays, lists, and sometimes also graphs, trees, sets, vectors, etc
- all possible kind of traversals of those collections: *loops* or ones

Slide 28

that require coordination/synchronisation e.g. *reductions*

Exploiting data-parallelism has not been restricted to functional languages: Fortran-90, C*, Nesl (nested data-parallelism).

Most important categories in the functional programming style:

- Implicitly parallel higher-order functions
- Data-flow approaches (Id, Sisal, etc) that introduce arrays and streams (pipeline parallelism)

Slide 29

4.2.2 Implicitly parallel higher-order functions

Every computation can be expressed as the functional composition of higher-order functions such as *Map, Reduce*. Those functions may be expressed using the Bird-Merteens Formalism (BMF)

Cost measures can be added and effectively executed at each step of the refinement of those higher-order functions.

Programs are very concise. (But) clustering of data and operation on those data is usually not exposed (it is hidden in libraries targetted for each different platforms) and thus is outside the programmer's control [3].

Those functions implement basic computations in a parallel way, such as follows.

Slide 30

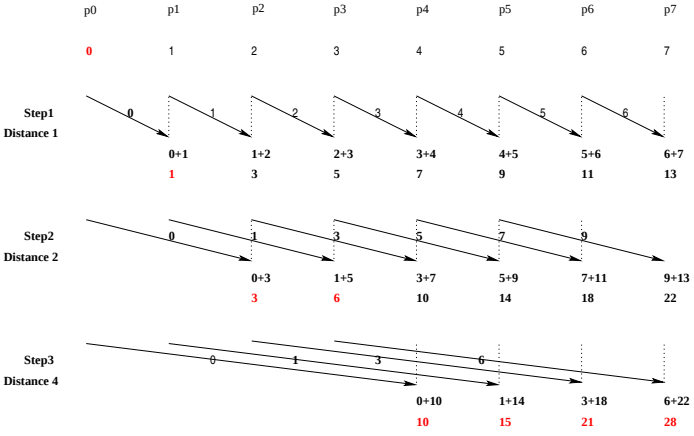
How to unfold a *prefix-sum* into a set of parallel and communicating tasks (given here in the PRAM model, i.e. a shared-memory parallel computation model): We are given n numbers, $x_0, x_1, ..., x_{n-1}$ assuming n is a power of 2. It is required to compute the following sums, called *prefix sums*:
$s_0 = x_0, s_1 = x_0 + x_1, s_2 = x_0 + x_1 + x_2, ..., s_{n-1} = x_0 + x_1 + ... + x_{n-1}$
A sequential algorithm :

$s_0 = x_0$
for i=1 to n-1 do
    $s_i = s_{i-1} + x_i$
endfor

A PRAM parallel algorithm consists of $\log n$ steps. Initially on every P_i, $s_i = x_i$, for $i = 0, 1, ..., n-1$. During the j^{th} step ($j \in [0..\log n - 1]$), the operation $s_i = s_{i-2^j} + s_i$ is performed simultaneously by the processors numbered 2^j to $n-1$. During each

Slide 31

step, two numbers are added whose distances are twice apart the distance in the previous step.

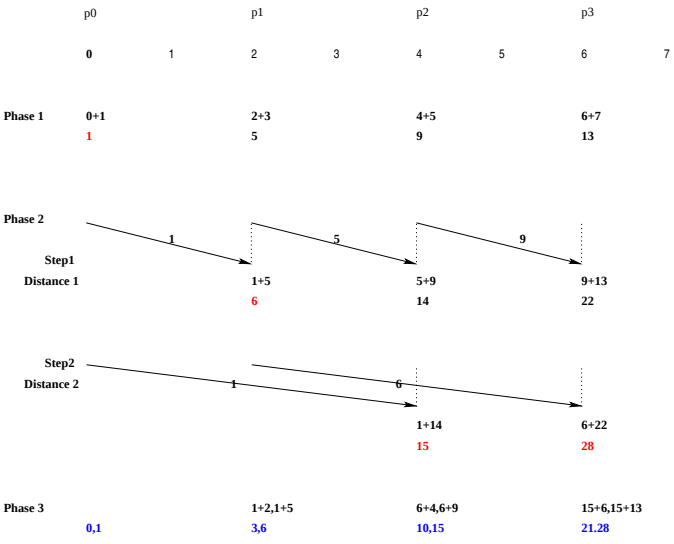


Slide 32

The number of used processors is n. The cost of this algorithm: parallel time ($\log n$) multiplied by number of processors n is $O(n \log n)$ which is not optimal w.r.t. to the sequential cost ($O(n)$).

A cost-optimal solution is to use only i.e. $p = n/\log n$ processors, and map $n/p = \log n = k$ numbers to each. In the first phase, a sequential sum of all the k values is executed on all processors in parallel (in $O(n/p)$ time); in the second phase, the parallel prefix-sum algorithm is executed on those values with a time cost of $O(\log p)$; in the final phase, each processor i in parallel compute the sequential prefix sum given its initial numbers and the prefix sum calculated on its predecessor.

Slide 33

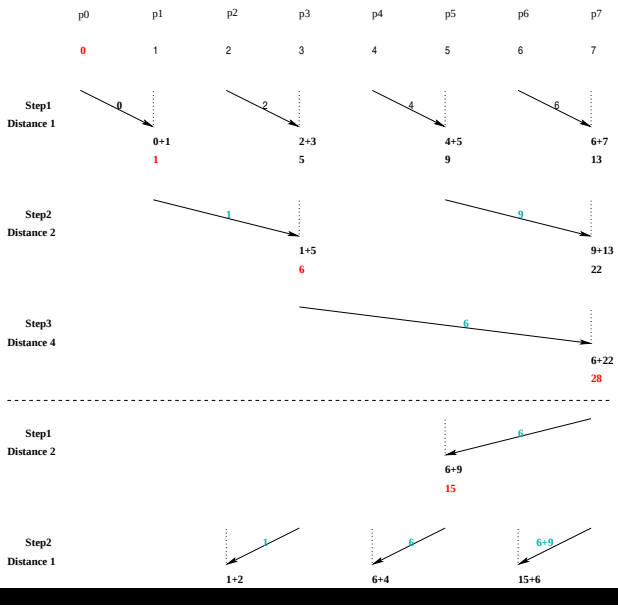


Slide 34

The cost is optimal: parallel time is $2 * \log n + \log(n/\log n) = O(\log n)$ with $n/\log n$ processors, i.e. cost is $O(n)$.

Distributed-memory machines How to execute the parallel prefix sum on an architecture without shared-memory and with only communications between neighbours processors ? Two phases, where each one is a reduction. For instance, if the underlying network has an embedded binary tree:

Slide 35



Slide 36

Examples in BMF [8] Higher-Order functions (operators):

- Building collections (e.g. lists): (1) Empty element: [] (2) Unity element (singleton) : [.] (3) Concatenation of two sub-collections: $\between$ (associative)
- Composition of functions: $\bullet$
- Map ("for all"): $f*$
- Reduction: $g/$
- Accumulate (Prefix Sum) (can be defined using * and /): $h//$

`Example:` inits $= \between // \bullet ([\cdot])*$
/* Compute inits[a,b,c] = [[], [a], [a,b], [a,b,c]] */
1) singleton [a,b,c] * = [[a],[b],[c]]
2.1) data transfer yields: [[],[[],[a]],[[],[a],[b]],[[],[a],[b],[c]]]
2.2) Apply the sum-prefix operator $\between$ to each list member

Slide 37

[[],[a],[a,b],[a,b,c]]

Slide 38

Maximum Segment Sum Example Given a list of ints, return the sub-list's sum which is maximum (obvious if only positive ints)

1. First solution:

$$mss = \uparrow_{max} / \bullet +/* \bullet segs$$
$$segs = \nparallel / \bullet tails* \bullet inits$$
where tails [a,b,c] = [[a,b,c],[b,c],[c]]

```
mss[a,b,c,...,n]
inits => [ [], [a], [a,b], [a,b,c], ..., [a,b,c,...,n] ]
tails* =>[ [[]], [[a]], [[a,b],[b]], [[a,b,c],[b,c],[c]],
..., [[a,b,c,...,n], [b,c,...,n], [c,...,n], ..., [n]] ]
∦/ =>[ [], [a], [a,b], [b], [a,b,c], [b,c], [c],
..., [a,b,c,...,n],[b,c,...,n],[c,...,n],...,[n] ]
+/* => [0, a, a+b, b, a+b+c, b+c, c, ...,
  a+b+c+...+n, b+c+...+n, c+...+n,..., n]
```

Slide 39

$\uparrow_{max}$`/ => the greatest value`

2. Second solution:

$$mss = \uparrow_{max} / \bullet \otimes//$$
$$a \otimes b = (a + b) \uparrow_{max} 0$$

$\otimes//_e[a, b, ..., x] = [e, e \otimes a, (e \otimes a) \otimes b, ..., ((e \otimes a) \otimes ...) \otimes x]$
`mss[6,4,-3,0,-7,5]`
$\otimes//_0[6, 4, -3, 0, -7, 5]$ `=>`
$[0, 0 \otimes 6, (0 \otimes 6) \otimes 4, ((0 \otimes 6) \otimes 4) \otimes -3, (((0 \otimes 6) \otimes 4) \otimes -3) \otimes 0, ((((0 \otimes 6) \otimes 4) \otimes -3) \otimes 0) \otimes -7, (((((0 \otimes 6) \otimes 4) \otimes -3) \otimes 0) \otimes -7) \otimes 5] =$
[0, 6, 10, 7, 7, 0, 5]
$\uparrow_{max} / = 10$

Slide 40

Some concrete languages based on the BMF formalism: PMLS [4], SCL, P3L, ...; usually strict semantics.

Implementation details Thread creation and coordination is transparent to the programmer. Selected higher-order functions serve as skeletons of parallel algorithms. With the help of a cost model (complexity + cost to create a thread holding p tasks, communication cost: L+$\beta \cdot n$ with n the data size and underlying topology), each skeleton is translated into a behaviour that is hacked for the specific architecture, yielding to an abstract parallel process topology, with parameters specifying details of the tasks that are to be performed.

How to efficiently *chain* the various topologies ? Compiler and runtime systems try to do their best and may be helped by the programmer [3]!

Slide 41

For instance how *map f* may be compiled for a distributed architecture, e.g. a Network of Workstations:

- construct a task farm skeleton with pre-loaded f on each PE
- records all workers as free
- repeatedly:
  - sends an unprocessed list element to a free worker and records it as busy;
  - receives a processed list element from a busy worker and records it as free;
- until all list elements have been processed;
- assembles the processed list in the appropriate order.

Alternatively, a pipe line of workers may be created and list elements may flow on it.

Slide 42

4.2.3 Data flow approaches

Belong to applicative programming.

A popular language: SISAL (Streams and Iterations in a Single Assignement Language)

The parallelism is extracted from the fact that data flow on the functions that need to be applied to it.

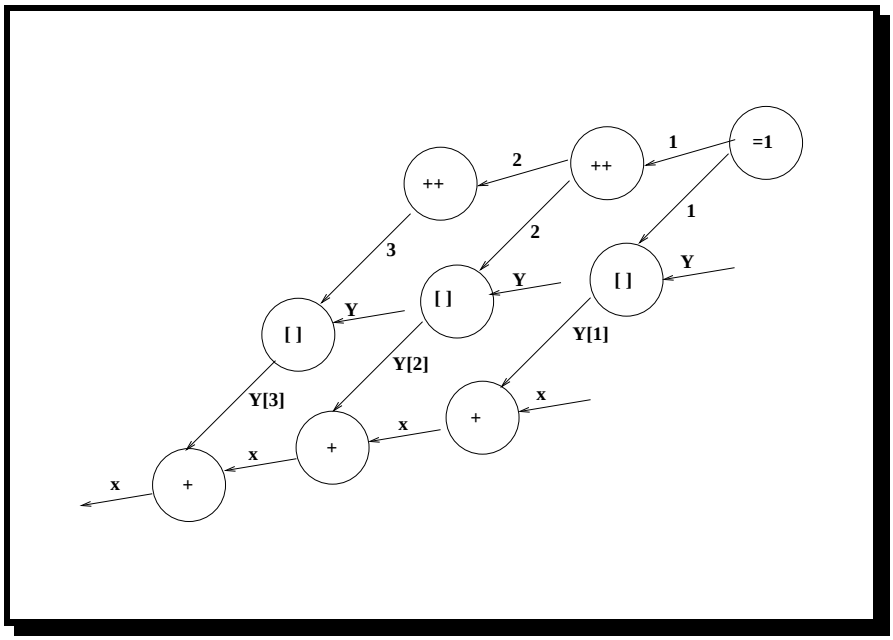
Example:

```
for // it is a ‘‘forall’’ construct in essence !
  i := 1    // Generator
  x := Y[1] // of the for loop
  while i < n repeat  // this will
    i := old i + 1    // compile into
    x := old x + Y[i] // parallel loops
  returns x
```

Slide 43

As the language is a strict single-assignement language (and only from Sisal v2.0 are higher-order functions allowed), it is quite easy to extract the dataflow graph at compile time and to exploit parallelism at runtime. In fact parallelism is exploited from loops on (mono/multi dimensional) arrays and from streams.

Slide 44



Slide 45

5 What about distributed functional computing ?

5.1 Principles

The idea is to exploit the parallelism "de situation" as opposed to the parallelism "de resolution".

Of course, this yields to explicit task creation, and in some sense, also explicit task comm and synchronisation.

build a distributed language based on location-aware creation and execution of remote (mobile) threads executing remote functions on shared or copied data.

A few examples in conventional functional languages are as Distributed Haskell [12], or ConcurrentClean

Slide 46

(`http://www.cs.kun.nl/~clean`) [7].

Slide 47

A toy clientS/server example written in GdH (inheriting from Concurrent Haskell and GpH) [5]:

```
main do =
 ch <- newChannel    pes <- allPEId
 q <- readQuadTree "geoMap" `usingIO` rnf //reduction to normal form
 mapM (rforkIO (runGUI (window ch))) pes //remote fork with IO
 let server = do
    respond ch ( b -> return (visible q b) `usingIO` rnf)
    server
 forkIO server

window ch gui = do
  init gui ...
  showit b = do
    m <- request ch b
```

Slide 48

```
draw gui m ‘demanding‘ rnf m
```

Slide 49

5.2 Process Networks

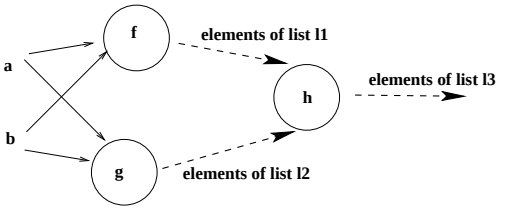
- The program is a network build with process (coarse grained computations) connected by ports (input and output) onto which data flow.

→ pipeline of data and parallel execution on those data by processes.

- Automatic synchronisation on ports in a producer/consumer way.
- No deadlock in the computation if the network is a DAG
- Such model may be well adapted to grid-computing as it emphasizes on the overlap of communications by computations (e.g. the PAGIS system and an implementation using ProActive, [13]).

Slide 50

```
l1 = f(a,b)
l2 = g(a,b)
l3 = h(l1, l2) // The function h is non strict in l1 & l2
```



Example of a node in a PNetwork

```
public class ProcessWriter extends thread{
  OutputStream w; Server s;
 // constructor
 void run() {
  Packet p;
  while ((p=s.server_consume())!=null)
      process_write(w,p);
 w.close();
 }
 }
```

Slide 51

6 Concluding remarks

To my point of view the field of distributed functional programming should be considered only if distribution of the various computations is a hard constraint (e.g. client/server interaction), not just for sharing load on many CPUs. In this last case, parallel compiler technologies should eventually succeed to really exploit distributed parallelism implicitly.

Some important problems that distributed functional programming should work on:

- take advantage of portable platforms like Virtual machines
- may be able to target NOWs and Grids
- may do at least as good or better than distributed object-oriented languages w.r.t. management of concurrency and

Slide 52

distribution (remote calls, exception handling). Still have many advantages over OO: equational reasoning, non strict evaluation modes, referential transparency, high-level notation, etc, ...

Slide 53

References

[1] K. Hammond. Paralell functional programming: An introduction. In *1st International Symposium on parallel Symbolic Computation*, 1994.

[2] Paul Hudak. Exploring parafunctional programming: Separating the what from the how. *IEEE Software*, 5(1):54–61.

[3] Gabrielle Keller and Manuel M.T. Chakravarty. On the distributed implementation of aggregate data structures by program transformation. In *IPDPS workshop on High-level parallel programming models and supportive environments HIPS'99*, number 1586 in LNCS.

[4] H.W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G.J. Michaelson, R. Pena, S. Priebe,

Slide 54

A.J. Rebon Portillo, and P.W. Trinder. Comparing parallel functional languages: Programming and performance. *Higher-Order and Symbolic Computation*, 16(3), 2003.

[5] R.F. Pointon, S. Priebe, H.W. Loidl, R. Loogen, and P.W. Trinder. Functional vs object-oriented distributed languages. In *EUROCAST*, number 2178 in LNCS, 2001.

[6] Jr R.H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[7] Pascal R. Serrarens and Marinus J. Plasmeijer. Explicit message passing for concurrent clean. In *Implementation of Functional Languages*, pages 229–245, 1998.

[8] D. Skillicorn. Architecutre-independant parallel computation. *IEEE Computer*, 1990.

Slide 55

[9] P. W. Trinder, E. Barry, Jr., M. K. Davis, K. Hammond, S. B. Junaidu, U. Klusik, H-W. Loidl, and S. L. Peyton Jones. GPH: An Architecture-Independent Functional Language. *IEEE Transactions on Software Engineering*, 1999.

[10] Philip W. Trinder, Hans-Wolfgang Loidl, Ed. Barry Jr., M. Kei Davis, Kevin Hammond, Ulrike Klusik, Simon L. Peyton Jones, and Álvaro J. Rebón Portillo. The multi-architecture performance of the parallel functional language GpH (research note). *Lecture Notes in Computer Science*, 1900:739–743, 2001.

[11] P.W. Trinder, K. Hammond, J.S. Mattson Jr., A.S. Partridge, and S.L. Peyton Jones. GUM: a portable implementation of Haskell. In *Proceedings of Programming Language Design and Implementation*, 1996.

[12] P.W. Trinder, H.W. Loidl, and R.F. Pointon. Parallel and

Slide 56

Distributed Haskells. *Journal of Functional Programming*, 2002.

[13] Darren Webb, Andrew Wendelborn, , and Kevin Maciunas. Process Networks as a High-Level Notation for Metacomputing. In *Proc. of the Int. Parallel Programming Symposium (IPPS'99), workshop on Java for Distributed Computing, Puerto Rico*, 1999.

[14] A. Al Zain, P.W. Trinder, H-W. Loidl, and G.J. Micaelson. Managing Heterogeneity in a Grid Parallel Haskell. In *ICCS 2005, Workshop on Practical Aspects of High-Level Parallel Programming (PAPP)*, number 3515 in LNCS.

Slide 57