Abstract, declarative control over partitioning in parallel functional programs: experiences with Caliban

Paul H J Kelly

phjk@doc.ic.ac.uk

(joint work with Frank Taylor, now with TECC Ltd, London)

Department of Computing,
Imperial College London, U.K.

September 1998

Introduction

Slogan:

"Parallel programming by programmed partitioning"

- Suppose automatic parallelisation isn't good enough
- The programmer has to decide how the data and computation is distributed across the processing elements
  - ⋆ This is a software engineering problem
  - ⋆ It needs a good programming language
  - ⋆ We need to support abstraction and re-use

The fully-dynamic approach

- Futures (lazy task creation etc)
- `par`, "sparking"
- threads, fork
- ParAlfl's "`$on`' operator

Processes are generated one-at-a-time as the computation progresses

Higher-level operators (pipelines, farms, arrays of processes etc) can easily be programmed, but the structure is not evident to the implementation

⋆ If we can separate the partitioning from the program execution, we can exploit broader knowledge when allocating resources

⋆ Start with fully static

⋆ Necessary precursor to semi-static

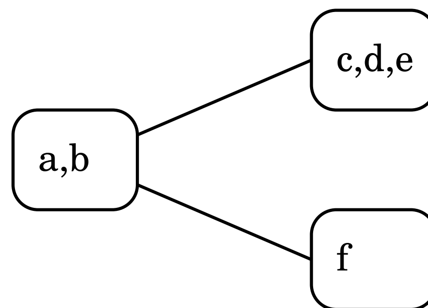
Compile-time vs run-time

Programmed partitioning depends on:

- the target machine: number of processors, communication vs computation performance, memory capacity
- the problem instance: size, shape of key data structures
- the evolution of the computation: e.g. where eddies, collisions or precipitation occur

Some information available at compile-time, some only later

Introducing Caliban



- This is expressed as an annotation (in fact a Haskell data structure):

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

- The Bundle assertions specify that
  - expressions a and b are allocated to the same processor
  - c, d and e to the same, presumably different processor
  - and f to a third.
  - The Arc a d assertion specifies a link in the task graph, either because a consumes d or *vice versa*.

The process placement rule

- The **moreover** assertion specifies the placement of named expressions onto the task graph.
- The "process placement rule" (analogous to "owner computes" in HPF) specifies where the computation of these expressions takes place.

⋆ In the absence of any annotations, *every* processor executes the *entire* program. Some arbitrarily-chosen processor's result expression is output.

⋆ The expression Bundle [x] asserts that x is computed on one processor only, and all non-local references to x involve communication.

Communication

- For simplicity, we assume that placed expressions such as x are of list type, and our implementation serialises such lists by evaluating and sending each element in its entirety in turn.

Explanation:

- Caliban is based on the model of a static network of processes communicating via streams of messages.
- To support this, we require that in an annotation such as Bundle [x], the name x should refer to an object which can be transmitted as a stream of messages.

This raises various issues...

- Isn't it unnecessarily restrictive?
- Doesn't it interfere with the semantics?

Compute-ahead and strictness

- Evaluation proceeds in anticipation of demand, so that the producer of a stream can operate in parallel with its consumer.
- This “compute-ahead” is restricted by the availability of buffer space in the consumer (and, of course, by availability of operands).

Streams are easy to compute ahead, because the consumer cannot choose what to demand next

Threads: “bundling”

- In an assertion such as Bundle [a,b], two expressions are assigned to the same processor.
- We create a thread for each expression, each charged with computing elements ahead of demand and sending the values to each of the consumer processors.

In principle, these threads should be pre-emptively scheduled, so that evaluation of all the expressions allocated to a processor proceeds even if one of the threads loops or blocks.

Strictness and semantics

- "Bundle" assigns a thread to evaluate the elements of the specified stream in order

An annotation may stop a working program from producing all its outputs:

1. Threads are evaluated in advance of demand. If the stream turns out not to be needed, may loop – but a consumer thread oughtn't be interfered with
2. Stream elements are evaluated in order. If one element turns out not to be needed may loop evaluating unwanted stream element, so subsequent elements are never reached
3. If threads are not pre-emptively scheduled, one expression may wait forever for evaluation of another expression on the same processor

Strictness and semantics - cont'd

One more reason an annotation may stop a working program from producing all its outputs:

4 To evaluate the annotation, the compiler may have to evaluate expressions which aren't needed by the computation by itself – the compiler may not terminate

Example: ray tracing

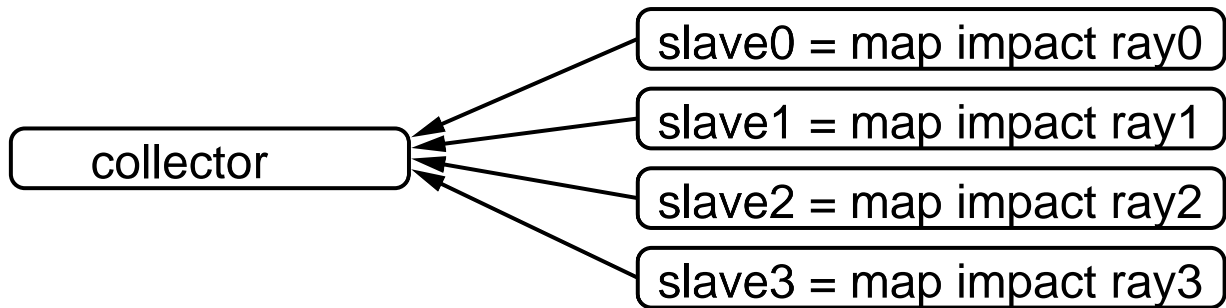
A simple ray-tracer can be reduced to the following Haskell program:

```
ray_trace scene viewpoint
  = map impact rays
    where
    rays = generate_rays viewpoint
    impact ray = fold earlier impacts
                 where
                 impacts = map (hit ray) scene
```

- hit ray obj tests whether a ray strikes a given object in the scene, and if so, returns details of the impact
- earlier impact_1 impact_2 returns the first impact struck by the ray

The closest impact to the viewpoint determines the colour of the pixel in the output.

First approach: using a processor farm



Idea: define a function to generate this pattern:

```
fan :: Stream → [Stream] → Placement
fan s [] = NoPlace
fan s (a:as) = (Bundle [a]) And (Arc a s) And (fan s as)
```

the expression `fan collector [slave0, slave1, slave2, slave3]` yields the annotation

```
Bundle [slave0] And (Arc slave0 collector) And
Bundle [slave1] And (Arc slave1 collector) And
Bundle [slave2] And (Arc slave2 collector) And
Bundle [slave3] And (Arc slave3 collector) And
NoPlace
```

("NoPlace" is the null annotation). "fan" is called an NFO - a network forming operator.

Skeletons

We can define a reusable function which encapsulates the processor farm behaviour; we use the fan operator to build its annotation:

```
farm :: (a→a) → [a] → [a]
farm func operands
  = farmed moreover fan farmed farmed
    where
    farmed = map func operands
```

- The assertion says that each element of the list farmed is evaluated on a separate processor, and the results (the actual list farmed) are collected onto a single processor for output
- The assertion is evaluated by the compiler to yield an annotation which places each of the slaves on a separate processor.
- number of operands must be known

Using the parallel operator

```
ray_trace scene viewpoint
  = farm impact rays
    where
    rays = generate_rays viewpoint
    impact ray = fold earlier impacts
                 where
                 impacts = map (hit ray) scene
```

This is unfolded by the compiler:

```
ray_trace scene viewpoint
  = farmed moreover fan farmed farmed
    where
    farmed = map impact rays
    rays = generate_rays viewpoint
    impact ray = fold earlier impacts
                 where
                 impacts = map (hit ray) scene
```

If the viewpoint is known at compile-time, we can calculate the list of rays:

```
ray_trace scene viewpoint
  = farmed moreover fan farmed farmed
    where
    farmed = map impact rays
    rays = generate_rays viewpoint = [r0, r1, r2, r3]
    ...
```

Using the definition of map the compiler can construct the list of unevaluated processes:

```
ray_trace scene viewpoint
  = farmed moreover fan farmed farmed
    where
    [farmed0, farmed1, farmed2, farmed3] = farmed
    farmed
      = [impact r0, impact r1, impact r2, impact r3]
    rays = generate_rays viewpoint = [r0, r1, r2, r3]
    ...
```

The compiler can expand fan farmed farmed and build the static process network....

```
ray_trace scene viewpoint
  = farmed moreover
    Bundle [farmed0] And (Arc farmed0 farmed) And
    Bundle [farmed1] And (Arc farmed1 farmed) And
    Bundle [farmed2] And (Arc farmed2 farmed) And
    Bundle [farmed3] And (Arc farmed3 farmed) And
    NoPlace
where
[farmed0, farmed1, farmed2, farmed3]
  = [impact r0, impact r1, impact r2, impact r3]
rays = generate_rays viewpoint = [r1, r2, r3, r4]
impact ray = fold earlier impacts
             where
             impacts = map (hit ray) scene
```

- You could have written the annotation above manually
- Caliban's compile-time symbolic evaluation allowed a more concise annotation

Aggregation by bundling

- The ray tracer above assigns one processor per ray
- With many (or an unknown number of) rays, we want to allocate many rays to each processor
- It seems natural to use Bundle:

```
farm :: (a→a) → [a] → [a]
farm func operands
  = farmed moreover fan farmed slaves
    where
    farmed = map func operands
    slaves = partition noOfProcessors farmed
```

where partition n xs simply splits a list xs into n sublists as equally as possible.

- So, if noOfProcessors is 2, farm f [x_0, x_1, x_2, x_3] expands to

```
farmed = map f [x0, x1, x2, x3]
moreover  Bundle p1 And (Arc p1 farmed) And
          Bundle p2 And (Arc p2 farmed) And
          NoPlace
where
p1 = [x0, x1]
p2 = [x2, x3]
```

- This is nice in that the code for the computation itself is completely unchanged

Problems with aggregation by bundling

- Unfortunately, using bundling alone for aggregation has some serious practical problems
  - → The compiler has to elaborate the entire list of rays, rather than the list of partitions
  - → There are many communication messages, one for each ray – we have aggregated computation but not communication
- Related to this:
  - → The compiler has to know when to stop evaluating p_1 and p_2 – our implementation always evaluates placed expressions to WHNF
  - → So the first element of each list x_0, x_1, x_2 and x_3 is computed at compile-time (unless further input is required)

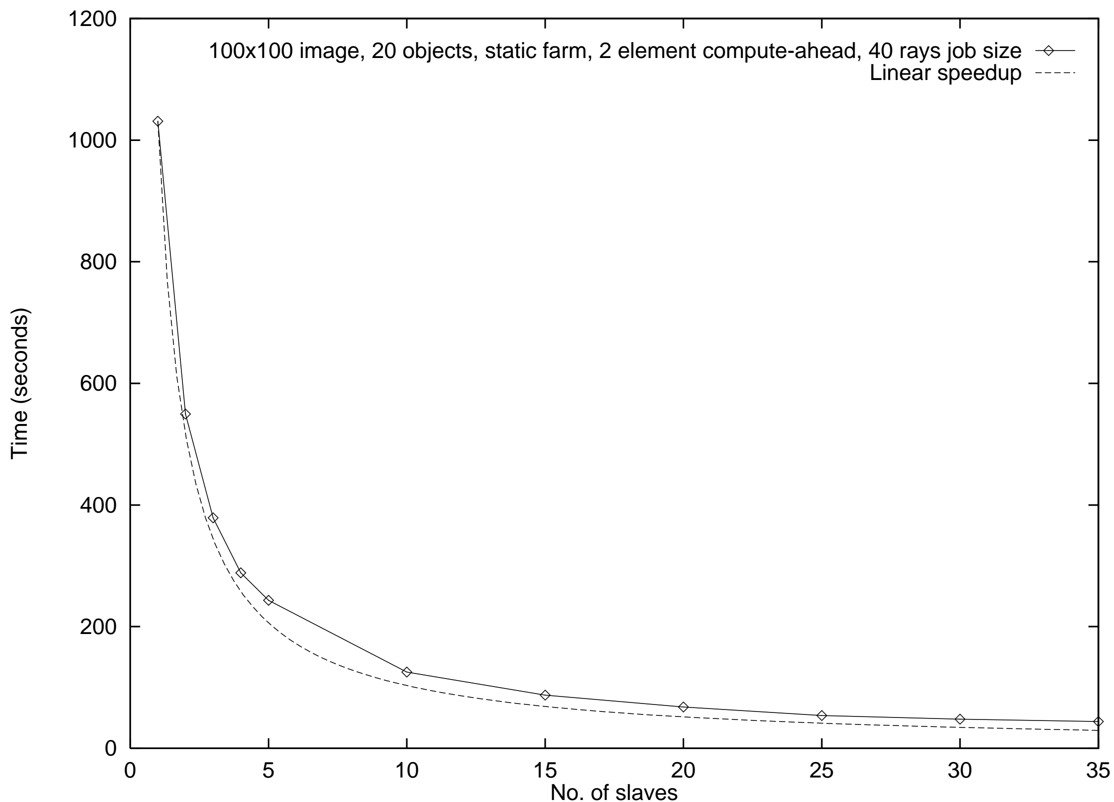
Aggregation by restructuring

We can fix most of the the problems by modifying the computation to partition explicitly:

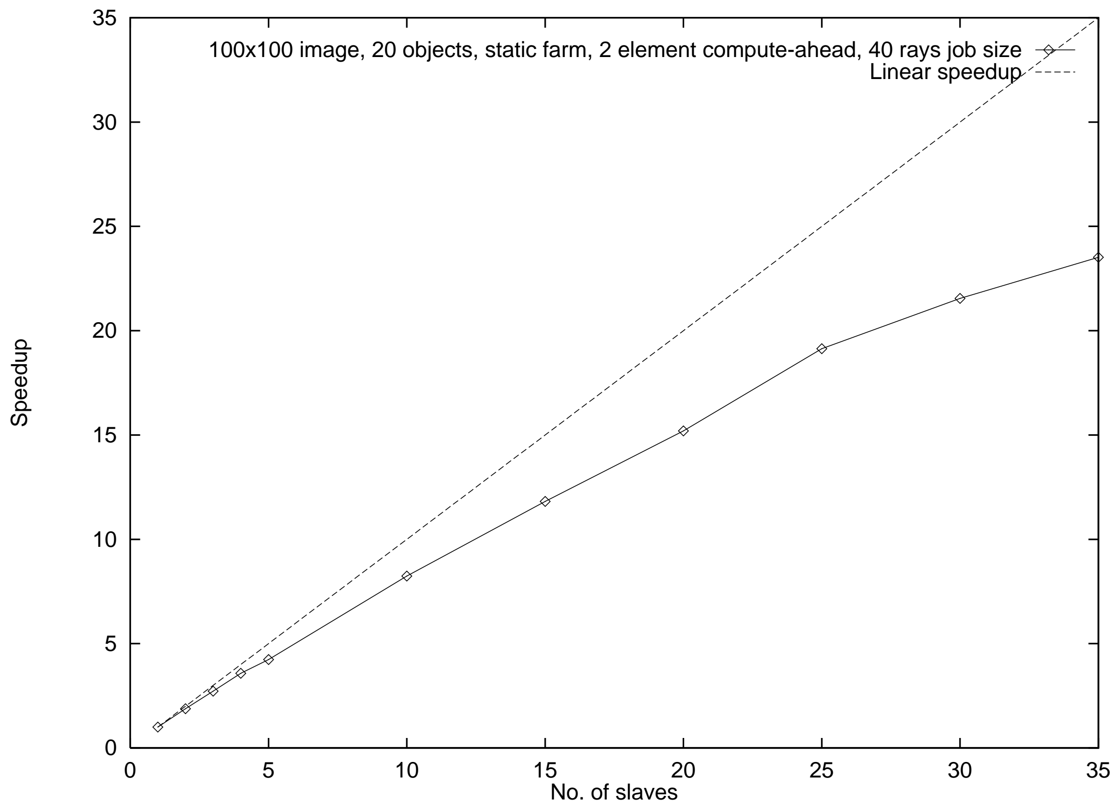
```
rayTrace scene viewpoint
  = farm noOfProcessors impact rays
    where
    rays = generateRays viewpoint
    impact ray = fold earlier impacts
                 where
                 impacts = map (hit ray) scene

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

Farmed ray tracer - performance



- The scene consisted on 20 simple objects
- There were 100×100 rays
- MPI on the Fujitsu AP1000 at Imperial College (25MHz Sparc processors)
- Implementation built on the Haskell^{-} compiler and a version of the FCG project back-end (with thanks to Pieter Hartel and others)



The graphs show execution time and speedup for the explicitly-partitioned ray tracer:

- A small further adjustment was made to avoid compile-time ray tracing
- Compute-ahead was adjusted to avoid excessive blocking
- Rays were partitioned into groups of 40 for maximum performance

Summary

- Motivation

  Controlling how a program is executed in parallel - declaratively

- Promise

  Use the power of the functional language, use powerful generic operators, re-use operators from the computation

- Implementation

  Use partial evaluation to specialise the program for each target machine, and perhaps even each problem instance

- Re-use, composition

- Problems

  Compilation time, controlling partial evaluation, surprising effects of demand-driven execution

Discussion: dynamic vs static

→ Can Caliban be extended to be more dynamic?

- Sure, we could interpret Bundle as a "spark"
- We lose the opportunity to schedule aggregate process networks to processors

  (eg to ensure processes in a pipeline aren't allocated to the same processor)
- We lose control over where communication happens

  (so where a given expression is computed may depend on a non-deterministic race between evaluation and sparking)

Directions for further work

→ Partial evaluation has lots of potential

→ Hybrid dynamic/static resource allocation is an interesting area

→ Caliban did not provide enough control over evaluation order and communication