Dynamic generation of parallel computations

James Hanlon, Simon J. Hollis
Many-core project

June 13, 2011

Background

- Concurrency is not a new area: originally developed as a key abstraction in the design of real-time systems
- Conventional thinking in academia and industry has largely ignored the vast amount of work in this area.
- Caused largely by preoccupation with frequency scaling, (between $\sim$1970-2005).
- *Parallelism* will be the primary means of increasing computational performance.
- But we don't know how to effectively architect or program parallel computers.

State of the art parallelism

- Parallelism now pervasive in systems design
  - HPC systems becoming increasingly important in science and industry.
  - Dual/quad core processors standard in desk and laptop computers.
  - Embedded systems using network-on-chip designs.

State of the art parallelism

- Parallelism now pervasive in systems design
  - HPC systems becoming increasingly important in science and industry.
  - Dual/quad core processors standard in desk and laptop computers.
  - Embedded systems using network-on-chip designs.
- *But*: parallelism is still deployed in specific areas, addressing specific requirements.
- Evident in wide the wide variety of designs, e.g. CMPs, GPUs, HPC systems.
- Emerging *gap* between architectures and languages, and application users.
- Very difficult for users to harness all available parallelism.

General-purpose parallel computers

- Sequential case: von Neumann architecture provides an *efficient abstraction* from the implementation of different computer systems.
  - Hides *irrelevant* details from the programmer
  - Makes possible *standardised languages* and *transportable software*

General-purpose parallel computers

- Sequential case: von Neumann architecture provides an *efficient abstraction* from the implementation of different computer systems.
  - Hides *irrelevant* details from the programmer
  - Makes possible *standardised languages* and *transportable software*
- *Universality* concept, introduced by Turing in 1937.
  - Computer both special purpose device for executing a program, as well as a device capable of simulating *all* programs.
  - Special purpose machines have no significant advantage (Valiant 1990).
- A universal *parallel* computer would allow parallelism to be exploited effectively with *high level, transportable* languages.

Language features supporting concurrency

- Programming languages must support high-level concurrent programming.
- Contribution of this work is to demonstrate the existence of simple language *features* supporting this.
- Process-to-processor allocation is the key issue.

Parallelism and channel communication

```
proc init() is
  var c: chan;
{ p1(c) | p2(c) }
```

```
proc p1
(c: chanend) is
  var x: integer;
{ x:=0 ; c!x ; c?x }
```

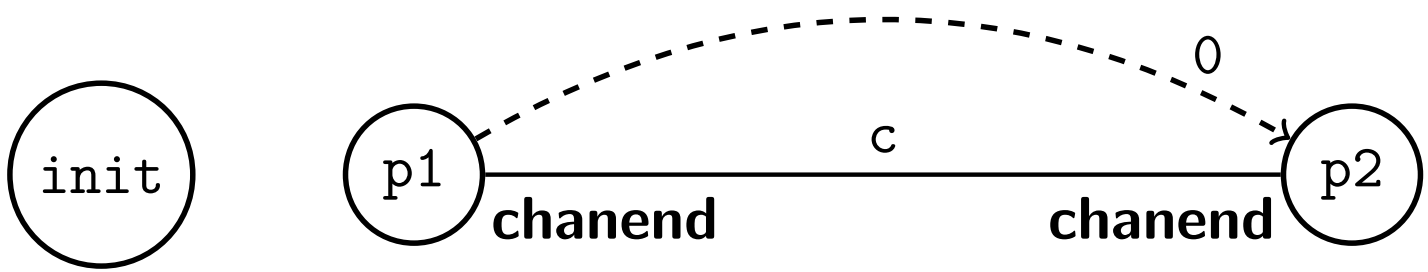
```
proc p2
(c: chanend) is
  var y: integer;
{ c?y ; c!y+1 }
```

Parallelism and channel communication

```
proc init() is
  var c: chan;
{ p1(c) | p2(c) }
```

```
proc p1
(c: chanend) is
  var x: integer;
{ x:=0 ; c!x ; c?x }
```

```
proc p2
(c: chanend) is
  var y: integer;
{ c?y ; c!y+1 }
```

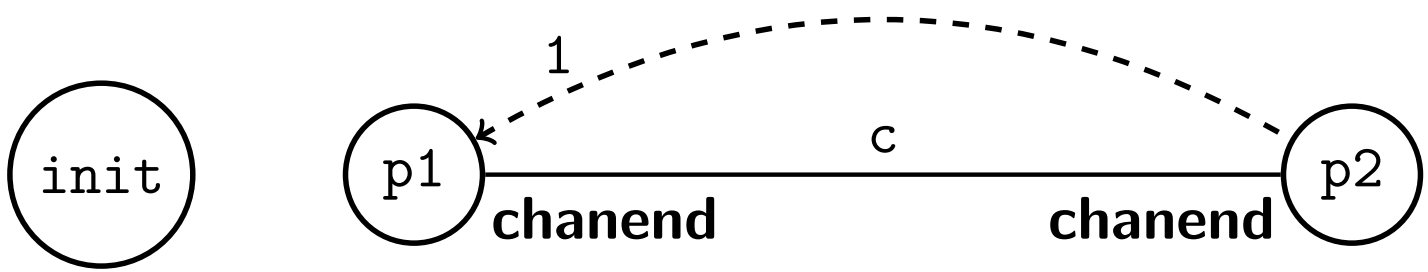


Parallelism and channel communication

```
proc init() is
  var c: chan;
{ p1(c) | p2(c) }
```

```
proc p1
(c: chanend) is
  var x: integer;
{ x:=0 ; c!x ; c?x }
```

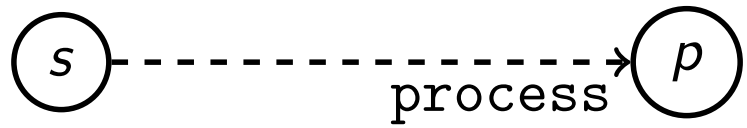
```
proc p2
(c: chanend) is
  var y: integer;
{ c?y ; c!y+1 }
```



Process migration

- Offload a process:

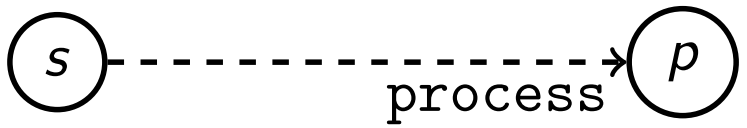
on p **do** `process()`



Process migration

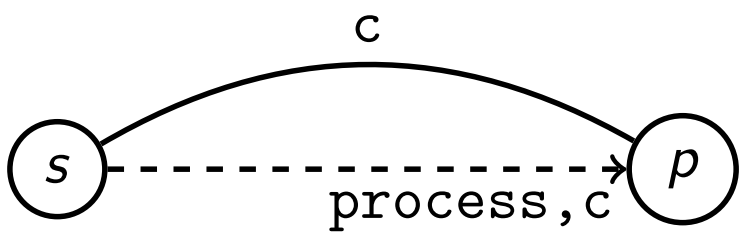
- Offload a process:

```
on p do process()
```



- Offload a process with a channel:

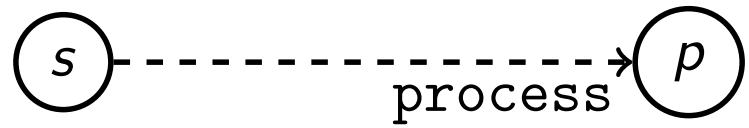
```
var c: chan
{ on p do process(c)
; c ! value
}
```



Process migration

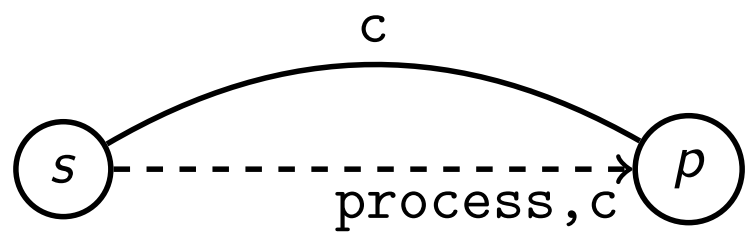
- Offload a process:

```
on p do process()
```



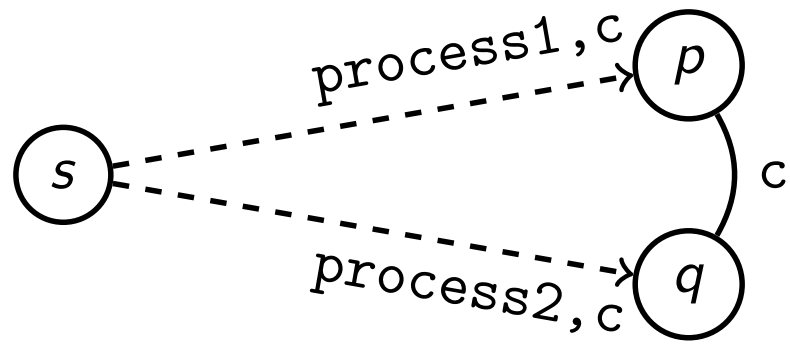
- Offload a process with a channel:

```
var c: chan
{ on p do process(c)
; c ! value
}
```



- Offload processes sharing a channel:

```
var c: chan
{ on p do process1(c)
; on q do process2(c)
}
```



Parallel recursion

- *Parallel recursion* is a natural tool for expressing concurrent program structures.

Parallel recursion

- *Parallel recursion* is a natural tool for expressing concurrent program structures.
- **Recursion**: solve a problem by solving smaller instances of the same problem.
- **Parallelism**: break a large computation down into smaller parts.

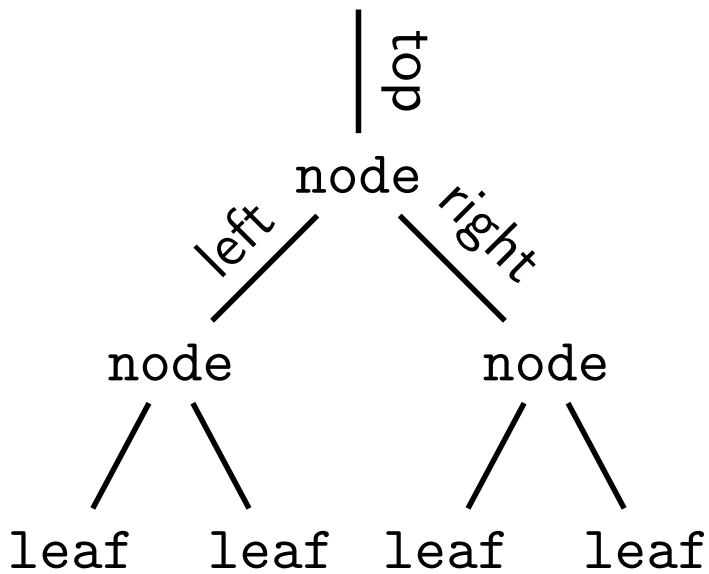
Creating a tree

```
proc tree(depth: int; top: chanend) is
  var left, right: chan
  if depth = 0 then leaf(top)
  else { node(top, left, right) |
    tree(depth-1, left) | tree(depth-1, right) }
```

Creating a tree

```
proc tree(depth: int; top: chanend) is
  var left, right: chan
  if depth = 0 then leaf(top)
  else { node(top, left, right) |
    tree(depth-1, left) | tree(depth-1, right) }
```

`tree(2, top)`:



Process structures

- A process structure is the communication topology of a set of concurrent processes.
- Simple structures such as the tree underpin many important parallel algorithms.
  - e.g. sorting and FFT.
- Other common process structures include arrays, meshes and hypercubes.
- Parallel recursion and process migration allow the style of programming to shift from *data structures* to *process structures*.

Example: rapid process spawning

- Combine parallel recursion and process migration to optimise the distribution of processes over a system.

```
proc d(t, n: int) is
  if n = 1 then node(t)
  else { d(t, n/2) | on t + n/2 do d(t + n/2, n/2) }
```

- Given a set of networked processors p_0, p_1, p_2, p_3, `d(0, 4)` executes in *time* and *space*:

Step	p_0	p_1	p_2	p_3

Example: rapid process spawning

- Combine parallel recursion and process migration to optimise the distribution of processes over a system.

```
proc d(t, n: int) is
  if n = 1 then node(t)
  else { d(t, n/2) | on t + n/2 do d(t + n/2, n/2) }
```

- Given a set of networked processors p_0, p_1, p_2, p_3, `d(0, 4)` executes in *time* and *space*:

Step	p_0	p_1	p_2	p_3
0	`d(0,4)`			

Example: rapid process spawning

- Combine parallel recursion and process migration to optimise the distribution of processes over a system.

```
proc d(t, n: int) is
  if n = 1 then node(t)
  else { d(t, n/2) | on t + n/2 do d(t + n/2, n/2) }
```

- Given a set of networked processors p_0, p_1, p_2, p_3, `d(0, 4)` executes in *time* and *space*:

Step	p_0	p_1	p_2	p_3
0	`d(0,4)`			
1	`d(0,2)`		`d(2,2)`	

Example: rapid process spawning

- Combine parallel recursion and process migration to optimise the distribution of processes over a system.

```
proc d(t, n: int) is
  if n = 1 then node(t)
  else { d(t, n/2) | on t + n/2 do d(t + n/2, n/2) }
```

- Given a set of networked processors p_0, p_1, p_2, p_3, `d(0, 4)` executes in *time* and *space*:

Step	p_0	p_1	p_2	p_3
0	`d(0,4)`			
1	`d(0,2)`		`d(2,2)`	
2	`d(0,1)`	`d(1,1)`	`d(2,1)`	`d(3,1)`

Example: rapid process spawning

- Combine parallel recursion and process migration to optimise the distribution of processes over a system.

```
proc d(t, n: int) is
  if n = 1 then node(t)
  else { d(t, n/2) | on t + n/2 do d(t + n/2, n/2) }
```

- Given a set of networked processors p_0, p_1, p_2, p_3, `d(0, 4)` executes in *time* and *space*:

Step	p_0	p_1	p_2	p_3
0	`d(0,4)`			
1	`d(0,2)`		`d(2,2)`	
2	`d(0,1)`	`d(1,1)`	`d(2,1)`	`d(3,1)`
3	`node(0)`	`node(1)`	`node(2)`	`node(3)`

Hardware support for concurrency

- It is essential for an efficient implementation of these mechanisms that the hardware directly supports them.
- Difficult in systems like MPI where communication predominantly software based.

Hardware support for concurrency

- It is essential for an efficient implementation of these mechanisms that the hardware directly supports them.
- Difficult in systems like MPI where communication predominantly software based.
- *Process* and *communication* primitives must be provided at the hardware level (in the instruction set).
- These primitives must complete in same magnitude of time as equivalent sequential operations such as subroutine calls & memory accesses.

A real implementation

- XMOS XCore processor architecture: general-purpose, scalable and provides low-level support for concurrency.
- Completed work:
  - Written bespoke compiler implementing a small language as platform for new features
  - A simple implementation of `on` statement.
- Initial exploration of approach has been promising. Results will follow in due course.

Conclusions

- The combination of parallel recursion and process migration allows the elegant expression of powerful concurrent programs.
- Rapid process distribution is an important mechanism in large scale systems & has a simple high level expression in this framework.
- The existence of the sympathetic XCore architecture proves implementation of efficient mechanisms supporting concurrent programming are feasible.
- The results will be very competitive when compared to leading parallel architectures.

Any questions?

Email: hanlon@cs.bris.ac.uk