Fast Distributed Process Creation with the XMOS XS1 Architecture

James Hanlon

Department of Computer Science
University of Bristol, UK

20th June 2011

Processors as a resource

- Current parallel programming models provide little support for management of processors.
- Many are closely coupled to the machine and parameterised by the number of processors.
- The programmer is left responsible for scheduling processes on the underlying system.
- As the level of parallelism increases (10^6 processes at exascale), it is clear that we require a means to automatically allocate processors.
- We don't expect to have to write our own memory allocation routines!

Scalable parallel programming

- For parallel computations to scale it will be necessary to express programs in an *intrinsically* parallel manner, focusing on *dependencies* between processes.
- Excess parallelism enables scalability (parallel slackness hides communication latency).
- It is also more expressive:
  - For *irregular* and *unbounded* structures.
  - Allows *composite* structures and construction of *parallel subroutines*.
- The scheduling of processes and allocation of processors is then a property of the language and runtime.
- But this requires the ability to rapidly initiate processes and collect results from them as they terminate.

Contributions

1. The design of an explicit, lightweight scheme for *distributed dynamic processor allocation*.
2. A convincing proof-of-concept implementation on a sympathetic architecture.
3. Predictions for larger systems based on accurate performance models.

Platform

- XMOS XS1 architecture:
  - General-purpose, multi-threaded, message-passing and scalable.
  - Primitives for threading, synchronisation and communication execute in same time as standard load/store, branch and arithmetic operations.
  - Support for position independent code.
  - Predictable.
- XK-XMP-64:
  - Experimental board with 64 *XCore* processors connected in a hypercube.
  - 64kB of memory and 8 hardware threads per core.
  - Aggregate 512-way concurrency, 25.6 GIPS and 4MB RAM.
- A bespoke language and runtime with a simple set of features to demonstrate and experiment with distributed process creation.

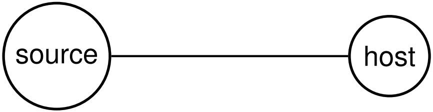
Explicit processor allocation: notation

- Processor allocation is exposed in the language with the **on** statement:
  $$\textbf{on}\ p\ \textbf{do}\ Q$$
  This executes process Q synchronously on processor p.
- The execution of all processes are implicitly **on** the current processor.
- We can compose **on** in parallel to exploit multi-threaded parallelism:
  $$\{\ Q_1 \parallel \textbf{on}\ p\ \textbf{do}\ Q_2\ \}$$
  which offloads and executes Q_2 while executing Q_1.
- Processes must be disjoint.

Explicit processor allocation: implementation

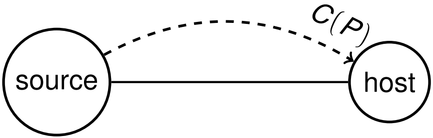
- **on** forms a *closure* C of process P including the variable context and a list of procedures including P and those it calls.

Explicit processor allocation: implementation



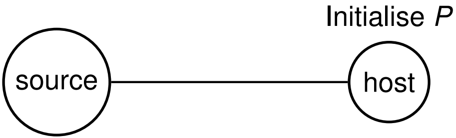
- **on** forms a *closure C* of process *P* including the variable context and a list of procedures including *P* and those it calls.
- A connection is initialised between the *source* and *host* processors and the host creates a new thread for the incoming process.

Explicit processor allocation: implementation



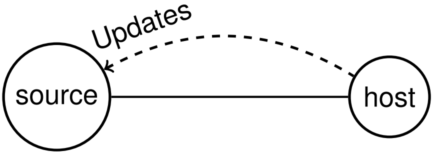
- **on** forms a *closure* C of process P including the variable context and a list of procedures including P and those it calls.
- A connection is initialised between the *source* and *host* processors and the host creates a new thread for the incoming process.
- It then receives $C(P)$ and initialises P on the new thread.

Explicit processor allocation: implementation



- **on** forms a *closure* C of process P including the variable context and a list of procedures including P and those it calls.
- A connection is initialised between the *source* and *host* processors and the host creates a new thread for the incoming process.
- It then receives $C(P)$ and initialises P on the new thread.
  - All call branches are performed through a table (with the instruction BLACP) so the host updates this to record the new address of each procedure contained in C.

Explicit processor allocation: implementation



- **on** forms a *closure* C of process P including the variable context and a list of procedures including P and those it calls.
- A connection is initialised between the *source* and *host* processors and the host creates a new thread for the incoming process.
- It then receives $C(P)$ and initialises P on the new thread.
  - All call branches are performed through a table (with the instruction BLACP) so the host updates this to record the new address of each procedure contained in C.
- When P has terminated, the host sends back any updated free variables of P stored at the source (as P is disjoint).

Rapid process distribution

- We can combine recursion and parallelism to rapidly generate processes:

  proc *distribute* (t, n) **is**
      if $n = 1$ **then** *node* (t)
      else
      $\{$ *distribute* $(t, n/2) \parallel$ **on** $t + n/2$ **do** *distribute* $(t + n/2, n/2)$ $\}$
- This distributes the process *node* over n processors in $O(\log n)$ time.
- The execution of *distribute* $(0, 4)$ proceeds in *time* and *space*:

p_0 $\qquad$ p_1 $\qquad$ p_2 $\qquad$ p_3

Rapid process distribution

- We can combine recursion and parallelism to rapidly generate processes:

  proc *distribute* (t, n) **is**
  if $n = 1$ **then** *node* (t)
  else
  $\{$ *distribute* $(t, n/2) \parallel$ **on** $t + n/2$ **do** *distribute* $(t + n/2, n/2)$ $\}$

- This distributes the process *node* over n processors in $O(\log n)$ time.
- The execution of *distribute* (0, 4) proceeds in *time* and *space*:

p_0	p_1	p_2	p_3
distribute (0,4)			

Rapid process distribution

- We can combine recursion and parallelism to rapidly generate processes:

  proc *distribute* (t, n) **is**
      if $n = 1$ **then** *node* (t)
      else
      $\{$ *distribute* $(t, n/2) \parallel$ **on** $t + n/2$ **do** *distribute* $(t + n/2, n/2)$ $\}$

- This distributes the process *node* over n processors in $O(\log n)$ time.
- The execution of *distribute* (0, 4) proceeds in *time* and *space*:

p_0	p_1	p_2	p_3
distribute (0,4)			
distribute (0,2)		*distribute* (2,2)	

Rapid process distribution

- We can combine recursion and parallelism to rapidly generate processes:

  proc *distribute* (t, n) **is**
      if $n = 1$ **then** *node* (t)
      else
      $\{$ *distribute* $(t, n/2) \parallel$ **on** $t + n/2$ **do** *distribute* $(t + n/2, n/2)$ $\}$

- This distributes the process *node* over n processors in $O(\log n)$ time.
- The execution of *distribute* (0, 4) proceeds in *time* and *space*:

p_0	p_1	p_2	p_3
distribute (0,4)			
distribute (0,2)		*distribute* (2,2)	
distribute (0,1)	*distribute* (1,1)	*distribute* (2,1)	*distribute* (3,1)

Rapid process distribution

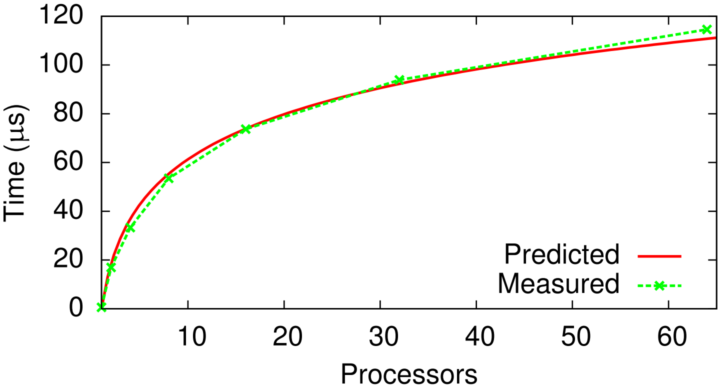
- We can combine recursion and parallelism to rapidly generate processes:

  proc *distribute* (t, n) **is**
      if $n = 1$ **then** *node* (t)
      else
      $\{$ *distribute* $(t, n/2) \parallel$ **on** $t + n/2$ **do** *distribute* $(t + n/2, n/2)$ $\}$

- This distributes the process *node* over n processors in $O(\log n)$ time.
- The execution of *distribute* (0, 4) proceeds in *time* and *space*:

p_0	p_1	p_2	p_3
distribute (0,4)			
distribute (0,2)		*distribute* (2,2)	
distribute (0,1)	*distribute* (1,1)	*distribute* (2,1)	*distribute* (3,1)
node (0)	*node* (1)	*node* (2)	*node* (3)

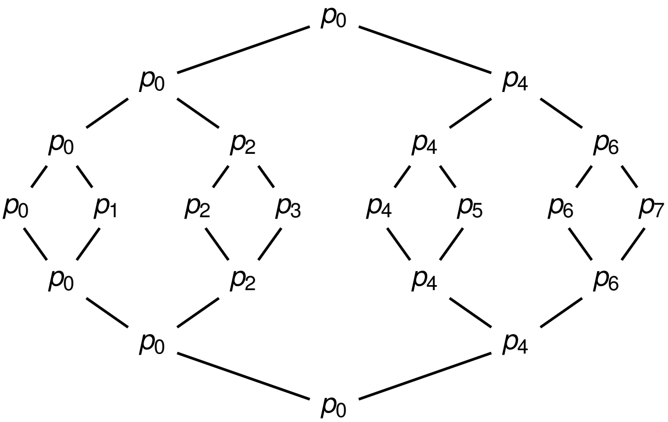
Rapid process distribution: execution time



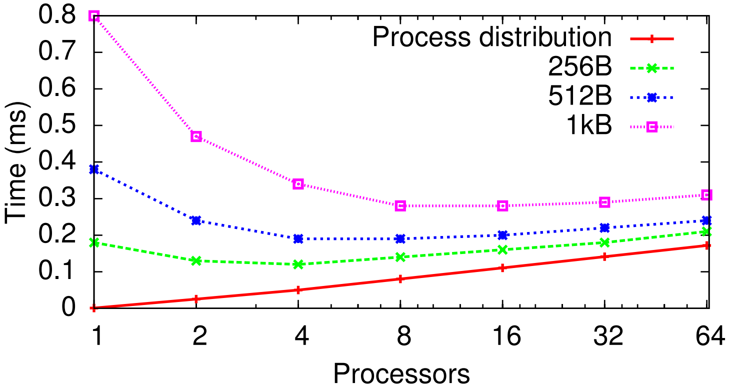
- 114.60μs (11,460 cycles) for 64 processors.
- Predicted 190μs for 1024 processors.

Mergesort

- Same structure as *distribute* but with work performed at leaves.



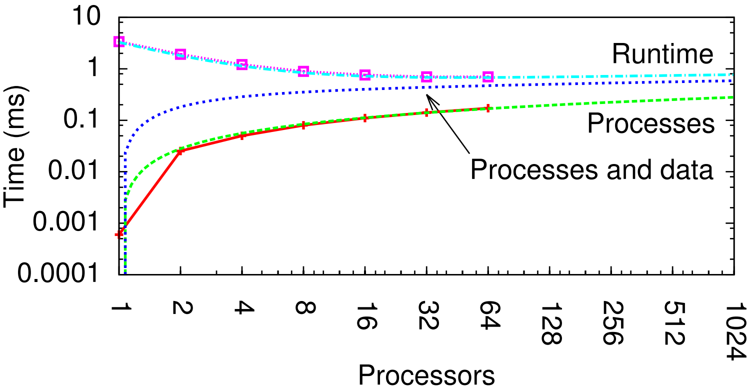
Mergesort: execution time I



- Minimum when input array is subdivided into 64B sections.

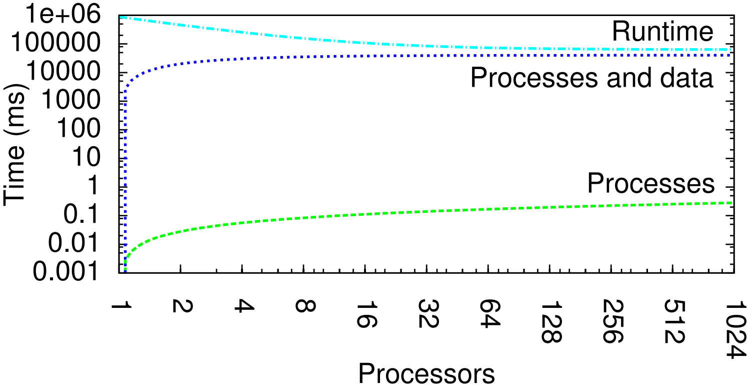
Mergesort: execution time II

- Measured (up to 64 cores) and predicted (up to 1024 cores) for 256B input.



Mergesort: execution time III

- Predicted up to 1024 cores for 1GB input.



- Single-source data-distribution is a worst-case.

Conclusions

- We have built a lightweight mechanism for dynamically allocating processors in a distributed system.
  - Combined with recursion we can rapidly distribute processes: over 64 processors in 114.60μs.
  - It is possible to operate at a fine granularity: creation of a remote process to operate on just 64B data.
- We can establish a lower bound on the performance of the approach.
  - Distribution over 1024 processors in $\sim$200μs (20,000 cycles).
- This scheme works well with large arrays of processors with small memories and allows you to express programs to exploit this.
  - Don't need powerful cores with large memories.
  - Emphasis changes from *data structures* to *process structures*.

Future work

1. Automatic placement of processes.
2. MPI implementation for evaluation on and comparison with supercomputer architectures.
3. Optimisation of processor allocation mechanism such as pipelining the reception and execution of closures.

Any questions?

Email:
hanlon@cs.bris.ac.uk

Project web page:
http://www.cs.bris.ac.uk/~hanlon/sire