

Multicore Scheduling for Lightweight Communicating Processes

Carl Ritson, Adam Sampson, Fred Barnes

Computing Laboratory
University of Kent
UK

10th June 2009

Introduction

- University of Kent at Canterbury (UK)
 - Systems Architecture Research Group
 - concurrency research sub-group
- Process-Oriented Programming



- Runtime kernel for multicore

Process-Oriented Programming (POP)

- Processes
 - Encapsulate state and code
 - No modifiable shared state
 - Execute concurrently
- Communication
 - Synchronise and exchange data
 - Interact to accomplish tasks
 - Connectivity, and communication patterns are run-time defined
- Concurrent processes
 - ⇒ parallel execution potential

Process-Oriented Programming (POP)

- Processes
 - Encapsulate state and code
 - No modifiable shared state
 - Execute concurrently
- Communication
 - Synchronise and exchange data
 - Interact to accomplish tasks
 - Connectivity, and communication patterns are run-time defined
- Concurrent processes
 - ⇒ parallel execution potential



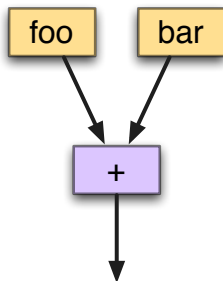
Process-Oriented Programming (POP)

- Processes
 - Encapsulate state and code
 - No modifiable shared state
 - Execute concurrently
- Communication
 - Synchronise and exchange data
 - Interact to accomplish tasks
 - Connectivity, and communication patterns are run-time defined
- Concurrent processes
⇒ parallel execution potential



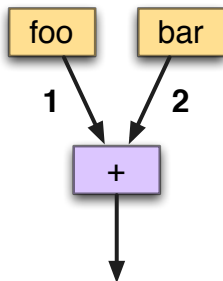
Process-Oriented Programming (POP)

- Processes
 - Encapsulate state and code
 - No modifiable shared state
 - Execute concurrently
- Communication
 - Synchronise and exchange data
 - Interact to accomplish tasks
 - Connectivity, and communication patterns are run-time defined
- Concurrent processes
⇒ parallel execution potential



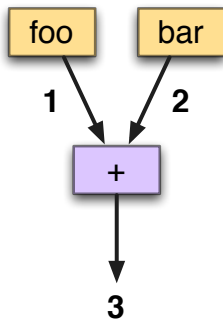
Process-Oriented Programming (POP)

- Processes
 - Encapsulate state and code
 - No modifiable shared state
 - Execute concurrently
- Communication
 - Synchronise and exchange data
 - Interact to accomplish tasks
 - Connectivity, and communication patterns are run-time defined
- Concurrent processes
⇒ parallel execution potential



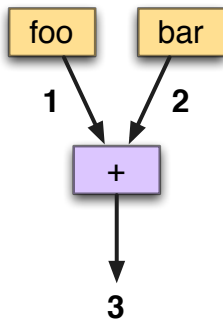
Process-Oriented Programming (POP)

- Processes
 - Encapsulate state and code
 - No modifiable shared state
 - Execute concurrently
- Communication
 - Synchronise and exchange data
 - Interact to accomplish tasks
 - Connectivity, and communication patterns are run-time defined
- Concurrent processes
⇒ parallel execution potential



Process-Oriented Programming (POP)

- Processes
 - Encapsulate state and code
 - No modifiable shared state
 - Execute concurrently
- Communication
 - Synchronise and exchange data
 - Interact to accomplish tasks
 - Connectivity, and communication patterns are run-time defined
- Concurrent processes
 - ⇒ parallel execution potential



Benefits

- Formal basis:
 - CSP
 - π -calculus
 - ... other process calculi
- Verify deadlock freedom
- ... or use deadlock-free pattern
- Freedom from race-hazards
- ... and aliasing errors

Benefits

- Formal basis:
 - CSP
 - π -calculus
 - ... other process calculi
- Verify deadlock freedom
- ... or use deadlock-free pattern
- Freedom from race-hazards
- ... and aliasing errors

Benefits

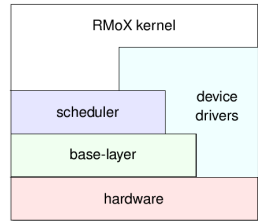
- Formal basis:
 - CSP
 - π -calculus
 - ... other process calculi
- Verify deadlock freedom
- ... or use deadlock-free pattern
- Freedom from race-hazards
- ... and aliasing errors

The RMoX Operating System

The RMoX Operating System

■ A process-oriented OS

- built from hundreds to thousands of lightweight communicating processes
- OS components written in occam-pi, scheduled using the infrastructure described here



■ Need effective process scheduling:

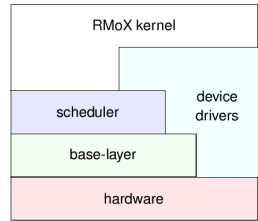
- utilising processor caches effectively
- transparently able to take advantage of multi-core and multi-processor hardware

■ Constantly changing process network

- new processes being created and old ones shut-down
- evolving network topology with moveable channel links

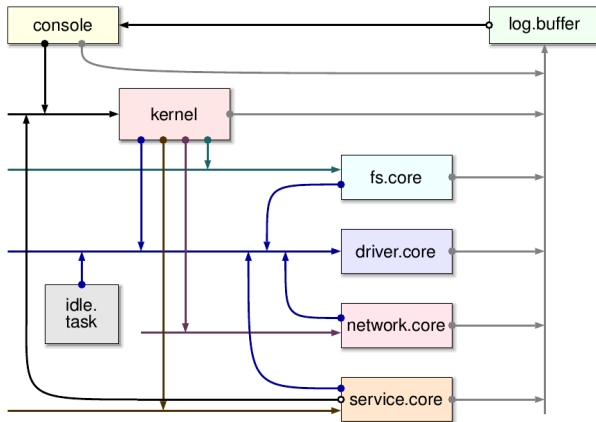
The RMoX Operating System

- A process-oriented OS
 - built from hundreds to thousands of lightweight communicating processes
 - OS components written in occam-pi, scheduled using the infrastructure described here
- Need effective process scheduling:
 - utilising processor caches effectively
 - transparently able to take advantage of multi-core and multi-processor hardware
- Constantly changing process network
 - new processes being created and old ones shut-down
 - evolving network topology with moveable channel links



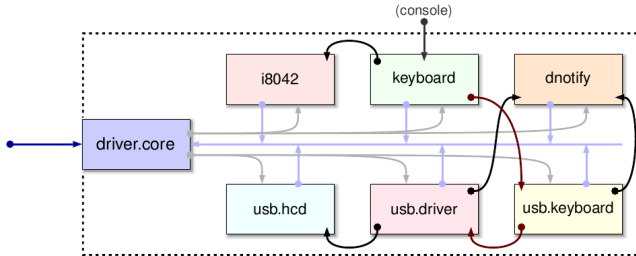
The RMoX Operating System

- Links between components are bundles of channels
 - able to carry data (and bundle-ends) in both directions



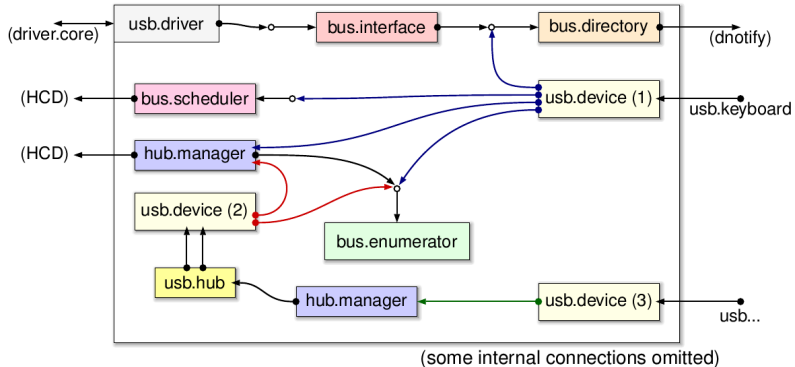
The RMoX Operating System

- Strong typing in the occam-pi language avoids many mishaps (e.g. incompatible plumbing)
 - client-server architecture gives a deadlock-free design
 - formal analysis for guaranteeing this (in progress!)

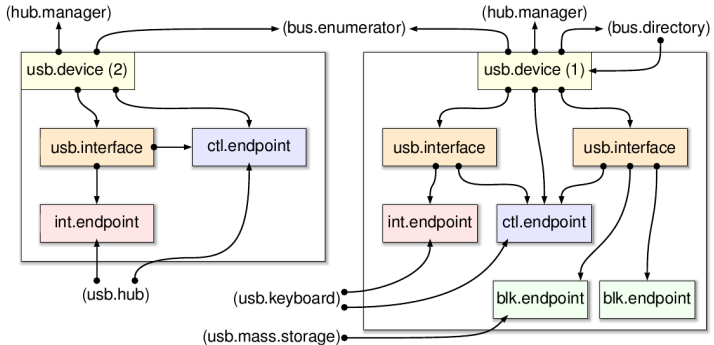


The RMoX Operating System

- Layering extends 'downwards' as necessary
 - resulting in significant numbers of parallel processes



The RMoX Operating System



- ... which communicate and interact frequently

Scheduling Process-Oriented Programs

- Process-oriented software can obviously be parallelised
 - Multiplex processes over available processors
- However, compared to typical multithreading:
 - Larger number of concurrent threads
 - A lot more inter-thread synchronisation

Scheduling Process-Oriented Programs

- Process-oriented software can obviously be parallelised
 - Multiplex processes over available processors
- However, compared to typical multithreading:
 - Larger number of concurrent threads
 - A lot more inter-thread synchronisation

Existing Frameworks

- Other frameworks for concurrency not ideal for *POP*
- Too heavyweight:
 - POSIX Threads
- Lack communication primitives:
 - Cilk
 - Java Fork/Join
 - Intel Thread Building Blocks
 - OpenMP
- Constrain communication patterns:
 - Brook
 - StreamIT

Existing Frameworks

- Other frameworks for concurrency not ideal for *POP*
- Too heavyweight:
 - POSIX Threads
- Lack communication primitives:
 - Cilk
 - Java Fork/Join
 - Intel Thread Building Blocks
 - OpenMP
- Constrain communication patterns:
 - Brook
 - StreamIT

Existing Frameworks

- Other frameworks for concurrency not ideal for *POP*
- Too heavyweight:
 - POSIX Threads
- Lack communication primitives:
 - Cilk
 - Java Fork/Join
 - Intel Thread Building Blocks
 - OpenMP
- Constrain communication patterns:
 - Brook
 - StreamIT

Existing Frameworks

- Other frameworks for concurrency not ideal for *POP*
- Too heavyweight:
 - POSIX Threads
- Lack communication primitives:
 - Cilk
 - Java Fork/Join
 - Intel Thread Building Blocks
 - OpenMP
- Constrain communication patterns:
 - Brook
 - StreamIT

How?

- Processes:
 - Use cooperative scheduling
 - Processes manage their own state (on stack)
 - Scheduler needs < 8 words of memory per process
- Communication:
 - Synchronous and unbuffered channels
 - Constrains memory
 - Aids reasoning (CSP, etc)

How?

- Processes:
 - Use cooperative scheduling
 - Processes manage their own state (on stack)
 - Scheduler needs < 8 words of memory per process
- Communication:
 - Synchronous and unbuffered channels
 - Constrains memory
 - Aids reasoning (CSP, etc)

How?

- Processes:
 - Use cooperative scheduling
 - Processes manage their own state (on stack)
 - Scheduler needs < 8 words of memory per process
- Communication:
 - Synchronous and unbuffered channels
 - Constrains memory
 - Aids reasoning (CSP, etc)

Context Switching

- Assuming very fast scheduling (e.g. round-robin)
- ... and we can context switch in $< 100\text{ns}$...
- ... that's 10,000,000 potential switches a second
- Rapid switching \implies bad cache utilisation
- Modern architectures depend on cache for performance

Context Switching

- Assuming very fast scheduling (e.g. round-robin)
- ... and we can context switch in $< 100\text{ns}$...
- ... that's 10,000,000 potential switches a second
- Rapid switching \implies bad cache utilisation
- Modern architectures depend on cache for performance

Context Switching

- Assuming very fast scheduling (e.g. round-robin)
- ... and we can context switch in $< 100\text{ns}$...
- ... that's 10,000,000 potential switches a second
- Rapid switching \implies bad cache utilisation
- Modern architectures depend on cache for performance

Locality

- Improve temporal locality (reduce cache pressure)
- Constrain context switching to group processes, *batches*
- Batch:
 - Group of processes
 - Repeatedly executed
 - Build cache footprint
 - ... and reuse it

Locality

- Improve temporal locality (reduce cache pressure)
- Constrain context switching to group processes, *batches*
- Batch:
 - Group of processes
 - Repeatedly executed
 - Build cache footprint
 - ... and reuse it

Batch Formation

- How do we form batches?
- Unrelated processes? (K.Vella - PDPTA 2002)
- Related parts of process network:
 - Determine groups dynamically by communication
 - Reduce large groups based on heuristic

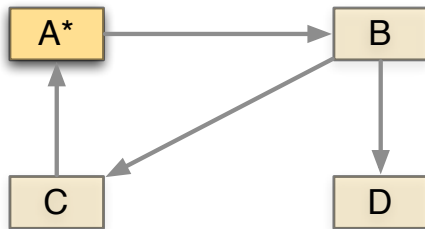
Batch Formation

- How do we form batches?
- Unrelated processes? (K.Vella - PDPTA 2002)
- Related parts of process network:
 - Determine groups dynamically by communication
 - Reduce large groups based on heuristic

Batch Formation

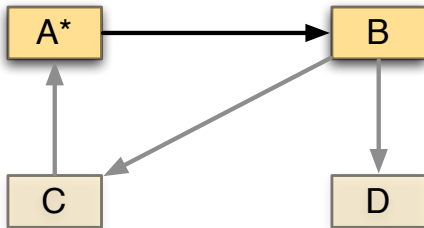
- How do we form batches?
- Unrelated processes? (K.Vella - PDPTA 2002)
- Related parts of process network:
 - Determine groups dynamically by communication
 - Reduce large groups based on heuristic

Batch Formation



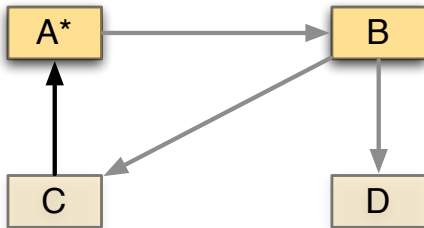
queue =

Batch Formation



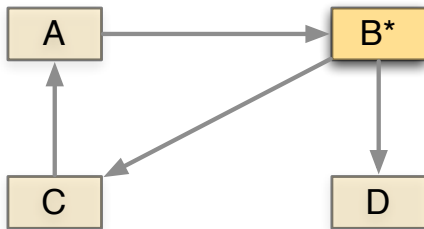
queue = B

Batch Formation



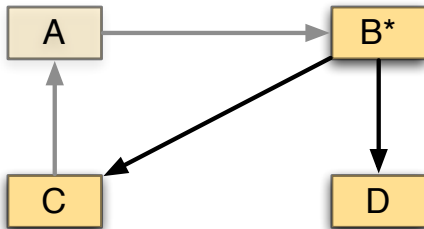
queue = B

Batch Formation



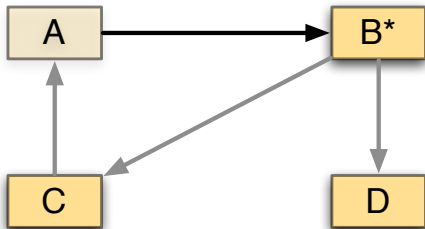
queue =

Batch Formation



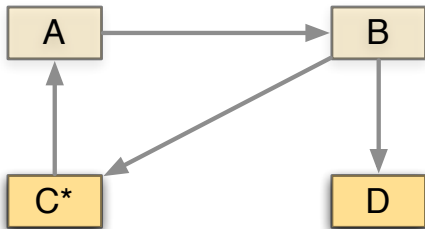
queue = C, D

Batch Formation



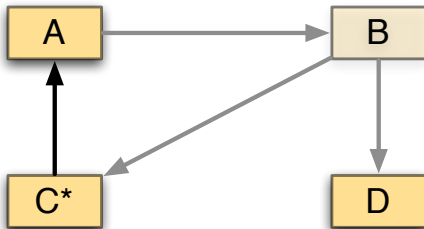
queue = C, D

Batch Formation



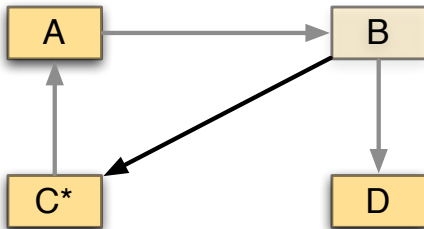
queue = D

Batch Formation



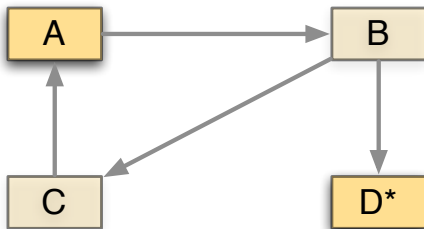
queue = D, A

Batch Formation



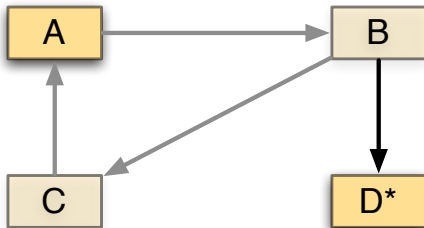
queue = D, A

Batch Formation



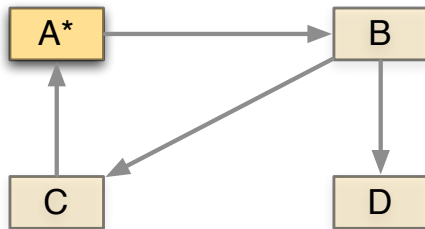
queue = A

Batch Formation



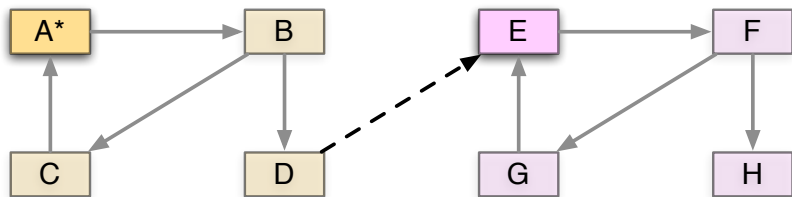
queue = A

Batch Formation



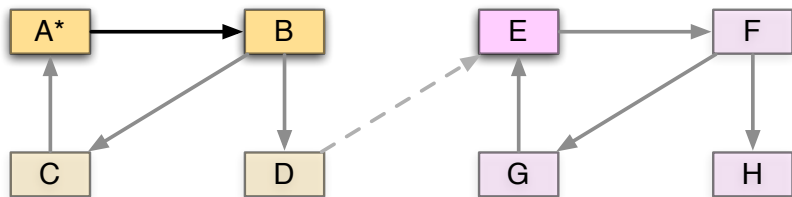
queue =

Batch Formation 2



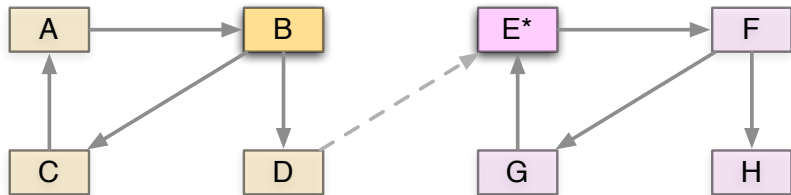
queue = E

Batch Formation 2



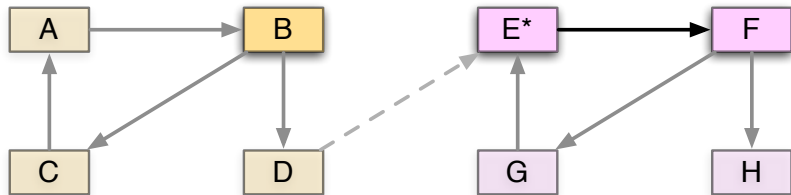
queue = E, B

Batch Formation 2



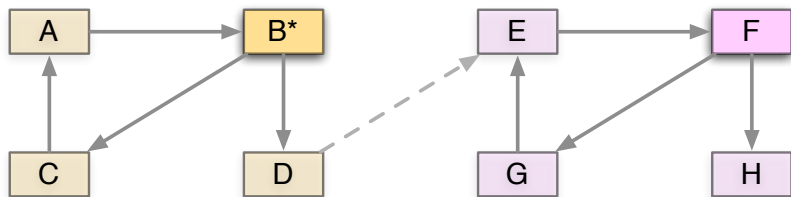
queue = B

Batch Formation 2



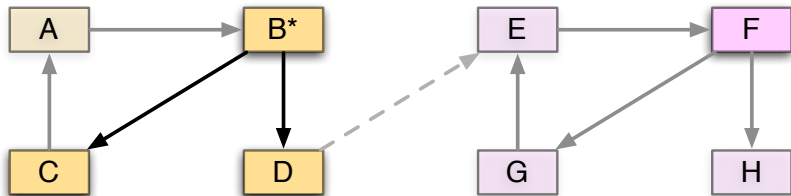
queue = B, F

Batch Formation 2



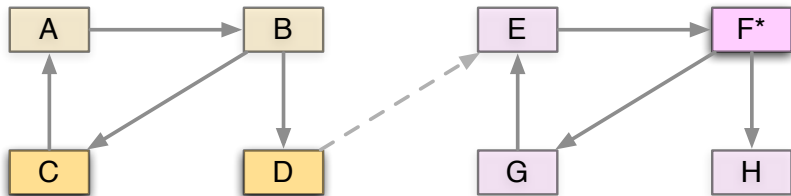
queue = F

Batch Formation 2



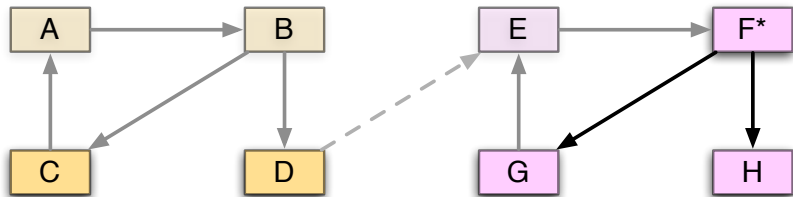
queue = F, C, D

Batch Formation 2



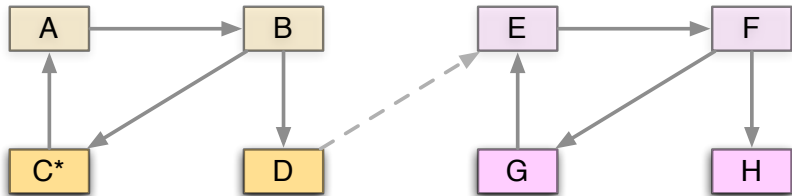
queue = C, D

Batch Formation 2



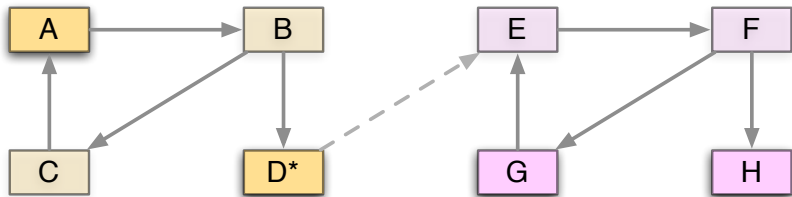
queue = C, D, G, H

Batch Formation 2



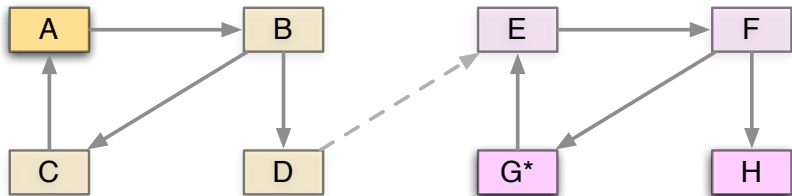
queue = D, G, H

Batch Formation 2



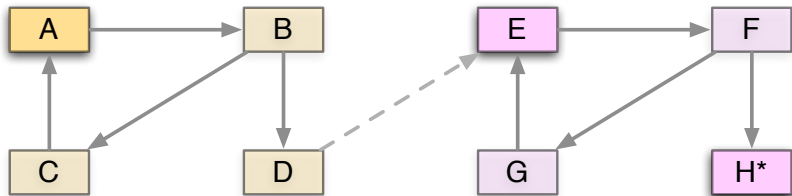
queue = G, H, A

Batch Formation 2



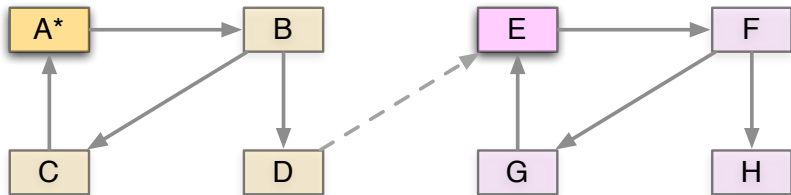
queue = H, A

Batch Formation 2



queue = A, E

Batch Formation 2



queue = E

Batching

- Form temporally isolated sub-networks in to batches
- Communication draws processes into batch
- Lack of dependency splits batch

Batching

- Form temporally isolated sub-networks in to batches
- Communication draws processes into batch
- Lack of dependency splits batch

Multicore

- Batches provide work units
- How do we distribute them between processors?
- Single run-queue?
 - Enqueue/dequeue \implies lock/transaction/etc
 - Single point of contention
 - Becomes bottleneck as number of processors rises

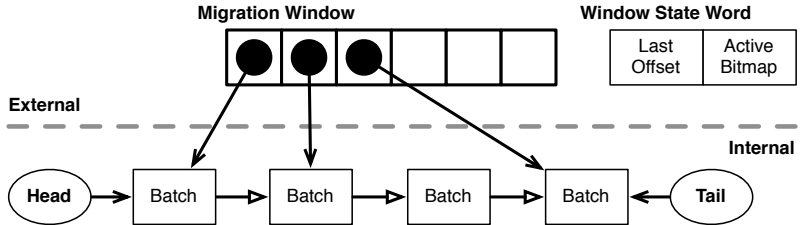
Multicore

- Batches provide work units
- How do we distribute them between processors?
- Single run-queue?
 - Enqueue/dequeue \implies lock/transaction/etc
 - Single point of contention
 - Becomes bottleneck as number of processors rises

Distributed Run-queues

- Processors run scheduler instances
 - Schedulers independent
- Batches are exposed via a migration window
- Idle schedulers steal batches

Migration Window

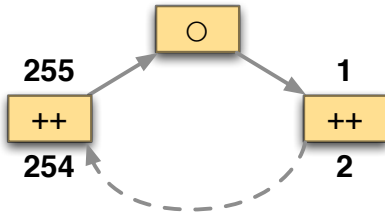


Minimising Atomic Operation Usage

- Migration Window
 - wait-free enqueue (1), dequeue (1), steal batch (2+)
- Communication
 - average 1, worst-case 2 atomic operations
- Choice over events (ALT)
 - maximum 4 operations per event, typically less
- Mutual exclusion
 - typically 1 atomic operation for lock/unlock
- Barriers
 - typically 1 atomic operation per synchronising process

Communication Time

- Use ring of processes to calculate communication time:



Communication Time

- Use ring of processes to calculate communication time:

Implementation	1-core (ns)	8-core (ns)
CCSP occam- π	46	39
CCSP C	73	75
Haskell	269	9892
Erlang	1697	1675
pthread	5013	3485
JCSP	7723	14905

The CoSMoS Project



The CoSMoS Project

- **Complex Systems Modelling and Simulation**
- Reusable best-practice techniques: the CoSMoS Process
- Case studies
 - Textbook examples: boid flocking, ants, AIS
 - Real research problems: lymphocyte migration, plant modelling, emergent behaviour in robot swarms
- Simulations built using process-oriented techniques in occam- π and JCSP
 - A good way to model systems that are by their nature massively concurrent – and may be distributed
 - Agents are processes, space is a network of processes. . .

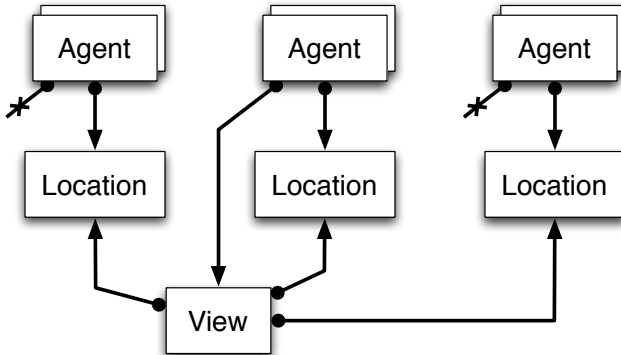
CoSMoS Inspired Benchmark

- Space divided into grid of *location* processes
- *Agent* processes avoid each other
- n-body problem where each body has internal personality:
 - Affects movement
 - Is updated based on position and encountered agents

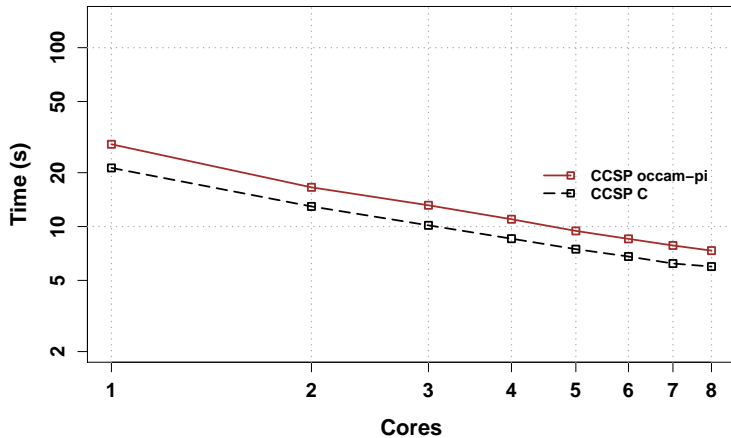
CoSMoS Inspired Benchmark

- Space divided into grid of *location* processes
- *Agent* processes avoid each other
- n-body problem where each body has internal personality:
 - Affects movement
 - Is updated based on position and encountered agents

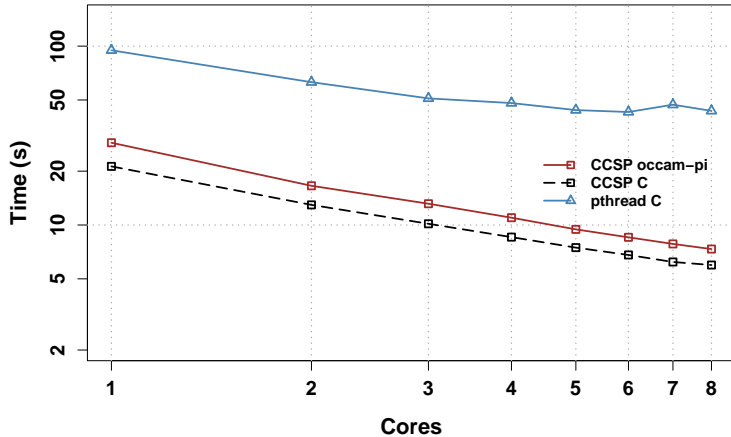
Process Network



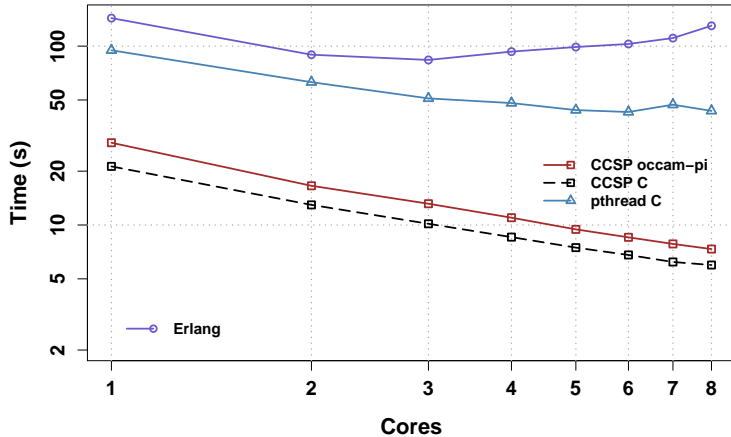
Increasing Cores



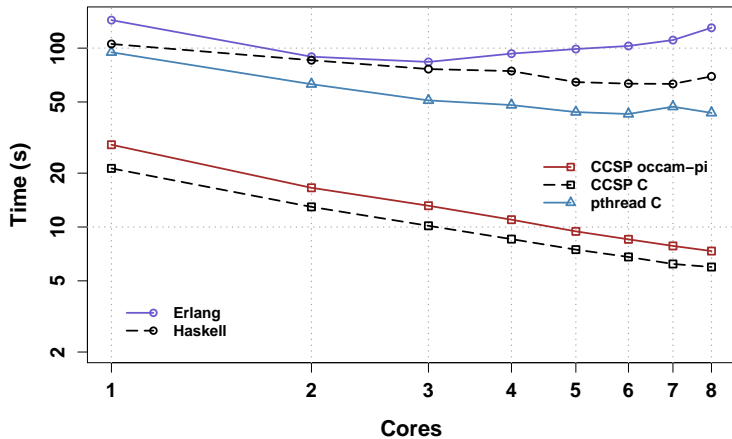
Increasing Cores



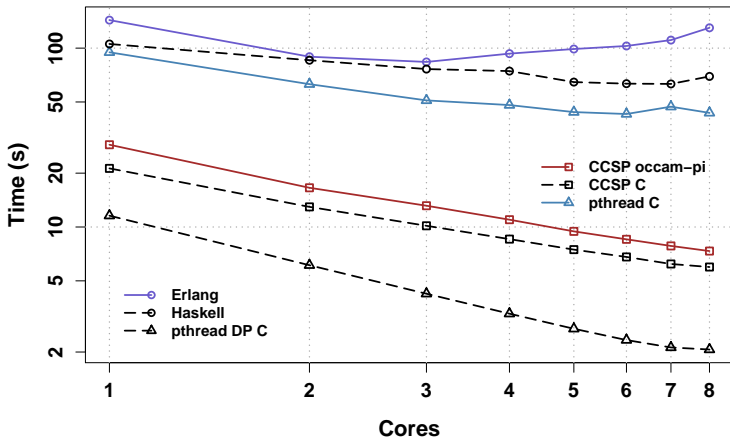
Increasing Cores



Increasing Cores



Increasing Cores



Conclusions

- Implemented and working
 - In use by CoSMoS and RMoX EPSRC funded projects
- Automatic parallelisation of concurrent software
 - No programmer intervention
 - Dynamic cache locality enhancement
 - *n*-core, scalable, asymmetric scheduling

Conclusions

- Implemented and working
 - In use by CoSMoS and RMoX EPSRC funded projects
- Automatic parallelisation of concurrent software
 - No programmer intervention
 - Dynamic cache locality enhancement
 - *n*-core, scalable, asymmetric scheduling

Future Work

- Scheduler improvements
 - Pipeline parallelisation
 - NUMA support
 - Memory management integration
- Portability
 - LLVM code generation
 - New language compiler

Future Work

- Scheduler improvements
 - Pipeline parallelisation
 - NUMA support
 - Memory management integration
- Portability
 - LLVM code generation
 - New language compiler

Questions?

- Questions...?
- Acknowledgements:
 - thank-you to the anonymous reviewers who provided valuable, insightful feedback
 - funded by EPSRC (UK), grant EP/D061822
- Formal Methods Week - CPA Conference (Nov. 2009):
 - <http://www.wotug.org/cpa2009/>

 Engineering and Physical Sciences
Research Council



<http://www.cs.kent.ac.uk/research/groups/sys/>

 University of
Kent  Computing