Multicore Scheduling for Lightweight Communicating Processes

Carl Ritson

Computing Laboratory
University of Kent
UK

17th October 2008

University of Kent | Computing

Motivation

- `http://www.cosmos-research.org`
  - Large-scale process-oriented models
  - Thousands to millions of agents (processes)
  - Real-time and interactive
- Demo

University of Kent | Computing

Motivation

- `http://www.cosmos-research.org`
  - Large-scale process-oriented models
  - Thousands to millions of agents (processes)
  - Real-time and interactive
- Demo

University of Kent | Computing

Process-Oriented Programming (POP)

- Processes
  - Encapsulate state and code (like objects?)
  - Execute concurrently with other processes
- Communication
  - Processes synchronise and exchange data
  - Share distributed state to accomplish tasks
- Concurrent processes $\Longrightarrow$ parallel execution potential

Process-Oriented Programming (POP)

- Processes
  - Encapsulate state and code (like objects?)
  - Execute concurrently with other processes
- Communication
  - Processes synchronise and exchange data
  - Share distributed state to accomplish tasks
- Concurrent processes $\Longrightarrow$ parallel execution potential

Process-Oriented Programming (POP)

- Processes
  - Encapsulate state and code (like objects?)
  - Execute concurrently with other processes
- Communication
  - Processes synchronise and exchange data
  - Share distributed state to accomplish tasks
- Concurrent processes $\Longrightarrow$ parallel execution potential

Efficiency

- Process-oriented software can be parallelised
- . . . but can it be used to develop efficient software?
- Yes.
- If:
  - Processes are lightweight
    - i.e. thousands of them with little memory
  - Communication is cheap
    - i.e. comparable to a method call

Efficiency

- Process-oriented software can be parallelised
- . . . but can it be used to develop efficient software?
- Yes.
- If:
  - Processes are lightweight
    - i.e. thousands of them with little memory
  - Communication is cheap
    - i.e. comparable to a method call

Efficiency

- Process-oriented software can be parallelised
- . . . but can it be used to develop efficient software?
- Yes.
- If:
  - Processes are lightweight
    - i.e. thousands of them with little memory
  - Communication is cheap
    - i.e. comparable to a method call

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)

Numbers - BIG and small

- Context switching:
  - No room for complex scheduling
  - Hybrid round-robin with priority
- If we can context switch in 100ns . . .
- . . . that's 10,000,000 potential switches a second
- Rapid switching $\Longrightarrow$ bad cache utilisation
- Modern architectures depend on cache for performance

Numbers - BIG and small

- Context switching:
  - No room for complex scheduling
  - Hybrid round-robin with priority
- If we can context switch in 100ns . . .
- . . . that's 10,000,000 potential switches a second
- Rapid switching $\Longrightarrow$ bad cache utilisation
- Modern architectures depend on cache for performance

Numbers - BIG and small

- Context switching:
  - No room for complex scheduling
  - Hybrid round-robin with priority
- If we can context switch in 100ns . . .
- . . . that's 10,000,000 potential switches a second
- Rapid switching $\Longrightarrow$ bad cache utilisation
- Modern architectures depend on cache for performance

Numbers - BIG and small

- Context switching:
  - No room for complex scheduling
  - Hybrid round-robin with priority
- If we can context switch in 100ns . . .
- . . . that's 10,000,000 potential switches a second
- Rapid switching $\Longrightarrow$ bad cache utilisation
- Modern architectures depend on cache for performance

What is a batch?

- Kevin Vella suggested sequential batching as solution
- Batch:
  - Group of processes
  - Repeatedly executed
  - Build cache footprint
  - . . . and reuse it

What is a batch?

- Kevin Vella suggested sequential batching as solution
- Batch:
  - Group of processes
  - Repeatedly executed
  - Build cache footprint
  - . . . and reuse it

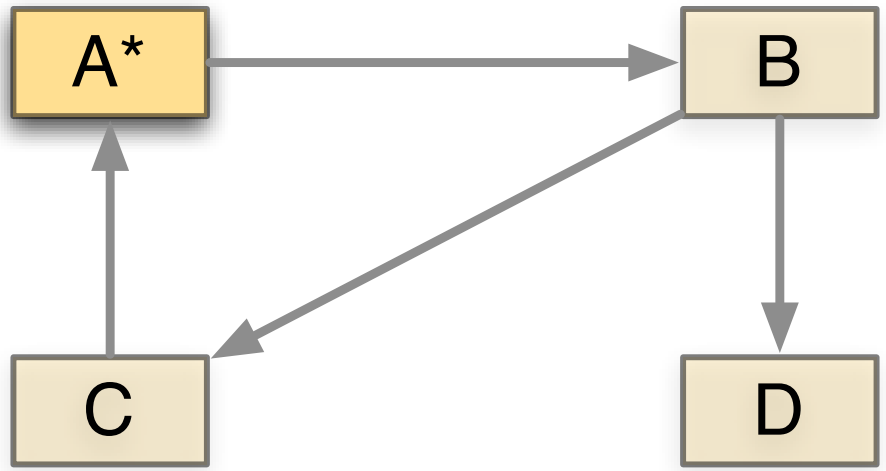
Batch Formation

- How do we form a batch?
  - Unrelated processes? (Vella)
- Related parts of process network:
  - Determine groups dynamically by communication
  - Reduce large groups based on heuristic

Batch Formation

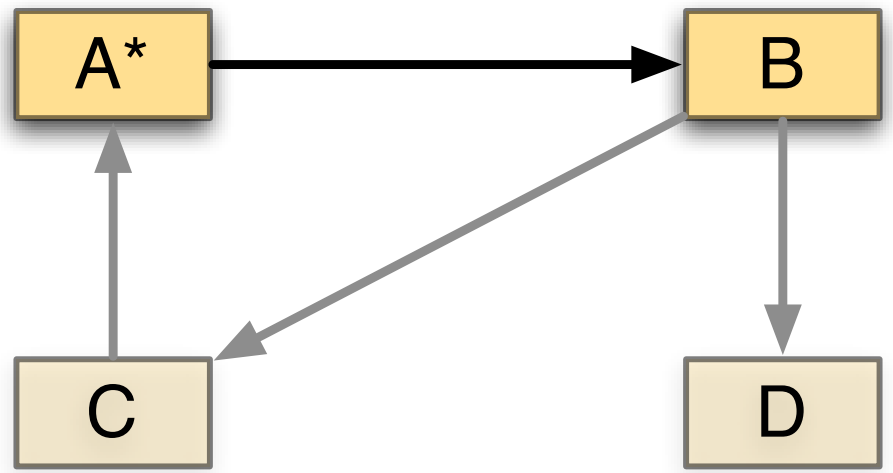
- How do we form a batch?
  - Unrelated processes? (Vella)
- Related parts of process network:
  - Determine groups dynamically by communication
  - Reduce large groups based on heuristic

Batch Formation



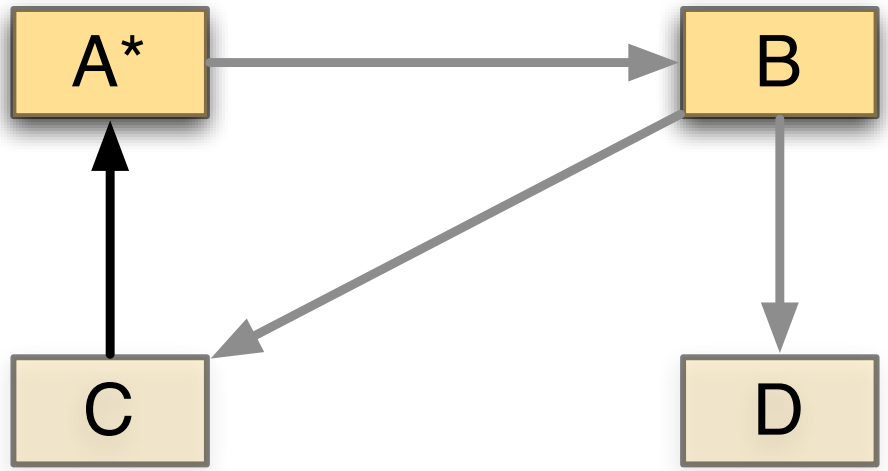
queue =

Batch Formation



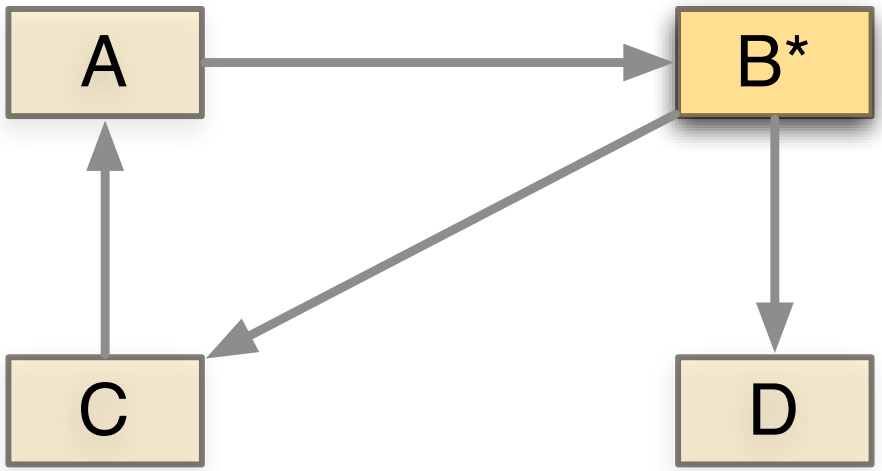
queue = B

Batch Formation



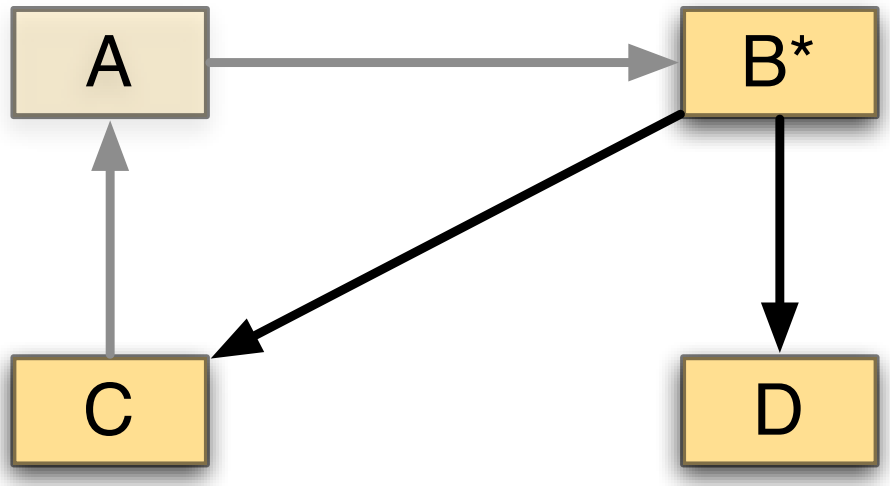
queue = B

Batch Formation



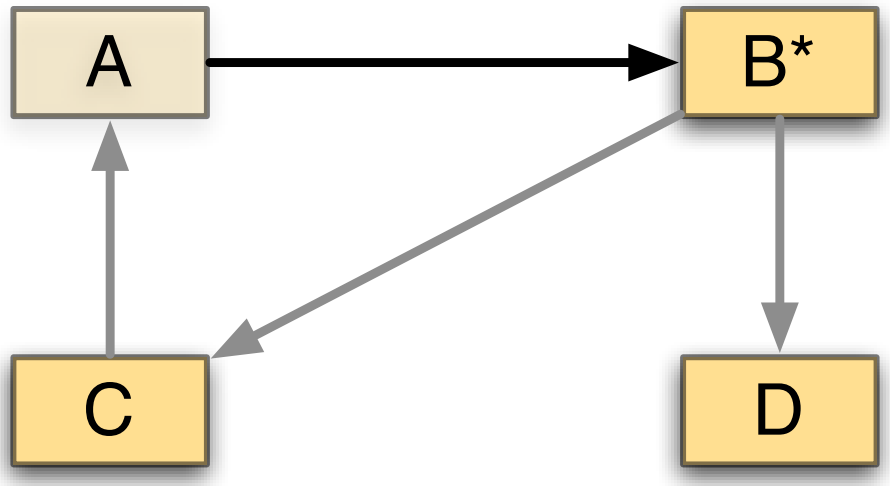
queue =

Batch Formation



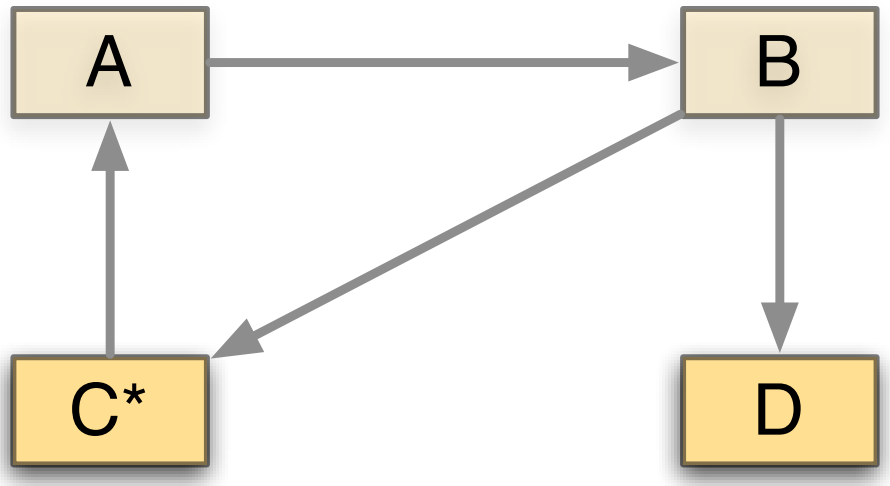
queue = C, D

Batch Formation



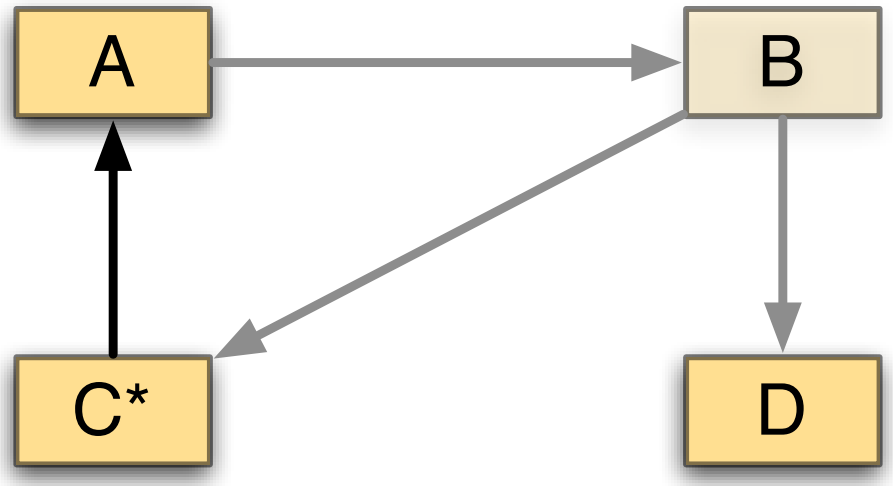
queue = C, D

Batch Formation



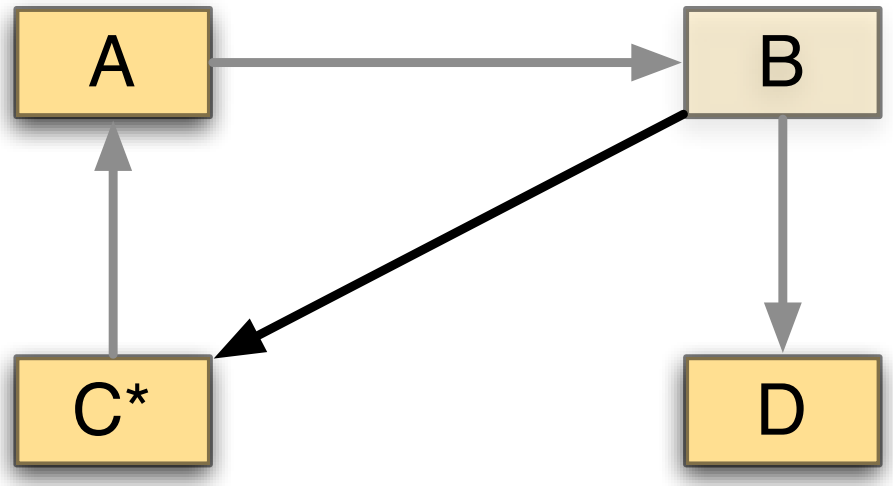
queue = D

Batch Formation



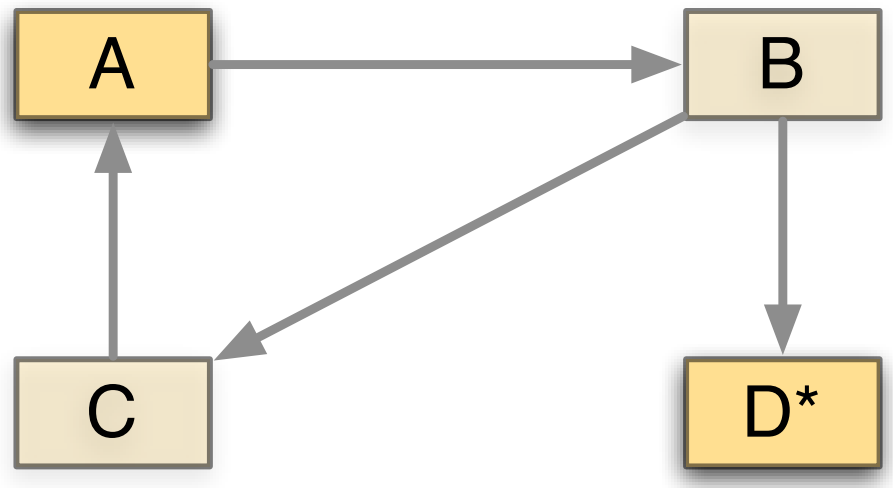
queue = D, A

Batch Formation



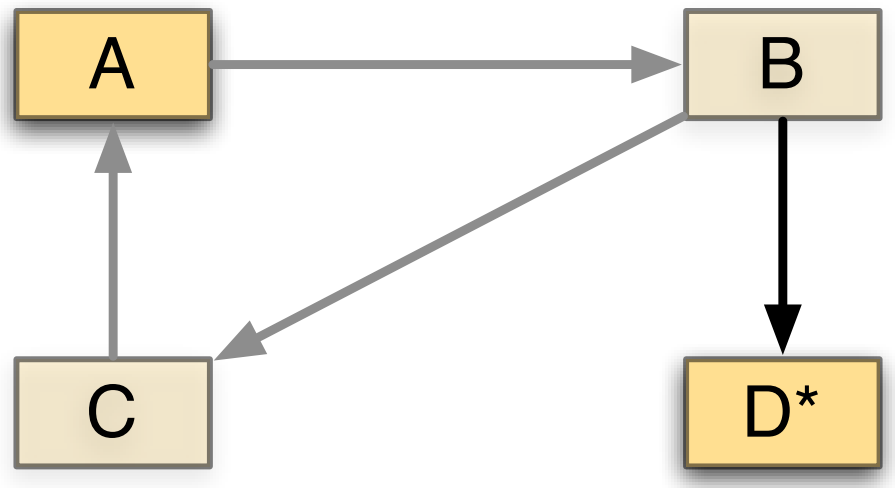
queue = D, A

Batch Formation



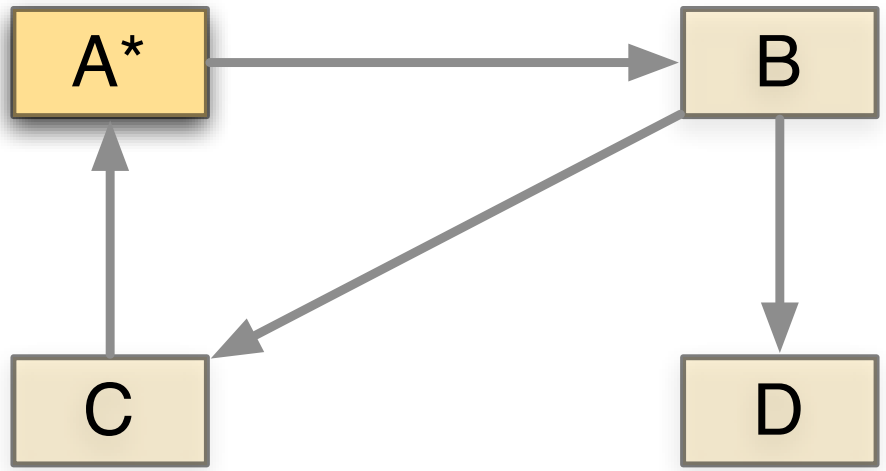
queue = A

Batch Formation



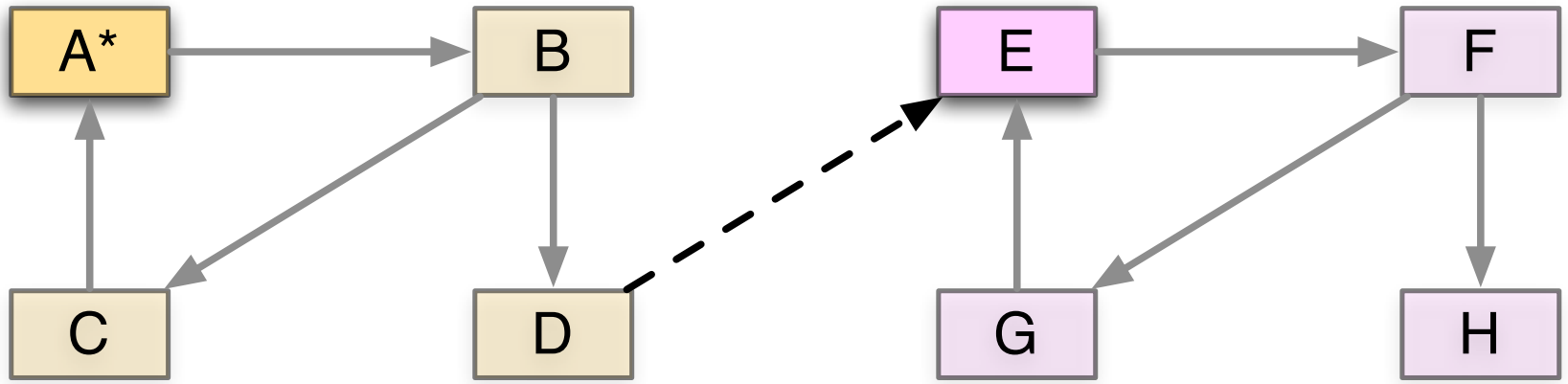
queue = A

Batch Formation



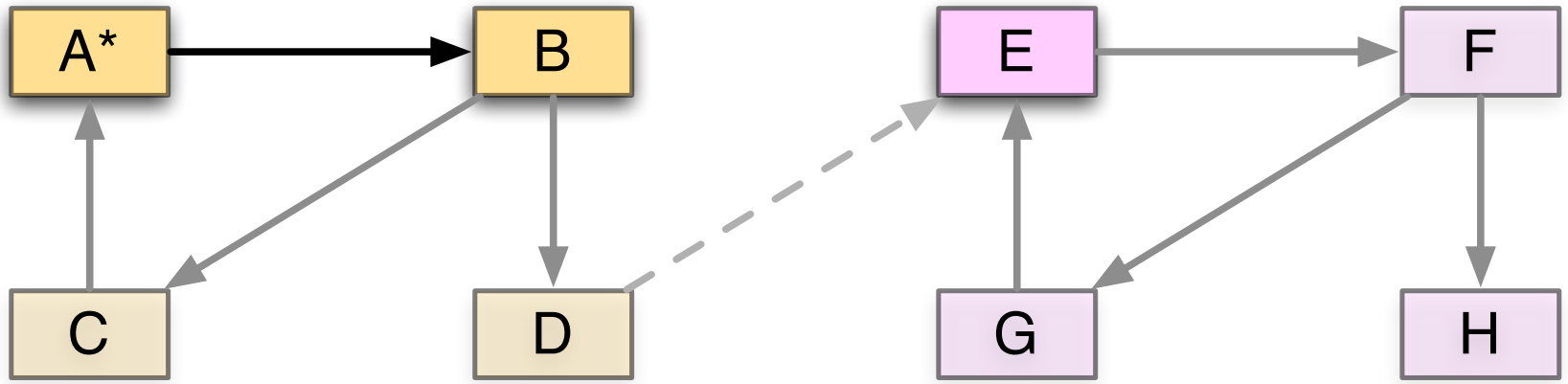
queue =

Batch Formation 2



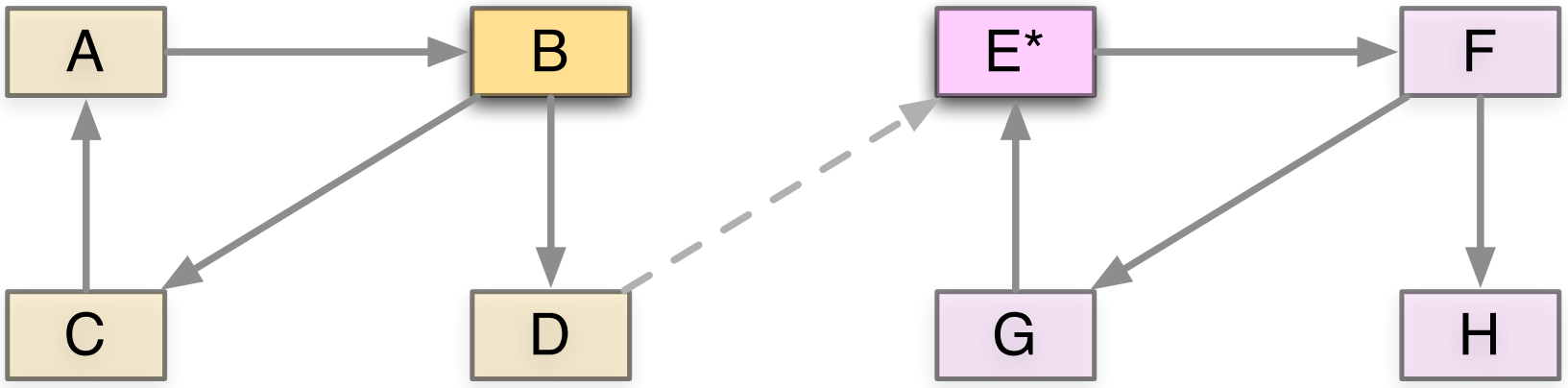
queue = E

Batch Formation 2



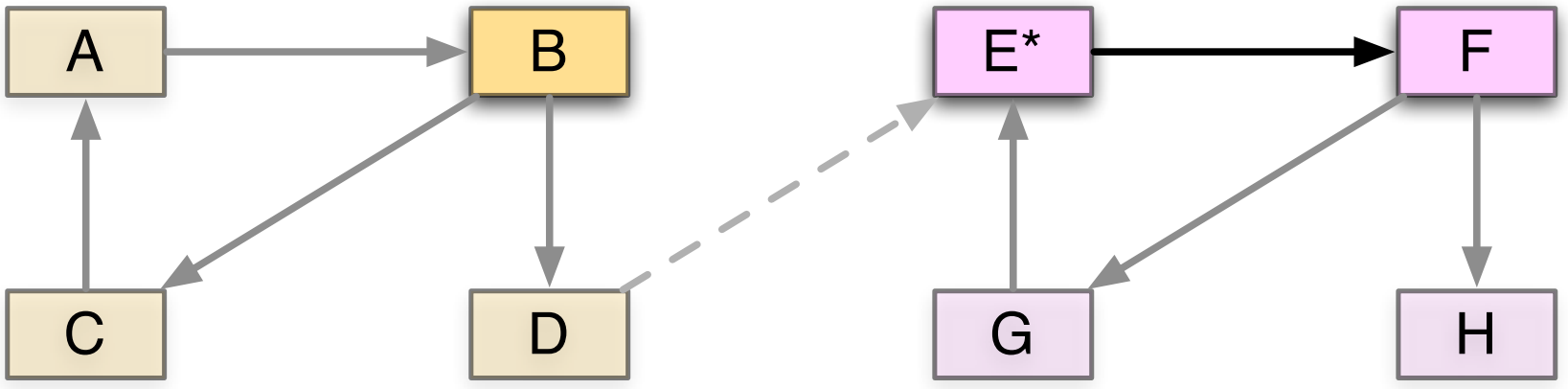
queue = E, B

Batch Formation 2



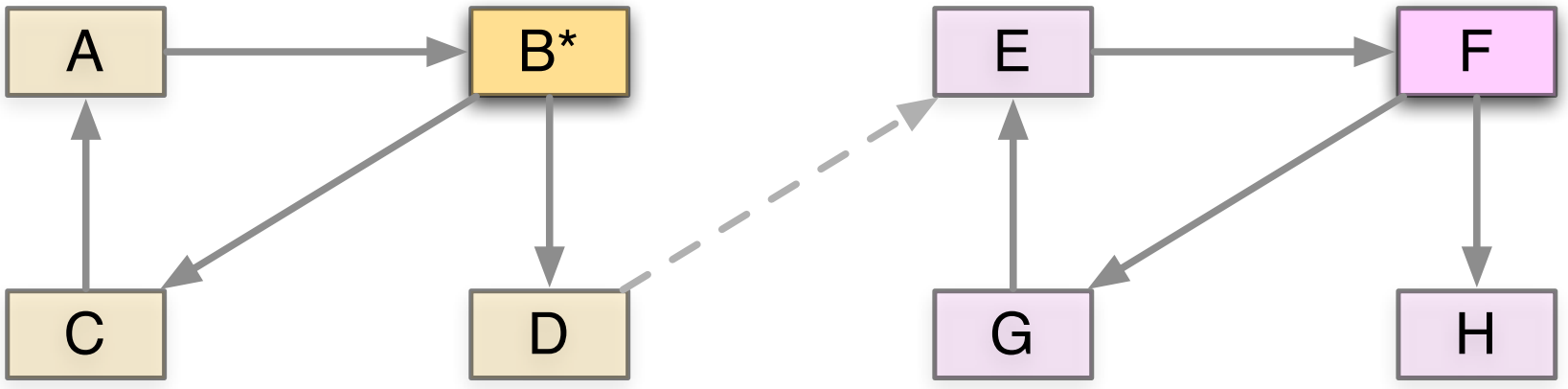
queue = B

Batch Formation 2



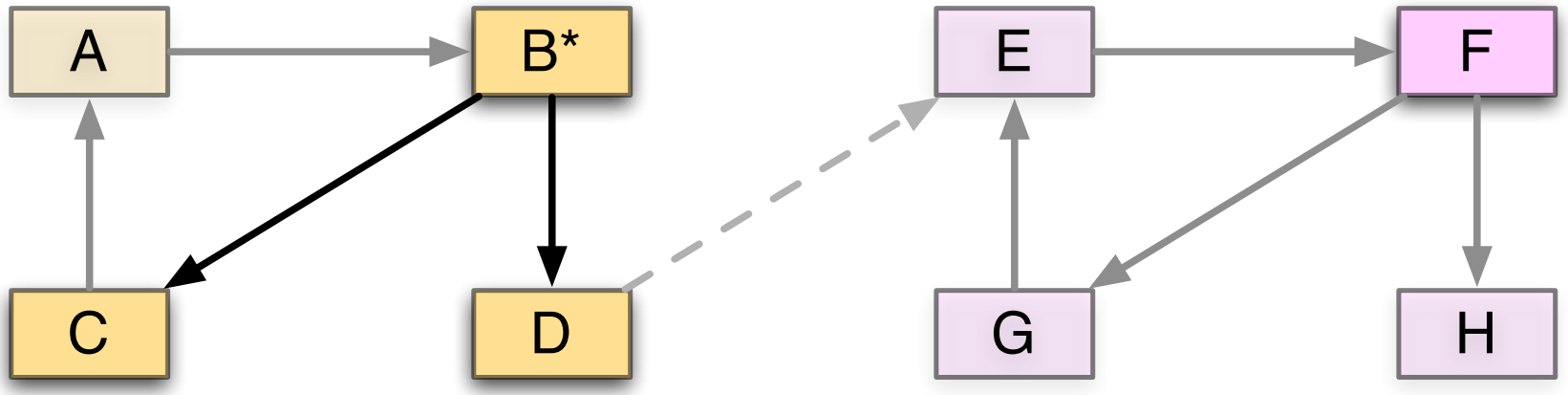
queue = B, F

Batch Formation 2



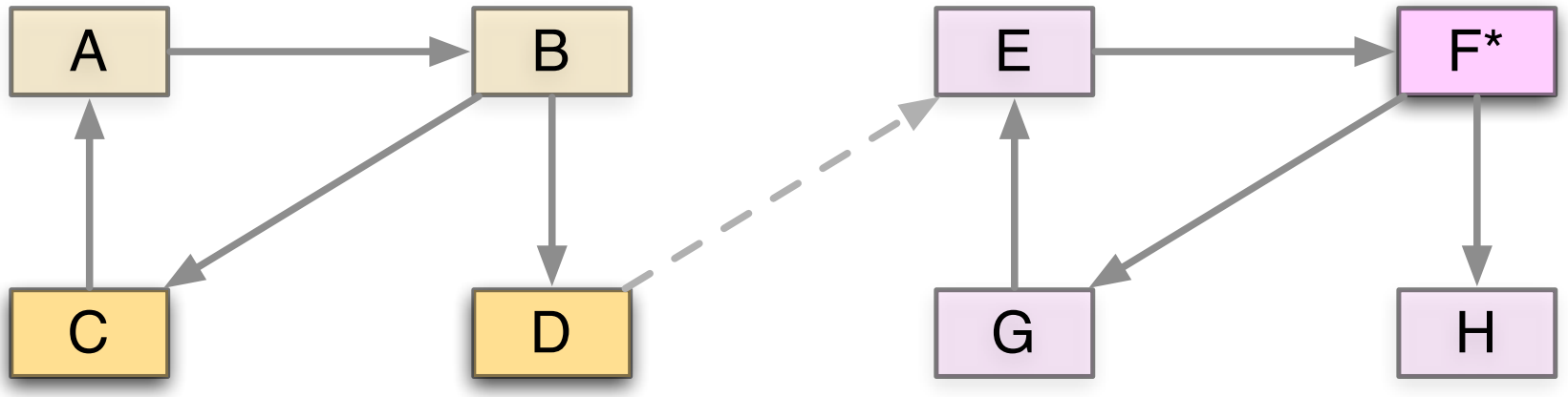
queue = F

Batch Formation 2



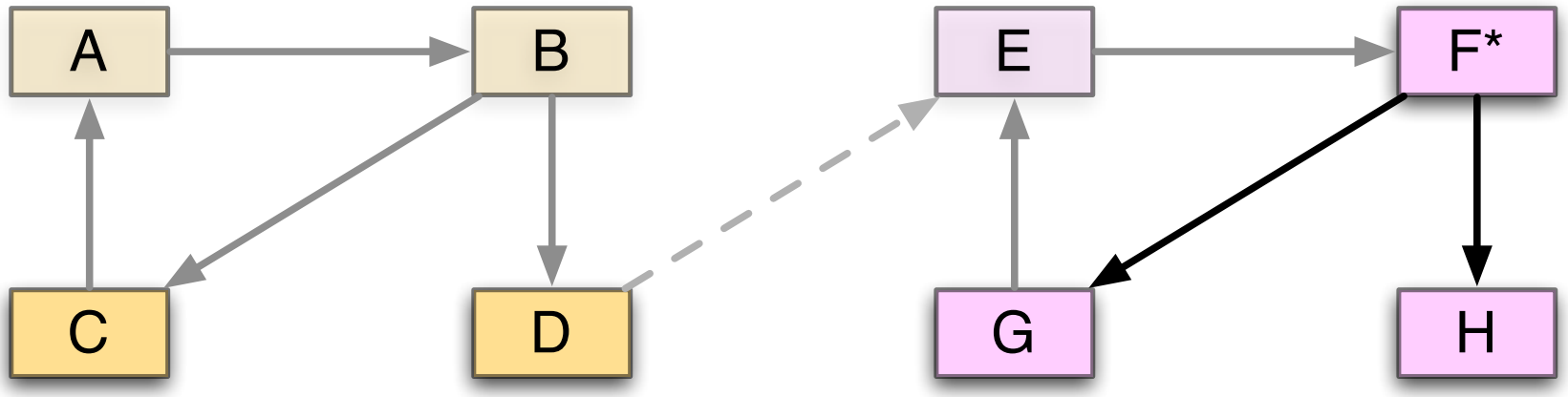
queue = F, C, D

Batch Formation 2



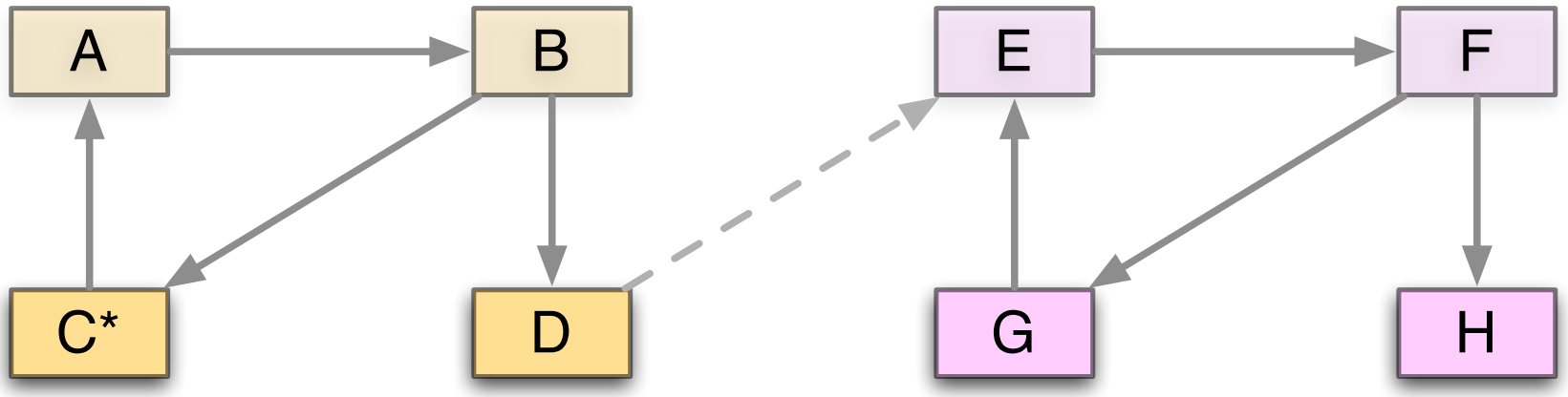
queue = C, D

Batch Formation 2



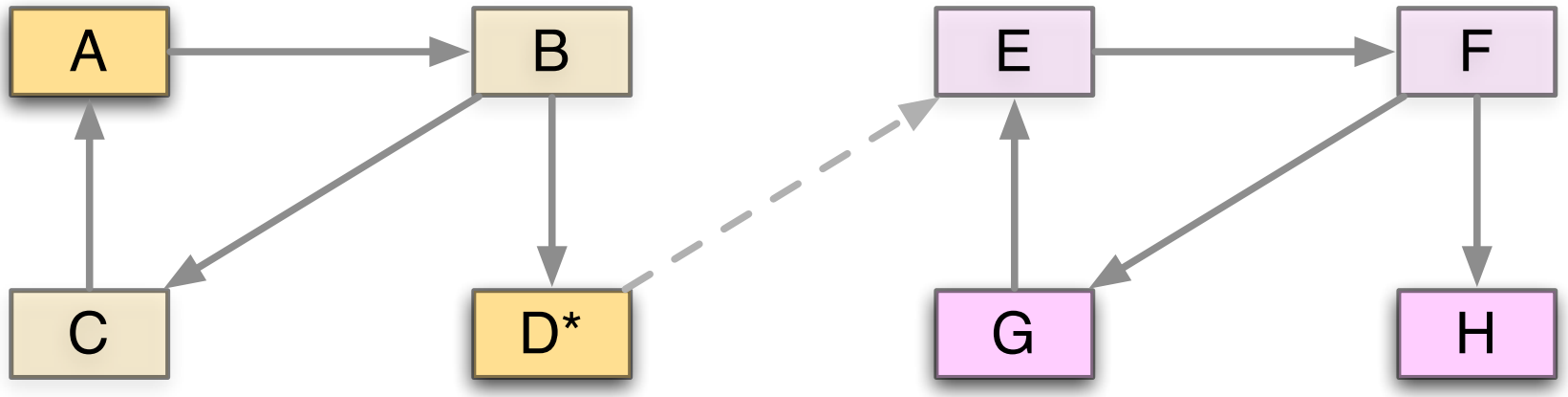
queue = C, D, G, H

Batch Formation 2



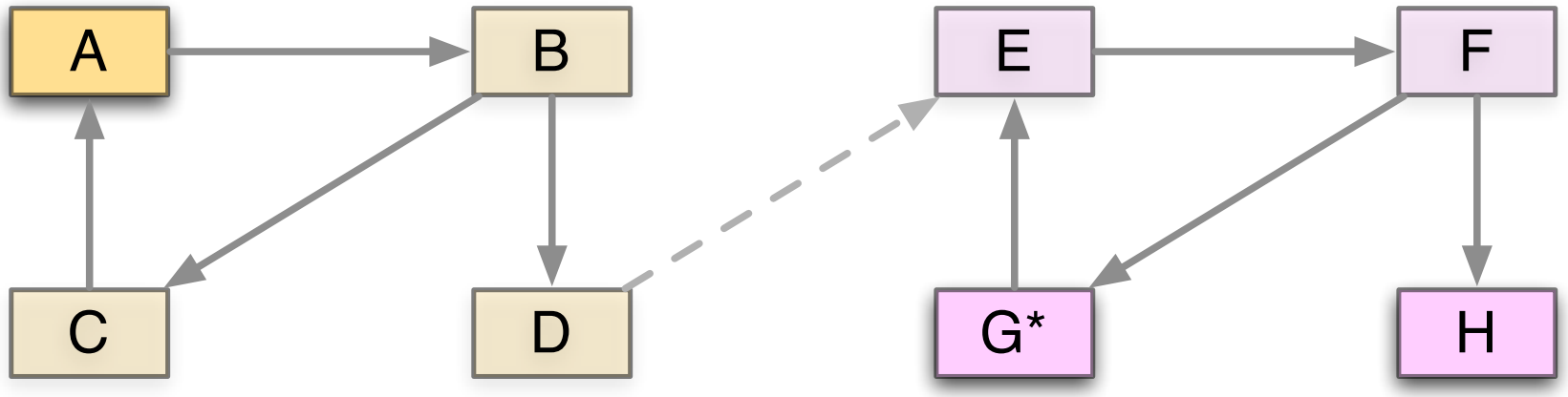
queue = D, G, H

Batch Formation 2



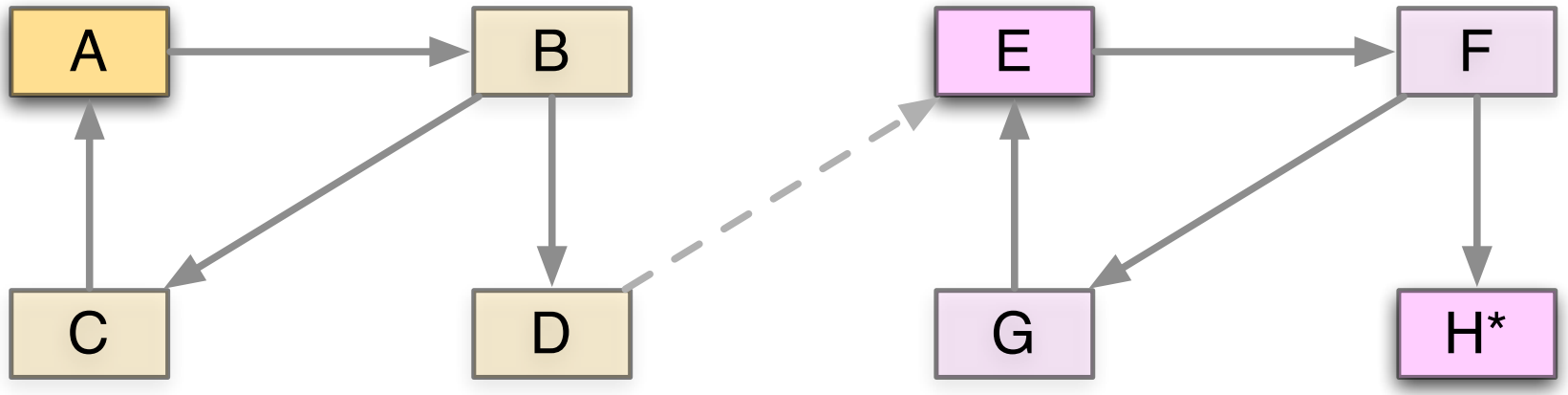
queue = G, H, A

Batch Formation 2



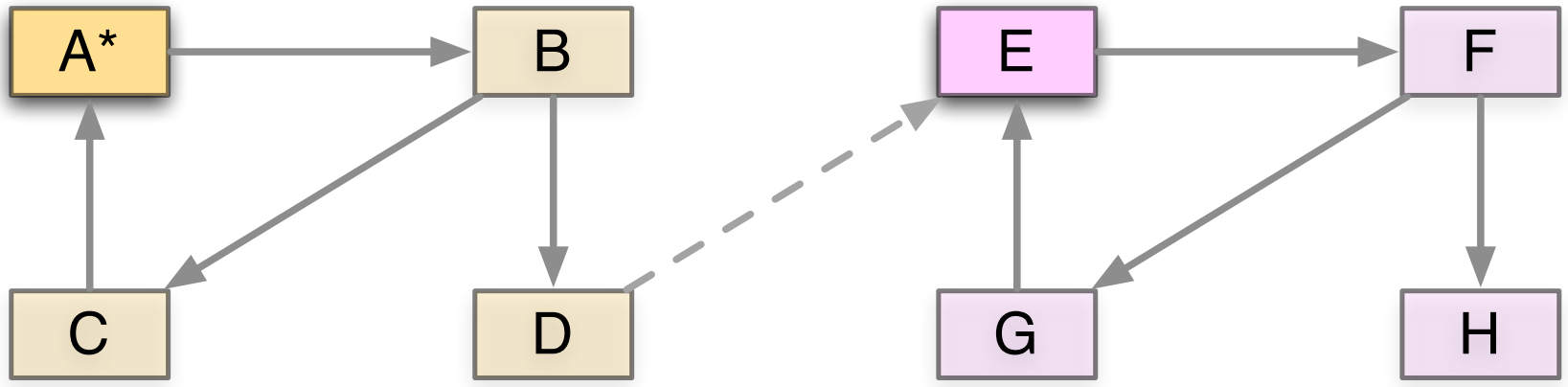
queue = H, A

Batch Formation 2



queue = A, E

Batch Formation 2



queue = E

Batching

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

Batching

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

Multicore

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

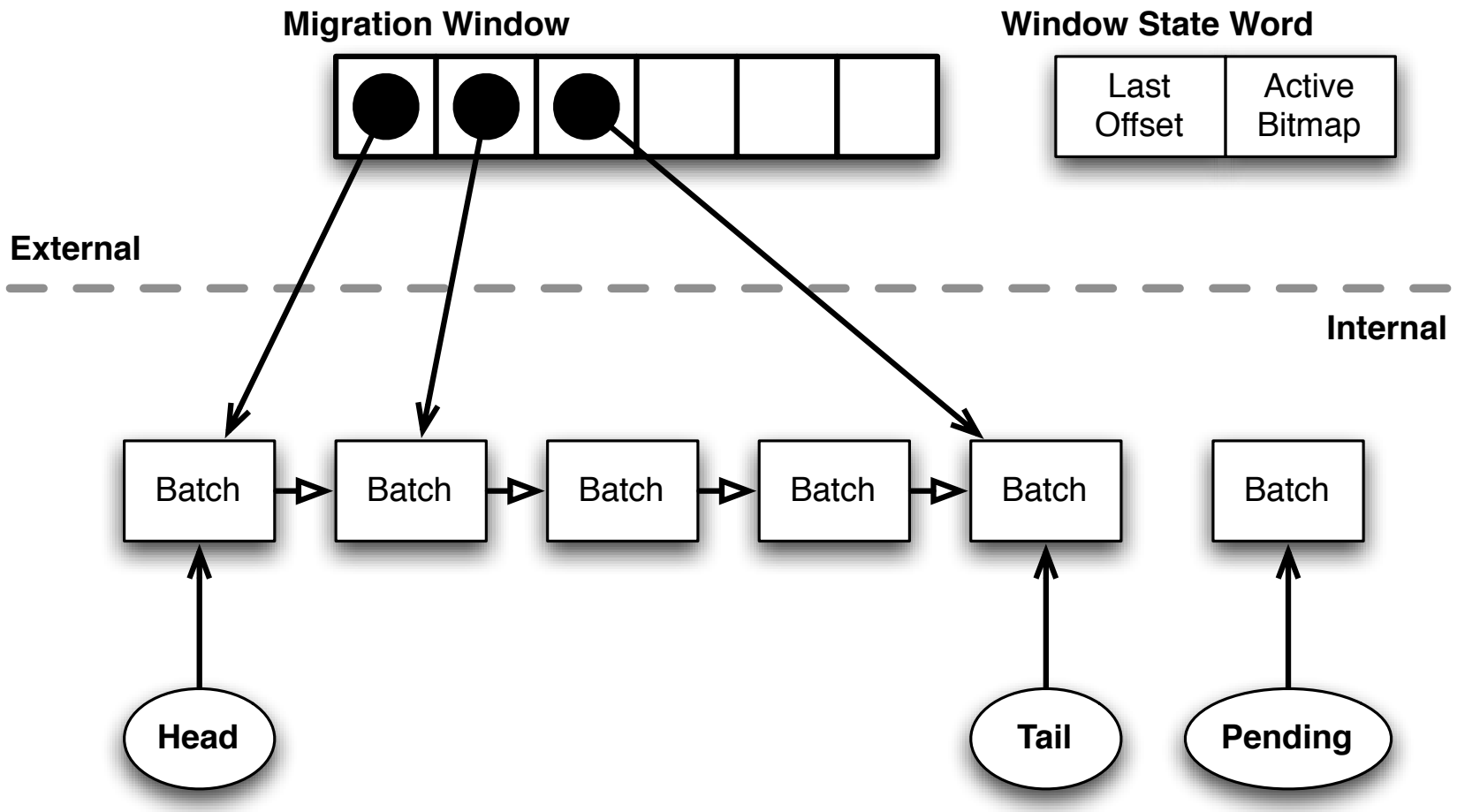
Multicore

- Batches provide work units
- How do we distribute them between processors?
- Single run-queue?
  - Enqueue/dequeue $\Longrightarrow$ 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



Lock Freedom

- *Non-blocking*:
  - cannot be indefinitely blocked by other threads
- *Lock-free*:
  - non-blocking (by inactive process), but interference possible
- *Wait-free*:
  - bounded number of steps regardless of other threads
- All scheduler algorithms, wait-free or lock-free

Lock Freedom

- *Non-blocking*:
  - cannot be indefinitely blocked by other threads
- *Lock-free*:
  - non-blocking (by inactive process), but interference possible
- *Wait-free*:
  - bounded number of steps regardless of other threads
- All scheduler algorithms, wait-free or lock-free

Atomic Operations

- Compare-and-swap
- Load-link and store-conditional
- Pipeline flush on most modern processors $\Rightarrow$ expensive
  - i.e. tens of nanoseconds
- Minimise their use
  - If we want fast scheduling and communication

Atomic Operations

- Compare-and-swap
- Load-link and store-conditional
- Pipeline flush on most modern processors $\Rightarrow$ expensive
  - i.e. tens of nanoseconds
- Minimise their use
  - If we want fast scheduling and communication

Atomic Operations

- Compare-and-swap
- Load-link and store-conditional
- Pipeline flush on most modern processors $\Rightarrow$ expensive
  - i.e. tens of nanoseconds
- Minimise their use
  - If we want fast scheduling and communication

Minimising Atomic Operation Usage

- Migration Window
  - wait-free enqueue, dequeue, steal batch
- 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:

Implementation	1-core (ns)	8-core (ns)
CCSP occam-π	46	39
CCSP C	73	75
Haskell	269	9892
Stackless Python	666	665
Erlang	1697	1675
pthread	5013	3485
JCSP	7723	14905

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
Stackless Python	666	665
Erlang	1697	1675
pthread	5013	3485
JCSP	7723	14905

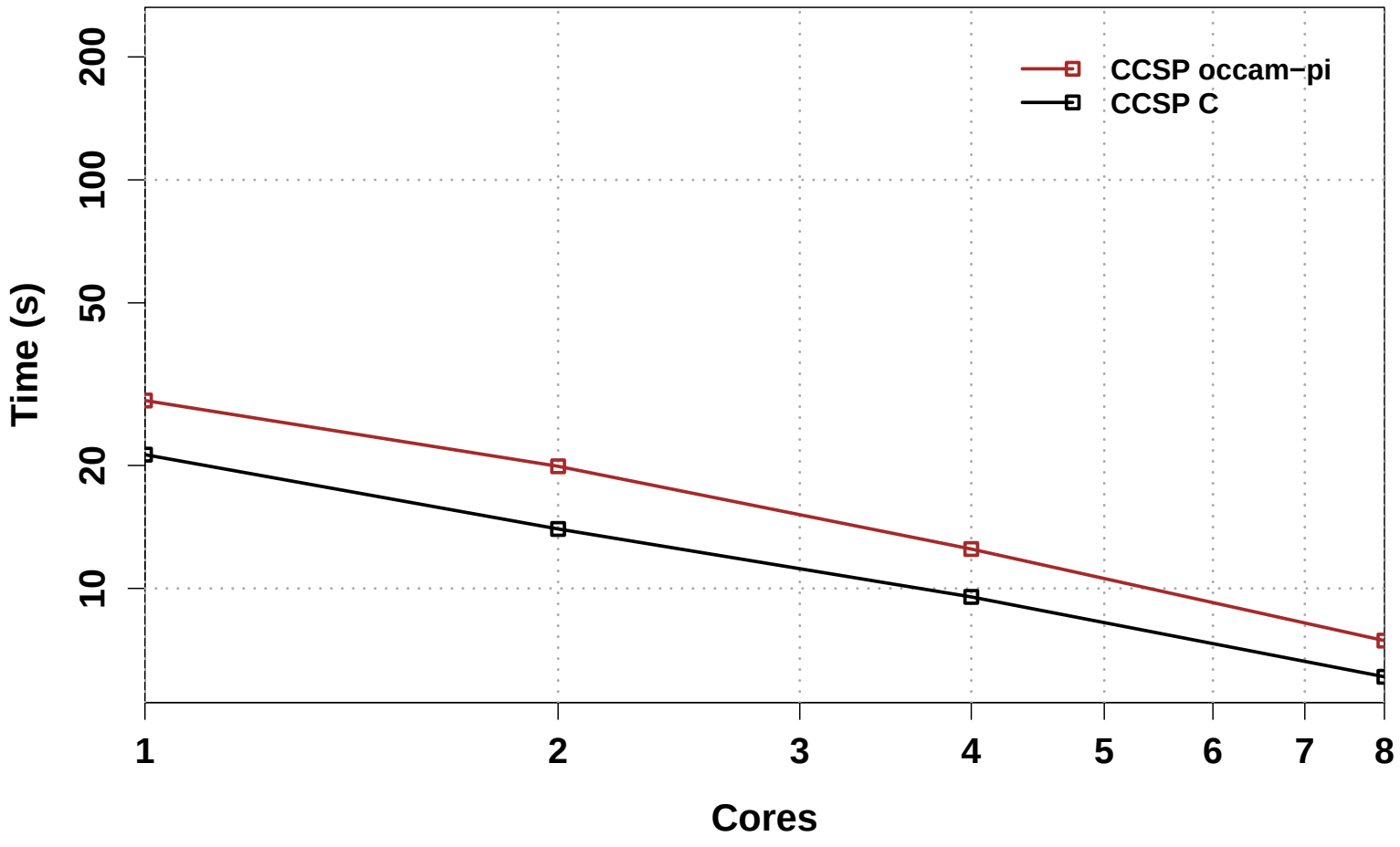
CoSMoS Inspired Benchmark

- 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

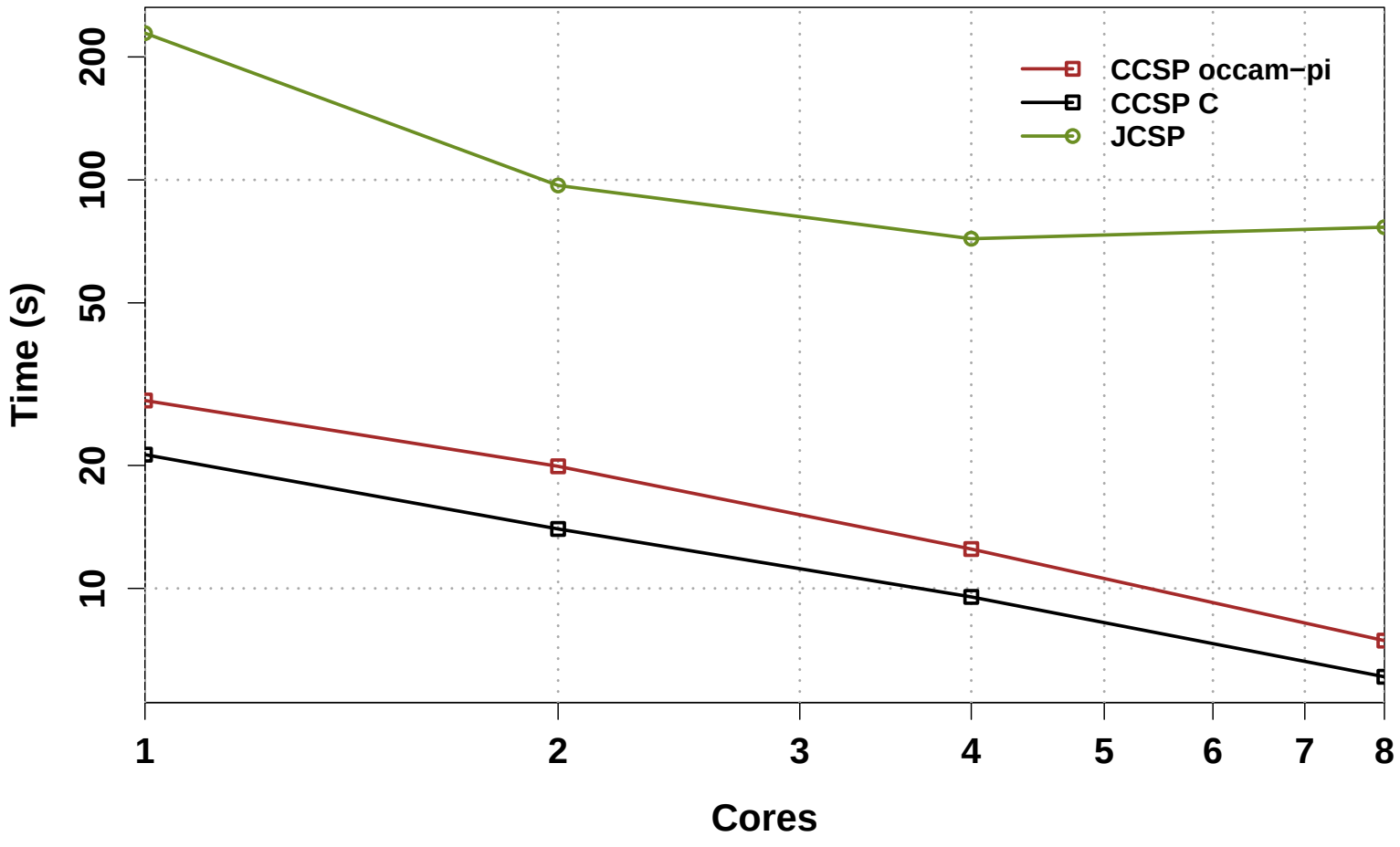
CoSMoS Inspired Benchmark

- 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

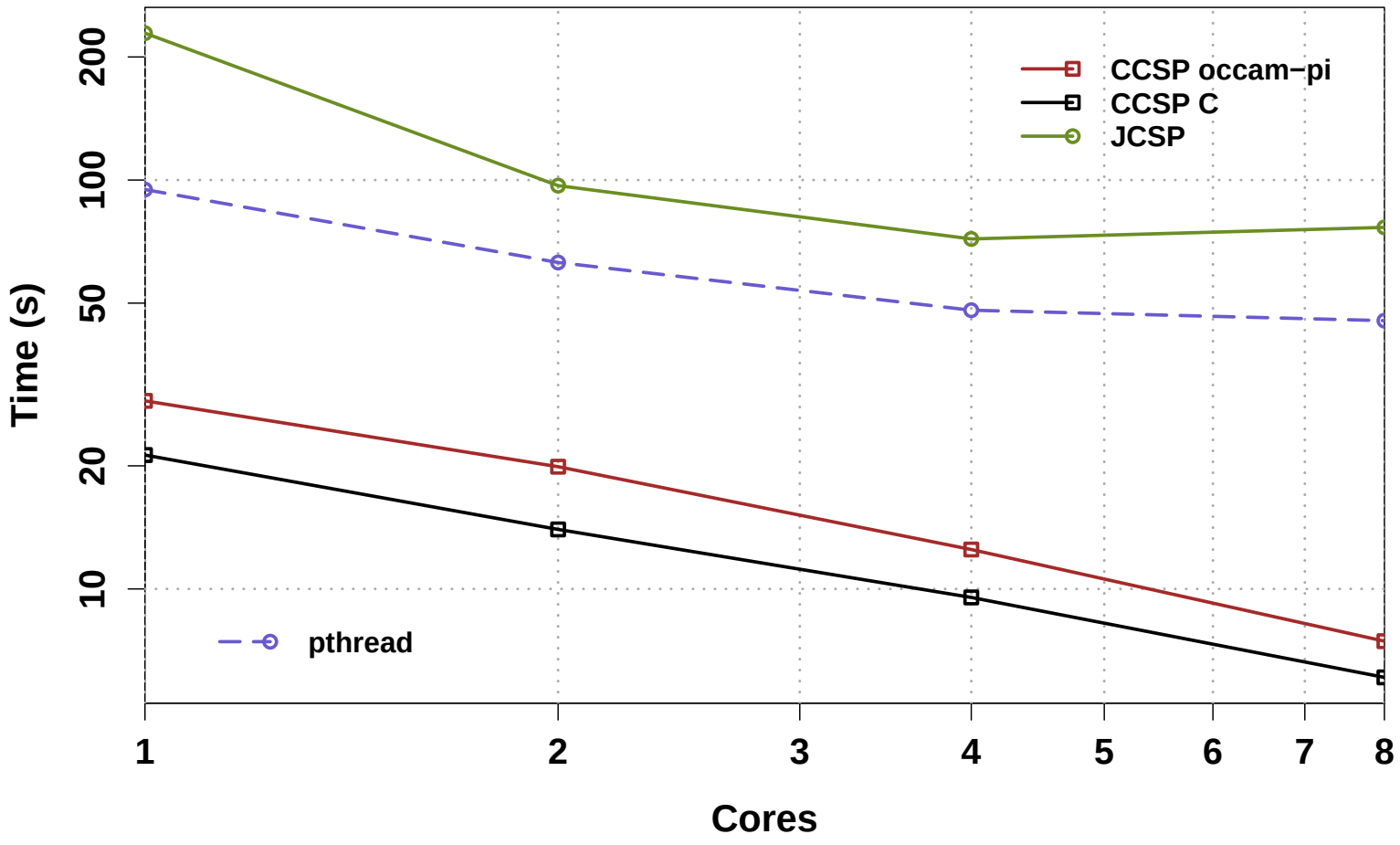
Increasing Cores



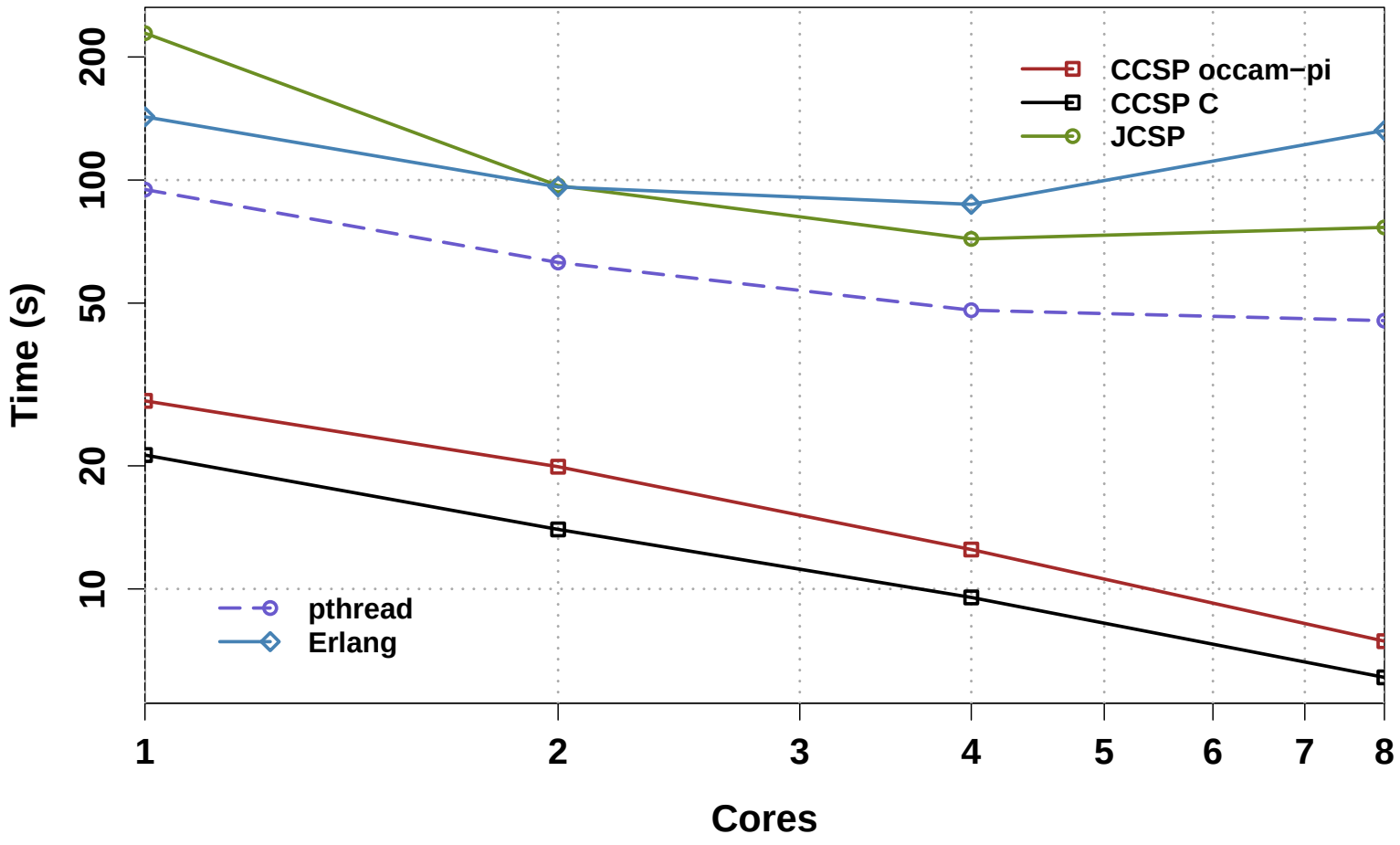
Increasing Cores



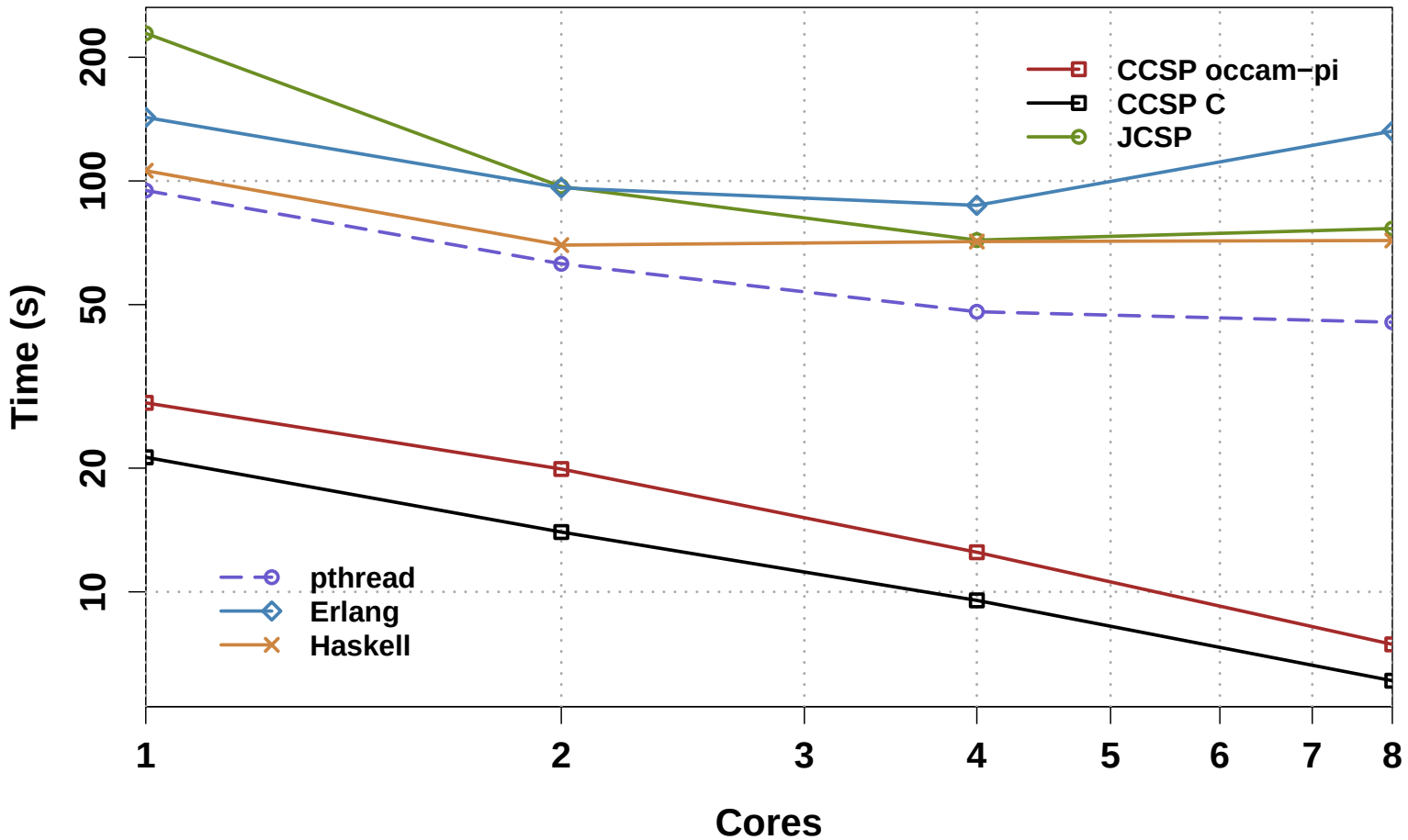
Increasing Cores



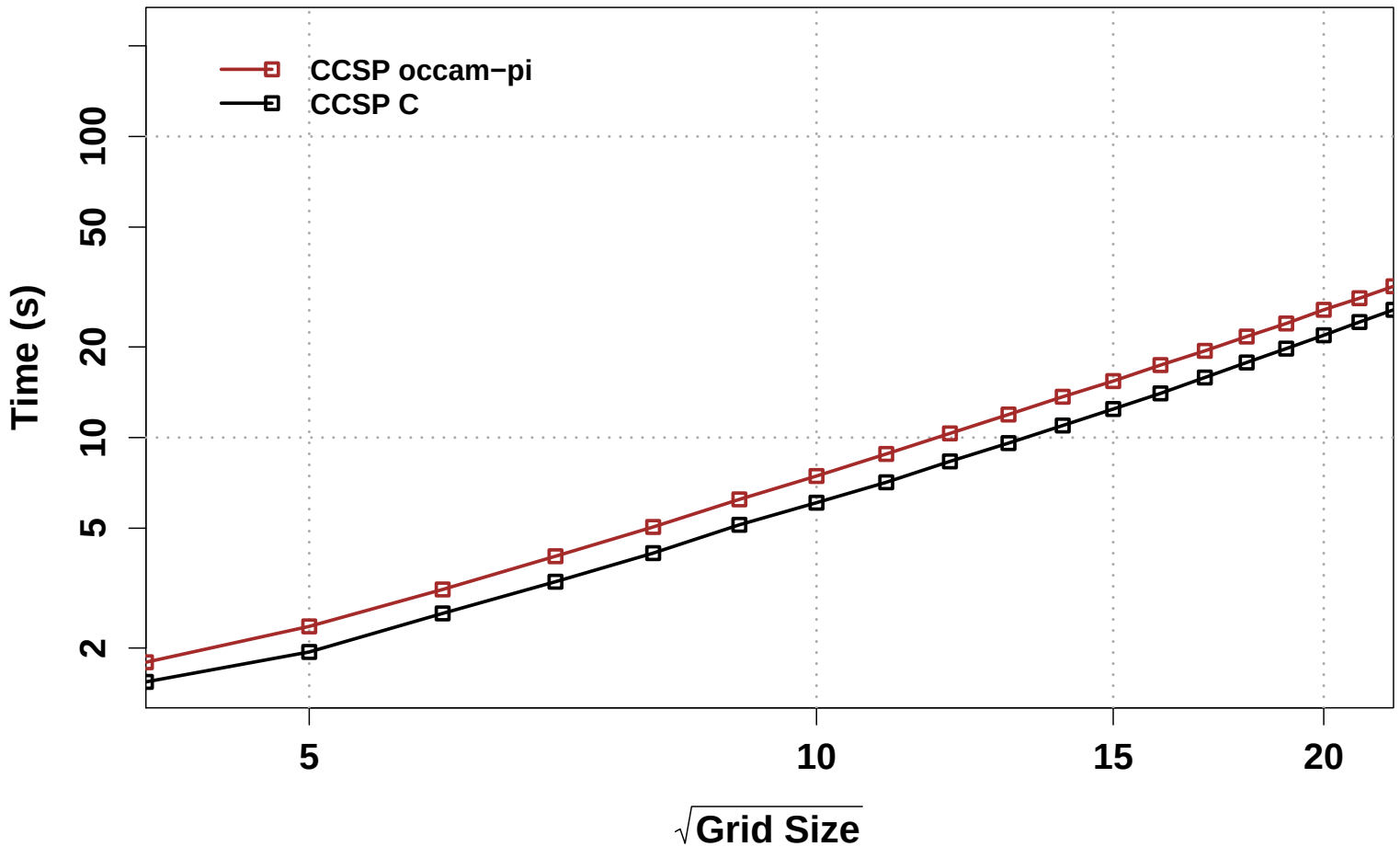
Increasing Cores



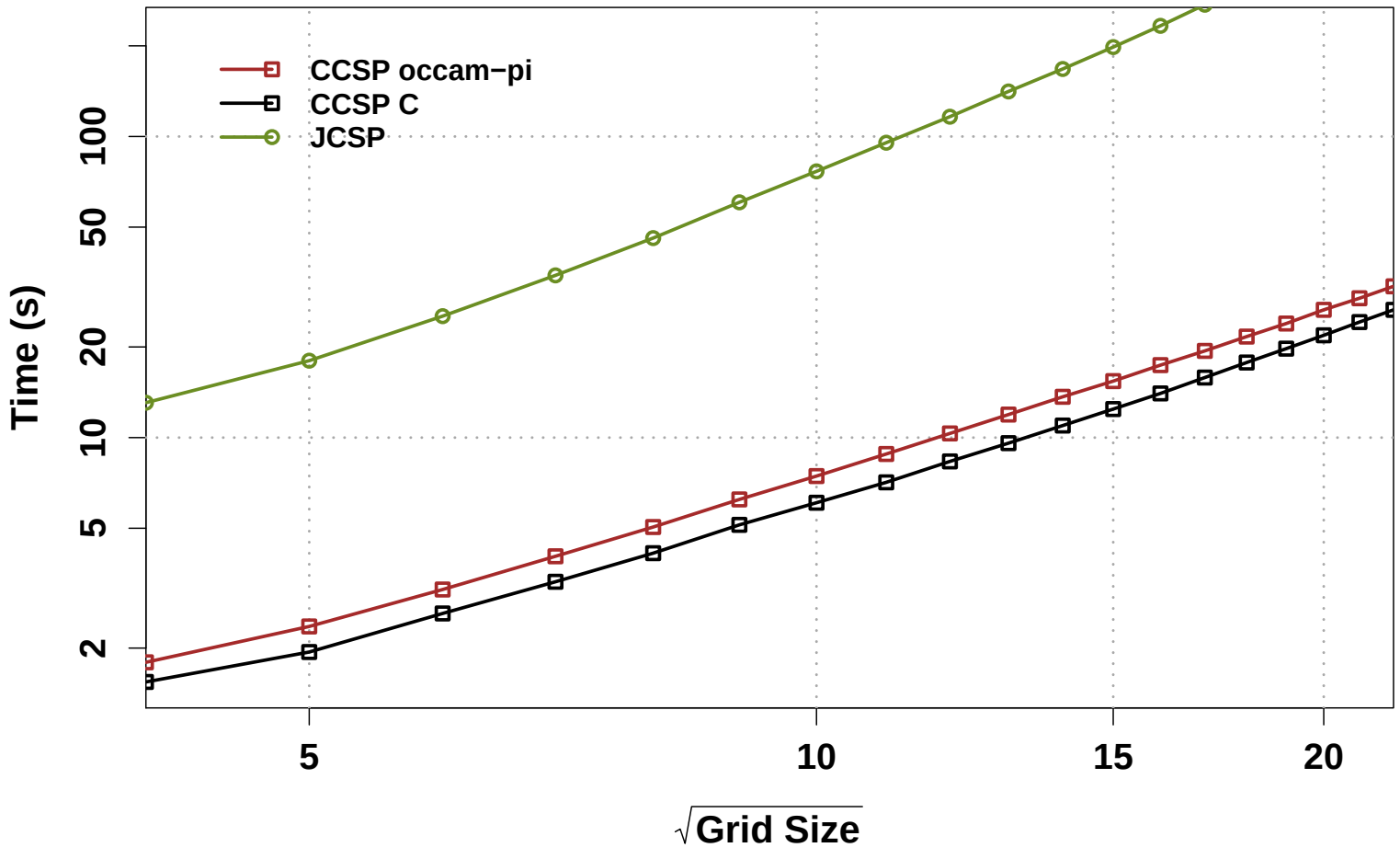
Increasing Cores



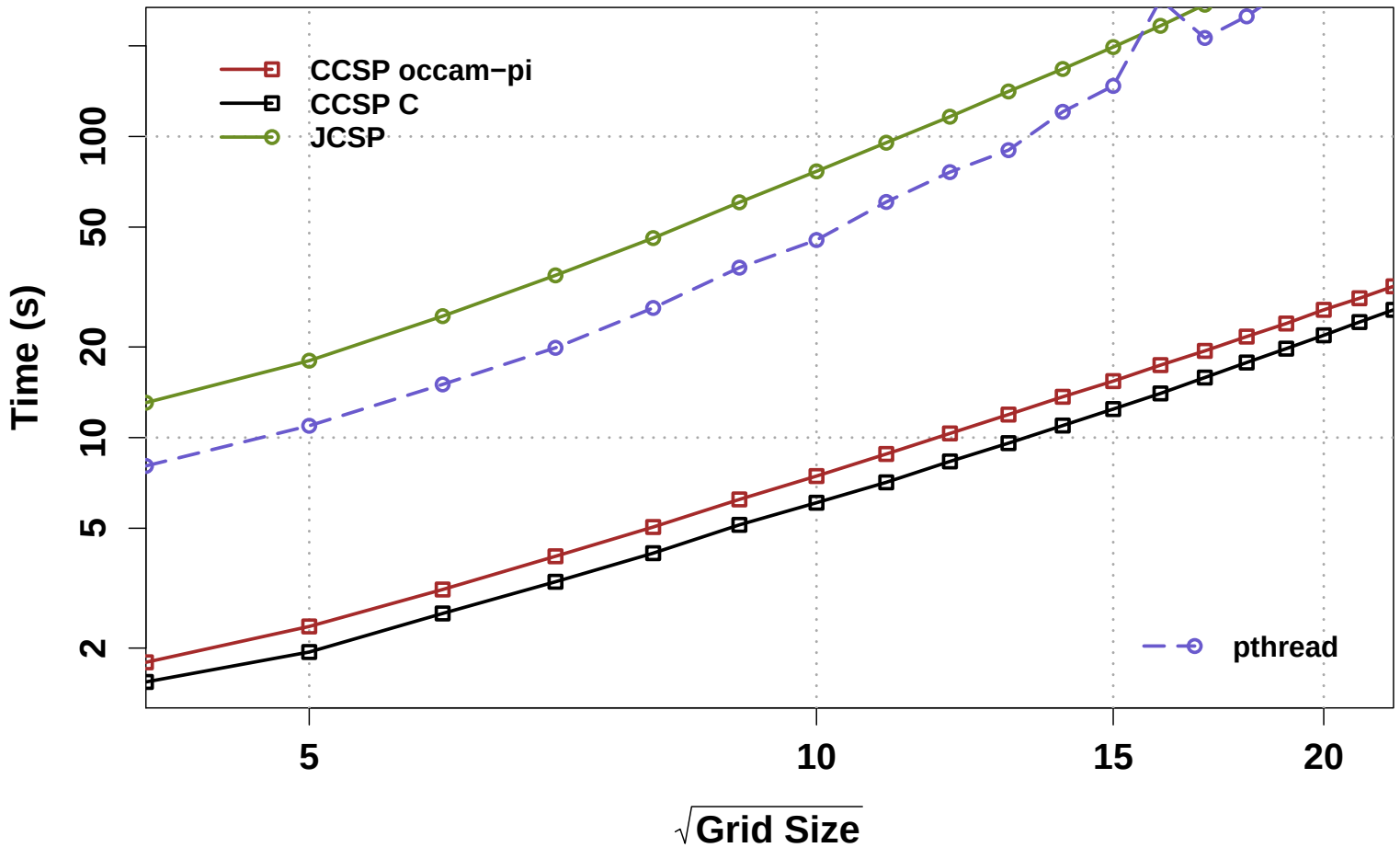
Increasing Problem Size (8-core)



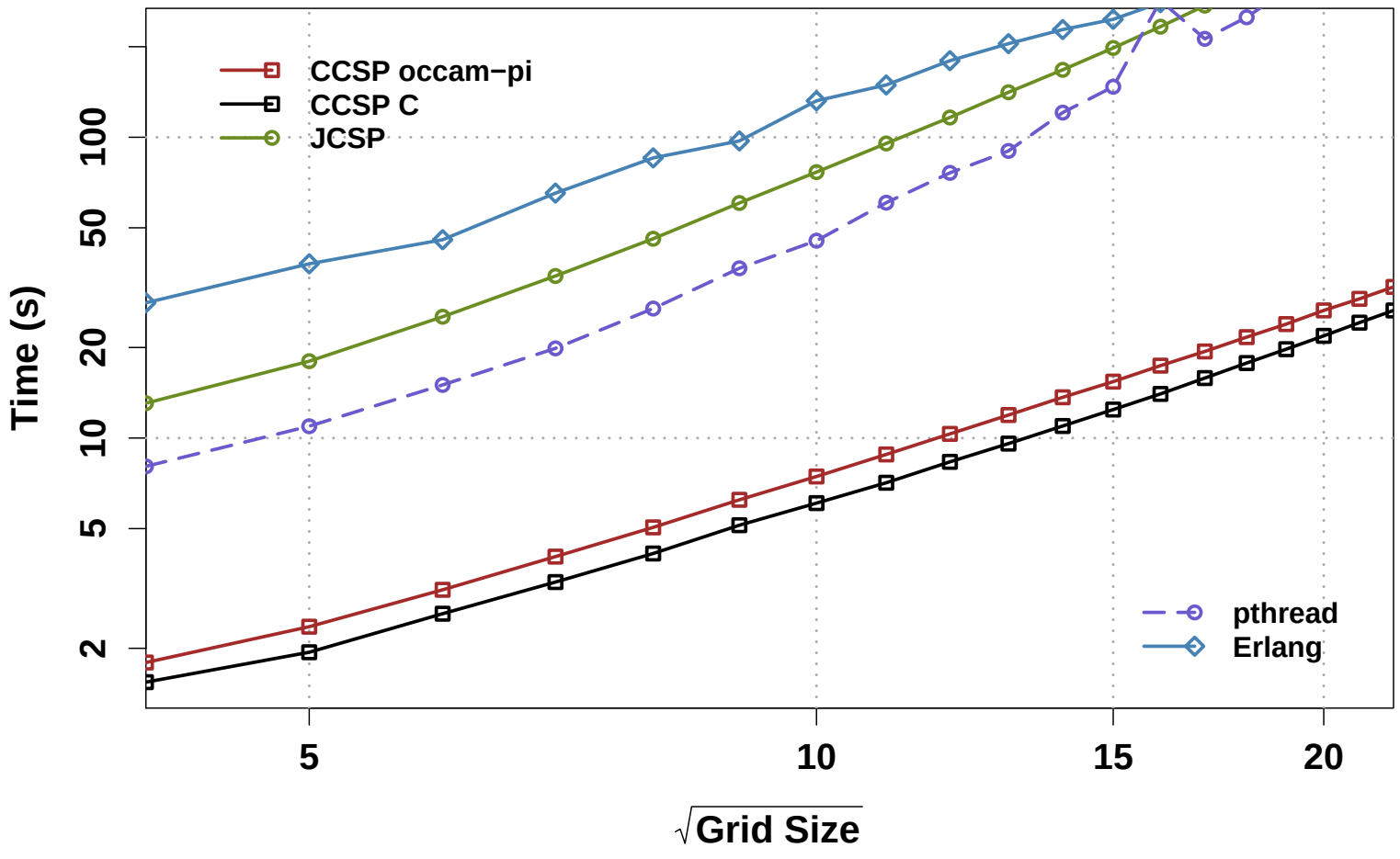
Increasing Problem Size (8-core)



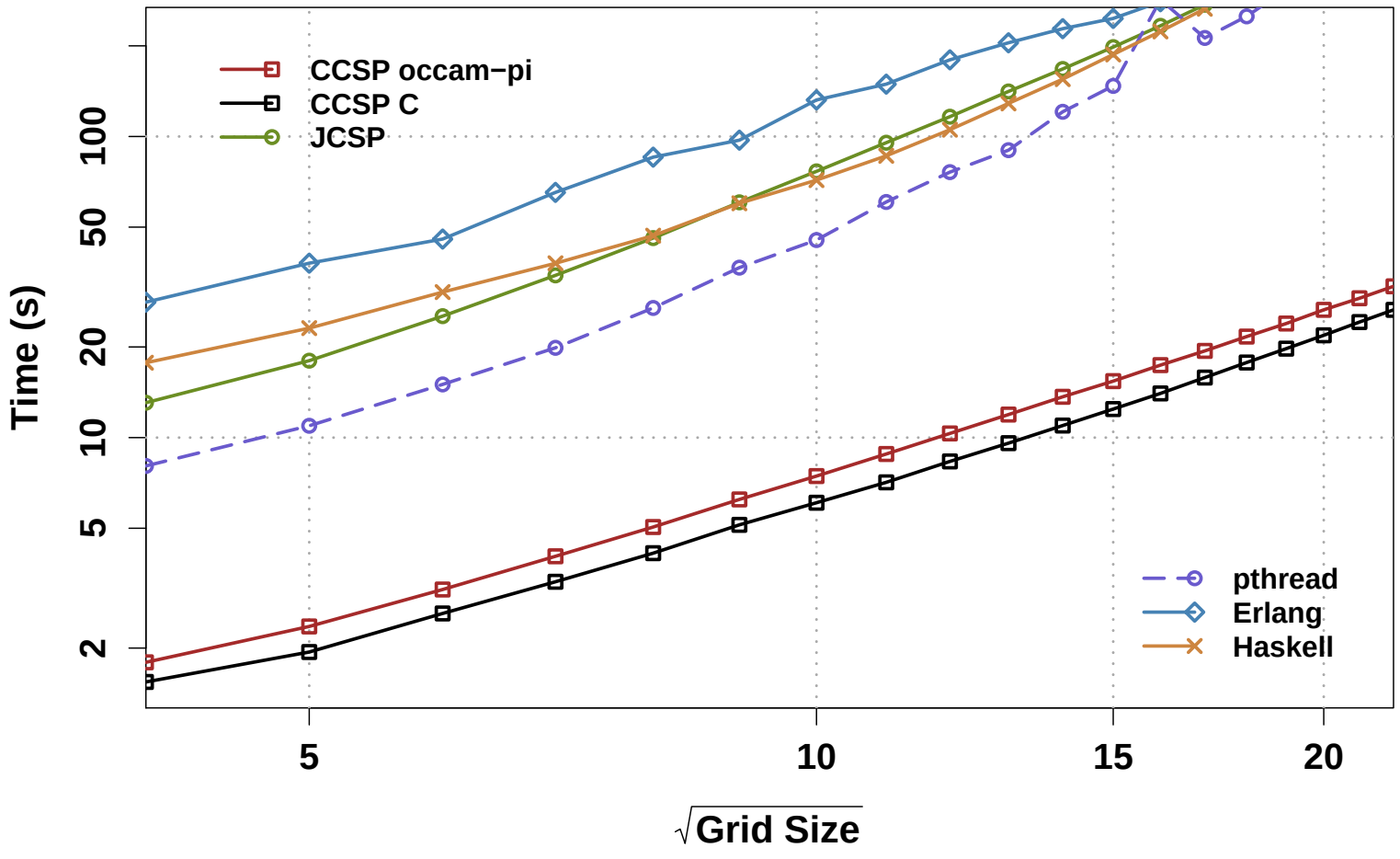
Increasing Problem Size (8-core)



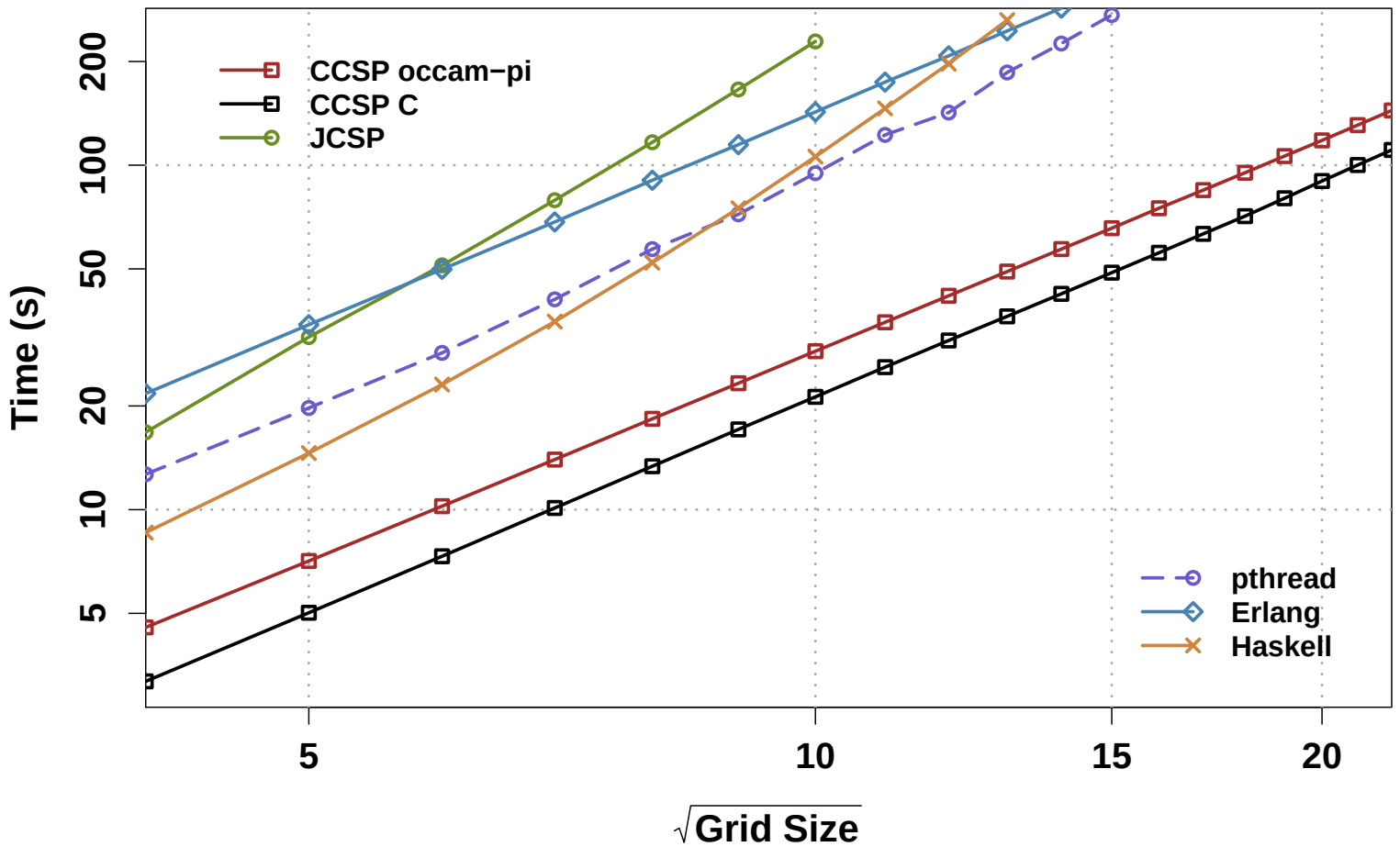
Increasing Problem Size (8-core)



Increasing Problem Size (8-core)



Increasing Problem Size (1-core)



Conclusions

- Implemented and working
  - In use by CoSMoS and RMoX EPSRC funded projects
- Automatic parallelisation of concurrent software
  - No programmer intervention
  - Dynamic batching
  - *n*-core, scalable, asymmetric scheduling
- Room for improvement
  - Smaller code
  - Fewer atomics
  - Portability

Conclusions

- Implemented and working
  - In use by CoSMoS and RMoX EPSRC funded projects
- Automatic parallelisation of concurrent software
  - No programmer intervention
  - Dynamic batching
  - *n*-core, scalable, asymmetric scheduling
- Room for improvement
  - Smaller code
  - Fewer atomics
  - Portability

Conclusions

- Implemented and working
  - In use by CoSMoS and RMoX EPSRC funded projects
- Automatic parallelisation of concurrent software
  - No programmer intervention
  - Dynamic batching
  - *n*-core, scalable, asymmetric scheduling
- Room for improvement
  - Smaller code
  - Fewer atomics
  - Portability

Questions?

- Questions...?