Process Cooperativity as a Feedback Metric in Concurrent Message-Passing Languages

Alexander Dean

Rochester Institute of Technology
Golisano College of Computer and Information Sciences

August 12, 2014

- Thank Fluet, Heliotis, and Raj.
- Dedicate to parents, who are unable to be present.

Process Cooperativity as a Feedback Metric

in Concurrent Message-Passing Languages

- Mouthful of a title, so I'll break it up:
  1. Runtime scheduling, to give some grounding in the area of study.
  2. Cooperativity, what it is and motivation to use it.
  3. Message Passing, because, as it turns out, it's a nice abstraction for our purpose of capturing cooperativity.
- The core of the work revolves around the toolkit I built.
  - A language/compiler/runtime/testing-framework
  - But also a *Simulator* which has a plug-and-play scheduler API. It let me test schedulers on a common test bed.
- Next, go over the list of schedulers & feedback mechanisms.
- Results, Conclusions, & Future Work.

Background

Introduce the new section:

1 Background
- Runtime Scheduling
- Cooperativity
- Message Passing

Background: Runtime Scheduling

- Schedulers can be defined in a discrete manner:
  1. *Choose* a process from set,
  2. *Reduce* it,
  3. *Update* private scheduler state.

- We can look at process schedulers like a function:
  - Takes a set of processes, and some private state.
  - Job of the function is to choose a process, and run it for a bit.
  - Then, based on what happened while running process, we update the state.
- Big questions: How are we choosing a process? What should effect our decision?

Background: Runtime Scheduling

- Schedulers can be defined in a discrete manner:
  1. *Choose* a process from set,
  2. *Reduce* it,
  3. *Update* private scheduler state.
- Statistics can be gathered at every step about process:
  - Timestamp of last run,
  - Number of reductions, *etc.*

- We can look at process schedulers like a function:
  - Takes a set of processes, and some private state.
  - Job of the function is to choose a process, and run it for a bit.
  - Then, based on what happened while running process, we update the state.
- Big questions: How are we choosing a process? What should effect our decision?
- Timestamp of last run? $\rightarrow$
  - Choose always most recent, it's a batch scheduler.
  - Choose oldest, we get something called Round-Robin.
- Number of reductions? $\rightarrow$ longevity $=$ might want to give someone else a go.
- What are useful, and what do they tell us about the state of the system? Well this leads us to cooperativity.

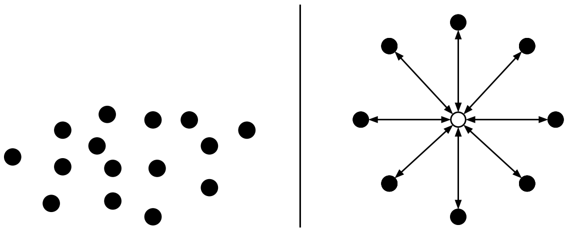
Background: Runtime Scheduling

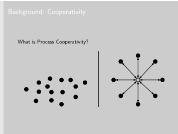
- Schedulers can be defined in a discrete manner:
  1. *Choose* a process from set,
  2. *Reduce* it,
  3. *Update* private scheduler state.
- Statistics can be gathered at every step about process:
  - Timestamp of last run,
  - Number of reductions, *etc.*
- *What statistics are useful?*

- We can look at process schedulers like a function:
  - Takes a set of processes, and some private state.
  - Job of the function is to choose a process, and run it for a bit.
  - Then, based on what happened while running process, we update the state.
- Big questions: How are we choosing a process? What should effect our decision?
- Timestamp of last run? $\rightarrow$
  - Choose always most recent, it's a batch scheduler.
  - Choose oldest, we get something called Round-Robin.
- Number of reductions? $\rightarrow$ longevity $=$ might want to give someone else a go.
- What are useful, and what do they tell us about the state of the system? Well this leads us to cooperativity.

Background: Cooperativity

What is Process Cooperativity?

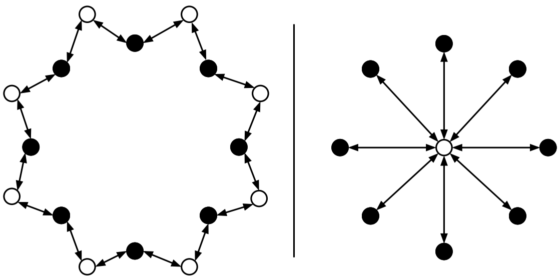


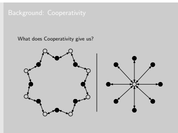


- What is Process Cooperativity?
- White = channel & black = a process.
- Can think of channel a mechanism for passing information between processes.
  - These are nice functional abstractions of things like locks, shared-memory, *etc.*
- Left: Cloud of processes with no interaction.
- Right: We see a definite structure caused by some sharing of information. This is the core of recognizing cooperation, namely, recognizing these structures when they exist.

Background: Cooperativity

What does Cooperativity give us?





What does Cooperativity give us? What's the difference in the behaviour of cooperation in the left/right applications?

- Left: A Ring,
  - the level of parallelism is nearly nil.
  - Each process is cooperating yes, but granularity is very fine.
- Right: A Star,
  - the level of parallelism is nearly full.
  - Each process is cooperating, **not reliant on more than one** other process.
- In both, the whole system is communicating, but with cooperation, we can find the level of parallelism possible.

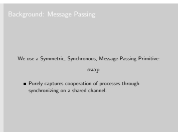
Next: Knowing this, how can we recognize cooperativity? Seems to be all about recording interactions with the channel.

Background: Message Passing

We use a Symmetric, Synchronous, Message-Passing Primitive:

`swap`

- Purely captures cooperation of processes through synchronizing on a shared channel.



- Symmetric, Synchronous, Message-Passing primitive.
- Symmetric:
  - Only one message passing primitive: SWAP
- Synchronous:
  - Blocks until it's partner gets there.
- Purely captures cooperation: Simple synchronization representation.
- This is really what I based the language on.
- So, what does the rest of the language look like.

ErLam Toolkit

Introduce the new section:

2 ErLam Toolkit
- The Language
- Channel Implementations
- Simulation & Visualization

ErLam Toolkit: The Language

```
<Expression> ::= <Variable>
              |  <Integer>
              |  'newchan'
              |  '(' <Expression> ')'
              |  <Expression> <Expression>
              |  'if' <Expression> <Expression> <Expression>
              |  'swap' <Expression> <Expression>
              |  'spawn' <Expression>
              |  'fun' <Variable> '.' <Expression>
```


- Extremely simple on purpose (5 keywords).
- Issue now began to be how to build up primitive test behaviours.
- Made a library which allowed for built ins.

ErLam Toolkit: The Language

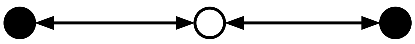
```
elib
    // ...
    ignore = (fun _.(fun y.y));
    omega = (fun x.(x x));
    // ...
    add = _erl[2]{ fun(X) when is_integer(X) ->
                        fun(Y) when is_integer(Y) ->
                            X+Y
                        end
                    end
                 };
    // ...
bile
```

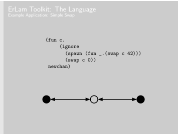

- There's options for built-ins as well as macros.
- Built-ins are raw Erlang, gets wrapped up into AST, and still "reduces" the same (*i.e.* no multi-variable functions).

ErLam Toolkit: The Language

Example Application: Simple Swap

```
(fun c.
    (ignore
      (spawn (fun _.(swap c 42)))
      (swap c 0))
 newchan)
```

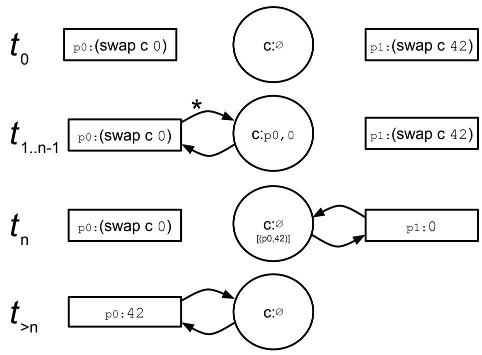


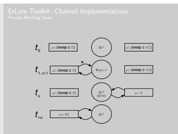


- Here is a simple application which:
  - Spawns a process to swap the number 42
  - Calls swap to get the value from the other process.
- Build up from here to simulate more complex behaviour.
- We can "do some work" before swaping, *etc.*
- This is how we built up or primitive test behaviours.

ErLam Toolkit: Channel Implementations

Process Blocking Swap

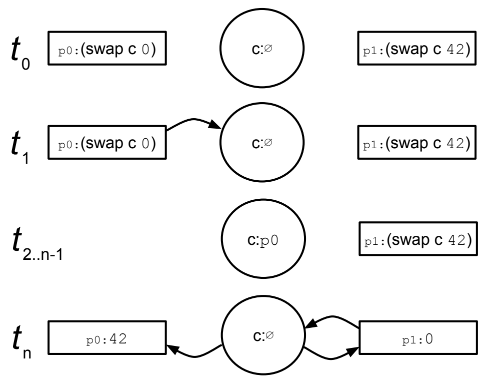


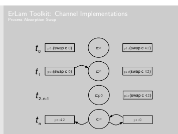


- Blocking: Maintains state of current and previous swap value until swap is completed.
- Mention expected effects on scheduler.
  - Scheduler will keep hold of the process, needs to recheck if blocked.
  - If all processes are communicating, large process queue of blocked processes.

ErLam Toolkit: Channel Implementations

Process Absorption Swap





1. Whole process gets absorbed by channel, away from scheduler.
2. When the second process completes the swap, the process gets removed from channel.
   - The p0 process can go back to its original scheduler,
   - OR to scheduler which unblocked it.

- Effects on scheduler:
  - Can loose or gain a extra process during communication.

ErLam Toolkit: Simulation & Visualization

Primitive Testing Behaviours:

- Degree of Parallelism
- Partial System Cooperativity

Logging & Report Generation

- To test cooperativity, we need a set of test behaviours.
- We want to analyize a large number of structure types.
- Two of them that I studied are:
  - Parallelism: Gets back to Ring vs Star, compare the two.
  - Full vs Partial Cooperation: Multiple groups of Stars or Rings.
- Finally, quickly, go over the current report generation that ErLam performs.

ErLam Toolkit: Simulation & Visualization

System Behaviours: Degree of Parallelism

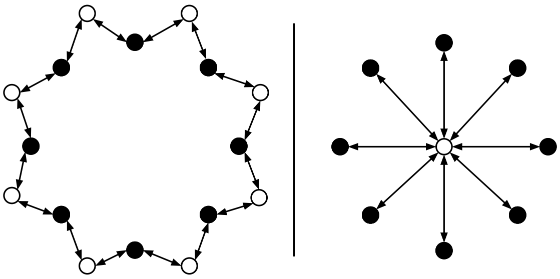


Figure: $PRing_N$, and $ClusterComm_{(N,1)}$ primitives to test degree of parallelism.

- Here are our Ring and Star behaviours, we call them by different names.
- We call the left, PRing, with the parameter N = number of processes.
- We call the right, *ClusterComm* with two parameters, N like *PRing*, $M = 1$ in this case.
- Note the generalization of the Star is a set of M channels that are randomly accessed.

ErLam Toolkit: Simulation & Visualization

System Behaviours: Consistency of Cooperation

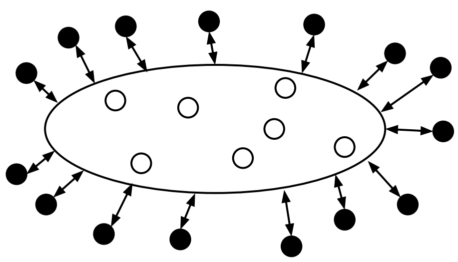


Figure: $ClusterComm_{(N,M)}$ to test effect of consistency on scheduler.

- Worst case scenario for C-C schedulers.
- However, this is still full system cooperativity. We want some minor work groups.

ErLam Toolkit: Simulation & Visualization
System Behaviours: Partial System Cooperativity

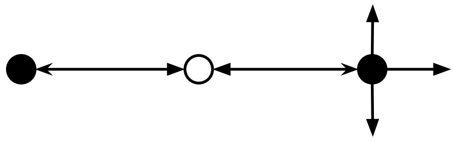
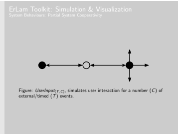


Figure: $UserInput_{(T,C)}$, simulates user interaction for a number (C) of external/timed (T) events.



- So we can make tiny work groups of two processes, like our simple swapping program earlier.
- The difference here is minor, we call this test UserInput because one process can hang for a set amount of time before triggering an event (effectively simulating user interation).
- But we need multiple of these work groups, running concurrently.
- Run a set of them in parallel, and we could even inject a bit of overhead at will by running any number of extra processes.

ErLam Toolkit: Simulation & Visualization
System Behaviours: Partial System Cooperativity

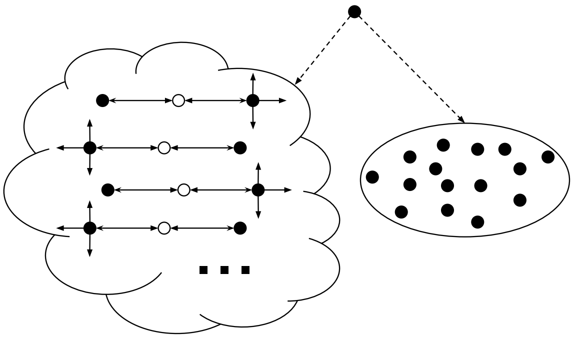
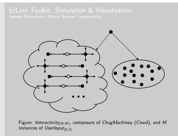


Figure: $Interactivity_{(N,M)}$, composure of $ChugMachine_N$ (Cloud), and M instances of $UserInput_{(5,2)}$.



- So we can make tiny work groups of two processes, like our simple swapping program earlier.
- The difference here is minor, we call this test UserInput because one process can hang for a set amount of time before triggering an event (effectively simulating user interation).
- But we need multiple of these work groups, running concurrently.
- Run a set of them in parallel, and we could even inject a bit of overhead at will by running any number of extra processes.

ErLam Toolkit: Simulation & Visualization
System Behaviours: Partial System Cooperativity

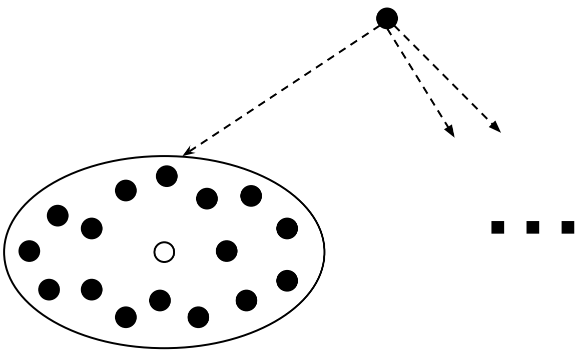
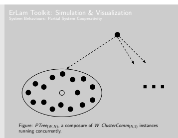


Figure: $PTree_{(W,N)}$, a composure of W $ClusterComm_{(N,1)}$ instances running concurrently.



- But we can run ClusterComm or PRing in parallel.

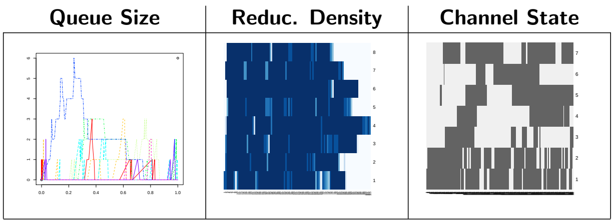
ErLam Toolkit: Simulation & Visualization
Logging & Report Generation

Things we could log:

- Process Queue Size (per LPU)
- Quantity of Reductions/Yields/Preempts
- State of the Scheduler (waiting/running)
- Channel State (Blocked/Unblocked)
- ...

- Queue-Length: work-stealing mechanics and saturation ability.
- Tick-Action: Visualize the density of computation/communication.
- Sched-State: Useful for comparing stealing/process selection mechanics.
- Chan-State: Tracking interactivity, speed of unblock=attentive to cooperation.
- Of course there are more, but we limited ourselves to the above for initial testing purposes.

ErLam Toolkit: Simulation & Visualization



- Three types of graphs:
  - Queue Size: X-axis is time, Y-axis is size of queue
  - Density charts: Darkness of the line represents fraction of ticks event happened in.
  - Channel State: dark=blocked, light=unblocked.

Scheduler Implementations

3 Scheduler Implementations

- Example Schedulers
- Feedback Mechanisms

Introduce the new section:

3 Scheduler Implementations
- Example Schedulers
- Feedback Mechanisms

Scheduler Implementations: Example Schedulers

- (MTRRGQ) - Round-Robin with Single Global Queue
  - All LPUs share a Process Queue.
- (MTRRWS-SQ) - Round-Robin with Work-Stealing via Direct Access
  - All LPUs have their own Process Queue.
  - LPUs can steal processes by grabbing them off the end of another LPU's queue.

Two of the basic schedulers built where:

1. RR w/ Global Queue: all synchronization around a single shared queue
2. RR w/ Work-Stealing: each scheduler gets their own queue but, they now need to steal work from others.
   – Implemented multiple types of work stealing, but we'll limit talk to one type:
   – Stealing directly from another LPUs by accessing the end of their process queue.

Scheduler Implementations: Feedback Mechanisms

Three types of mechanics:

- Longevity-Based Batching
- Channel Pinning
- Bipartite-Graph Aided Sorting

- Instead of a single cooperativity-conscious scheduler, we implemented three mechanics which take cooperativity into account on top of the basic schedulers.

Scheduler Implementations: Feedback Mechanisms

Longevity-Based Batching

- Choose via Round-Robin
  - from batch rather than queue
  - keeps track of number of rounds (batch size)
- Work-Steal whole batches
- Spawn to batch unless: $|b_i| \geq B$
  - Make singleton with new process.
  - Push parent and child into new batch.

- Batching processes based on longevity.
  – Based on occam-Π.
  – if a process communicates frequently then it will be batched (absorption), singleton if very computation-bound.
- We are normal RR but with one extra layer.
- If batch is too big during spawns we can:
  – Make singleton, best if child is needed to start work right away. Map-Reduce.
  – Make push-back, parent can get another chance to spawn more children sooner.
- GOAL here is to analyze the effects of grouping the frequently communicating.

Scheduler Implementations: Feedback Mechanisms
Longevity-Based Batching

- Choose via Round-Robin
  - from batch rather than queue
  - keeps track of number of rounds (batch size)
- Work-Steal whole batches
- Spawn to batch unless: $|b_i| \geq B$
  - Make singleton with new process.
  - Push parent and child into new batch.

GOAL: Can batching based on longevity account for fine/coarse parallelism in application?

- Batching processes based on longevity.
  - Based on occam-Π.
  - if a process communicates frequently then it will be batched (absorption), singleton if very computation-bound.
- We are normal RR but with one extra layer.
- If batch is too big during spawns we can:
  - Make singleton, best if child is needed to start work right away. Map-Reduce.
  - Make push-back, parent can get another chance to spawn more children sooner.
- GOAL here is to analyze the effects of grouping the frequently communicating.

Scheduler Implementations: Feedback Mechanisms
Channel-Pinning

- Upon call to *newchan*, pin to LPU based on spread algorithm:
  - *same* - LPU *newchan* is called is where it is pinned.
  - *even* - Cycle through LPUs and pin based on that.
  - ...
- Work-steal based on channel that's been pinned to you.

- Pin channels to LPUs.
  - Pinning a channel means to set a process affinity to a LPU based on the channels it uses.
  - Work-Stealing works like Go-Fish.
- Absorption channels would work well here to relocate processes based on channel usage.
- GOAL: can we improve on work-stealing by being selective.

Scheduler Implementations: Feedback Mechanisms
Channel-Pinning

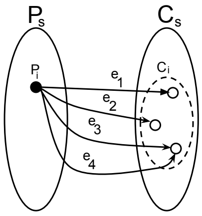
- Upon call to *newchan*, pin to LPU based on spread algorithm:
  - *same* - LPU *newchan* is called is where it is pinned.
  - *even* - Cycle through LPUs and pin based on that.
  - ...
- Work-steal based on channel that's been pinned to you.

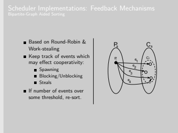
GOAL: Can an *even*-like spread increase early saturation?

- Pin channels to LPUs.
  - Pinning a channel means to set a process affinity to a LPU based on the channels it uses.
  - Work-Stealing works like Go-Fish.
- Absorption channels would work well here to relocate processes based on channel usage.
- GOAL: can we improve on work-stealing by being selective.

Scheduler Implementations: Feedback Mechanisms
Bipartite-Graph Aided Sorting

- Based on Round-Robin & Work-stealing
- Keep track of events which may effect cooperativity:
  - Spawning
  - Blocking/Unblocking
  - Steals
- If number of events over some threshold, re-sort.

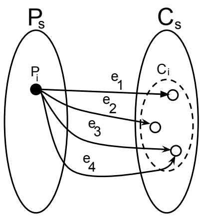




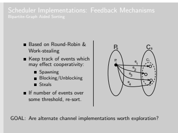
- Based on RR and WS.
- Keep a list of all communications as a graph between set of processes and channels.
- Keep track of all queue events: Spawn, Block, Steal
- We can resort using some function with these edges.
- GOAL: If we block, we wont loose order. If we had absorption channels like the previous two schedulers, all sorting would be in vain.

Scheduler Implementations: Feedback Mechanisms
Bipartite-Graph Aided Sorting

- Based on Round-Robin & Work-stealing
- Keep track of events which may effect cooperativity:
  - Spawning
  - Blocking/Unblocking
  - Steals
- If number of events over some threshold, re-sort.



GOAL: Are alternate channel implementations worth exploration?

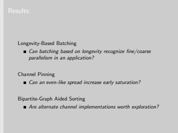


- Based on RR and WS.
- Keep a list of all communications as a graph between set of processes and channels.
- Keep track of all queue events: Spawn, Block, Steal
- We can resort using some function with these edges.
- GOAL: If we block, we wont loose order. If we had absorption channels like the previous two schedulers, all sorting would be in vain.

Results

Introduce the new section:

Results:



Longevity-Based Batching

- *Can batching based on longevity recognize fine/coarse parallelism in an application?*

Channel Pinning

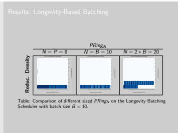
- *Can an* even-*like spread increase early saturation?*

Bipartite-Graph Aided Sorting

- *Are alternate channel implementations worth exploration?*

- Remind about the goals of the talk.
- LBB: Would like to take advantage of the frequency of communication.

Results: Longevity-Based Batching



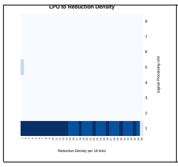
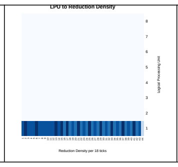
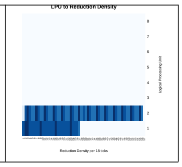
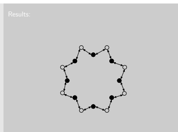
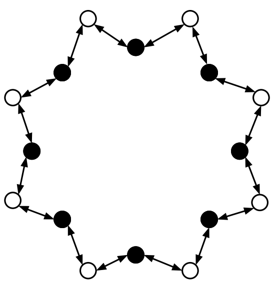
	$PRing_N$		
	$N = P = 8$	$N = B = 10$	$N = 2 * B = 20$
Reduc. Density			

Table: Comparison of different sized $PRing_N$ on the Longevity Batching Scheduler with batch size $B = 10$.

- Early tests gave promising results.
- Here is $PRing_N$ which shows the reabsorption and containment on a single LPU as expected and hoped.
- So does batching based on longevity really recognize fine/coarse parallelism in an application?
- Sort of, if you know what the right time-quantum is to make that distinction.

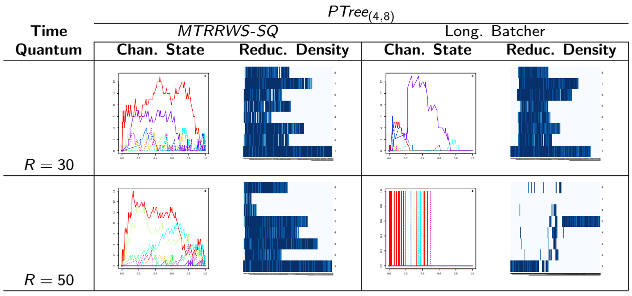
Results:

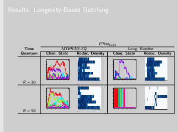




Reference Slide.

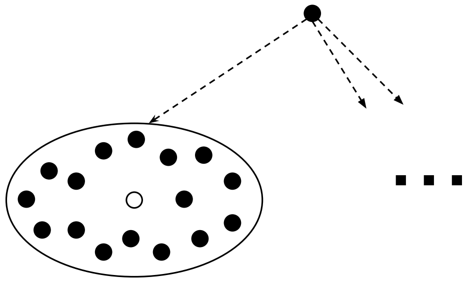
Results: Longevity-Based Batching

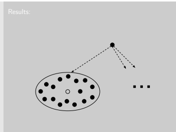




- Comparison of $PTree_{(4,8)}$ running with the Longevity-Based Batching Scheduler and *MTRRWS-SQ* at different time-quantums.
- Absorption channels help here to relocate processes.
- At lower time quantums Long. Batcher starts to look like RRWS-SQ, however batching and absorption channels tend to lead to consolidation.
- At higher time quantums Long. Batcher results in the originally expected work-groups. But it turns out to be inefficient due to lost chances of parallelism of each "star" of each group.
- Heuristical adjustment of the time-quantum would definitely be possible.
- NOTE: We don't capture overhead of stealing. Batching has obvious gains here.

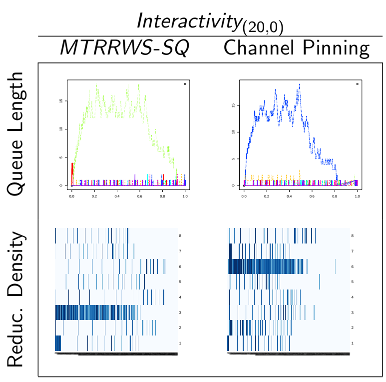
Results:

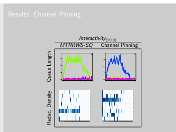




Reference Slide.

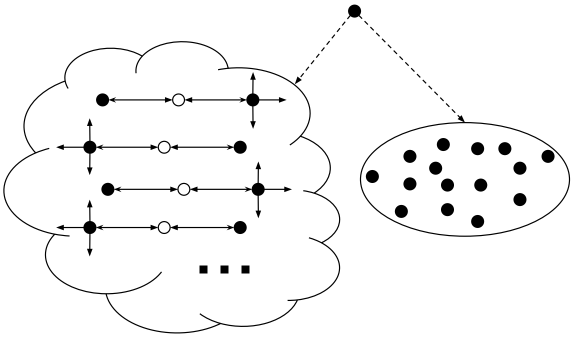
Results: Channel Pinning

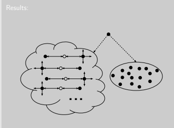




- Comparison of Uniform synchronization for *MTRRWS-SQ* and the Channel Pinning Scheduler on Absorption Channels.
- This used the *even* spread type.
- Note the speed at which it saturates all cores.
- Despite Naive WS, we still have decent spread.

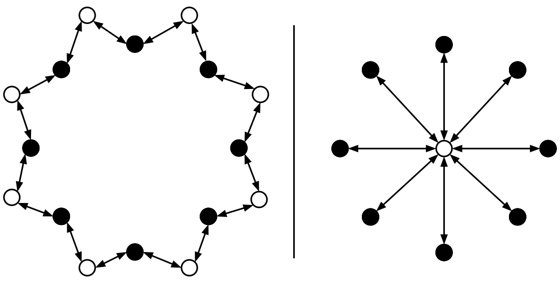
Results:

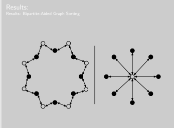




Reference Slide.

Results:
Results: Bipartite-Aided Graph Sorting

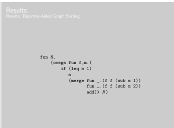




- There is an issue with our next test.
- Our base primitive behaviours don't have a preferred order of execution.
- Ring does, but there is nothing parallel about it.

Results:
Results: Bipartite-Aided Graph Sorting

```
fun N.
    (omega fun f,m.(
        if (leq m 1)
           m
           (merge fun _.(f f (sub m 1))
                  fun _.(f f (sub m 2))
                  add)) N)
```

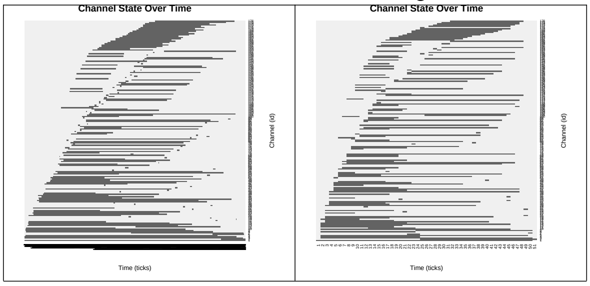



- We therefore used a parallel Fibonacci program.
- Merge function waits for each sub process to finish before running Add.
- If we sort based on who hasn't had a chance to communicate yet, we can preferentialize these MapReduce style applications, while not decrementing possible execution in parents.

Results: Bipartite-Aided Graph Sorting

Parallel Fibonacci

MTRRWS-SQ	Sorting Scheduler



- PFib has a strong reliance on order of execution.
- Channel State comparison of Parallel Fibonacci executed on *MTRRWS-SQ* and the Bipartite-Graph Aided Sorting Scheduler.
- Note the large reduction in number of ticks.

Conclusions & Future Work

5 Conclusions & Future Work
- ErLam Toolkit
- Cooperative Schedulers
- Cooperativity as a Metric

Introduce the new section:

5 Conclusions & Future Work
- ErLam Toolkit
- Cooperative Schedulers
- Cooperativity as a Metric

Conclusions & Future Work: ErLam Toolkit

- Test Primitives were nicely composable process behaviours.
  - More research into generating behaviours.
  - More compositions: PTree with Rings.
- Log generation, lots of overhead, but good observations.

- Overall pleased with simulator and achieved its goal.
- Future Work:
  - Generate more test primitives, a good library of them would be nice.
  - Compose them easier and more frequently. Perhaps generating work groups as PRing might have made a better comparison than Interactivity.
  - Clean up log generation (reduce overhead).
  - Process evaluation uses alpha-reduction (can be sped up substantially).
  - Make schedulers more adjustable (different spawn/yields/etc).
  - More Channel implementations

Conclusions & Future Work: Cooperative Schedulers

- **Longevity Batching:**
  - Would benefit from heuristic based Quantum selection.
  - As it stands, limited gain from longevity recognition.
- **Channel Pinning:**
  - Promising saturation and work-stealing mechanic.
- **Bipartite-Graph Aided Sorting:**
  - Supprising results on MapReduce style applications.
  - Worth studying Blocking-Channels further for gains from sorting.

- SS: Would benefit from merging with Channel Pinning. Increase likelihood that sorting puts channel partners close.
- LBB + CP might be interesting as channels could own batches.
- Overhead of Sorting? If implemented in a practicle language, would it be worth it? Seems counter-intuitive but promising as is.

Conclusions & Future Work: Cooperativity as a Metric

- Possible to recognize and benefit from.
- The three example mechanics are promising and can be extended for practicle modern languages, despite simplistic simulation language.
- More to explore:
  - Alternate Message-Passing Types (Asymmetric?)
  - ...

- Promising scheduling mechanics
- More research is necessary in the message-passing implementation for it to be practical in common languages.
- (Asymmetric/Directionality would be first on the list)

Questions/Comments?

Questions/Comments?

Questions/Comments?

Thank You!

Questions/Comments?
Thank You!

Links

- https://github.com/dstar4138/erlam
- https://github.com/dstar4138/thesis_cooperativity
- http://dstar4138.com

Links
https://github.com/dstar4138/erlam
https://github.com/dstar4138/thesis_cooperativity
http://dstar4138.com