

# Impact of Workload and System Parameters on Next Generation Cluster Scheduling Mechanisms

Yanyong Zhang, Anand Sivasubramaniam, José Moreira, and Hubertus Franke

Presented by Gary Jackson

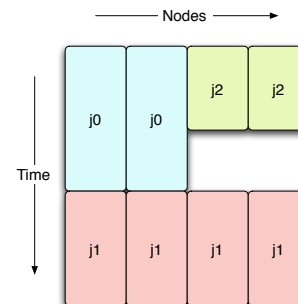
# Introduction

- Scheduling is important for communication
- After simulating some scheduling schemes
  - Periodic Boost (PB) is the best
  - Certain PB heuristics are better than others

# Need

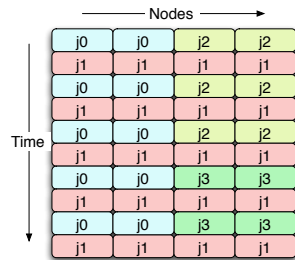
- Chaos on clusters does not work well
- Basic problem: waiting on a message from an unscheduled process
- Thus, scheduling and communication need to be considered together

# Existing Schemes: Space Sharing



- Wasted time on unscheduled nodes
- Wasted time if there's a lot of I/O

## Existing Schemes: Coscheduling (GS)



- Not robust for node failure
- Requires good synchronization
- Long time quanta
- Poor performance for many classes of jobs
- Wasted time waiting for I/O

## Solution: Dynamic Coscheduling

- Approximating coscheduled execution
- Based on messaging events
- In this paper, relies on User Level Networking (ULN)
- Have already demonstrated PB and Spin Yield (SY) on an existing cluster

## User Level Networking (ULN)

- Send:
  - Process enqueues message
  - NIC polls memory, dequeues and sends
- Receive:
  - Process busy-waits on receive queue
  - NIC enqueues received message
  - Process dequeues message

## Why is this good?

- Normally, processes block when reading messages from a network
  - This causes a context switch
- In ULN, the busy-wait receive means less frequent context switches
  - But nothing is getting done

## Scheduling Strategies

- Multiprogramming Level (MPL): number of simultaneous processes on a node
- Local Scheduling: unmodified local scheduler
- Gang Scheduling (GS): conventional coscheduling

## When waiting...

- Spin Block (SB)
  - Spin for a fixed amount of time
  - Then block
- Spin Yield (SY)
  - Spin for a fixed amount of time
  - Then lower own priority

## When a message arrives...

- Demand-Based Coscheduling (DCS)
  - NIC checks the running process periodically
  - On message arrival
    - If the recipient is not running
      - Interrupt the processor to raise the priority of the recipient

## When a message arrives...

- Periodic Boost (PB)
  - Kernel checks arrival queues periodically
  - Uses a heuristic to raise the priority of receiving processes
  - Heuristic used here:
    - Boost receiving processes that are waiting on a message

# Scheme Summary

What do you do on message arrival?	How do you wait for a message?		
	Busy Wait	Spin Block	Spin Yield
No Reschedule	Local	SB	SY
Interrupt & Reschedule	DCS	DCS-SB	DCS-SY
Periodically Reschedule	PB	PB-SB	PB-SY

# Experimental Platform

- Sophisticated simulator
- Workloads
  - Statistical models based on real-world observation
  - Eight different workloads
- Four different communication patterns
- Twelve different parameters for experiments

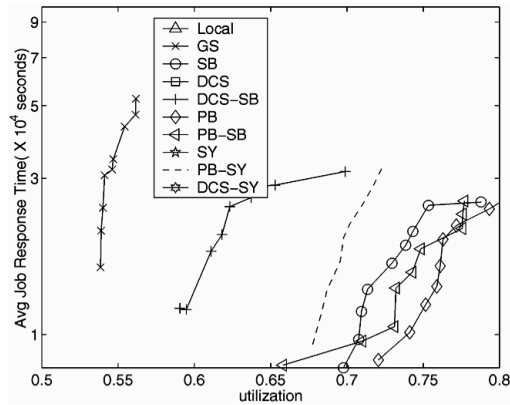
# Metrics

- **Response = Wait + Execution time**
- Slowdown
- Utilization
- Fairness
- **Performance Profile**

# Experiment: Load

- Test with a fixed frequency and nominal length of job
- Measure average response time and utilization
- Conclusion: PB gets better response time and utilization than the others

## Experiment: Load



- What does this graph mean?  
There are two dependent axes.
- What do they mean by “saturation”?

## Experiment: Nature of Workload

- Test on different workload types
- Measure response time
- Conclusions
  - PB is superior for communication
  - PB, SB, PB-SY, and PB-SB were otherwise indistinguishable
  - Local and SY were universally bad

## Experiment: Multiprogramming Level

- Test the simulator with three different workloads and MPL=2, 5, and 16
- Measure response time and performance profile
- Conclusion: PB is better at lower MPL, and is about the same as SB at higher MPL

## Experiment: Skewness

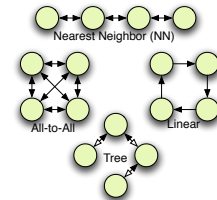
- Skewness: variation introduced in to the computation and I/O phases to cause the programs to be out of synchronization
- Test with 20% skewness and 150% skewness and an even workload
- Measure response time and performance profile

## Experiment: System Overhead

- Test with three sets of overhead costs
- Measure response time
- Conclusion: blocking based schemes reacted the best to shortened overhead costs

## Experiment: Communication Patterns

- Test with four different communication patterns
- Measure response time
- Conclusion: GS is best for AA, PB is best overall



## Experiment: Number of Nodes

- Test with 16 and 32 nodes
  - Job sizes remain the same
- Measure response time and performance profile
- Conclusion: Idle times go up in the blocking schemes, indicating limited scalability
- Also, partitioning doesn't do any good

## Experiment: Fairness

- Test mix of CPU, I/O, and communication intensive jobs
- Measure slowdown for each individual job type
- Conclusion: GS is the best, local is the worst, and blocking schemes do better than spinning schemes (no surprises)

# Experiment Summary

Experiment	Winners	Losers
Load	PB	Local
Nature of Workload	PB	Local, SY, DCS variants
MPL	PB lower, SB higher	Local, DCS, SY
Skewness	PB and SB variants	
System Overheads	SB and PB-SB benefited from lower	
Communication Pattern	GS for AA, PB overall	Local, SY, DCS variants
Number of Nodes		SB variants
Fairness	GS	Local

# Performance Boost Heuristics

- Decisions based on process states
- Can be divided along
  - What the process is doing
  - Whether the process has unconsumed messages

Unconsumed Messages	Phase	
	Compute/Send	Receive
No	S1	S4
Yes	S2	S3

# Performance Boost Heuristics

<b>A</b>	$S3 \rightarrow \{S2, S1\}^*$
<b>B</b>	$S3 \rightarrow S2 \rightarrow S1$
<b>C</b>	$\{S3, S2, S1\}$
<b>D</b>	$\{S3, S2\} \rightarrow S1$
<b>E</b>	$S2 \rightarrow S3 \rightarrow S1$

\*brackets mean equal round-robin consideration

# Tests

- Tested all five with three different workloads
  - CPU intensive, balanced, communication intensive
- Measured response time, conditions causing boost, position of boosted process

## Tests

- Conclusion: **D** performs the best
- **B** fails in communication due to lower likelihood of coscheduling
- **D** is more fair than **E**

## More Fairness

- Added fair-share policy to all heuristics
- Compared heuristics with and without fair-share
- Conclusion
  - Fair sharing makes **A'**, **B'**, **C'**, and **D'** more fair
  - Slight improvement in response time

## Conclusion

- ULN requires changes in scheduling schemes
  - As evidenced by local scheduler performance
- GS as tested doesn't do as well as dynamic coscheduling
- PB is the best until skewness is large, then PB-SB might be better
- Heuristic **D'** is the best PB heuristic

The End.