Coordinated Coscheduling in Time-Sharing clusters through a Generic Framework

Saurabh Agarwal
IBM Research (India Research Labs)
IIT New Delhi – 110016, India

Co-authors:
Gyu Sang Choi : PhD Student, CSE Dept. , Penn State University.
Dr. Chita Das : Prof. (& my Advisor), CSE Dept. , Penn State University.
Dr. Andy B. Yoo : LLNL, Livermore, CA
Dr. Shailabh Nagar: IBM T.J. Watson research center, NY
Clusters Today

- Scientific Applications
- Bio-Informatics Applications
- Multimedia Applications
- Web / Wireless Application Servers
- Other Real-Time Applications
- SAN (Myrinet, cLan, Dolphin, IBA)

Why?
- Low cost
- High performance
- Scalability
- Fault-tolerance
Typical cluster based Application Characteristics

- Parallel and/or Distributed.

- Possibly Communication Intensive (High/Medium/Low).
  - Very common and is our research focus.

- Requirement for progress?
  - Timely Inter-Node communication.

- Implication?
  - Parallel Jobs must be co-scheduled!
Outline Today

- What is the coscheduling problem?
- How has it been solved earlier?
  - Batch, Gang, DCS, ICS (SB), PB
- Why prior solutions not enough?
- What do we propose?
- What are the results?
- What can we conclude?
- Is there a future for this?
What is CoScheduling?

- **CoScheduling**: Concurrently Schedule processes of a parallel job on individual nodes of a time-sharing cluster.

- **Why Needed?**: Performance

*Defined by Ousterhout-92*
Parallel job Scheduling Techniques
(Hierarchy)

Scheduling Alternatives for Parallel jobs

Local Scheduling (No concurrency)

Coscheduling (Concurrent Scheduling)

Explicit Coscheduling (Uses Global scheduler)

Implicit Coscheduling (Relies on Local schedulers)

Batch (Space shared)

Gang Scheduling (space shared & time shared)

Dynamic (MIT)

Implicit (UCB)

Periodic Boost (PSU)

How has it been solved earlier? (The history Tree!)
Un-coordinated (Local) scheduling

How has it been solved earlier? (Local policy, no explicit solution)
Explicit coscheduling: Batch Scheduling

- 1 job runs on all CPUs until completion
- Other jobs wait in scheduler queue for their turn
- Problems:
  - Low utilization
  - Low response times
- Examples:
  - IBM-SP2 (Uses Load Leveler)
  - Intel Paragon (Uses Network Queuing System (NQS))
  - Many research COTS clusters use PBS

How has it been solved earlier? (Batch scheduling – space sharing)
Explicit Coscheduling: Gang Scheduling

- Mixed workloads
- Global, Logical Schedule

Examples:
- IBM-ACSI White
- IBM ASCI Blue-Pacific

How has it been solved earlier? (Gang scheduling – space + time sharing)
Why no Gang Scheduling on Clusters?

- Need a Global scheduler controlling jobs on ALL nodes
  - Impractical to implement as :-
    - Requires Frequent Synchronization (Order milli-sec)
      - Problem in NOW because of higher wire latencies (relatively).
    - Requires change in ‘time quantum’ (of order seconds).
      - Reduces system responsiveness.

- Not scalable for NOW (For same reason as above).
What is the solution?

Gang Scheduling

Implicit Co-scheduling

Co-scheduled only for part of the time slot.

Introducing alternatives: communication-driven implicit coscheduling.
NIC Control Program (NCP) senses incoming message.

NCP raises interrupt(s) to NIC Driver, if required.

NIC Driver finds & places relevant process into the highest-priority queue.

Scheduler schedules that process.

“Dynamic Adjustment” of process priorities by using “messages” (from communicating jobs) as hints.

How has it been solved earlier? (DCS – time sharing, interrupt driven)
SB : Spin Block

“Implicit” blocking if no message arrives within spin_time. Wakeup on arrival.

- MPI (or VIA) spins to wait for a message
- If not arrived, block in driver, register for interrupt in NIC.
- Upon message arrival, NIC raises interrupt, driver unblocks the process.
- Scheduler schedules that process as per local policies.

How has it been solved earlier? (SB – interrupt, spin driven, no boosting)
PB : Periodic Boost

“Periodically” check end-points in host memory, “Boost” the process with pending messages.

- Driver polls VI end-points in user-pinned memory, every 10 ms.
- Picks 1st pending-message VI, finds corresponding process, boosts it priority.
- Scheduler schedules that process next.

How has it been solved earlier? (PB – spin only, periodic boosting)
‘Implicit’ Coscheduling: Various Issues

- **DCS**
  - Too many interrupts (< 1ms)
  - Reason: Per VI, instead of per process

- **SB**
  - No boosting.
  - No sender-side optimization.
  - Variant: ICS → For Tightly coupled (Split-C) environment.

- **PB**
  - Unfair in current form.
  - Delayed accuracy: Polling done after DMAs
  - Low performing in fair version. (Too many polls)
    - Reason: Per VI, instead of per process.

Why prior solutions not enough?
No Commercial Implementation. WHY?

1. Lack of exhaustive experimentation on multiple platforms.
   - Not easy to code custom-solutions for each platform.
   - No generic / standard approach available.

2. No scheme best for all types of workloads.
   - None promises extensibility, generality, adaptability?
   - Support for QoS not addressed at all.

3. No real incentives (results) demonstrated yet
   - Coscheduling against batch scheduling?
   - Presence of other sequential workloads (CPU, I/O)?
   - High Multi-programming degree?
Addressing global issue 1: Prior Design

Parts of code to change:
- MPI Library (SB)
- VIA Library (SB, PB)
- Device driver (DCS, SB, PB)
- Firmware (DCS, SB, PB)

Disadvantages:
- Flexibility: Tight driver/firmware coupling
- Generality: Different implementations for each scheme.
- Modularity: No re-use across platforms.
- Portability: No standard interface
- Integrity: Local scheduler isolated.

What do we propose? - Where to change in previous policies
Proposed Design

Big Picture:
- Identify and isolate “policies” and “mechanisms”.
- Re-use “policies”
- Re-implement “mechanisms”

Advantages:
- Flexibility: Coscheduling independent of device driver / firmware.
- Generality: Same module for all schemes.
- Modularity: Re-use across platforms.
- Portability: Standard interfaces defined.
- Integrity: Local scheduler involved.

What do we propose? - A Generic framework for coscheduling
Addressing global issue 2 -
Coordinated Coscheduling

What do we propose? - A customizable coscheduling scheme (CC)
Addressing global issue 3: - Workload / Environment

- 16 node Myrinet connected cluster, 1GB RAM.
- Linux 2.4, MPI over VIA, Berkeley-VIA
- Lanai-9 Myrinet NI cards, 8MB on-chip memory.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Applications</th>
<th>Communication Intensity</th>
</tr>
</thead>
<tbody>
<tr>
<td>W13</td>
<td>(MG, MG, MG, MG, MG, MG)</td>
<td>hi:hi:hi:hi:hi:hi</td>
</tr>
<tr>
<td>W15</td>
<td>(LU, LU, LU, LU, LU, LU)</td>
<td>me:me:me:me:me:me</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Category</th>
<th>Workload Mix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel Only</td>
<td>w11, w12, w13, w14, w15</td>
</tr>
<tr>
<td>Parallel + CPU</td>
<td>(w11...w15) + 1 sb</td>
</tr>
<tr>
<td></td>
<td>(1,2,4,6) sb + 2 MGs</td>
</tr>
<tr>
<td>Parallel + IO</td>
<td>(w11...w15) + 1 sched.bench</td>
</tr>
</tbody>
</table>

(a) Parallel Workload Composition (MPL6) (b) Executed Combinations (sb: sched.bench)

Table: Workload mixes used in this study.
Performance : Execution Time

MPL = 6
CC > SB > PB > DCS > Local

What are the results? - Average wall clock time of various workloads
Where does time go?

- Total CPU Time = Useful work time + Non-useful spin time
- Wall Clock Time = Total CPU time + Block time + others...

What are the results? - Application profiling to isolate useful work..
Tolerance : Standard Deviation

$W_{L2}, W_{L4} : $ Mixed

$W_{L1}, W_{L3}, W_{L5} : $ Uniform

What are the results? - Average wall clock time of various workloads
Fairness : CPU Utilization

What are the results? - CPU utilization on a representative node.

Figure: CPU Utilization results for w13 (MPL=4)
Scalability (vertical)
Effect of MPL

- Rate of increase higher in Local, DCS, PB than SB, CC.
- Glimpse comparison to batch scheduling.

What are the results? - Effect of MPL on the average wall clock time.
Mixing CPU intensive jobs (WI3)

- CC & SB tolerate load better (low *overhead* in (a)).
- CC and SB exploit idle cycles well (low *slowdown* in (b)).

(a) Effect on parallel job
(b) Effect on CPU job
Mixing I/O intensive jobs (WI3)

- Jobs in all schemes get equally affected (similar overheads).
- Very Insignificant slowdown in I/O across all schemes.

What are the results? - Parallel + I/O intensive mix.
Conclusions

- **Primary Contributions :-**
  - Modular, Flexible framework for deploying coscheduling.
  - Fair, high performing, new CC scheme.

- **Significant findings :-**
  - Blocking based schemes better than spinning (Linux).
  - CC and SB scale well (Vertically) at high MPL of 6.
  - CC and SB get equal or better than Batch scheduling.
  - CC marginally better than SB, added QoS Potential.

- **Other Advantages :-**
  - Can implement all policies with CC approach.
  - Can use framework with all ULN libraries.

What can we **conclude**?
Future Work

- Horizontal scalability ( > nodes, GM)
- Release assumptions:
  - No paging (All apps fit well in memory).
  - Homogeneous nodes availability.
  - All apps of same priority.
- Optimizations in CC mechanism.
- Allocation problems in coscheduling.
- Dynamic Communication pattern identification.
- Integrated coscheduling as a feature in OS.
- True end-to-end QoS with support from scheduler.

Is there a future for this? Lots still to do….!! 😊
Thanks for your Time !! 😊
Difference with ICS ?
(Optional slide)

- SB registers for ALL incoming messages.
- ICS registers for expected messages for which a send has been done earlier.
- ICS is more tightly coupled (works on a send-recv pair).
- ICS is not too suitable in MPI environment.