Scheduling on Clusters

Presented by

Eric Hardisty
Effective Distributed Scheduling of Parallel Workloads

Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler
Motivation

• “...communication and synchronization events within the parallel applications provide sufficient information for coordinating the scheduling of cooperating processes.”

• Try to use the info *implicit* in the synchronization behavior of cooperating processes to schedule them.

• Seeks to improve upon the *explicit* nature of basic coscheduling.
Coscheduling Background

- Each process in a “task force” of cooperating processes is scheduled concurrently on the system.
- Similar to virtual memory--failure to schedule all the processes in a task force’s working set results in process thrashing.
- Scheduling done using an Ousterhout matrix.
Weaknesses of Coscheduling

- “Fault-tolerant, scalable coscheduling is non-trivial to design and implement.”
- The failure mode (i.e. if one processor dies, then the whole machine dies) does not work well in a distributed environment.
- The needs of I/O bound jobs are ignored.
- Processes must busy wait during I/O which is wasteful of CPU cycles.
Weaknesses of Previous Work

• Previous work seemed to indicate that coscheduling was necessary for fine-grained parallel applications.

• Many did not consider different blocking algorithms, local schedulers or programming models.
Weaknesses of Previous Work (con’t)

- Local scheduling with two-phase block was examined.
- Little improvement was seen because the spin time was short when compared to the cost of a context switch.
Bulk-synchronous SPMD Simulator Design

- $g =$ Computational granularity
- $v =$ Variation (load imbalance)
- $c =$ Communication
- Two barriers: one just before communication starts, and one just after it ends before the next round begins.
- $L =$ Latency. Constant.
Simulation Workload Patterns

• Barrier -- Processes only synchronize at the opening barrier.

• NEWS -- (North, East, South, West) Each process needs information from four of its neighbors.

• Transpose -- Each process communicates with every process.
Local Scheduler

• Overall system performance is highly dependent on this. More later...

• Taken from SVR4 Unix (Solaris 2.4)

• Key features:
  • Processes waking up from waiting on an event are given a priority boost that likely makes them immediately get the CPU.
  • Priority boost for yielding the CPU before time quantum expiration. (i.e. the process blocks for whatever reason)
  • Priority penalty for using the entire time quantum.
  • Starvation prevented using a timer that triggers once per second.
Local Scheduler (con’t)

- They assume, pessimistically, that the fairness timer (once per second) and the quantum timer (once per 10 ms) expire independently across processes.

- They assume, optimistically, that the cost of a coscheduling context switch is equal to a local one. In actuality, it can be an order of magnitude slower (as modeled in the next paper).
Slowdown

\[ S = \frac{\text{Local}}{\text{Cosched}}. \]
Local Scheduling with Immediate Blocking
Varying Latency and Context Switch Cost

• Previous chart assumed constant latency and context switch costs.

• Both of these can vary widely in current systems. How sensitive is the analysis to these changes?

• Vary imbalance, latency, and context switch costs. What happens?
### Impact of Workload and System Parameters on Immediate Blocking

<table>
<thead>
<tr>
<th>Barrier</th>
<th>Computation Granularity</th>
<th>Fine</th>
<th>Medium</th>
<th>Coarse</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Low</td>
<td>1.05</td>
<td>1.01</td>
<td>0.97</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>0.97</td>
<td>0.82</td>
<td>0.93</td>
</tr>
<tr>
<td></td>
<td>Low</td>
<td>1.01</td>
<td>1.07</td>
<td>0.95</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>1.01</td>
<td>0.92</td>
<td>0.92</td>
</tr>
<tr>
<td></td>
<td>Low</td>
<td>1.09</td>
<td>0.96</td>
<td>0.94</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>1.09</td>
<td>0.96</td>
<td>0.94</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>News</th>
<th>Computation Granularity</th>
<th>Fine</th>
<th>Medium</th>
<th>Coarse</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Low</td>
<td>0.96</td>
<td>0.96</td>
<td>0.91</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>1.01</td>
<td>0.94</td>
<td>0.93</td>
</tr>
<tr>
<td></td>
<td>Low</td>
<td>3.64</td>
<td>3.08</td>
<td>1.11</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>3.07</td>
<td>2.93</td>
<td>1.36</td>
</tr>
<tr>
<td></td>
<td>Low</td>
<td>13.1</td>
<td>10.8</td>
<td>1.50</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>13.1</td>
<td>10.8</td>
<td>1.50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transpose</th>
<th>Computation Granularity</th>
<th>Fine</th>
<th>Medium</th>
<th>Coarse</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Low</td>
<td>0.82</td>
<td>0.82</td>
<td>0.91</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>0.81</td>
<td>0.82</td>
<td>0.91</td>
</tr>
<tr>
<td></td>
<td>Low</td>
<td>4.31</td>
<td>4.07</td>
<td>1.50</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>4.31</td>
<td>4.07</td>
<td>1.50</td>
</tr>
<tr>
<td></td>
<td>Low</td>
<td>2.46</td>
<td>2.42</td>
<td>1.89</td>
</tr>
<tr>
<td></td>
<td>High</td>
<td>2.46</td>
<td>2.42</td>
<td>1.89</td>
</tr>
</tbody>
</table>

The tables show the impact of different workload and system parameters on immediate blocking, with values indicating the blocking probability under various conditions.
Two-phase fixed-spin

- Immediate blocking does not work well at all for fine-grained applications.

- Try two-phase fixed-spin blocking--the waiting process spins for a predetermined time, then blocks if the time expires.

- When acquiring shared memory locks, a good time to pick is the context switch time. What about for clusters when communication enters into the equation?
Spin time = context switch cost
Impact of Scheduler

• The improvement occurs because cooperating processes become dynamically coordinated after a barrier.

• This is a side effect of the SVR4 scheduler’s policy that boosts the priority of a woken up process.

• Beneficial because the processes enter the communication phase immediately after the first barrier.
What happens if we vary the spin-time?
Varying the Fixed-Spin Time

2xCS

4xCS

8xCS
Adaptive Blocking

- Adjust the spin-time based on observed program behavior.
- They develop a load imbalance oracle to compare their adaptive system to.
- The idea is that they know exactly when to block at a barrier vice spin.
Some Definitions

- **S**: The minimum spin-time before blocking that ensure coordinated processes remain coordinated. We *always* spin for this long on a read.

- **V**: The load imbalance past which it’s better to block than spin.
  - If \(\nu\), the load imbalance > \(V\), then spin for \(S\).
  - \(\nu < V \Rightarrow\) spin for \(\nu + \text{(network RTT)}\)
• The global loss when a process blocks for a barrier is the resulting increase in the time for all processes to be scheduled after completing the barrier: $2 \times CS$

• Benefit is $\nu/2 - CS - S$

• Blocking is beneficial whenever the global benefit exceeds the global cost.

• Block when $\nu > 10 \times CS$
Determining $v$

- **Local Approximation**
  - Record the max wait time while waiting for the barriers.
  - Can result in some processes approximating values $> v$ if the processes are not coordinated.
  - As a heuristic, they disregard the top 10% of the generated values as outliers.

- **Global Approximation**
  - Use the barrier.
  - Root node knows the imbalance is equal to the longest waiting time of any given process.
  - Root calculates the spin time and propagates it with the barrier completion message.
  - Root ignores the top 10%
Scheduler Sensitivity

- Previous work used Round-Robin scheduling instead of the MLF type scheduling they use.
- What happens when they use RR scheduling in their simulator?
Nothing good!
Wrapping up...
“Our hope is that an implementation of implicit scheduling on a network of workstations will be more portable, scalable, and fault-tolerant, and yet less complex, than a coscheduling implementation with equivalent performance.”
Impact of Workload and System Parameters on Next Generation Cluster Scheduling

Yanyong Zhang, Jose Moreira
Motivation

• Pure coscheduling does not work well with non-scientific, I/O intensive jobs such as databases and web services.

• Dynamic coscheduling seems to work better, but how much better? What factors affect the performance?

• Provides a comparison between the various dynamic coscheduling techniques.
Scheduling Strategies

- Coscheduling (aka Gang Scheduling): Coordinated decision at the end of a quantum.
- Local: The native OS on a node makes the scheduling decision independently of the other nodes.
- These two are used as a basis for comparison.
Scheduling Strategies

- Dynamic Coscheduling: Rely on messaging to guide the system toward coscheduled execution with no coordinated effort. Like the Arpaci-Dusseau paper (Ref 1).

- Several different flavors depending on how a process waits (i.e. block, spin-block, or spin-yield) and what it does when a message arrives (i.e. do nothing, interrupt, or periodically examine message queues).
Dynamic Coscheduling Varieties

- **Spin Block**: Like the Arpaci-Dusseau paper.
- **Spin Yield**: Lower priority to lowest, then boost the priority of another process. Continues to spin, just at a lower priority.
- **Demand-based Coscheduling (DCS)**: Incoming message causes preemption if recipient is not scheduled.
Dynamic Coscheduling Varieties

- Periodic Boost (PB): A kernel thread becomes active every millisecond, checks the message queues, and boosts the priority of a process based on some heuristic.

- They use a simple heuristic (which they change later)--use RR, stopping at the first process with a message waiting but not consumed.
Dynamic Coscheduling Varieties

- Combinations allowed as well.
- DCS with Spin-Block (DCS-SB)
- Periodic Boost with Spin-Block (PB-SB)
- DCS with Spin-Yield (DCS-SY)
- Periodic Boost with Spin-Yield (PB-SY)
Simulator

- Models processes arriving at arbitrary times.
- Uses the Solaris MLF queueing discipline.
- 8 Workloads created to model data obtained from the Lawrence Livermore Supercomputing lab.
- 4 Communication patterns...
Communication Patterns
Skewness

- Imbalance/skewness can have a significant impact on results.
- Larger skewness implies a greater mismatch between when processes reach the communication events.
- Overall execution time depends on who gets there last.
- Effectiveness can be evaluated by how well they hide the increase in execution time.
Impact of Load
Impact of Nature of Workload
Impact of MPL

- Local: overhead dominates since less time is spent on useful computation. (Figure 5)
- PB and SB perform well, but cannot keep the CPU busy for small MPL values.
- Higher MPL implies more context switching, but the reduction in idle time makes up for it.
Impact of Skewness
Impact of System Overheads

- Lowering context switch time from 200 to 100 usec results in faster response times. Not surprising.

- More interesting: doubling the cost of interrupt handling doesn’t have a huge impact on the response times.

- Lesson: make your context switches fast. :)

Impact of Communication Pattern

(a) Lin Tree

(b) NN AA

(c) NN Tree

(d) AA Lin
Impact on System Size

- For a cluster of 32 workstations, is it better to partition it into two logical clusters of 16 workstations or treat it as one unit?
- Better to treat it as one unit.
- Doubling system size more than halves the wait time. More than makes up for the minor increase in execution time.
Fairness

• Mixed workload ran on 32 node system.

• Calculated normalize average slowdown and coefficient of variation of slowdown (lower is more fair).

• Local: 1.632, GS = 0.099, SB = 0.203, PB = 0.504

• My question: Do we really want fair?
Periodic Boost
Heuristics

- When the PB thread is invoked, it finds processes in one of four positions:
  - S1: In the compute/send phase with no pending messages.
  - S2: In the compute/send phase with at least one pending message.
  - S3: In the receive phase and the matching send has arrived.
  - S4: In the receive phase and the message it waiting for has not arrived.
Periodic Boost
Heuristics

• A: S3 -> {S2, S1}
• B: S3 -> S2 -> S1
• C: {S3, S2, S1}
• D: {S3, S2} -> S1
• E: S2 -> S3 -> S1
Fairness of Heuristic

• Instead of boosting the next process of the appropriate type in RR order, boost the process that has had the CPU the least.

• It turns out adding this fairness quality doesn’t cause an increase in response time.
Conclusions

• There is a need for some coordinated scheduling effort between the nodes to accommodate the parallel jobs—Local’s performance is dismal.

• Dynamic coscheduling, in general, is a better alternative than GS (except when AA communication is frequent).
Conclusions

• GS does not perform as well when:
  • Skewness is high
  • I/O bound jobs are present
  • There’s a high degree of multiprogramming.
  • When the cost of explicit scheduling is high (i.e. large distributed context switching cost)