CMSC 714

Scheduling

Guest Lecturer: Sukhyun Song

Scheduling Techniques for Concurrent Systems

John K. Ousterhout

IEEE ICDCS'82

Introduction

- Motivation
  - People assume in OS scheduling that interactions between processes are the exception rather than the rule.
  - But it is not true any more.
    - Multiprocessor systems are appearing.
    - Cooperation between processes becomes widespread.
    - Traditional techniques will break down.
- Short-term scheduling
  - Two-phase blocking scheme
- Long-term scheduling
  - Three algorithms of coscheduling

Short-Term Scheduling

- Waiting is a fundamental aspect of communication.
  - It is unlikely that two processes reach the rendezvous point at exactly the same time.
  - A sender or a receiver should wait for the other party.
- Most short-term schedulers are inefficient.
  - Immediate blocking: When a process waits for some event, it is thrown off its processor and another process is activated.
  - It always requires two context swaps.
- Solution: to divide waiting into two phases
  - Pause: A process pauses until the event occurs.
  - Block: If pause time is exceeded, it relinquishes a processor.
  - Effective only for fine-grained communication
    - The event should occur very soon so that we do not get into the block phase.
    - Duration of the event wait < 2 x CS (context swap cost)

Thrashing and Process Working Sets (1)

- Naïve long-term scheduling
  - Processes are sending messages among themselves.
  - Half scheduled in odd time slices, the other in even slices
  - Most processes in the executing half will block awaiting messages from processes in the non-executing half.
- Process thrashing
  - Progress of parallel program is limited by scheduling decision rather than communication primitive speed.
  - Demand paging: Progress of program is limited by speed of swapping rather than memory reference speed.
- Solution: coscheduling
  - We should schedule a group of closely-interacting processes (process working set) for execution simultaneously.
  - Make parallel program progress fast
  - Estimating process working set dynamically is not easy.
    - To record message trace details: expensive

Thrashing and Process Working Sets (2)

- Assumptions
  - Process working sets statically specified by programmers.
  - A process can be loaded onto any processor, but cannot move after it starts execution.
  - A task force (TF) means a process working set.
  - TF coscheduled: All runnable processes are executing simultaneously on different processors.
  - TF fragmented: not coscheduled
- Goal
  - maximize avg # processors executing coscheduled processes
- Simulation parameters
  - P=50 processors / system
  - At most Q=16 processes / processor
  - Q large enough that allocation always succeeds
  - No TF has > P processes.

Coscheduling: Matrix Method

- Matrix of process slots (Figure 1)
  - P columns (processors) x Q rows (processes)
- Allocation
  - Find a row to accommodate all processes in a task force
- Scheduling
  - Round-robin mechanism
    - In time slice 0, we run processes in row 0.
- Alternate Selection
  - Executing processes can block awaiting terminal input.
  - Each processor scans its column to find a runnable process and runs it as a TF fragment.
- Drawback
  - Process space is partitioned into disjoint rows.
    - Internal fragmentation
  - Alternate selection may miss coscheduling.

Coscheduling: Continuous Algorithm

- Sequence of process slots (Figure 2)
  - Slots for P processors x Q processes
- Allocation
  - Place a window of width P slots at the left end of sequence
  - Move the window until finding enough empty slots to accommodate the new task force.
  - Space is more packed than the matrix method.
- Scheduling
  - Each time slice, we move the window until the first process is the leftmost process of a task force that has not been coscheduled in the current sweep. (Figure 3)
  - Alternate selection
- Drawback
  - A new task force may be divided between several holes.
    - External fragmentation

Coscheduling: Undivided Algorithm

- Allocation
  - The same as the continuous algorithm, but no holes
- Scheduling
  - The same as the continuous algorithm
- Good
  - Eliminate holes and increase coscheduling
- Bad
  - Space is less packed.

Simulation: Effect of System Load

- Coscheduling Effectiveness
  - Average (across time slices) of # processors executing coscheduled processes / # processors w/ runnable processes
  - 1 is ideal.
- System Load
  - # runnable processes / # processors (TF arrival rate)
- Figure 4
  - System load vs coscheduling effectiveness
  - Load increases, then effectiveness decreases.
    - Straddling (continuous, undivided)
    - Alternate selection (all three)

Simulation: Effect of Task Force Size

- Average Task Force Size
  - # processes in a task force
- Figure 5
  - Average TF size vs. coscheduling effectiveness
  - TF size increases, then effectiveness decreases.
  - Matrix method
    - Worst at large average task force size (>15)
    - It does not use space efficiently.
  - Continuous algorithm
    - Worst at small average task force size (5)
    - Large TF can be fragmented on many holes.
  - Undivided algorithm
    - It performs the best. (efficient space, no holes)

Simulation: Effect of Idle Processes

- Idle process
  - Waiting for external event such as terminal input

- Figure 7
  - Idle fraction vs. coscheduling effectiveness
  - Continuous
    - Worst at high idle fraction (> 0.8)
    - TF can be fragmented on holes.
  - Matrix and Undivided
    - Good at high idle fraction
    - Tend to allocate TFs in contiguous slots

Comparison

- Matrix
  - Fast allocation / scheduling
  - Internal fragmentation
- Continuous
  - Fast allocation / scheduling
  - Dense packing
  - External fragmentation
- Undivided
  - Slow allocation, fast scheduling
  - Dense packing

Effective Distributed Scheduling of Parallel Workloads

Andrea C. Dusseau et al.

ACM SIGMETRICS'96

Introduction

- Motivation
  - Coscheduling: Processes of a parallel job are run at the same time across processors in an explicit manner.
  - Fault-tolerant scalable coscheduling is non-trivial.
  - Round-robin mechanism for interactive job is bad.

- Introduce a local implicit scheduling
  - Philosophy: Communication events within the parallel applications provide sufficient information for coordinating the scheduling of cooperating processes.
  - Each scheduler is able to make independent decisions.
  - Implicit scheduling is a feasible alternative to coscheduling.
    - Previous researches: Local scheduling is insufficient for fine-grained parallel applications.

Background (1)

- Programming model
  - Bulk-synchronous: sequence of supersteps (Figure 1)
    - computation, opening barrier, communication, closing barrier
  - Single-Program Multiple-Data
- Parameters
  - P: # processes in a job
  - g: computation granularity (time)
  - v: load imbalance (difference of max and min g)
  - c: time between read events in comm.
  - L: network latency (each read time in comm.)
- Communication patterns
  - BARRIER: no communication
  - NEWS: grid communication, four neighbors
  - TRANSPOSE: all-to-all communication

Background (2)

- Basic design of local scheduling
  - Dynamic priority allocation scheme
    - A job’s priority is lowered if it runs w/o relinquishing a processor.
    - A job’s priority is raised if it sleeps frequently.
  - Pessimistic assumptions
    - Clock and timer expire independently across processors.
    - Multiple jobs arrive in the system at the same time, processes are randomly ordered in the local scheduling queues.
  - Optimistic assumptions about coscheduling
    - No skew of time quanta across processors
    - Global context-switch cost = local scheduler cost

Background (3)

- Synthetic workload
  - Parameter space is huge. (5 parameters)
  - c: communication granularity is fixed to 8us.
  - P: 32 processes in a job
  - 10 seconds / job

- Figure 2
  - Coscheduling: processes of a parallel job are run at the same time across processors in an explicit manner.
  - Time breakdown when the programs are coscheduled
  - Load imbalance v increases, then sync time increases.
  - NEWS and TRANSPOSE involve communication.
    - Computation granularity g decreases, then communication time increases.

Verification of Literature

- Local scheduling w/ immediate blocking
  - A process blocks immediately on sync. and comm.
  - In literature, it is worse than coscheduling at fine granularity.
- Figure 3
  - Slowdown compared to coscheduling
  - Load imbalance up: local gets better
    - We can run other processes by context-switching.
  - Coarse-grain computation (g>=5ms): Local scheduling wins.
  - Fine-grain computation: Coscheduling wins.
    - Many steps of sync. and comm.: Context-switching happens a lot for local scheduling.
- Figure 4
  - Varying latency L and context-switching cost CS
  - Coarse-grain, high load imbalance: Local scheduling wins.
  - Low CS, High L: local is good, comp-comm overlap

Two-Phase Fixed-Spin

- Algorithm
  - A waiting process spins for a predetermined spin time.
  - If a response is received before time expires, it continues executing, o.w., it blocks and moves on to another process.
- Figure 5 (spin time = 1CS = 200us)
  - Better than immediate blocking (compared to Figure 3)
  - Still bad at fine-grain computation
- Figure 6 (spin time = 2, 4, 8CS = 400us)
  - Varies spin time for NEWS communication pattern
  - Better than 1CS
- Figure 7 (scheduling skew of 2CS)
  - Shows why spin time >= 2CS
  - P2 started CS to Job B right before Barrier is done.
  - P3 wants to read data from P2, spins until P2 finishes 2CS.

Adaptive Blocking

- Problem of fixed-spin
  - Hard to decide a proper spin time that is always beneficial
- Adaptive blocking strategy w/ load-imbalance oracle
  - Suppose we know load imbalance v of a program.
  - We'd like to decide a threshold parameter V s.t. it is beneficial if we take the following strategy of spinning time.
    - v > V: spin for 2CS, otherwise spin for v + 2L
  - Think of a case of spinning for 2CS
    - Benefit: time to run other jobs v/2 – 2CS – CS
    - Cost: scheduling skew 2CS (Figure 7)
    - Benefit > cost: v > 10CS
    - V = 10CS
- Figure 8
  - Closer to coscheduling

Approximation of Load-Imbalance v

- Local approximation
  - Each process uses max waiting time for barrier as the approximation of v
    - Handling outliers: disregard the top 10% of data points
- Figure 9
  - Worse than oracle at fine-grain computation
  - Underestimate v because of disregarding outliers
- Global approximation
  - If the barrier operation is implemented in software, each process sends a message to a root process.
  - The root process can record max waiting time for barrier and determine approximation of v.
    - Again, remove the top 10% of outliers
- Figure 10
  - Better than local approximation (Figure 9)

Sensitivity to the Local Scheduler

- Timer skew
  - Up to this point: pessimistic assumption
    - Timers are independent across processors.
- Figure 11
  - What if we synchronize timers?
  - Closer to coscheduling

- Round-robin scheduling
  - So far, we used priority-based scheduling.
- Figure 12
  - Round-robin scheduler is less robust.
  - Slowdown is 3.4x for some cases.