Lecture 7b: Parallel Scheduling
Parallel Job Scheduling

• “Job” is a collection of processes/threads that cooperate to solve some problem (or provide some service)
  • *Not* independent!

• How the components of the job are scheduled has a major effect on performance
  • Want scheduler to be aware of dependences

• We will look at two major strategies
  • *Space sharing* – each job has dedicated processors
  • *Time sharing* – multiple jobs share same processors
Why Job Scheduling Matters?

• Recall threads in a job are not independent
  • Synchronize over shared data
  • De-schedule lock holder, other threads in job may not get far
• Cause/effect relationships (e.g. producer-consumer problem)
  • Consumer is waiting for data on queue, but producer is not running
• Synchronizing phases of execution (barriers)
  • Entire job proceeds at pace of slowest thread
Forms of scheduler-awareness

1. Know threads are related, schedule all at same time
   • Space sharing: all threads are from same job
   • Time sharing: group threads that should be scheduled together

2. Know when threads hold spinlocks
   • Don’t deschedule lock holder
   • Extends timeslice, but not indefinitely

3. Know about general dependences
   • E.g. infer producer/consumer relationships
Space Sharing Scheduling

- Divide processors into groups
  - Fixed, variable, or adaptive
- Assign job to dedicated set of processors
  - Ideally one CPU per thread in job
- Job waits until required number of CPUs are available (batch scheduling)

- Fixed: Always 2 groups of 4 CPUs.

- Variable: Currently 3 groups of 2, 4, and 2 CPUs; changes as jobs come and go.

- Adaptive: Job can ask for more CPUs as it runs.
Space Sharing

- Typically used on supercomputers
- Pros:
  - All runnable threads execute at the same time
  - Reduce context switch overhead (no involuntary preemption)
  - Strong affinity
- Cons?
  - Inflexible
    - CPUs in one partition may be idle while another partition has multiple jobs waiting to run
  - Difficult to deal with dynamically-changing job sizes
    - Adaptive scheme is complicated and uncommon
Choosing Jobs to Run

- At job creation, specify number of threads
- Scheduler finds set of CPUs
  - May negotiate with application

- How should scheduler choose jobs to assign to CPUs? What is optimal (in terms of average wait time)?
  - Uniprocessor scheduling $\rightarrow$ Shortest Job First (SJF) (shortest expected next CPU burst)
  - MP version $\rightarrow$ smallest expected number of CPU cycles ($\text{cycles} = \text{num}_\text{cpus} \times \text{runtime}$)
Estimating Runtime

- Estimates typically come from users who submit the jobs
  - Low estimates make it “easier” to do scheduling
  - But cause trouble if not accurate!
  - Soln: kill jobs that exceed estimate
    - What user behaviour does that incentivize?
- How accurate are user estimates?
- Can automatic estimates based on history do better?
- How much does it matter?
Space Sharing - FCFS

- Scheduling convoy effect
  - Long average wait times due to large job
  - Exists with FCFS uniprocessor batch systems
  - But becomes much worse in parallel systems
    - Leads to fragmentation of CPU space

Scheduler queue (CPUs, time)

| 1 | 3 | 2 | 2 | 4 | 2 | 2 | 2 | 1 | 4 |

Time

CPU allocation diagram:
- A
- B
- C
- D
- E
Solution: Backfilling

- Fill CPU “holes” from queue in FCFS order
- Not FCFS anymore. What can happen?
- Want to prevent “fill” from delaying threads that were in queue earlier
  - EASY (Extensible Argonne Scheduling System)
  - Make reservation for next job in queue

Scheduler queue (CPUs, time)

<table>
<thead>
<tr>
<th></th>
<th>E</th>
<th>D</th>
<th>C</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>1,3</td>
<td>2,2</td>
<td>4,2</td>
<td>2,2</td>
<td>1,4</td>
</tr>
</tbody>
</table>
Variations on Backfilling

- **EASY**
  1. Used FCFS to order jobs in queue
  2. Made reservation for first blocked job in queue
  3. Backfilled jobs by looking at queue one at a time

- **1. Ordering alternative**: include priority in queue
  - Administrative to distinguish between users
  - User to distinguish between own jobs
  - Scheduler to prevent starvation

- **2. Reservation alternatives**
  - All queued jobs get a reservation (too much can go wrong)
  - Queued job gets a reservation if it has been waiting more than a threshold

- **3. Queue lookahead**
  - Use dynamic programming to determine optimal packing
Parallel Time Sharing

- Each CPU may run threads from multiple jobs
  - But with awareness of jobs
- Co-scheduling (Ousterhout, 1982)
  - Identify “working set” of processes (analogous to working set of memory pages) that need to run together
- Gang scheduling
  - All-or-nothing: co-scheduled working set is all threads in the job
- Get scheduling benefits of dedicated machine
- Allows all jobs to get service
Multiprogramming level is typically controlled by either:

- Monitoring memory demand, or
- Fixed number of slots (rows)

- e.g. IBM LoadLeveler Gang Scheduling allows up to 8 sets of jobs to be multiprogrammed on a set of CPUs
Gang Scheduling Issues

• All CPUs must context switch together
  • To avoid fragmentation, construct groups of jobs that fill a slot on each CPU
    • E.g., 8-CPU system, group one 4-thread job with two 2-thread jobs
  • Inflexible
    • If 4-thread job blocks, should we block entire group, or schedule group and leave 4 CPUs idle?

• Alternative 1: Paired gang scheduling
  • Identify groupings with complementary characteristics and pair them. When one blocks, the other runs

• Alternative 2: Only use gang scheduling for thread groups that benefit
  • Fill holes in schedule with any single runnable thread from those remaining
Example: Effect of Gang Scheduling

- LLNL gang scheduler on 12-CPU Digital Alpha 8400
- Parallel gaussian elimination program
- Run concurrently with 12 single-threaded interfering processes
- Benefits due to synchronization effects and better cache use

Source:
Forms of scheduler-awareness

1. Know threads are related, schedule all at same time
   - Space sharing: all threads are from same job
   - Time sharing: group threads that should be scheduled together

2. Know when threads hold spinlocks
   - Don’t deschedule lock holder
   - Extends timeslice, but not indefinitely

3. Know about general dependences
   - E.g. infer producer/consumer relationships
Knowing about Spinlocks

- Thread acquiring spinlock sets **kernel-visible flag**
- Clears flag on release
- Scheduler will **not immediately deschedule a** thread with the flag set
  - Gives thread a chance to complete critical section and release lock
  - Spinlock-protected critical sections are (supposed to be) short
  - Does not defer scheduling indefinitely
Forms of scheduler-awareness

1. Know threads are related, schedule all at same time
   - Space sharing: all threads are from same job
   - Time sharing: group threads that should be scheduled together

2. Know when threads hold spinlocks
   - Don’t deschedule lock holder
   - Extends timeslice, but not indefinitely

3. Know about general dependences
   - E.g. infer producer/consumer relationships

Generally harder to figure out. Not going to discuss this one.
OS Noise

• Or: how to schedule OS activities
• Massively parallel systems are typically split into I/O nodes, management nodes & compute nodes
  • Compute nodes are where the real work gets done
  • Run customized, lightweight kernel on compute nodes
  • Run full-blown OS on I/O nodes and mgmt nodes
  • Why?
• Asynchronous OS activities perturb nice scheduling properties of running jobs together
  • Up to a factor of 2 performance loss in real large-scale jobs
  • Need to either eliminate OS interference, or find ways to coordinate it as well
Illustration of OS Noise Issue

- The following 4 slides are extracted from a tutorial given at EuroPar 2004
  (www.di.unipi.it/europar04/Tutorial2/tutorial_europar04.ppt)
  - They illustrate the issue with OS Noise on the ASCI Q supercomputer, published in Supercomputing 2003 ("The case of the missing supercomputer performance")
  
  "Achieving Usability and Efficiency in Large-Scale Parallel Computing"
  
  - Kei Davis and Fabrizio Petrini {kei,fabrizio}@lanl.gov
  - Performance and Architectures Lab (PAL), CCS-3
    - (Computer and Computational Sciences Division, Los Alamos National Labs)
Performance of SAGE on 1024 nodes

- Performance consistent across QA and QB (the two segments of ASCI Q, with 1024 nodes/4096 processors each) \( \iff 4 \text{ processors / node} \)
- Measured time 2x greater than model (4096 Processor Elements)

There is a difference why?

OS activity is the culprit!

Lower is better!
The effect of the noise

- An application is usually a sequence of a computation followed by a synchronization (collective):

- But if an event happens on a single node, it can affect all the other nodes.
Effect of System Size

- The probability of a random event occurring increases with the node count
We can tolerate the noise by coscheduling the activities of the system software on each node.
Example Cluster Scheduler: SLURM

- **Simple Linux Utility for Resource Management**
  - Performs resource management within single cluster => 3 roles:
    - Allocates access to computer nodes and their interconnect
    - Launches parallel jobs and manages them (I/O, signals, time limits, etc.)
    - Manages contention in the queue
  - Developed by Lawrence Livermore National Lab (LLNL)
    - With help from HP, Bull (European high performance computing company), Linux NetworX, and others
  - Open Source
  - Extensible, provides flexible plugin mechanism
  - Active development still on-going
  - Widely used on high performance compute clusters
SLURM Features

- Plugins support multiple scheduling policies
  - FIFO
  - Backfilling
  - Gang Scheduling
    - Requires multi-core awareness at slurmctld
  - Priority-based preemption
  - Topology-aware scheduling
    - Reduce contention on interconnect
- Includes many management & accounting features
SLURM Architecture

- Graphic from https://computing.llnl.gov/linux/slurm/slurm.sc08.bof.pdf
Further readings

- Scheduling problem encountered in many contexts
  - e.g., cluster scheduling in datacenters
- Mesos (NSDI 2011)
  - [https://people.eecs.berkeley.edu/~alig/papers/mesos.pdf](https://people.eecs.berkeley.edu/~alig/papers/mesos.pdf)
- Sparrow (SOSP 2013)
  - [https://dl.acm.org/citation.cfm?id=2522716](https://dl.acm.org/citation.cfm?id=2522716)
- Borg (Eurosys 2015)
  - [https://ai.google/research/pubs/pub43438](https://ai.google/research/pubs/pub43438)
- Hawk (USENIX 2015)
  - [https://www.usenix.org/conference/atc15/technical-session/presentation/delgado](https://www.usenix.org/conference/atc15/technical-session/presentation/delgado)
- Firmament (OSDI 2016)