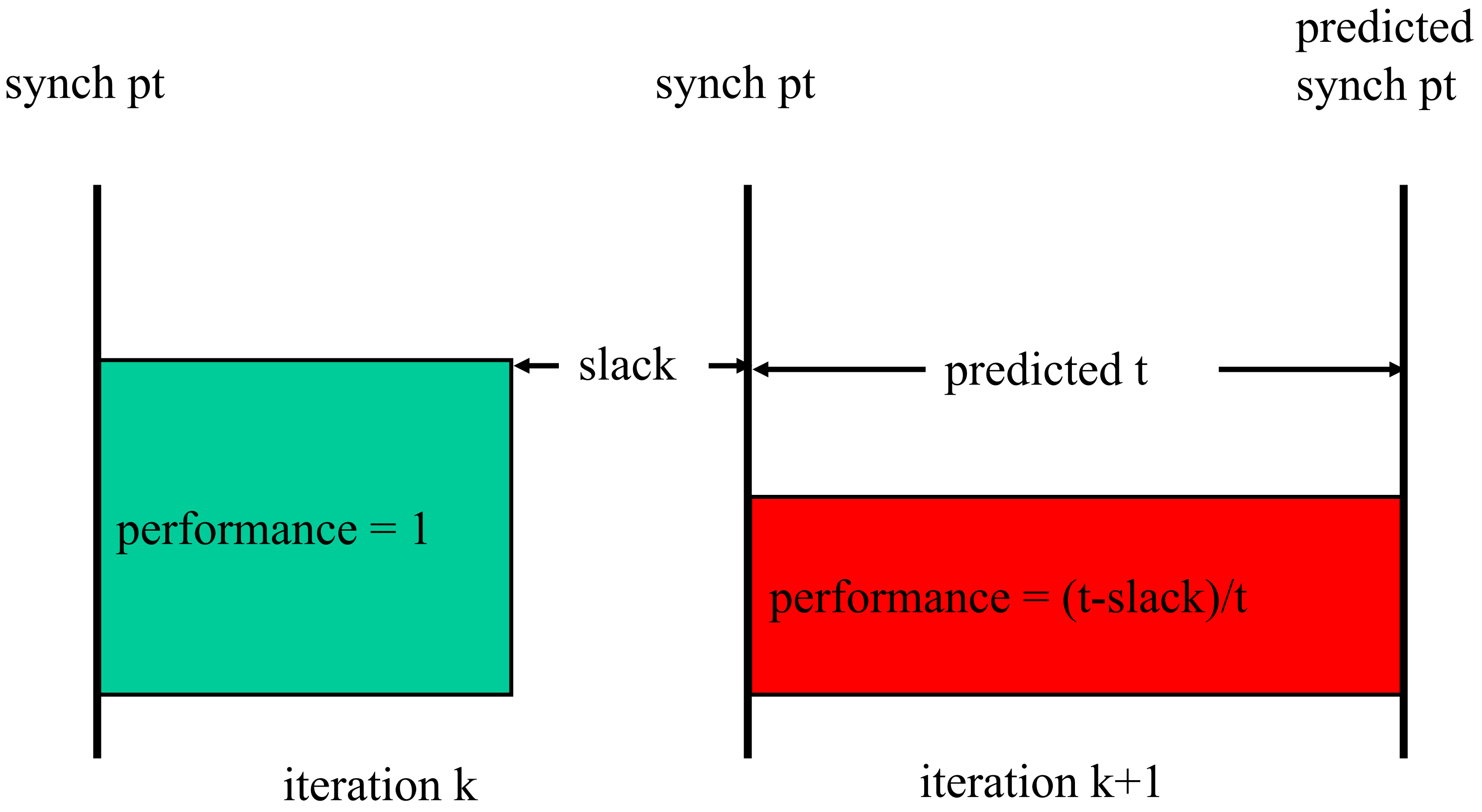
# Another reason to slow down CPU: leveraging load imbalance

- Best course is to keep load balanced
  - Load balancing is hard
  - Decrease frequency/voltage to save energy if not critical node
- How to tell if not critical node?
  - Assume global synchronization (e.g., barrier) occurs after each program iteration
    - No benefit to arriving early
  - Measure blocking time
  - Assume program behavior (mostly) the same between iterations

# Example



synch pt

synch pt

predicted synch pt

slack

predicted t

performance = 1

performance = (t-slack)/t

iteration k

iteration k+1

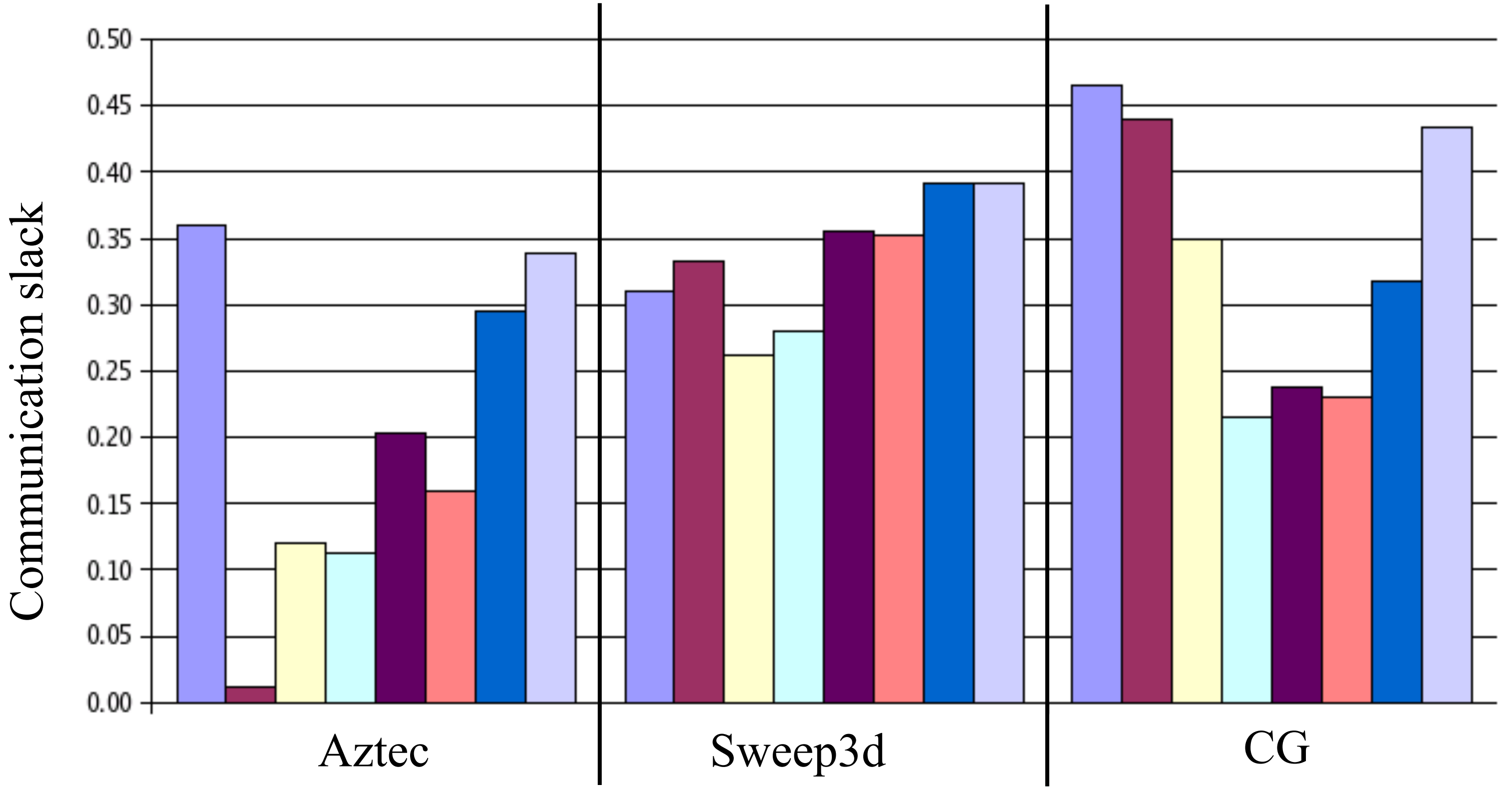
Reduced performance & power
→ Energy savings

# Measuring slack

- Measure blocking operations by intercepting MPI calls
  - Receive
  - Wait
  - Barrier

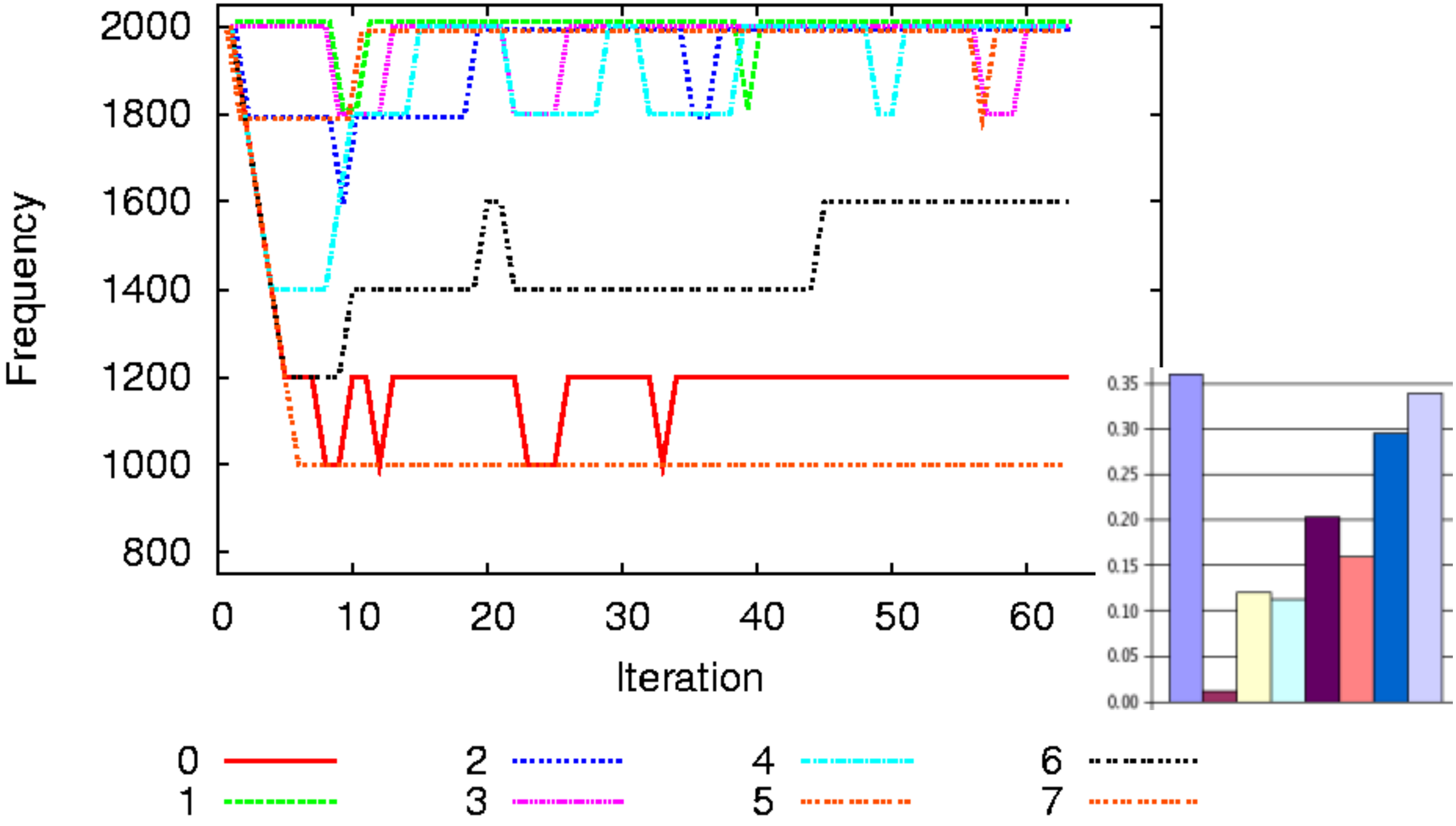
- Compute slack over one or more iterations
  - Measure times for computing and blocking phases
    - $T = C_1 + B_1 + C_2 + B_2 + \ldots + C_n + B_n$
  - Determine aggregate slack
    - $S = (B_1 + B_2 + \ldots + B_n)/T$

# Per-node slack



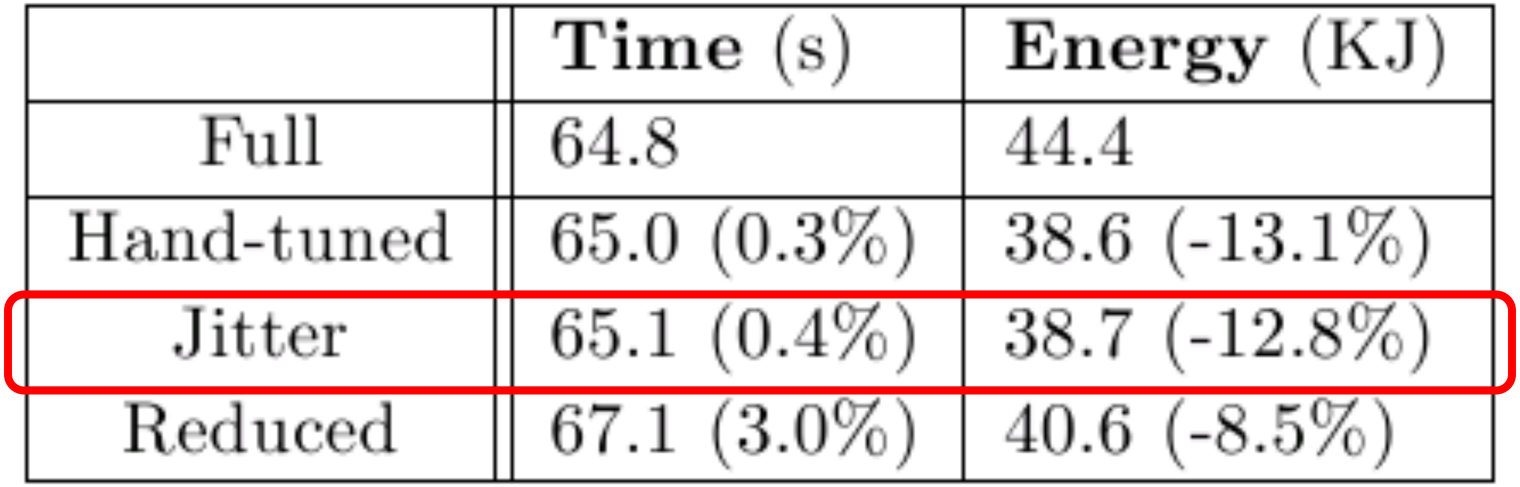
- Slack
  - Varies between nodes
  - Varies between applications
- Use net slack
  - Each node individually determines slack
  - Reduction to find min slack

# Aztec frequencies



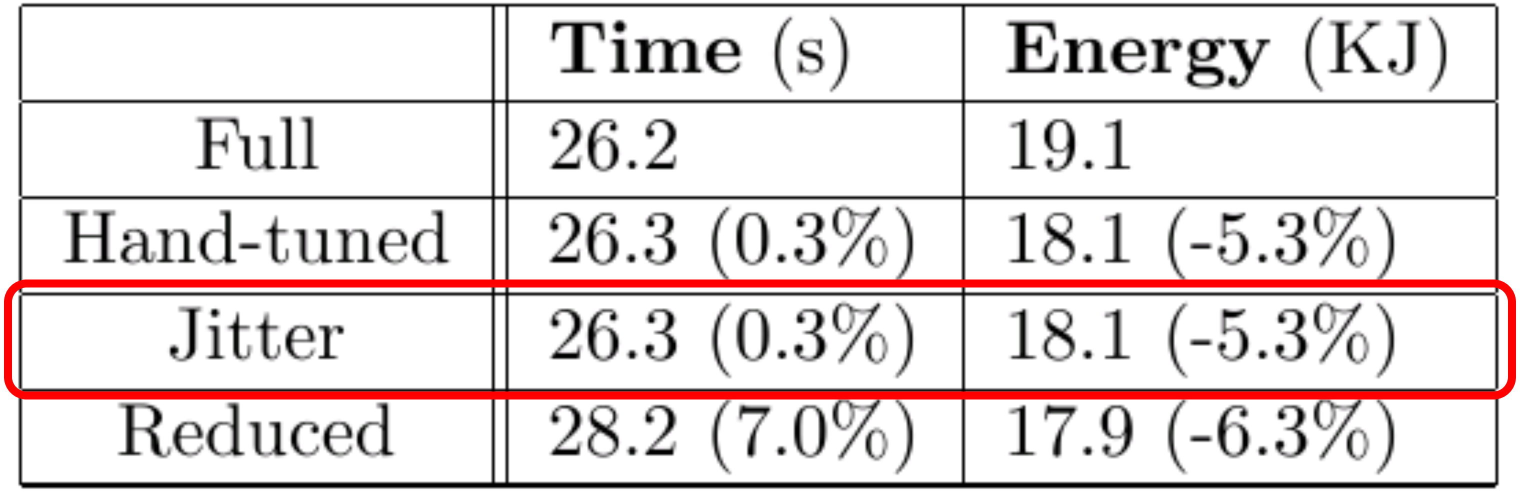
# Results

Aztec



| | Time (s) | Energy (KJ) |
|---|---|---|
| Full | 64.8 | 44.4 |
| Hand-tuned | 65.0 (0.3%) | 38.6 (-13.1%) |
| Jitter | 65.1 (0.4%) | 38.7 (-12.8%) |
| Reduced | 67.1 (3.0%) | 40.6 (-8.5%) |

Sweep3d



| | Time (s) | Energy (KJ) |
|---|---|---|
| Full | 26.2 | 19.1 |
| Hand-tuned | 26.3 (0.3%) | 18.1 (-5.3%) |
| Jitter | 26.3 (0.3%) | 18.1 (-5.3%) |
| Reduced | 28.2 (7.0%) | 17.9 (-6.3%) |

# Stage 2: don’t increase execution time

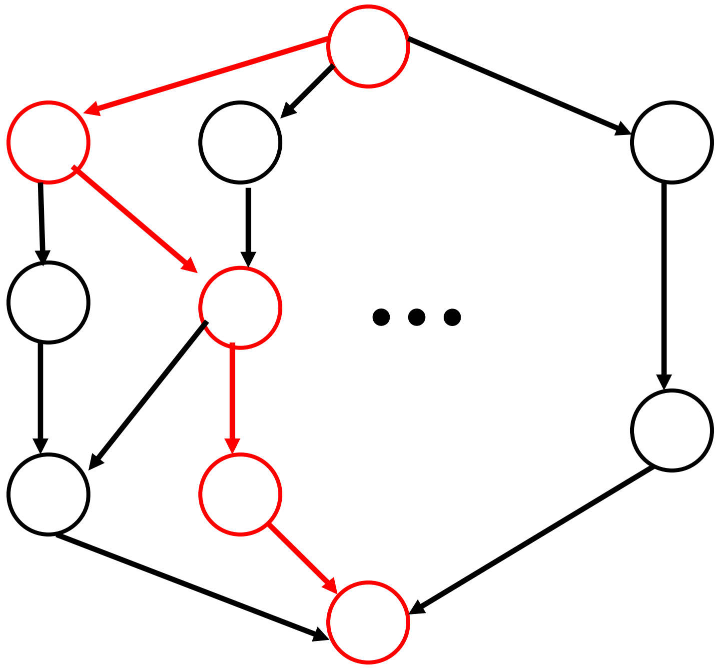
- HPC application programmers ***do not*** care about saving energy
  - If you can save energy with no increase in execution time, great!
  - Otherwise, go away: they won’t think a tradeoff of, say, 20% energy savings for a 1% time increase is a good thing

# Overall approach

- Divide the application into discrete “tasks”
- Create a task graph to represent execution behavior
- Execute the tasks on a process at every frequency
- Use linear programming to determine frequency per task
  - Constraint: do not slow down program
- Validate by re-running application, using schedule

# Program execution time

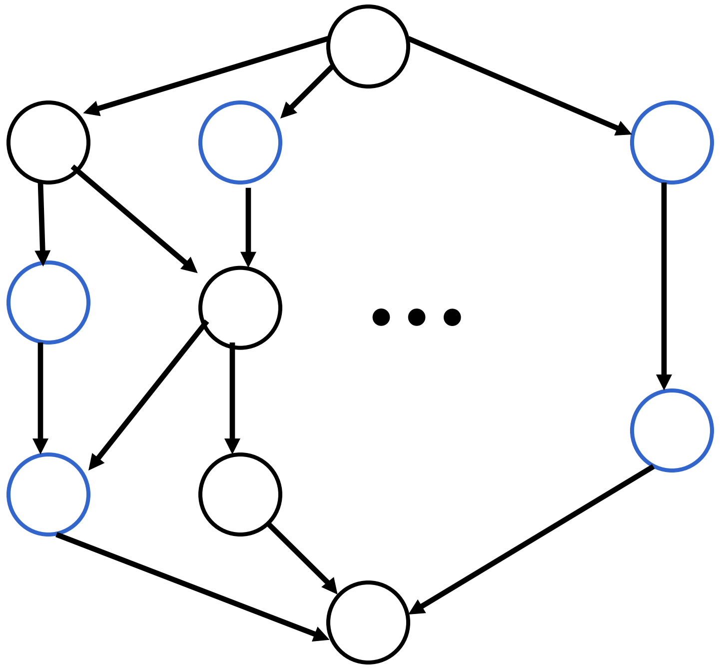
- Determined by *critical path* (you know this!)
  - Tasks not on critical path are (potentially) scalable
    - Running slower may not impact execution time



Tasks on critical path must execute at the fastest frequency

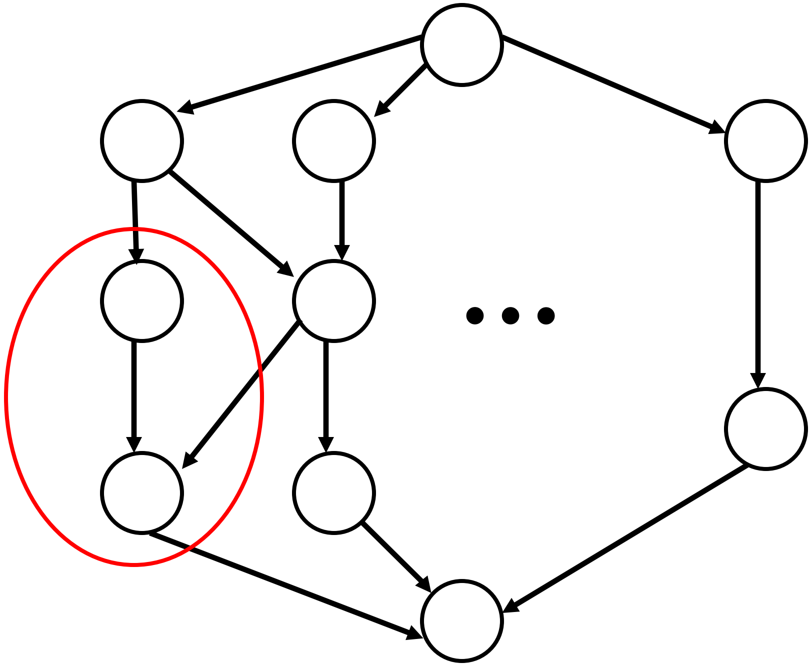
# Program execution time

- Determined by *critical path*
  - – Tasks not on critical path are (potentially) scalable
    - Running slower may not impact execution time



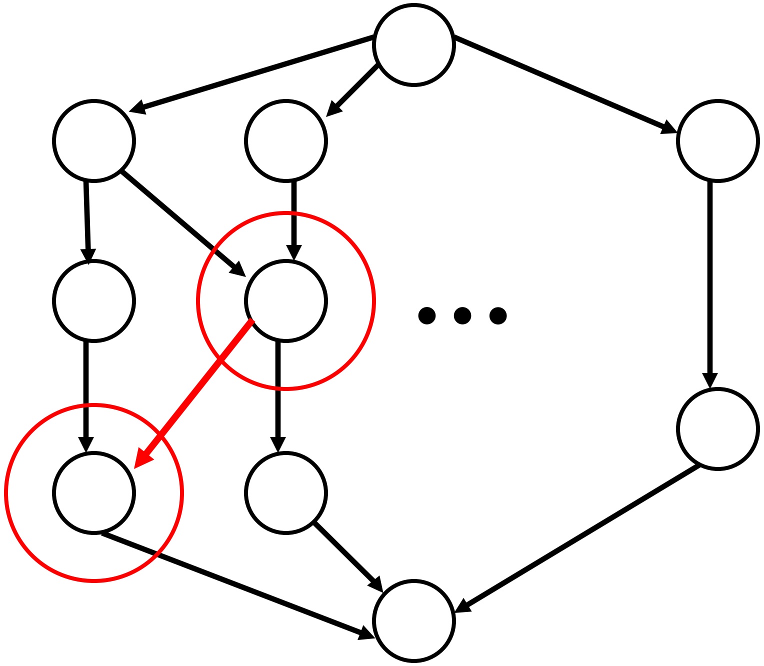
Tasks not on the critical path can stretch to fill in the time between processes on the critical path

# Task Precedence Constraint (1)



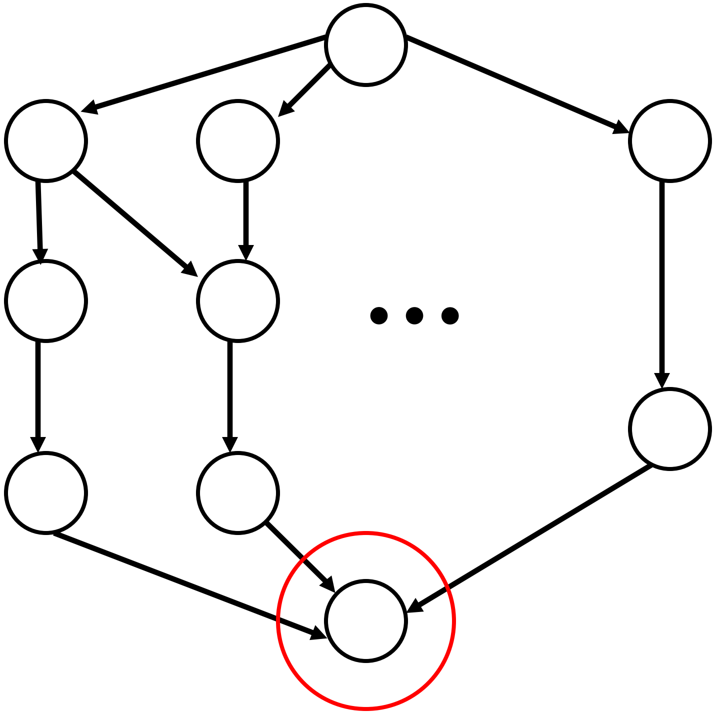
Task Cannot Start
Until Same-Process
Predecessor Done

# Task Precedence Constraint (2)



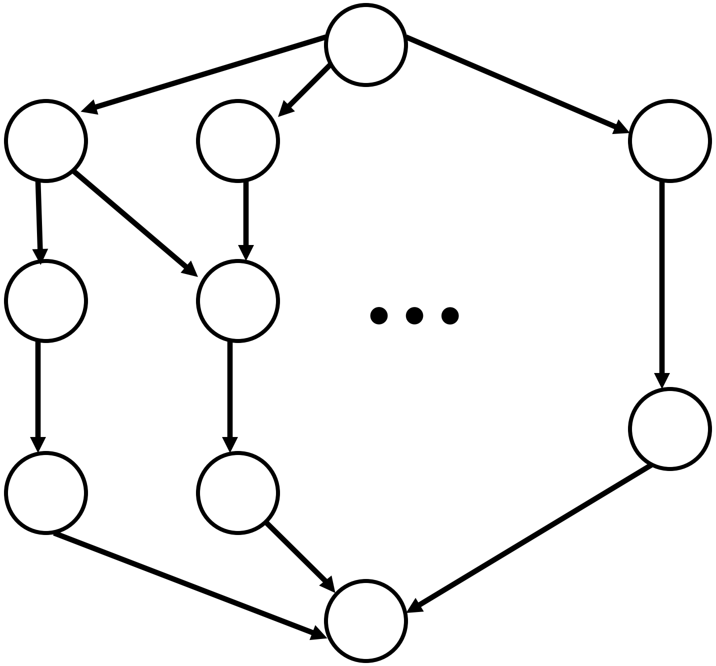
Task Cannot Start
Until Cross-Process
Predecessor Done
(Plus Message Latency)

# Application Timing Constraint



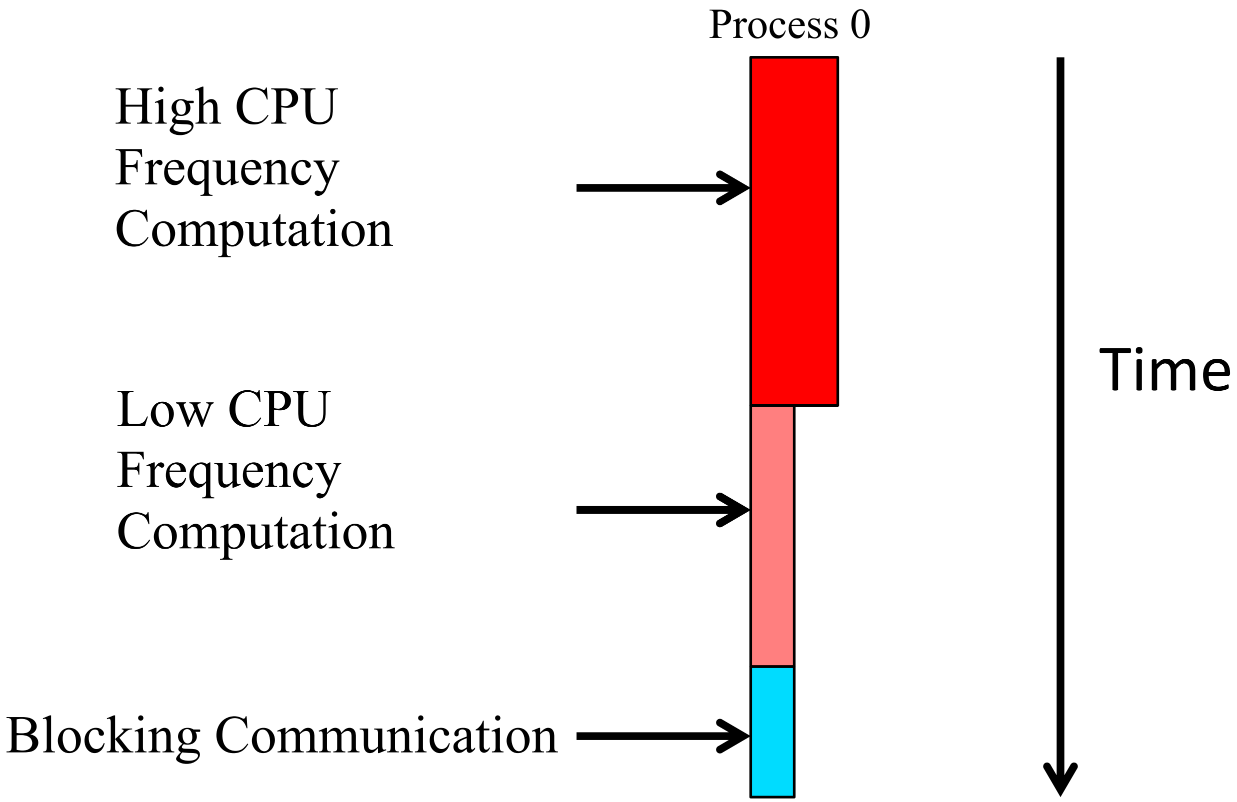
Sink Vertex Must
Complete Within
Original Execution Time

# Objective Function



Minimize Sum of Tasks' Energy
Plus "Idle Energy"

# Adagio: Converting the theoretical to the practical



Process 0

High CPU Frequency Computation

Low CPU Frequency Computation

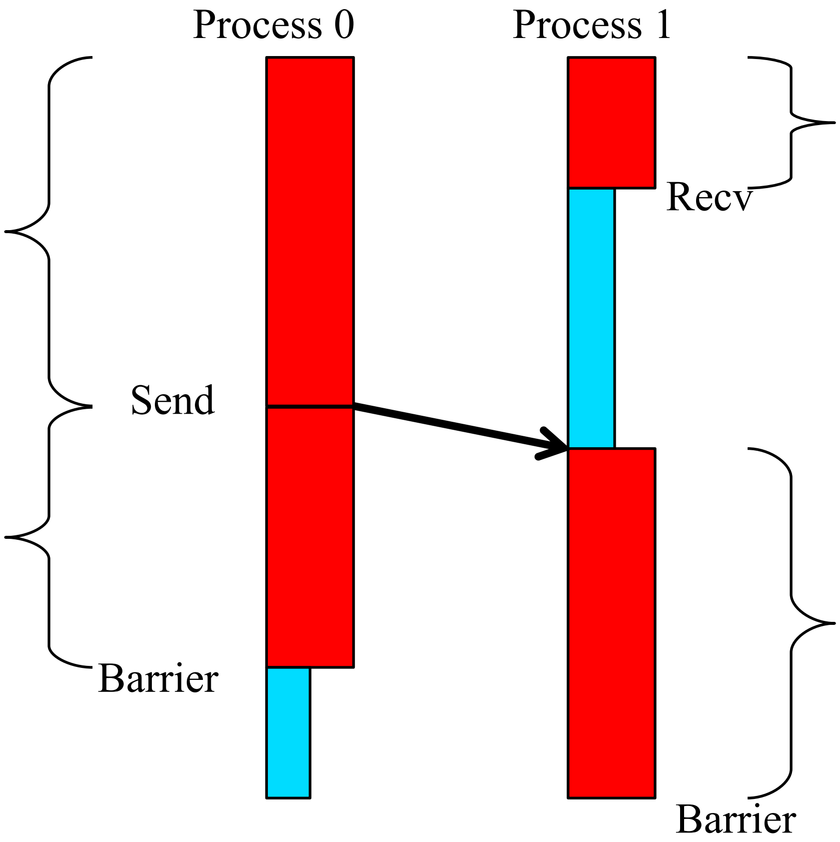
Blocking Communication

Time

# Another picture of tasks

Begins at end of
previous MPI call

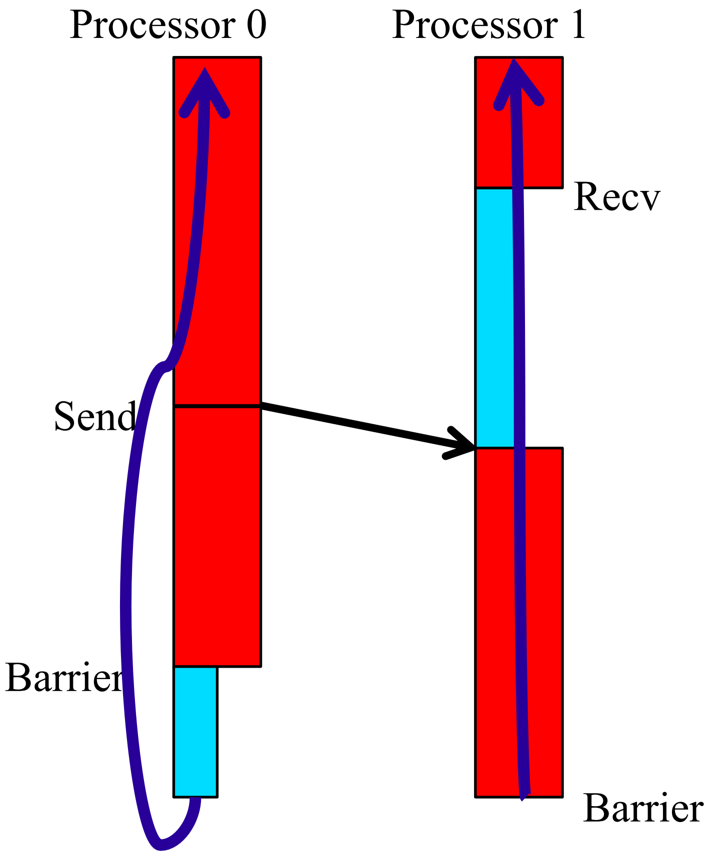
Ends at beginning
of following MPI call



# Assumptions

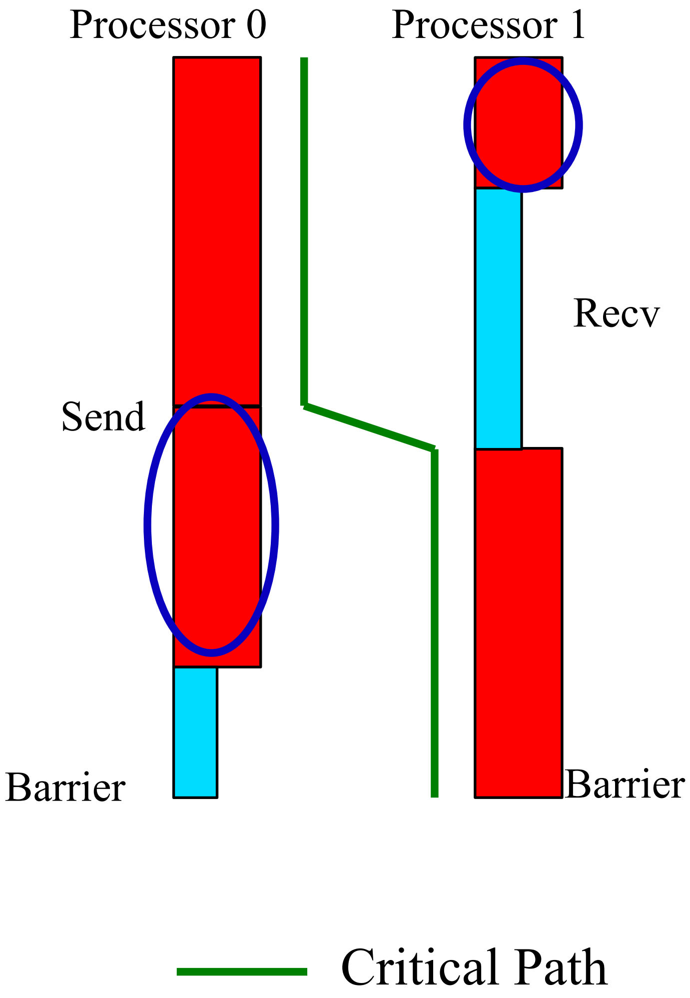
Iterative code

Per core DVFS



# Critical Path Approximation

1. Identify Tasks ***off*** the critical path--blocking

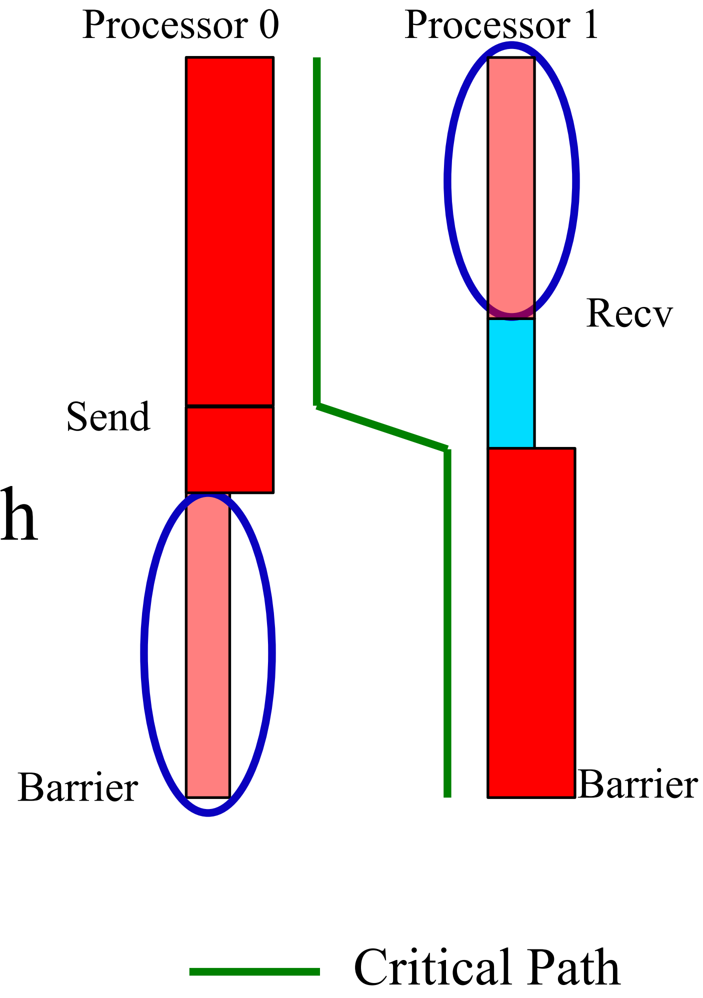


If task does not block, assume on critical path

# Critical Path Approximation

1. Identify Tasks off the critical path--blocking

2. On following iteration, slow off-critical path to (approximately) meet critical path

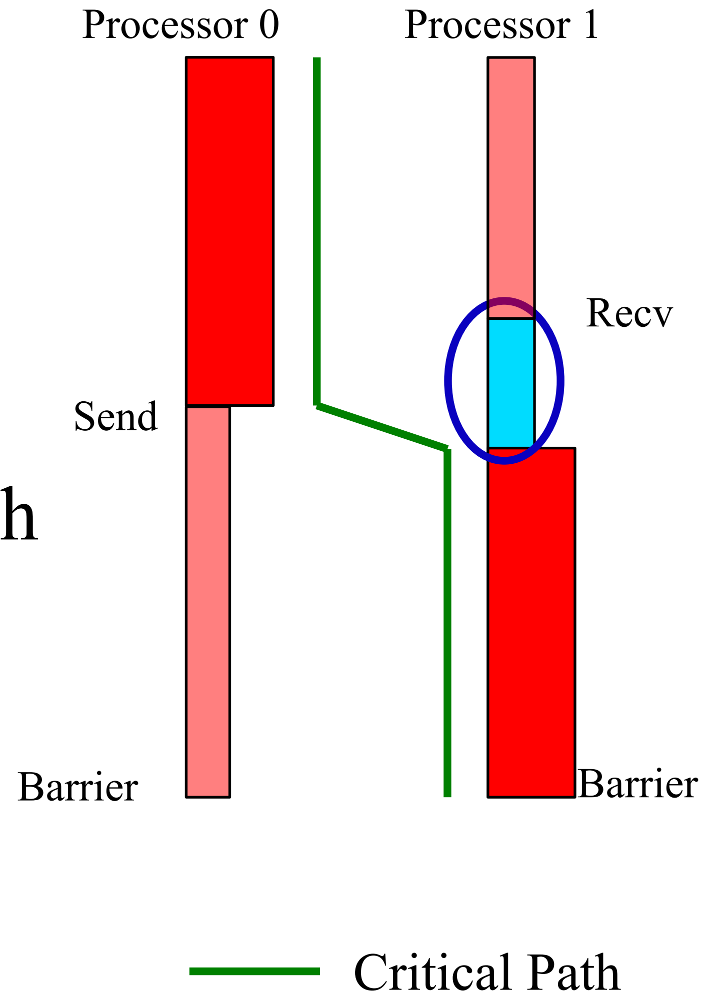


# Critical Path Approximation

1. Identify Tasks off the critical path--blocking

2. On following iteration, slow off-critical path to (approximately) meet critical path

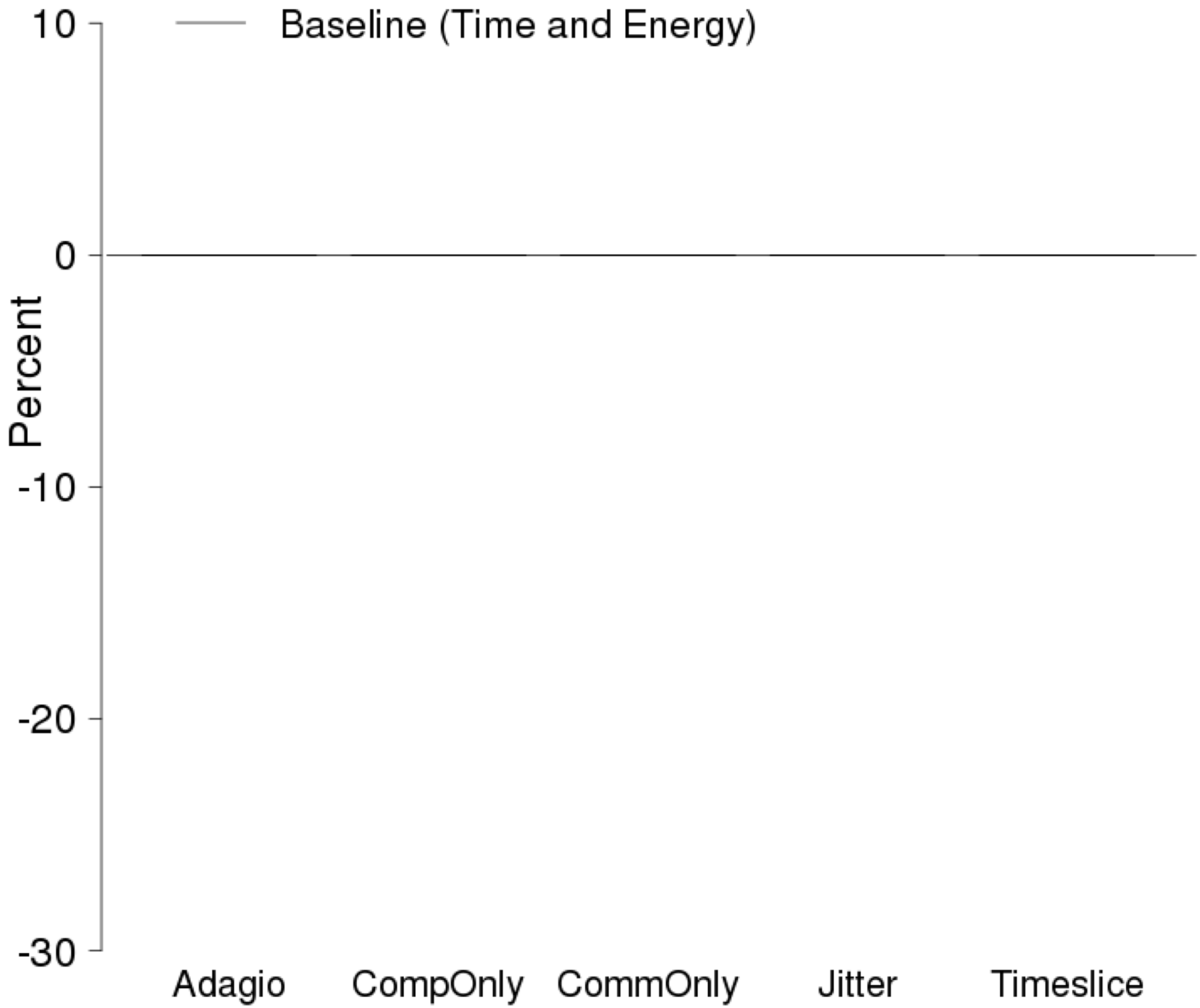
3. Slow remaining communication



# Experiments

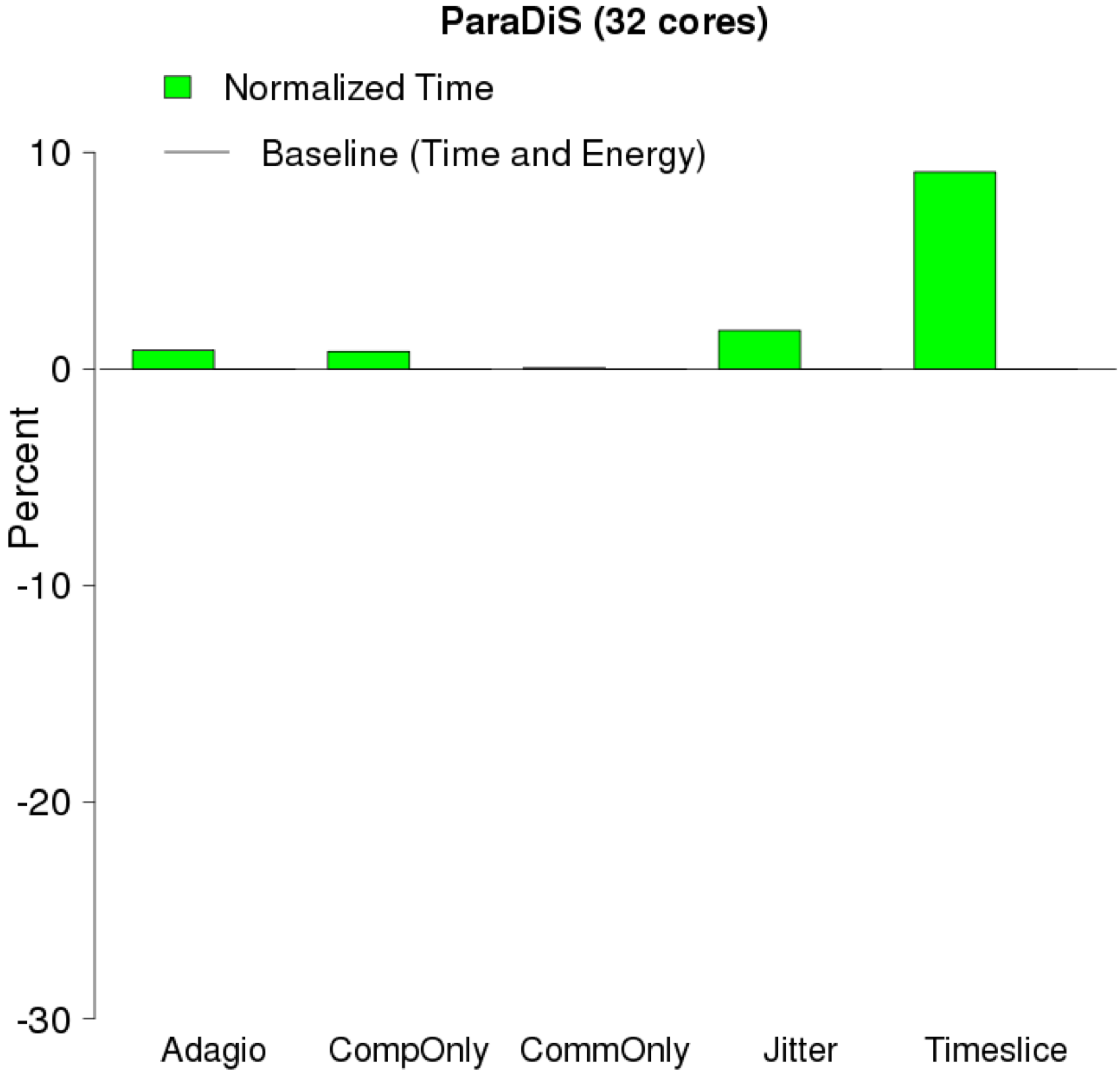
- 16-node, dual socket, dual-core Opteron 265s
  - Single core per socket used
  - 1.0-1.8 GHz in steps of 0.2 GHz
  - Power measurements taken from wall socket

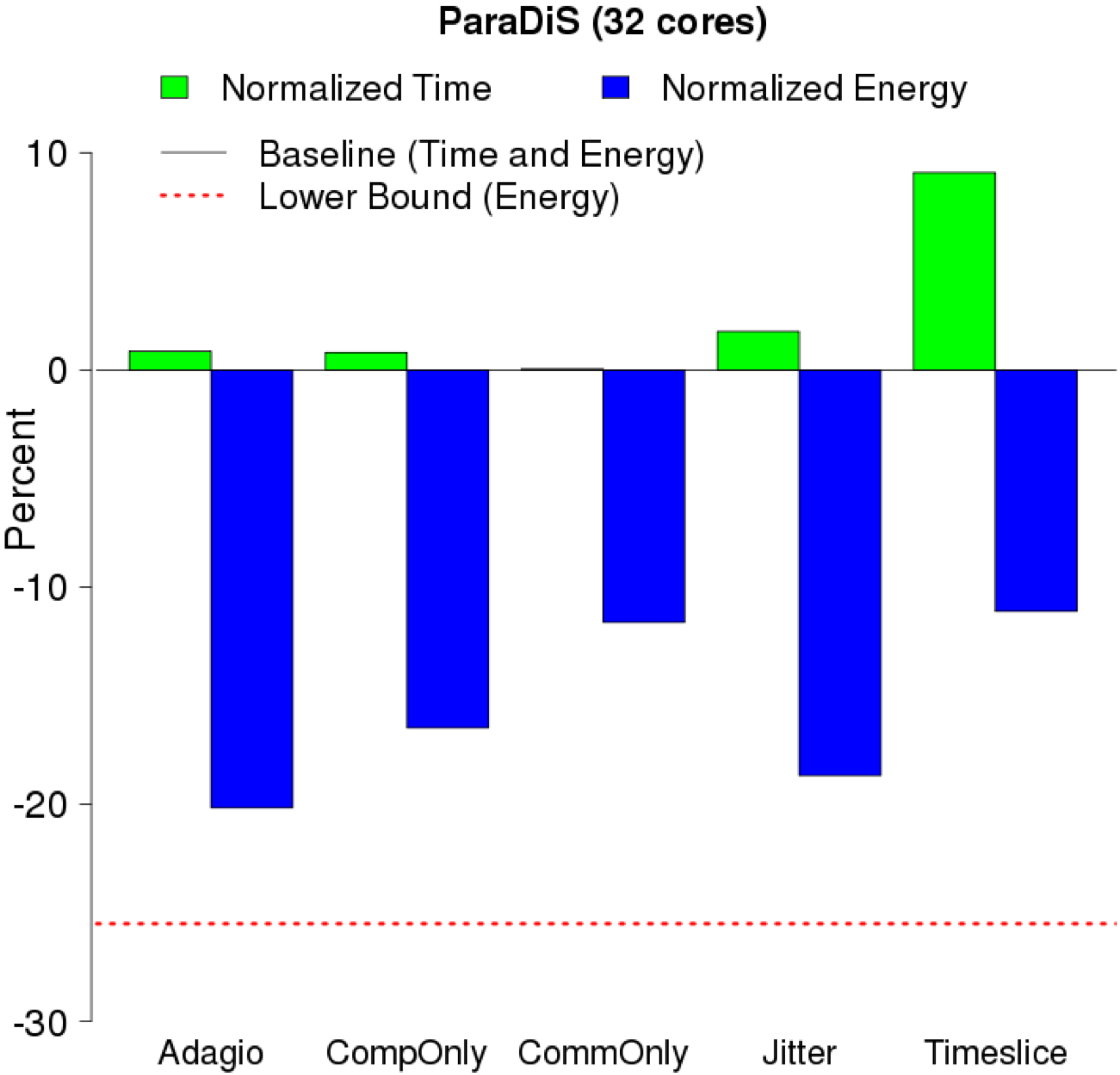
# ParaDiS (32 cores)

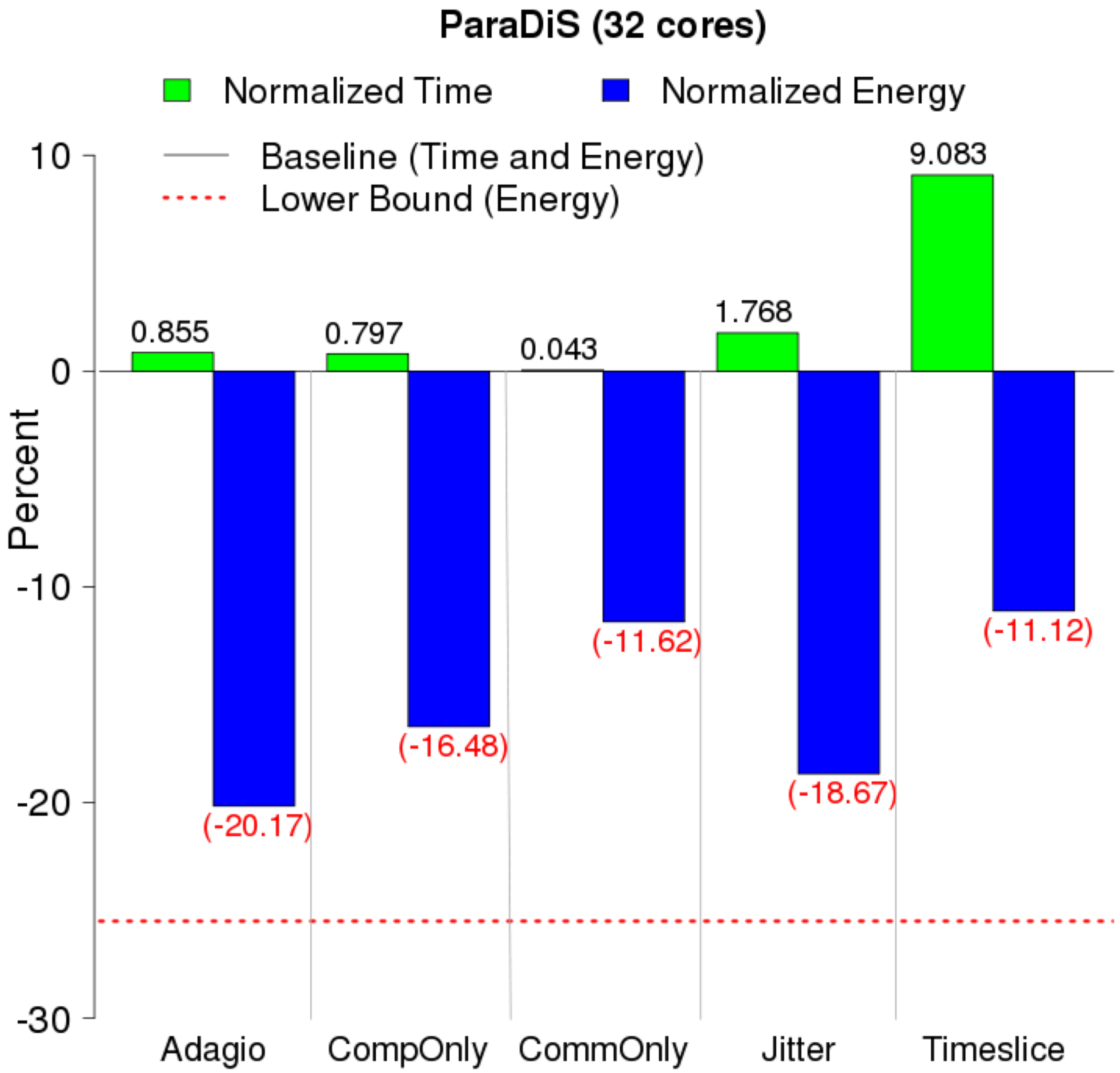


Freeh ‘05

Ge ‘07


ParaDiS (32 cores)
Normalized Time
Baseline (Time and Energy)
10
0
-10
-20
-30
Percent
Adagio
CompOnly
CommOnly
Jitter
Timeslice


ParaDiS (32 cores)
Normalized Time
Normalized Energy
Baseline (Time and Energy)
Lower Bound (Energy)
10
0
-10
-20
-30
Percent
Adagio
CompOnly
CommOnly
Jitter
Timeslice


ParaDiS (32 cores)
Normalized Time
Normalized Energy
Baseline (Time and Energy)
Lower Bound (Energy)
Percent
10
0
-10
-20
-30
0.855
(-20.17)
0.797
(-16.48)
0.043
(-11.62)
1.768
(-18.67)
9.083
(-11.12)
Adagio
CompOnly
CommOnly
Jitter
Timeslice

# Current Issue: Power a Problem at Exascale

- DOE originally stated that 20 MW is the limit for Exascale
  - Now appears to be 40 MW
- Unlimited power, though, is not tenable

# Stage 3: Power-Constrained HPC

- Traditional (wrong) thinking: it's a power/energy/delay problem
  - Derive metrics (and argue about them)
  - Energy-delay, Energy-delay-squared, etc
  - Save as much energy as possible subject to a fixed delay
- Alternative (correct) thinking: it's a performance problem
  - Limited power into the HPC facility
  - Machine peak power > HPC facility power
  - So have a power budget for the machine and thus per application
  - Goal: Maximize performance subject to the power budget

# Therefore: Manage Machine Resources

- Direct power to where it’s most useful

Cores: 10W-50W
DRAM: 15W-30W
Cache: 5W-10W
GPU: 100W-200W
FPGA: 20W-30W
Disk: 5W-10W

Total: 155W-330 W

Hypothetical Future Machine: Max < 330W **(* 10,000?)**

- How do we manage these resources in a holistic manner?
  - Requires fine-grain control, models, and system software

Note: possible to run a given node hotter if we run another node cooler

# Fine-Grain Control on Modern Machines: Power Measurement *and* Power Capping

- Power measurement: cores, DRAM
- Limit power to a node and its components
  - Example: Node allocated 200 watts, and user/runtime/OS directs 150 to the sockets/cores, 40 to the DRAM, and 10 to everything else
    - Also, possibly dynamic over the program run