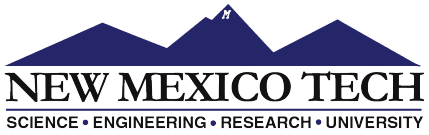


STEPPING TOWARDS A NOISELESS LINUX ENVIRONMENT

Hakan Akkan*, Michael Lang¶, Lorie Liebrock*

Presented by: **Abhishek Kulkarni¶**

* New Mexico Tech
¶ Ultrascale Systems Research Center
New Mexico Consortium
Los Alamos National Laboratory

Motivation

- HPC applications are *unnecessarily* interrupted by the OS far too often
- OS noise (or jitter) includes interruptions that increase an application's time to solution
- Asymmetric CPU roles (OS cores vs Application cores)
- Spatio-temporal partitioning of resources (Tessellation)
- LWK and HPC Oses improve performance at scale

OS noise exacerbates at scale

- OS noise can cause a significant slowdown of the app
- Delays the superstep since synchronization must wait for the slowest process: $\max(w_i)$

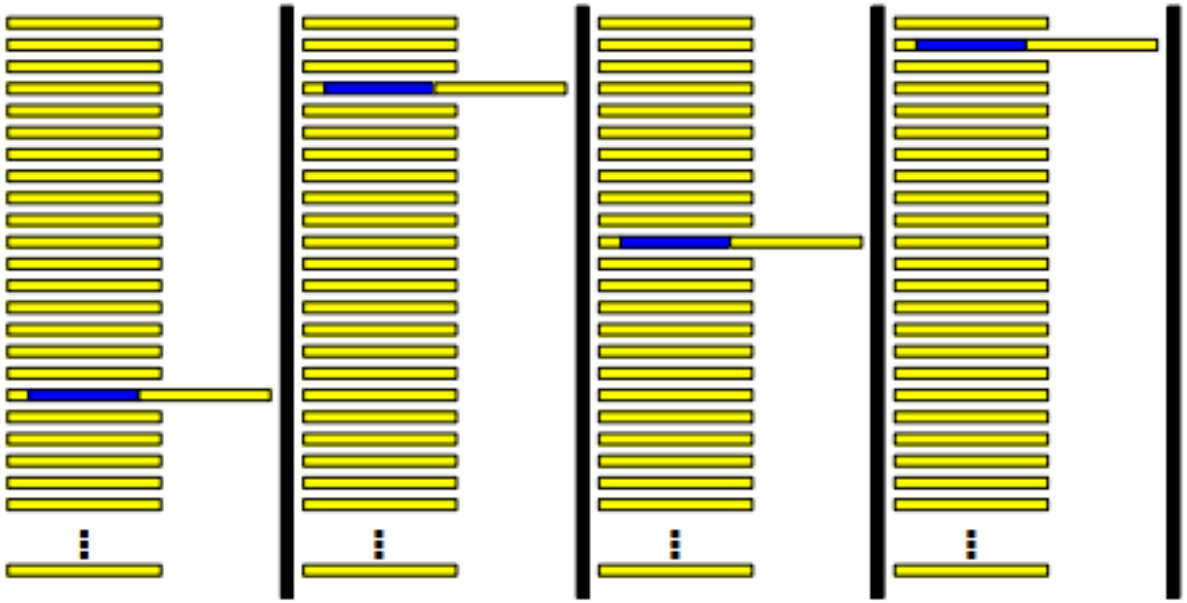


Image: The Case of the Missing Supercomputer Performance, Petrini et. Al, 2003

Noise co-scheduling

- Co-scheduling the noise across the machine so all processes pay the price at the same time

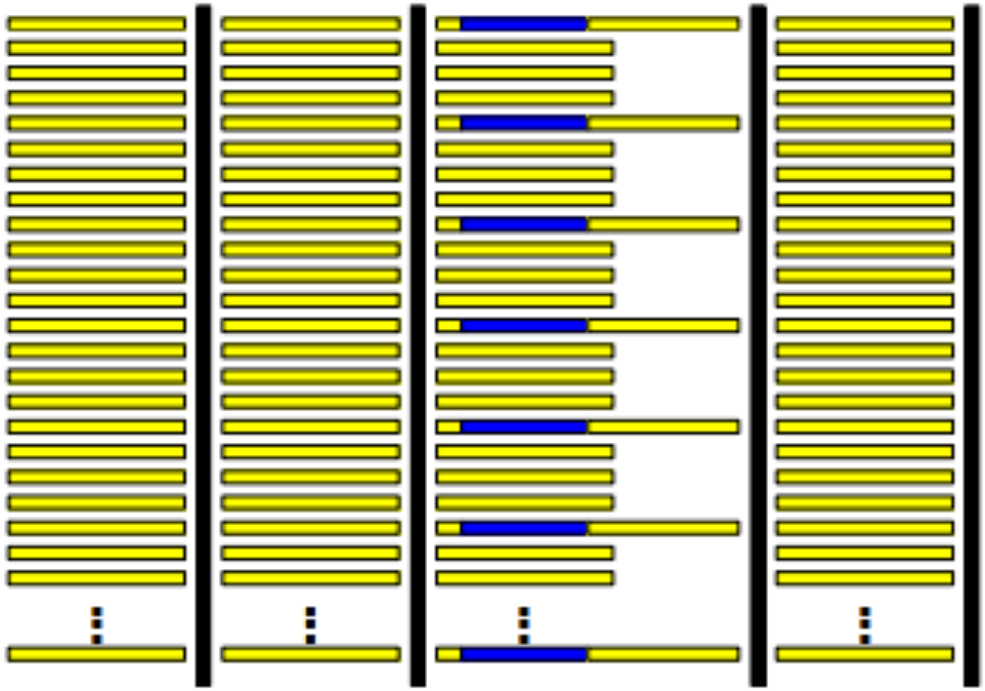


Image: The Case of the Missing Supercomputer Performance, Petrini et. Al, 2003

Noise Resonance

- Low frequency, Long duration noise
  - System services, daemons
  - Can be moved to separate cores
- High frequency, Short duration noise
  - OS clock ticks
  - Not as easy to synchronize - usually much more frequent and shorter than the computation granularity of the application
- Previous research
  - Tsafrir, Brightwell, Ferreira, Beckman, Hoefler
- Indirect overhead is generally not acknowledged
  - Cache and TLB pollution
  - Other scalability issues: locking during ticks

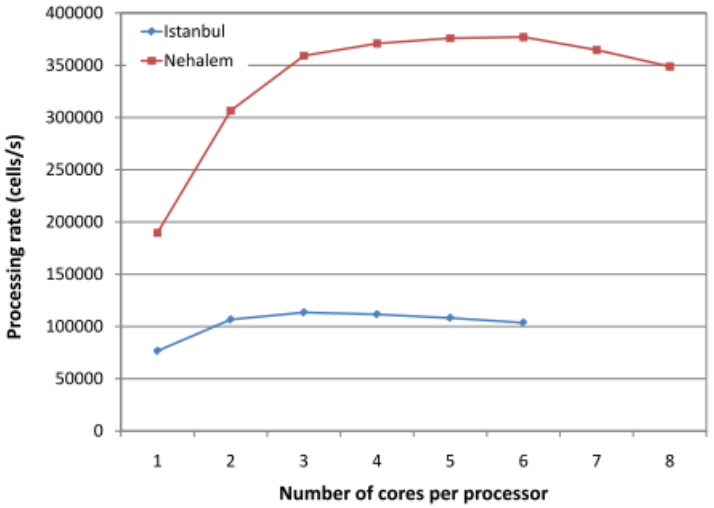


Fig. 3. SAGE performance scaling.

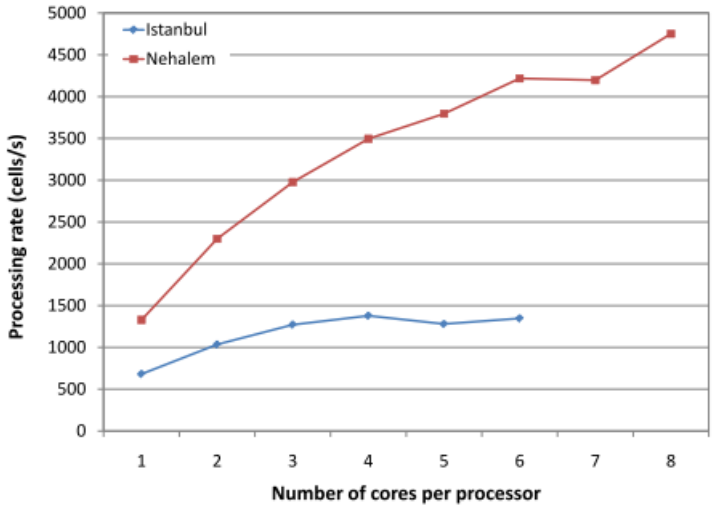


Fig. 4. PARTISN performance scaling.

Some applications are memory and network bandwidth limited!

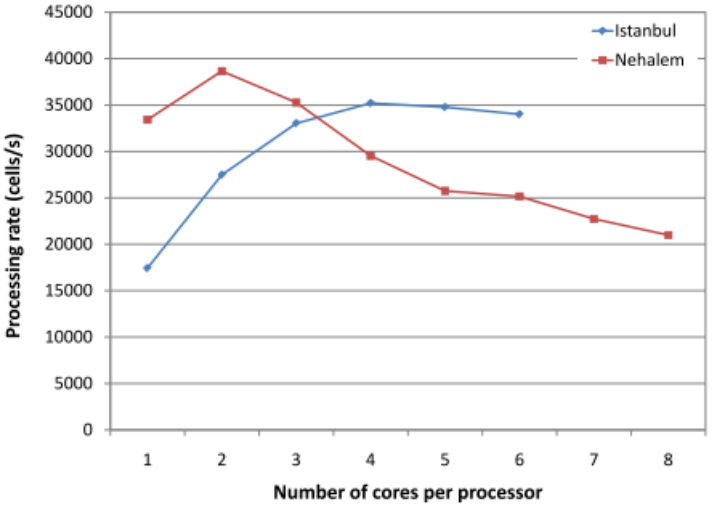


Fig. 5. XNOBEL performance scaling.

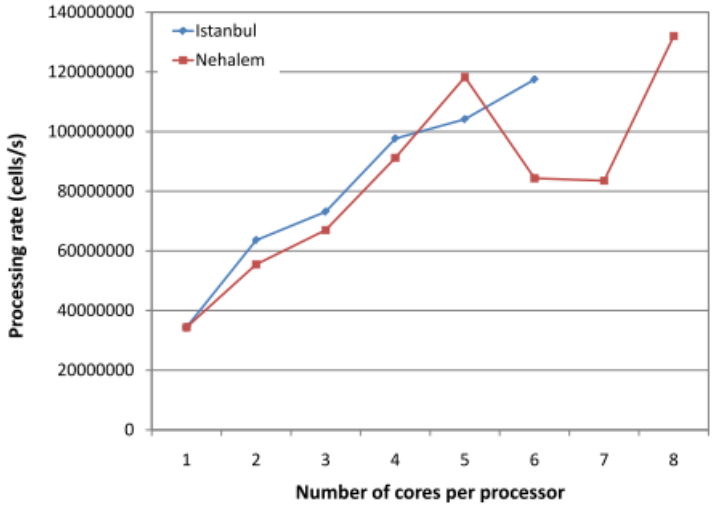


Fig. 6. SWEEP3D performance scaling.

*Sancho, et. al

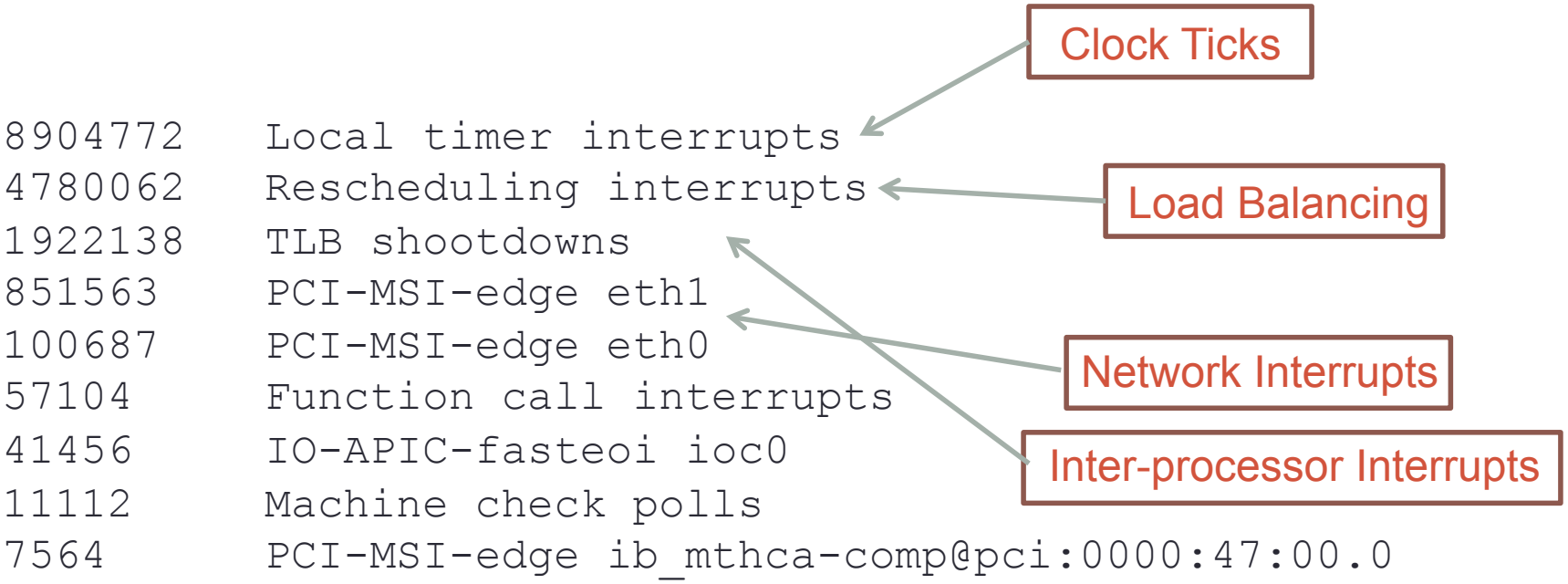
Recent Work

- Tilera Zero-Overhead Linux (ZOL)
  - Dataplane mode
  - Eliminates OS interrupts, timer ticks
- Cray Compute Node Linux
- Linux Adaptive Tickless Kernel
- We take a step-by-step approach quantifying the benefits of each configuration or optimization to Linux

Challenges

- Can we stop the ticks on application cores and move all OS functionality onto these spare cores?
- What would be the benefit in turning off the ticks? Are timer interrupts necessary for all cores?
- How close can we get to a LWK with Linux?

Interrupts in Linux



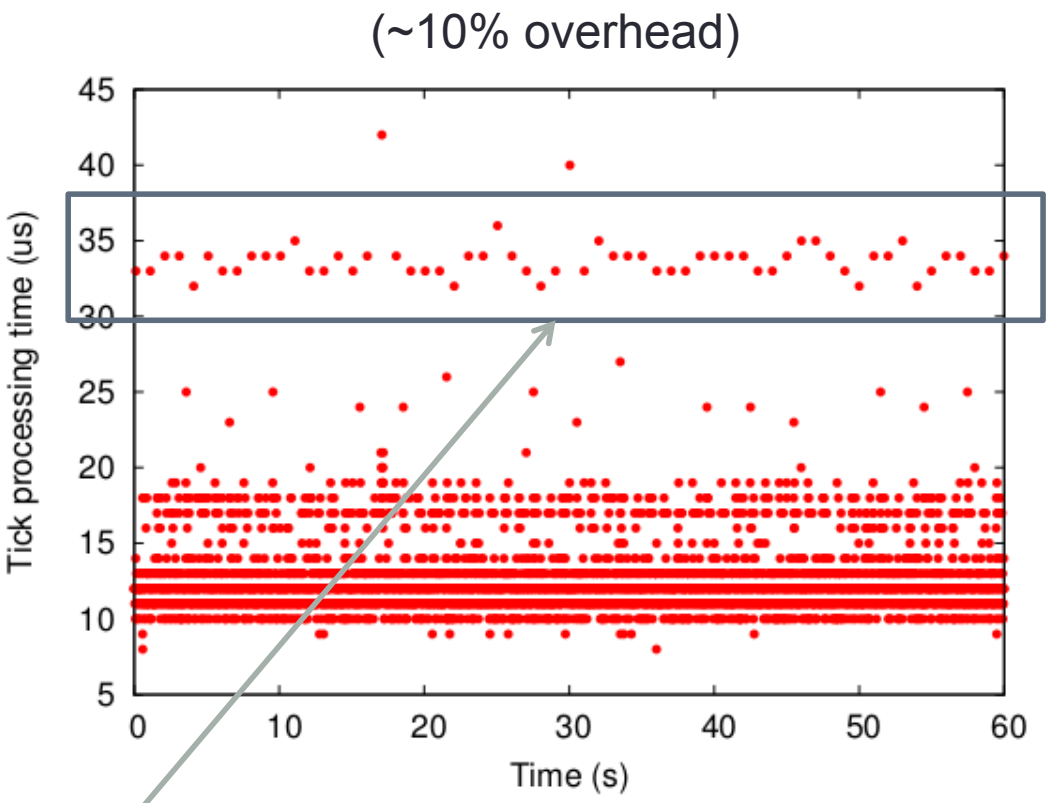
(on a 24 core Linux 2.6.x machine with hz=100)

What happens during a tick?

- Updating the kernel time
- Resource accounting
- Running expired timers
- Checking for preemption
- Performing delayed work
- Subsystems that need collaboration from all CPUs use IPIs
  - Read Copy Update (RCU): Expects every CPU to report periodically. Interrupts the silent ones.

Tick Processing Times

- Variance is due to locking and cache line bouncing caused by accessing and/or modifying global data such as the kernel time



A kernel thread was woken up periodically (every second) to refresh VM statistics!

Towards Noiseless Linux

- Measure tick processing times to characterize the effect of noise
- Ignore overhead caused by TLB shootdowns, page faults.
  - Not as easy to mitigate
- Task Pinning
- Turn off load balancing and preemption
- Move device interrupts to separate cores

Challenge: Preventing Preemption

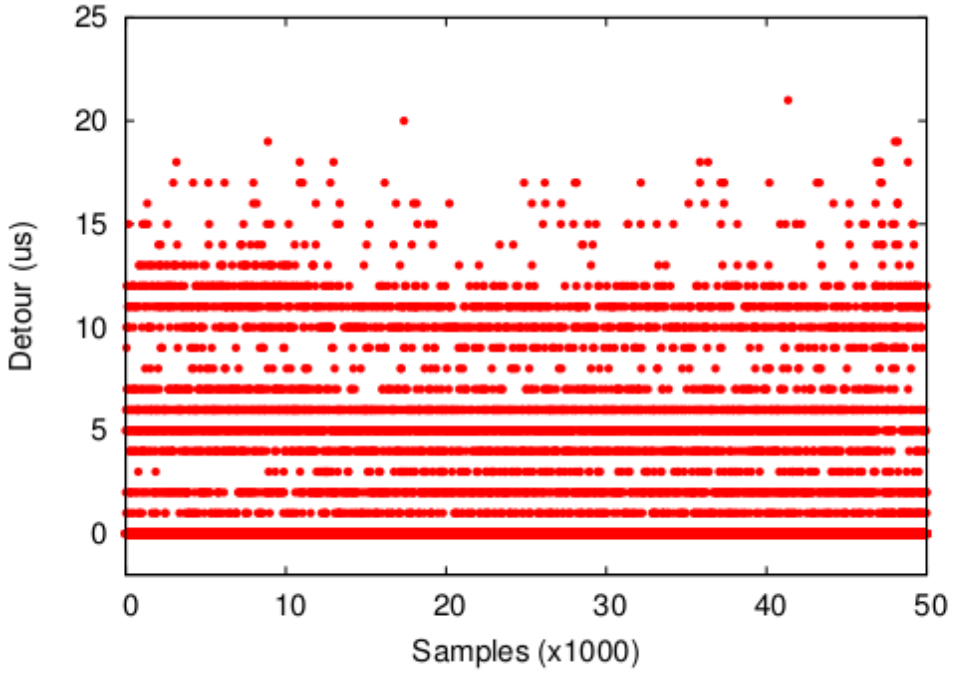
- Exclude CPUs from load balancing domains
  - *isolcpus* boot argument
    - Static, and nearly obsolete
  - Process Containers aka Kernel Control Groups (*cgroups*)
    - Dynamic
    - But harder manageability
- Difficult to disable certain kernel threads (such as *kworker*) without source-level changes

Measuring OS Noise

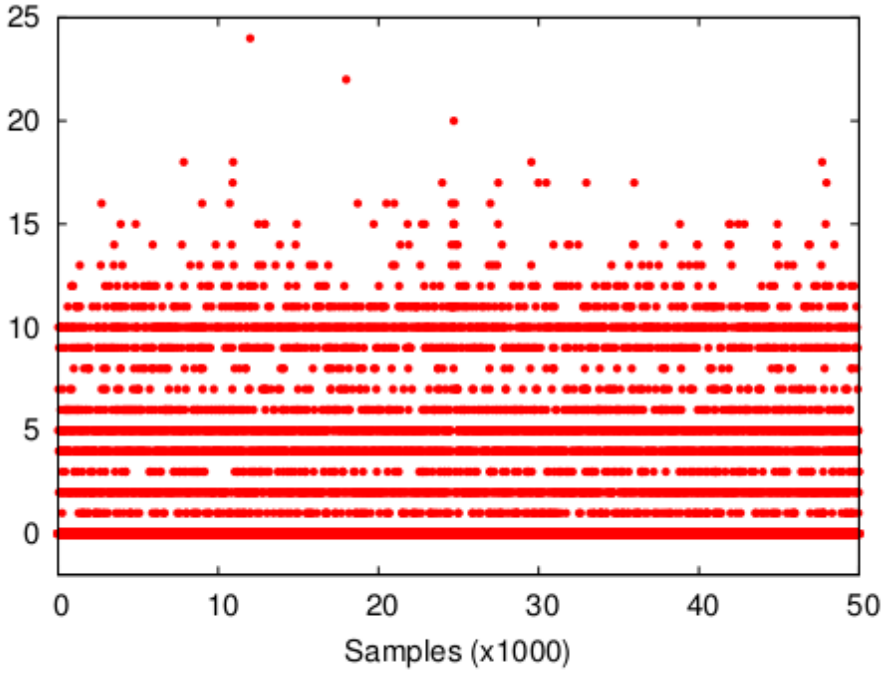
- Fixed Work Quanta (FWQ) benchmarks
  - Repeat a fixed amount of short work and record the time it takes at each iteration
  - Detour: How long does an iteration take?
- Tests run on a 4 socket, 6 core AMD machine with 16 MPI processes
  - Pinned to cores 3,4,5,6 on each NUMA domain (first 2 cores were reserved for the OS)

Measuring OS Noise

No attempts to reduce the noise vs task pinning



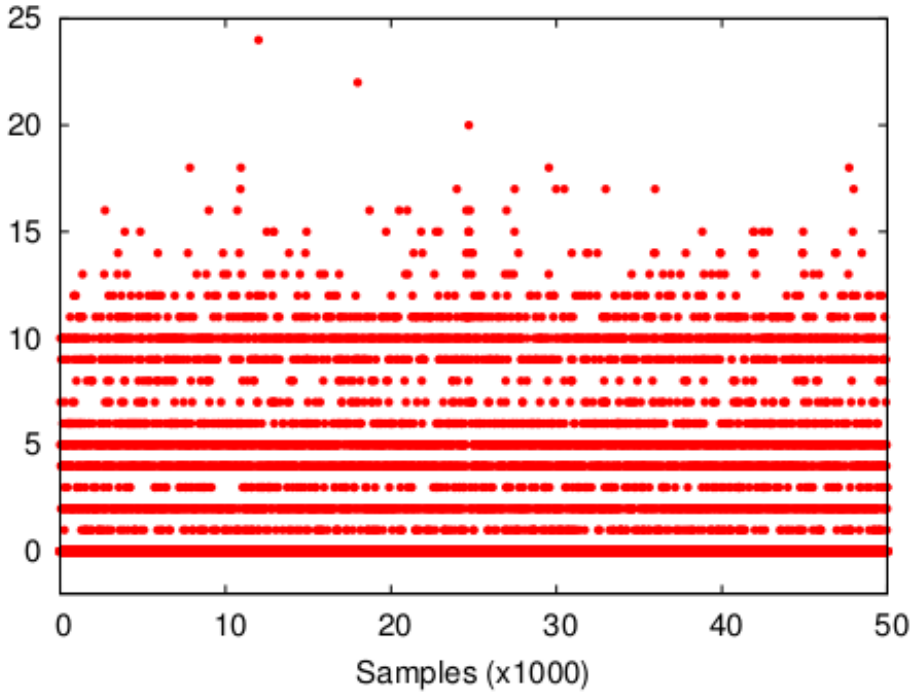
(a) Normal



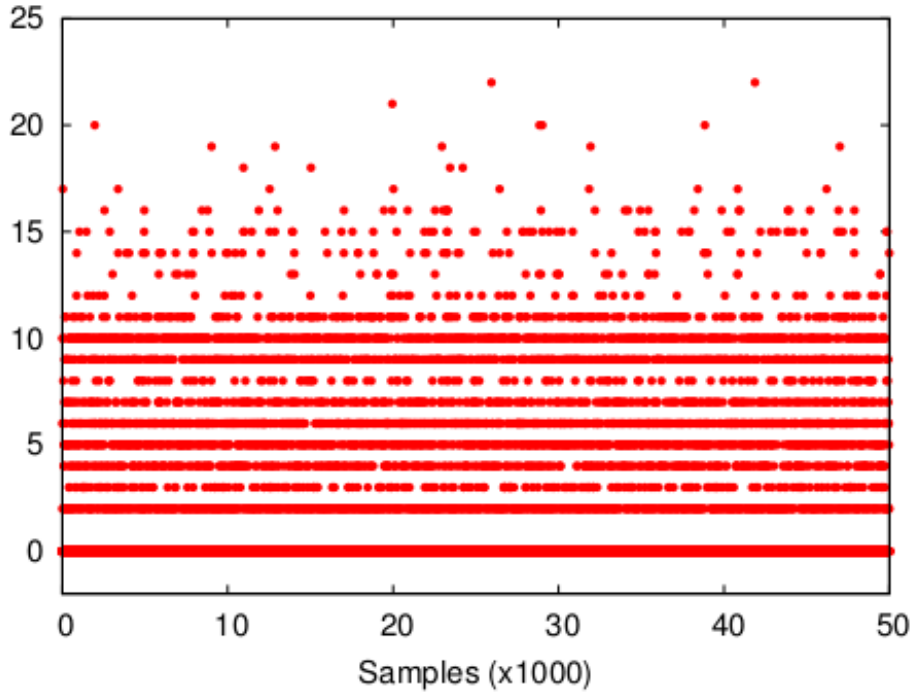
(b) Task affinity

Measuring OS Noise

Task pinning vs cgroups with load balancing



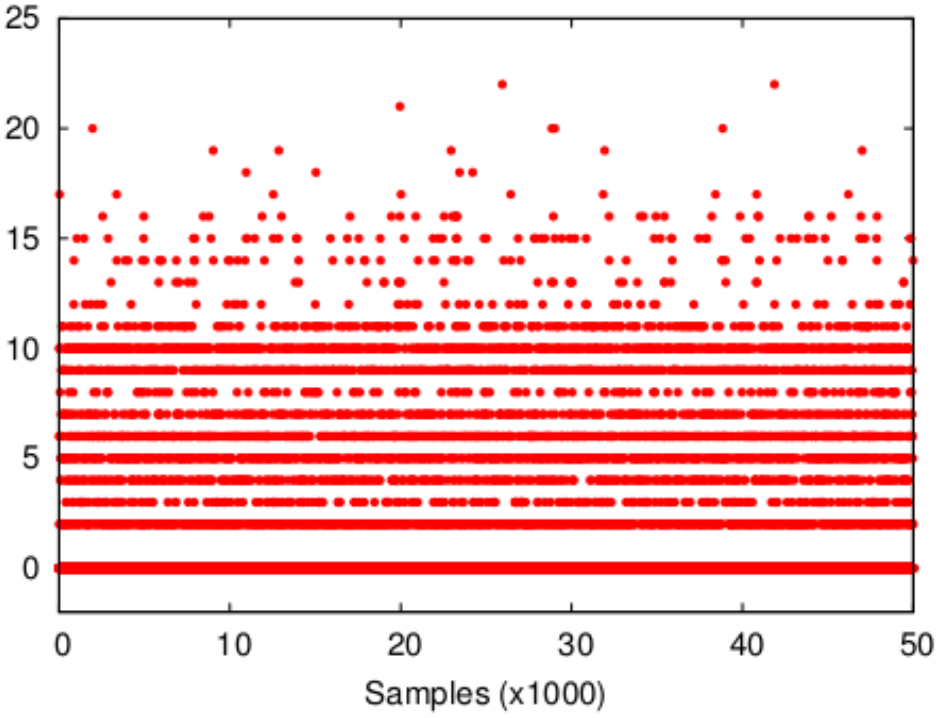
(b) Task affinity



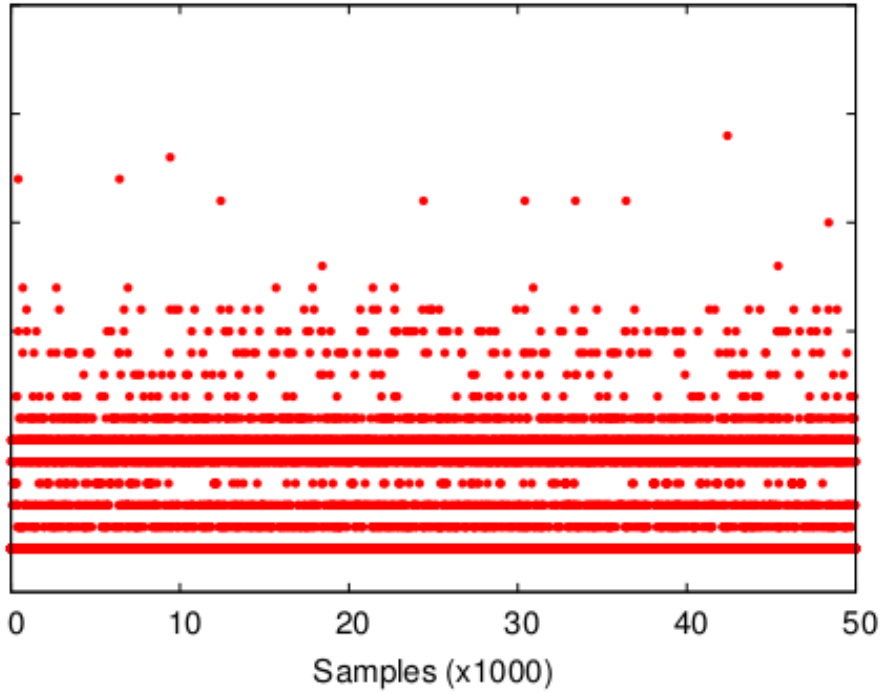
(c) cgroups (load balancing on)

Measuring OS Noise

cgroups with and without load balancing



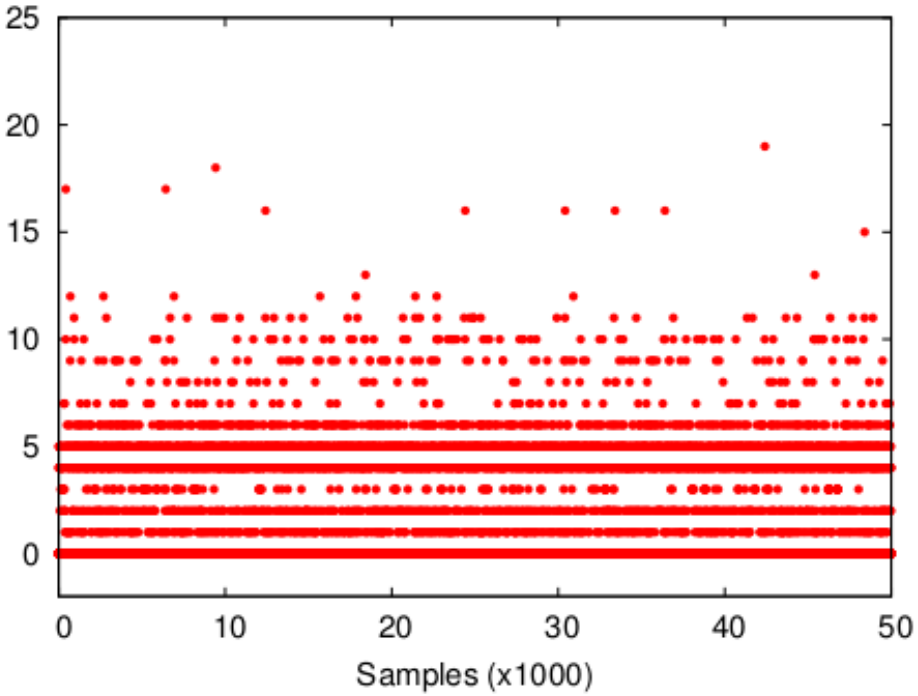
(c) cgroups (load balancing on)



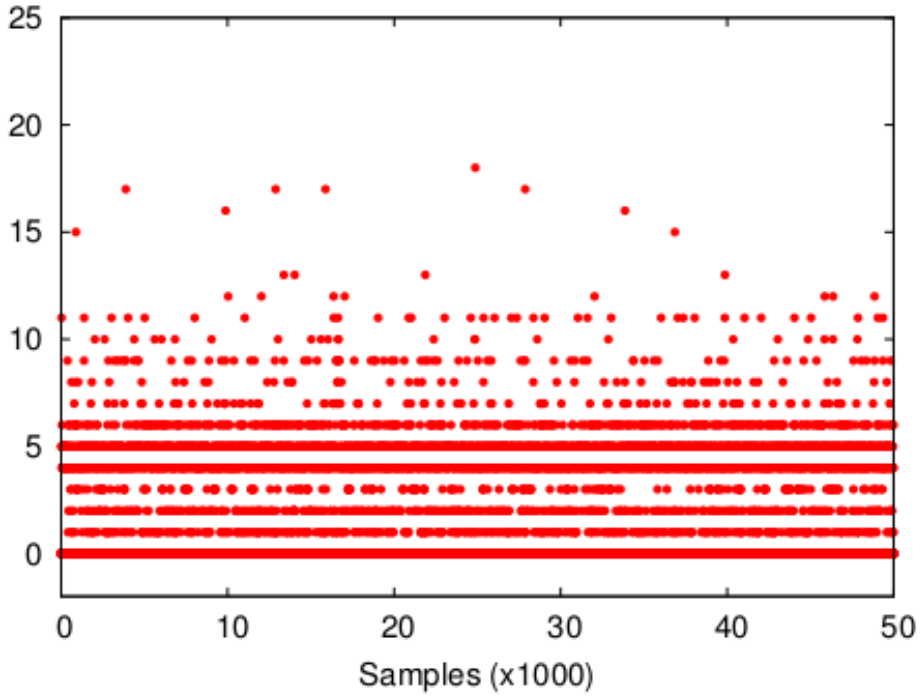
(d) cgroups (load balancing off)

Measuring OS Noise

cgroups without load balancing vs isolcpus



(d) cgroups (load balancing off)



(e) Isolated CPUs

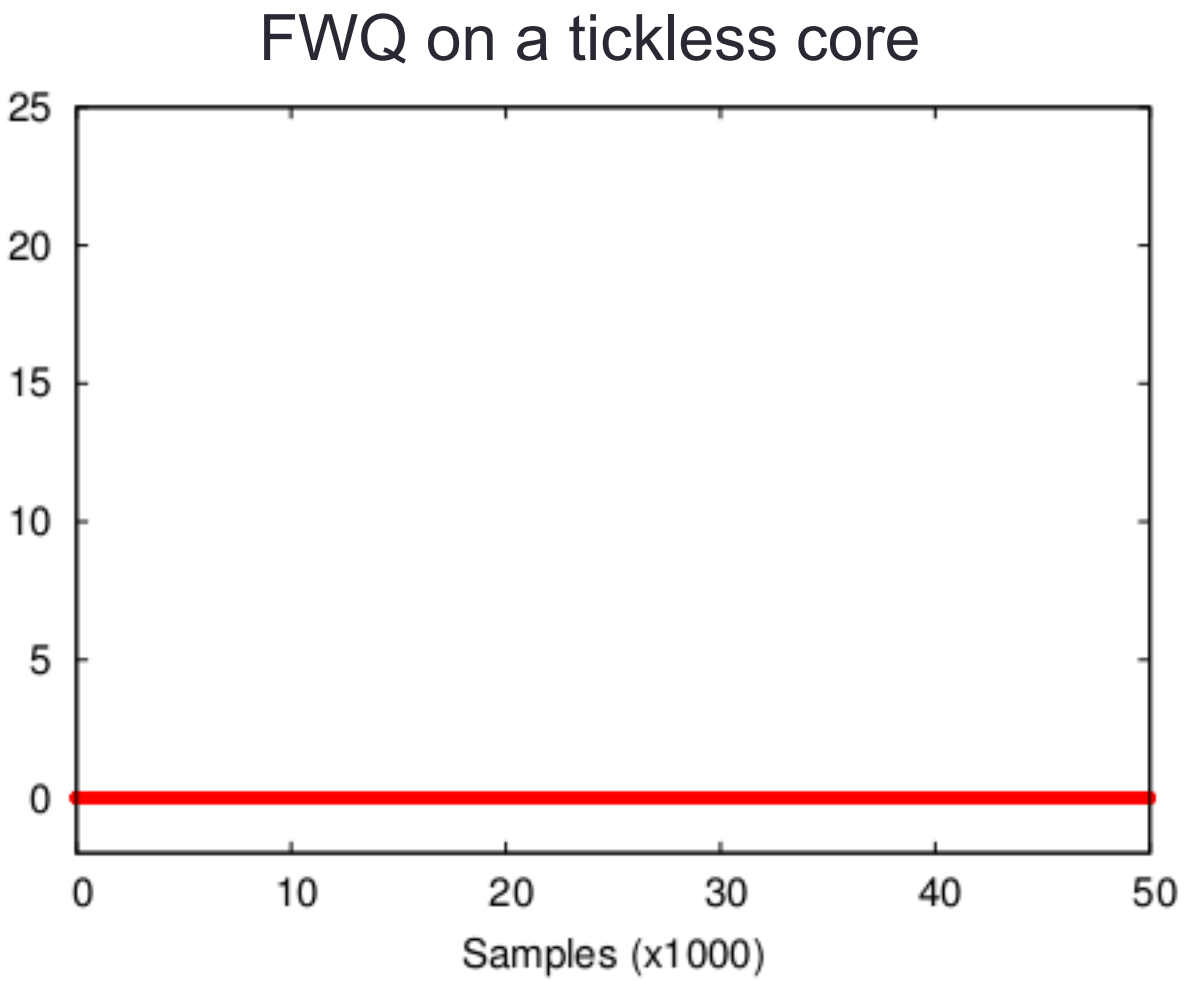
Challenge: Turning off ticks

- Ticks cause application runtime variability
- Cache pollution, TLB flushes and other scalability issues
- We also want realtime guarantees, predictability and deadline-driven scheduling
- Timers and delayed work items are problem
  - No interrupt -> no irq_exit -> no softirq
  - These usually reference local CPU data so running them on a separate CPU is not trivial

Challenge: Turning off ticks

- Our tickless Linux prototype:
  - Application requests a tickless environment
  - Kernel advances the tick timer much further in time and starts queuing any timer and workqueue requests to separate OS cores
  - Tells other subsystems to leave the application core alone and prevent inter-processor interrupts (IPI)
    - e.g. RCU subsystem

Tickless Linux

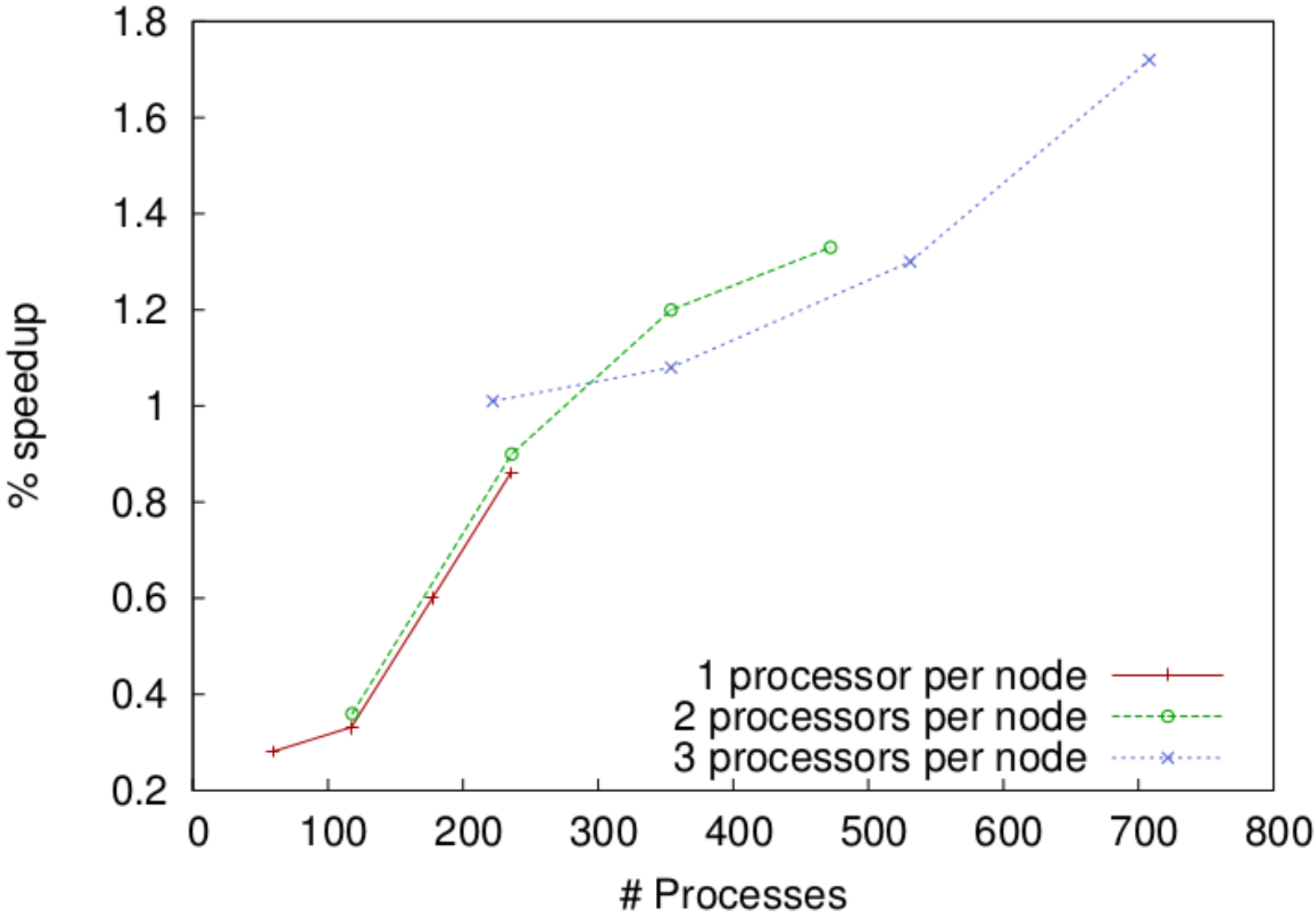


(f) Tickless mode

POP Performance

- Experimental Setup
  - 2 socket dual core processors x 236 nodes
  - Connected with a SDR InfiniBand network
  - Ran tests with 1, 2, and 3 ranks per node

POP Performance

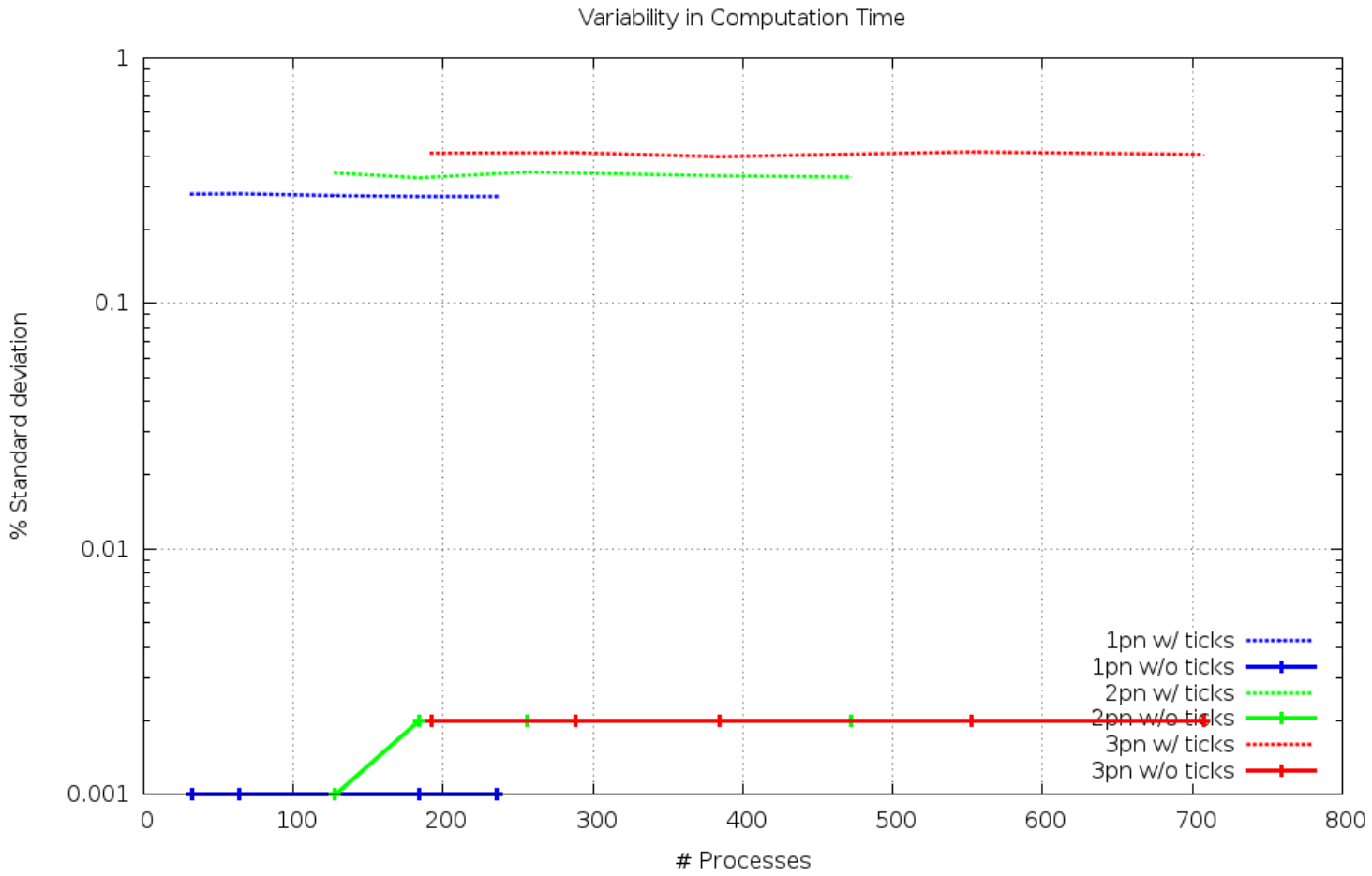


Variability Tests

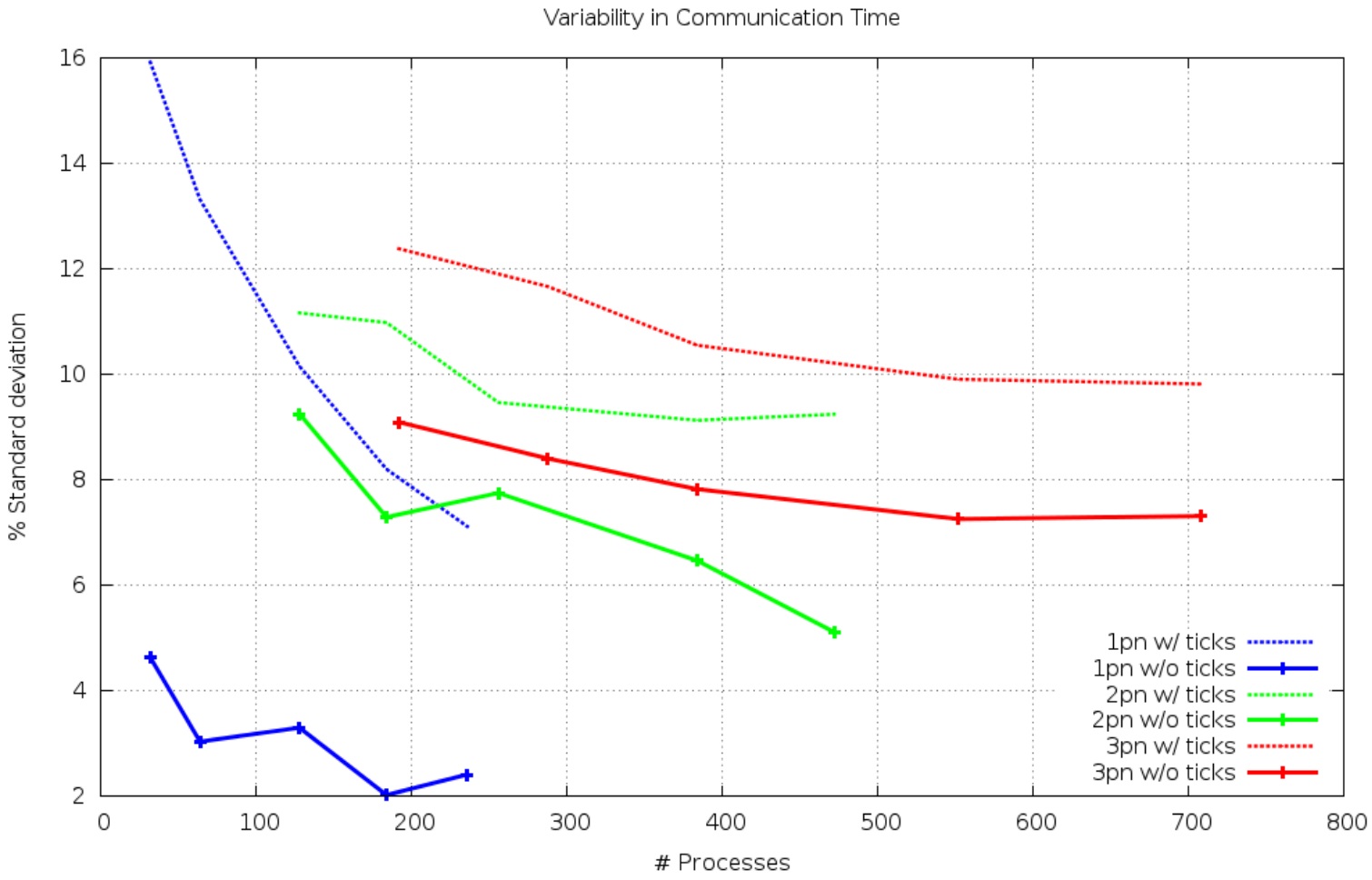
- Simple compute and synchronize benchmark

```
for(i = 0; i < iter; i++) {
     do_fixed_amount_of_work();
     timestamp[2 * i] = get_ticks();
     MPI_Allreduce();
     timestamp[2 * i + 1] = get_ticks();
}
```

Variability Tests



Variability Tests



Problems

- No softirq runs on the tickless core
  - I/O that depends on softirqs is slow/broken, e.g. Ethernet network
- Solution: Queue incoming packets to only OS cores
  - Resulted in unbalanced load making it slower by ~10%.
- IB works great because it does not depend on softirq processing
- Sometimes timekeeping was off by a bit

Prototype solutions

- To alleviate reduced network bandwidth, allow bottom-half handlers on OS cores to do larger batch processing
- Timekeeping issues can be dealt with by keeping one OS core running all the time (prevent going idle)
- Some device drivers depend on ticks: equip work items with HZ frequency

Future Work

- Collaboration with Linux developers to implement a tickless mode
- Implement
  - Accounting and timekeeping
  - Bottom-half handlers with higher batching
  - Disabling kernel threads or moving them to OS cores
- Test at higher scales with other applications

Conclusion

- We identified the primary events that happen during ticks and discussed their relevance in HPC context
- We proposed methods to move the ticks away from application cores
- We created a tickless Linux prototype with promising intial results
- We showed the benefits to noise-sensitive applications

80% of the Top500 are running Linux and losing compute cycles to ticks!

Questions?