Applications of Parallel Computers

Dynamic Load Balancing

https://sites.google.com/lbl.gov/cs267-spr2021

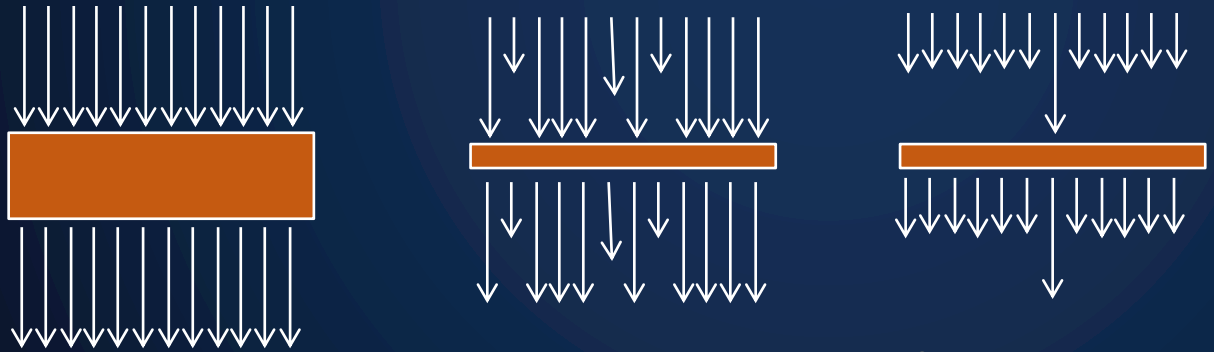
Causes of Poor Scaling

- **Amdahl's Law**
  - Serial code, outside OpenMP regions, inside critical regions, etc.
- **Insufficient parallelism**
  - Size of data N < P number of processors
- **Too much parallelism overhead**
  - Thread creation, synchronization, communication, scheduling
- **Poor locality**
  - E.g., false sharing
- **Load imbalance**
  - Different amounts of work across processors
  - Different data movement costs, compute cost, etc.

Looking for Load Imbalance

- High synchronization overhead
  - Expensive barriers or locks
- Load imbalance
  - Load imbalance
  - Extreme: Lack of parallelism including serial bottleneck (Amdahl's law)

May look similar. Need individual (not average) running times to measure load imbalance!



Good load balance
Expensive barriers

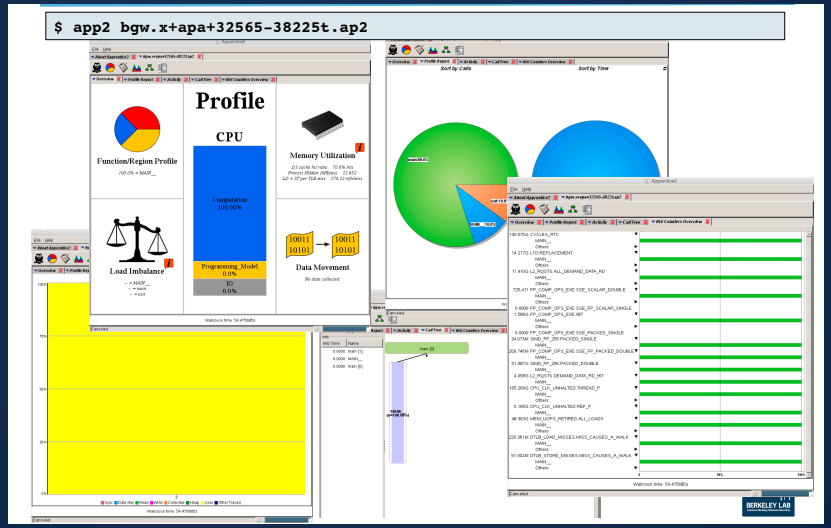
Load imbalance

Serial bottleneck
Worst case speedup

Measuring Load Imbalance

- Besides basic timers + printf, which can produce an enormous amount of output, use tools
- Cray Pat
- HPC Toolkit
- TAU

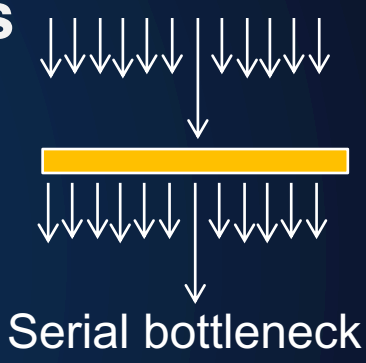
Beware of Heisenbugs in performance.



https://docs.nersc.gov/development/performance-debugging-tools/

Measuring Load Imbalance

- **Basic measurement: timers around barriers**
  - Don’t just average!
  - Need to look at all values or histogram
  - Load imbalance: average time wasted
    (max – average) / average
  - Especially subtle if not bulk-synchronous
  - “Spin locks” looks like useful work



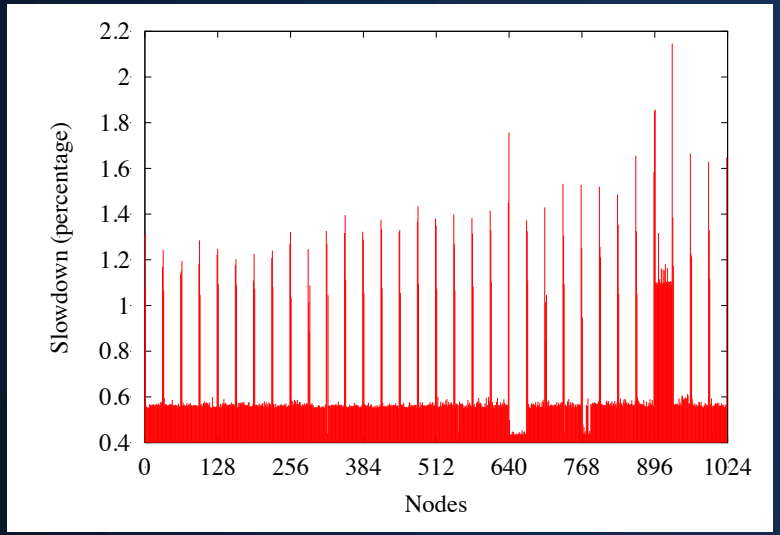
Sources of Load Imbalance

- Variable and often unknown:
  - Cost of tasks
  - Cumber of tasks
  - Structure of task tree (or graph)
- Machine variability

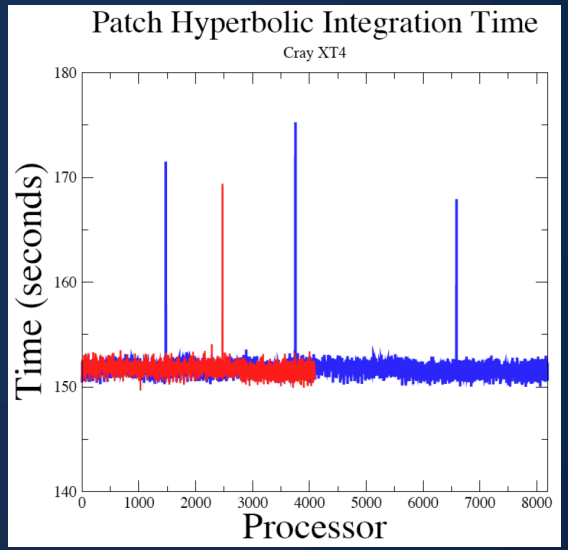
System performance variability

Load imbalance isn't always an application level problem

Performance variability can come from hardware or operating system effects



Petrini, Kerbyson, Pakin, "Case of Missing Supercomputer Performance," SC03

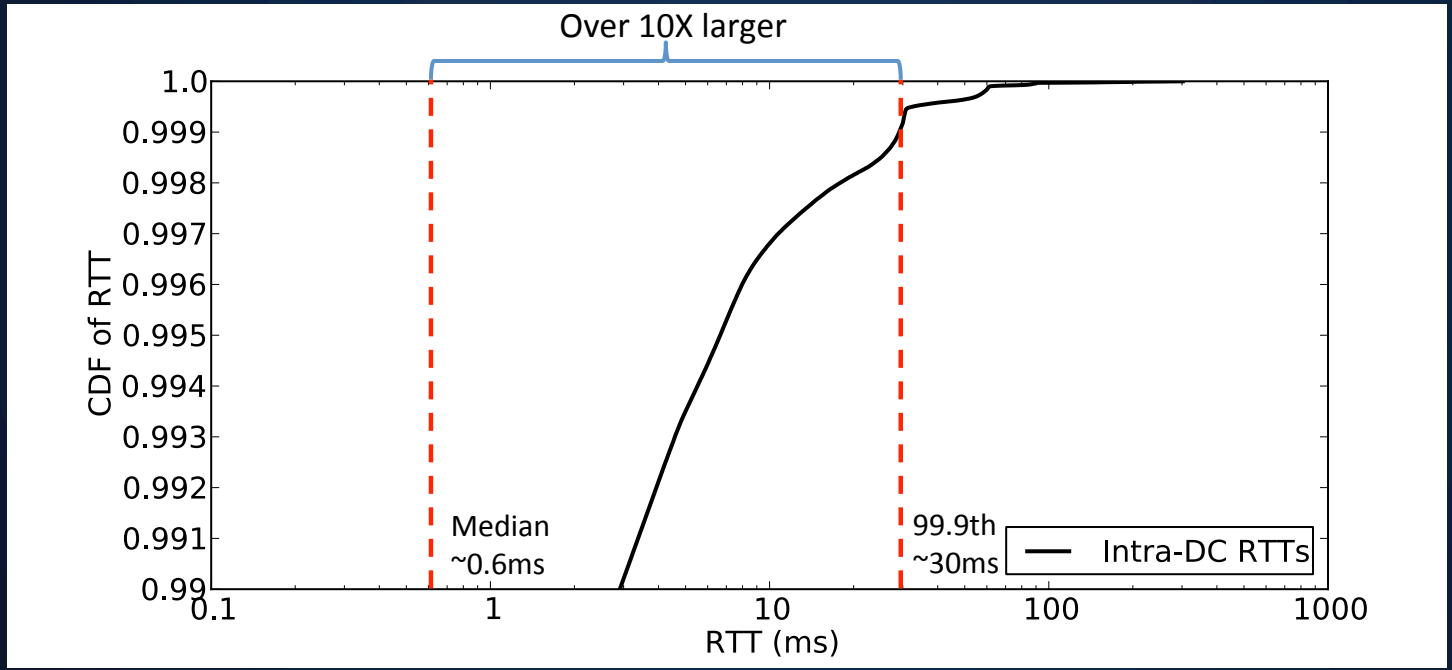


Brian van Straalen, DOE Exascale Research Conference, April 16-18, 2012. *Impact of persistent ECC memory faults.*

Public clouds (e.g., AWS) have high variability

- Variability of node performance (Round-Trip-Time) in the cloud

Data and image from, “Bobtail: Avoiding Long Tails in the Cloud,” by Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey, *University of Michigan*



Load Balancing Approaches

- Static Load balancing:
  - Divide into equal size parts
  - Straightforward if you know the problem size (n, nnz…)
  - Can be expensive to estimate / optimize (graph partitioning)
- Dynamic load balancing:
  - When work costs are unknown
- **Observation**: Load balancing and locality often trade off

Case 1: Independent Tasks

E.g., a Loop

Load balancing independent tasks

- Loop over a set of independent computations

```
for (i = 0, i < n; i++)  {

}
```

Easy Statements and loops with fixed bounds, no conditionals to vary workload

Harder Switch where each branch has "known" work, e.g., images of different sizes

Hardest Conditionals, computed loop bounds, breaks, exceptions, etc.

- None of these have communication / dependencies between iterations

Task cost variability (from the application)

- **Easy: Equal costs, fixed count**

Regular meshes, dense matrices, direct n-body

- **Hard: Tasks have different but estimable times, fixed count**

Adaptive and unstructured meshes, sparse matrices, tree-based n-body, particle-mesh methods

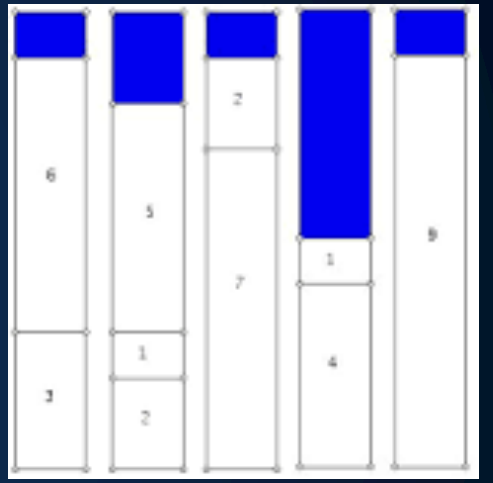
- **Hardest: Times or count are not known until mid-execution**

Search (UTS), irregular boundaries, subgrid physics, unpredictable machines

Note that good load balancing can't fix lack of parallelism.

Bin-packing for statically known costs is hard

Source: Wikipedia



- **Hard: Tasks have different but estimable times; known count**
  - When memory is limited this is the classic bin-packing problem
  - NP-hard (only exponential time algorithms are known)
  - Some easy cases (zig-zag)

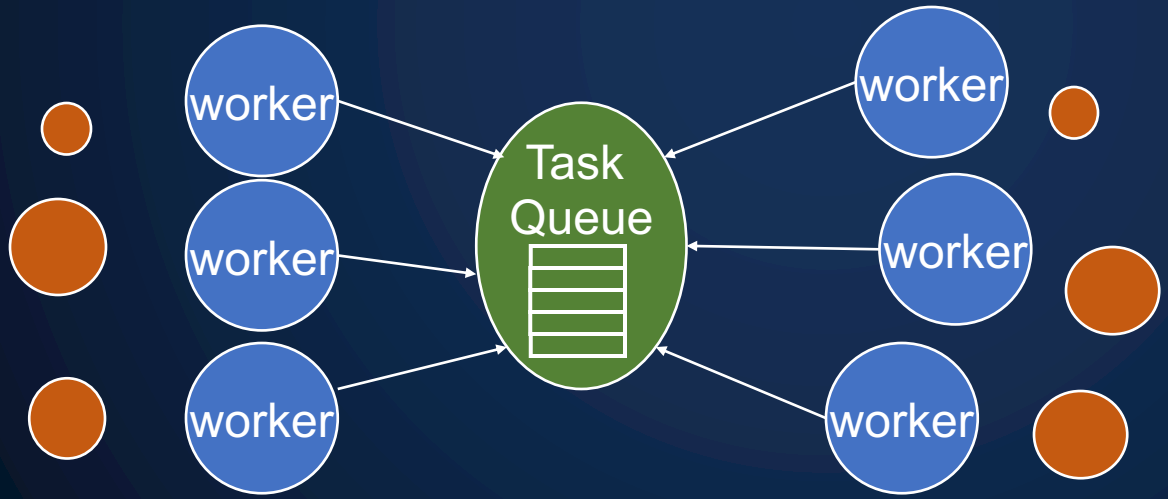
  - But good load balance is usually done approximately

Dynamic Scheduling

- What if work estimates are unknown?
  - Until mid-execution of an iteration / chunk
  - E.g., Mandelbrot, game trees

Self Scheduling

- "Self Scheduling" [Tang and Yew '86]
  - Shared queue of tasks
  - Processors (workers) take / add tasks to queue



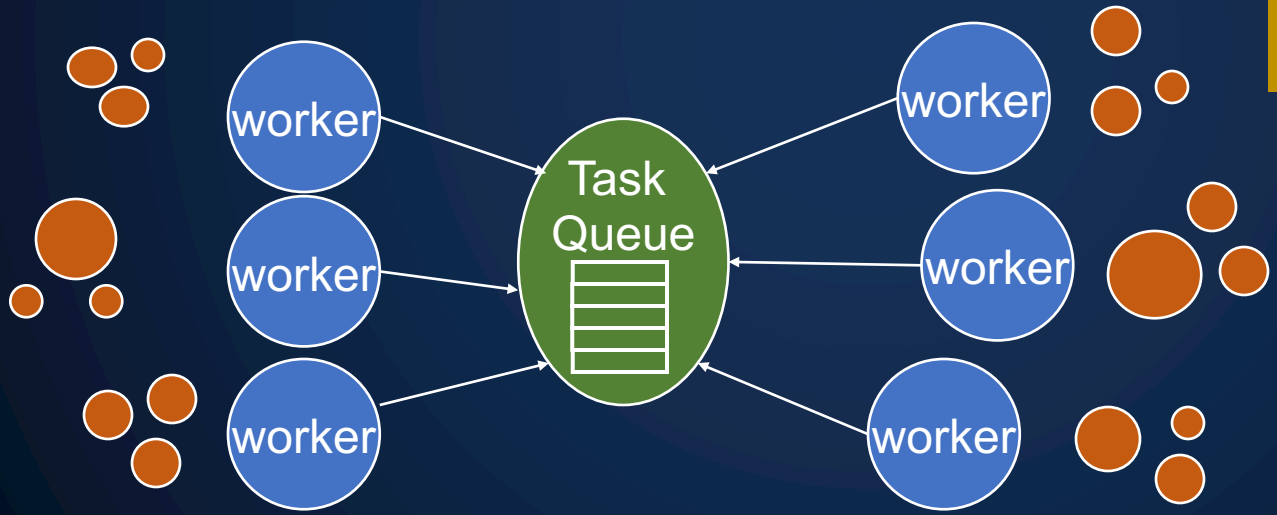
Chunked Scheduling

- "Chunked" (Self) Scheduling [Kruskal and Weiss '85]
  - Shared queue of tasks, workers remove chunks of size K
  - Reduce queue contention (locking)

Chunked Scheduling

- "Chunked" (Self) Scheduling [Kruskal and Weiss '85]
  - Shared queue of tasks, workers remove chunks of size K
  - Reduce queue contention (locking)

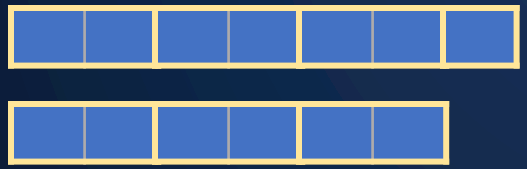
What is k is too big? Too small?



Chunking Size Trade-Off

- Large chunks: lower contention / overhead

- Small chunks: even finish time

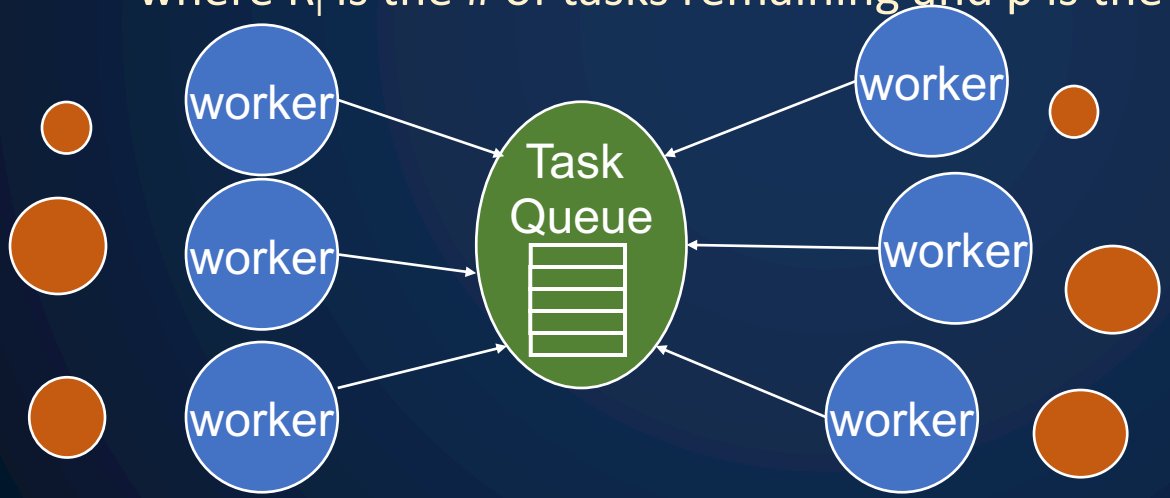


Guided Self Scheduling

- "Guided" Self Scheduling [Kuck and Polychronopolous '87]
  - The chunk size K_i at the ith access to the task pool is given by

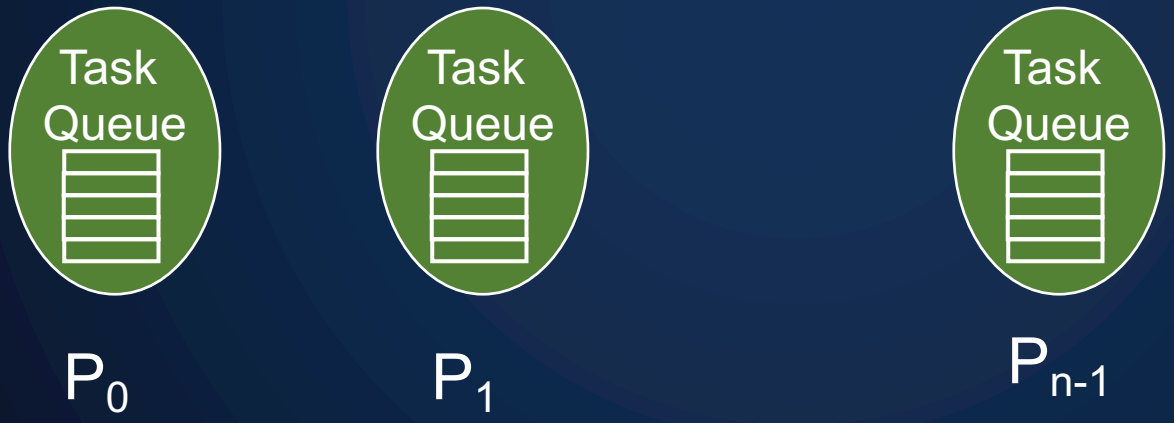
$$K_i = \text{ceiling}(R_i/p)$$

  - where R_i is the # of tasks remaining and p is the number of processors



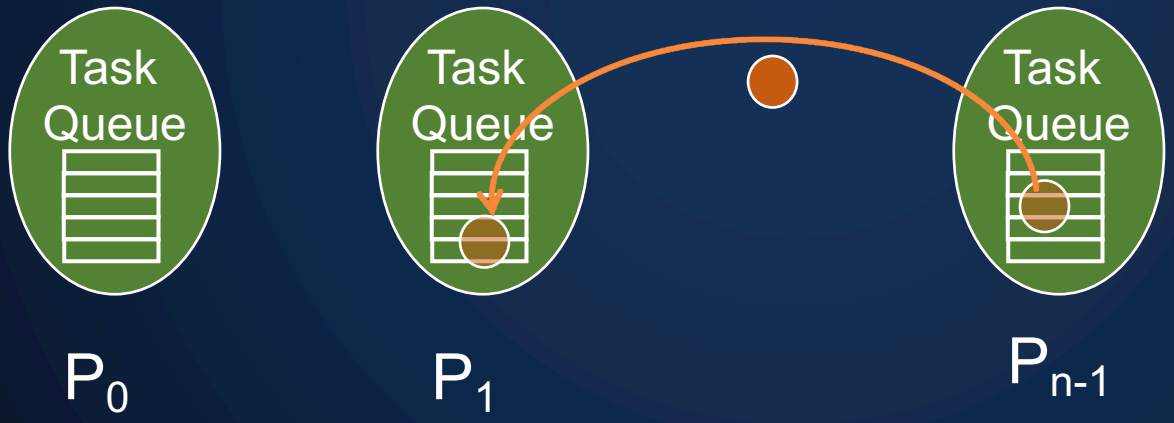
Privatizing the Queue

- Avoid bottleneck of a shared queue
- Use a “distributed” queue (1 per processor)



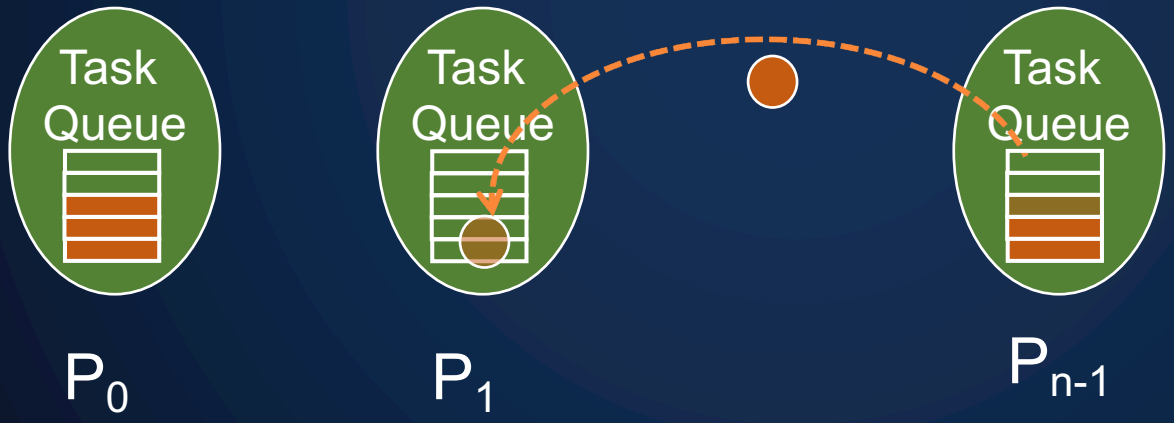
Task Stealing

- Use your own queue unless it's empty
- Then steal from another processor's queue



Task Stealing

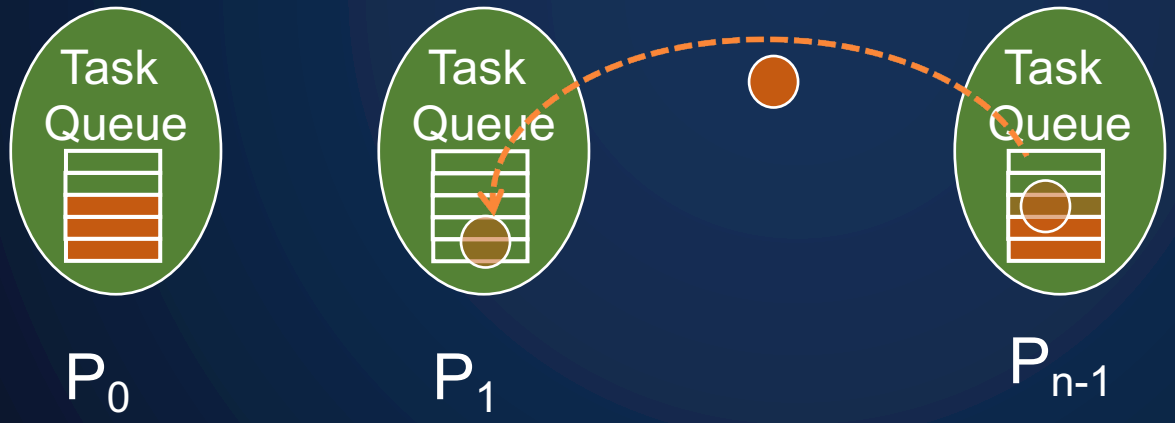
- How many tasks (see previous chunking GSS)
- Which one to steal?



Task Stealing

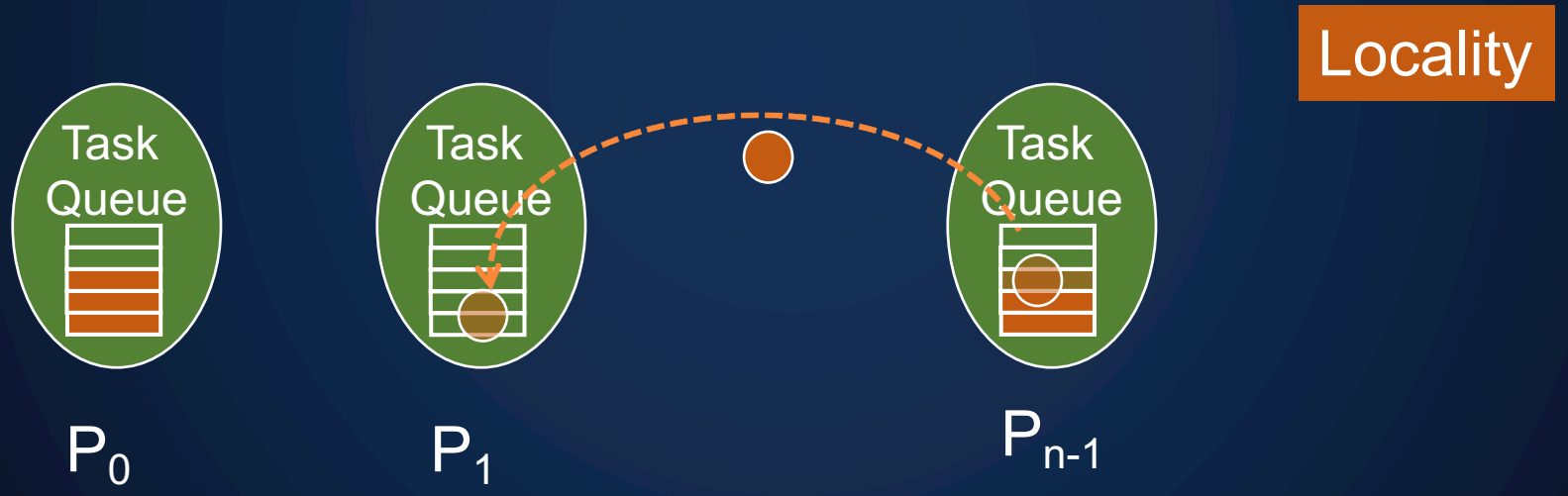
- Which one to steal?
  - Take my own tasks from the “top” like a LIFO stack

Why?



Task Stealing

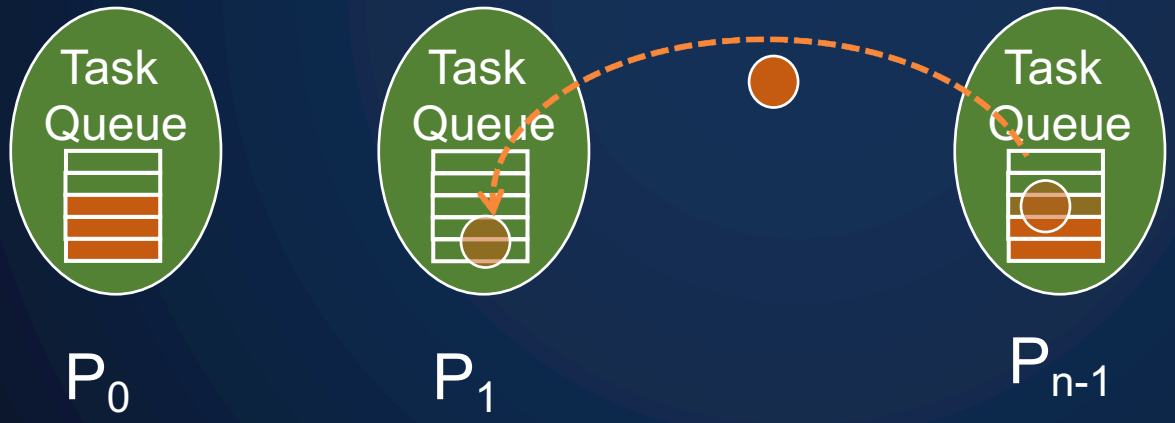
- Which one to steal?
  - Take my own tasks from the “top” like a LIFO stack



Task Stealing

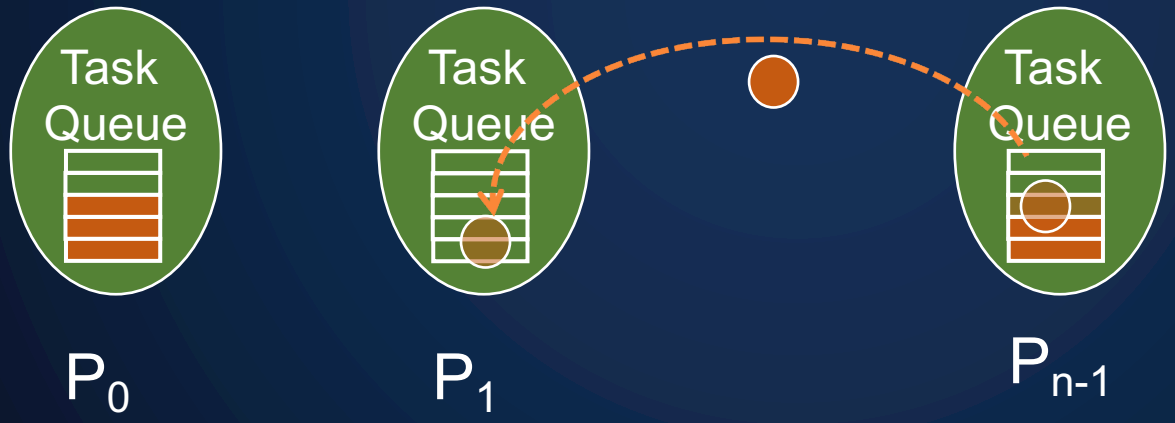
- Which one to steal?
  - Take my own tasks from the “top” like a LIFO stack
  - Steal from another like a FIFO queue

Why?



Task Stealing

- Which one to steal?
  - Take my own tasks from the “top” like a LIFO stack
  - Steal from another like a FIFO queue



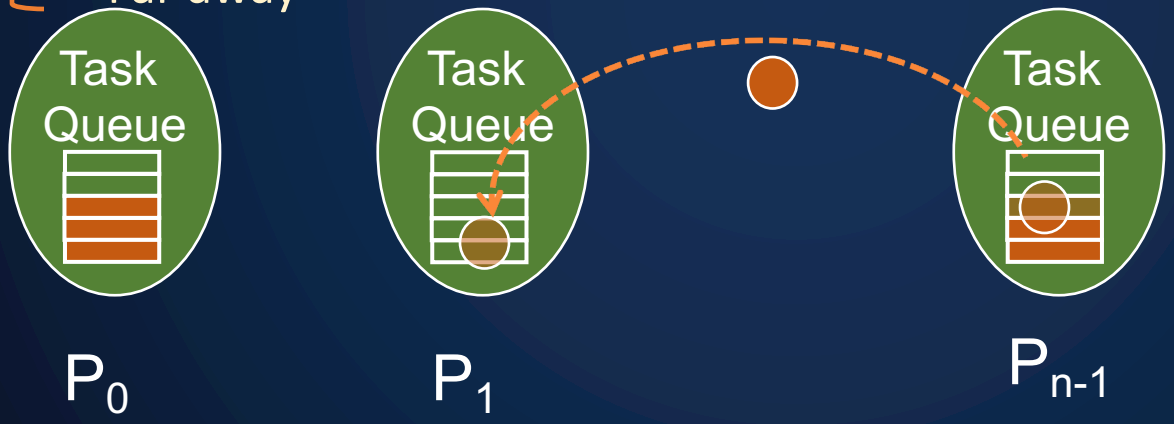
Older, more work (e.g., larger subtree)

Task Stealing

- Which queue to steal from?
  - A different one each time
  - Neighbors
  - Far away

vs

Why?

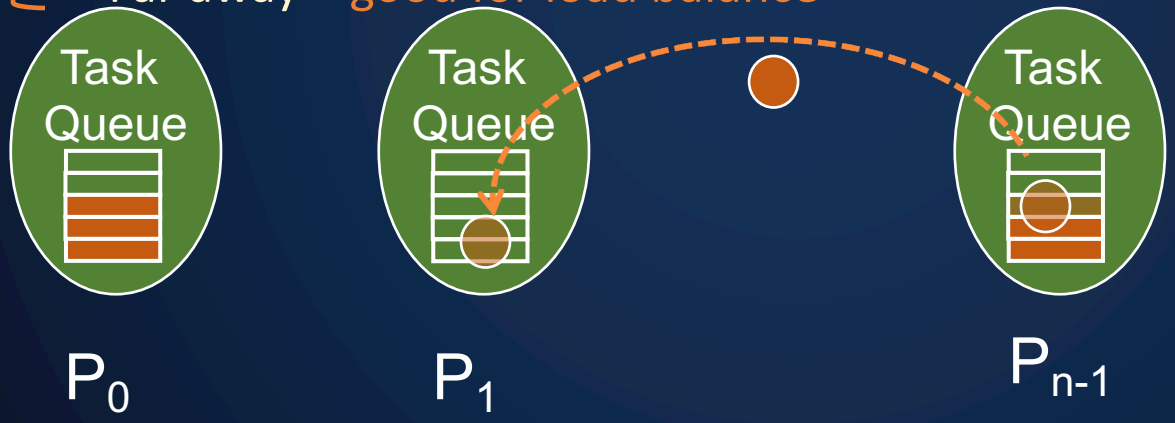


Task Stealing

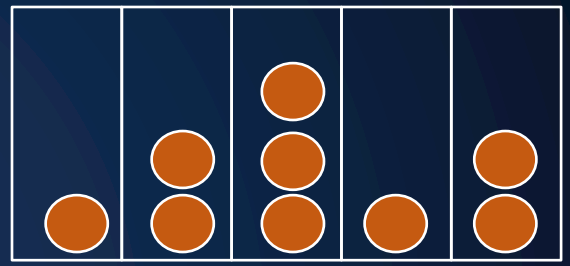
- Which queue to steal from?
  - A different one each time
  - Neighbors – good for locality
  - Far away – good for load balance

vs

But picking a random processor is best in general



Balls-in-bins: Randomization



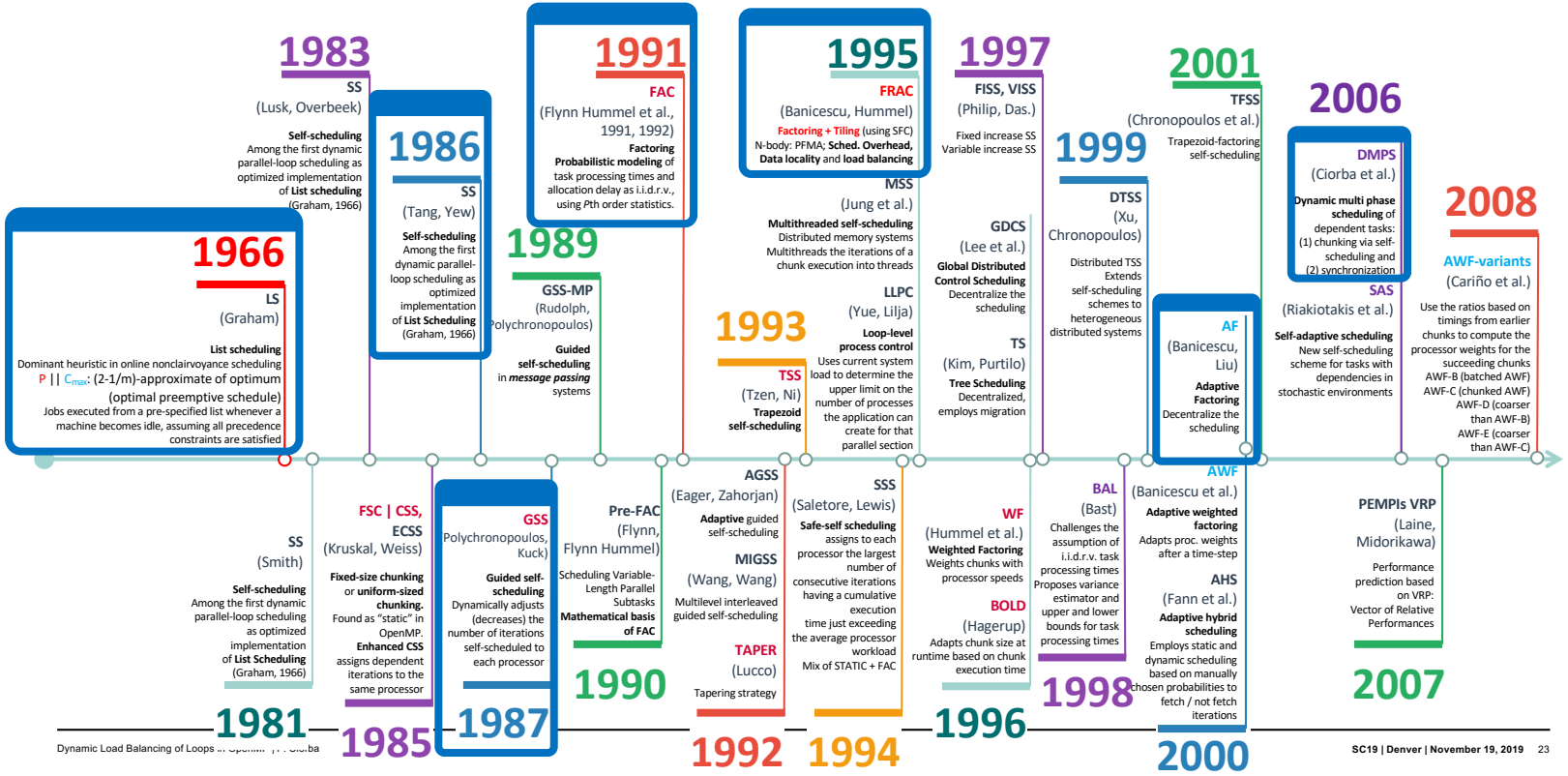
P = 5 (number of bins)

- Classic balls-and-bins result [Kotz '97, Kolchin '98]
  - Tasks are balls and processors are bins
- Can prove properties "with high probability" using randomization
- Given n balls thrown randomly into p bins: With high probability, the maximum load in any bin is:

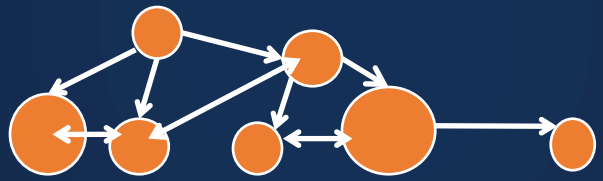
If n > p log p	If n = p
$\frac{n}{p} + \Theta\left(\sqrt{\frac{n \log(p)}{p}}\right)$	$\frac{\log(n)}{\log \log (n)} \bullet (1 + o(1))$

- Note that n/p is optimal, so this is "close"

Selected Self-scheduling (List Scheduling) Algorithms



Case 2: Tasks with Dependencies



E.g., a tree or graph (directed acyclic, i.e., DAG)

Task Dependency Spectrum

- **Easy: Set of ready tasks**

Matrix multiply, domain decomposition (loops over space)

- **Medium: Tasks have known relationship (task graph)**

Chains **Trees** **General Graphs**

Chains: Loops over time; iterative methods (outer loop)

Trees: (in –tree and out-trees); divide-and-conquer algorithms

Graphs: Direct solvers (dense and sparse), i.e., LU, Cholesky…

- **Hard: Task structure is not known until runtime**

Trees: Search

Graphs: Discrete events

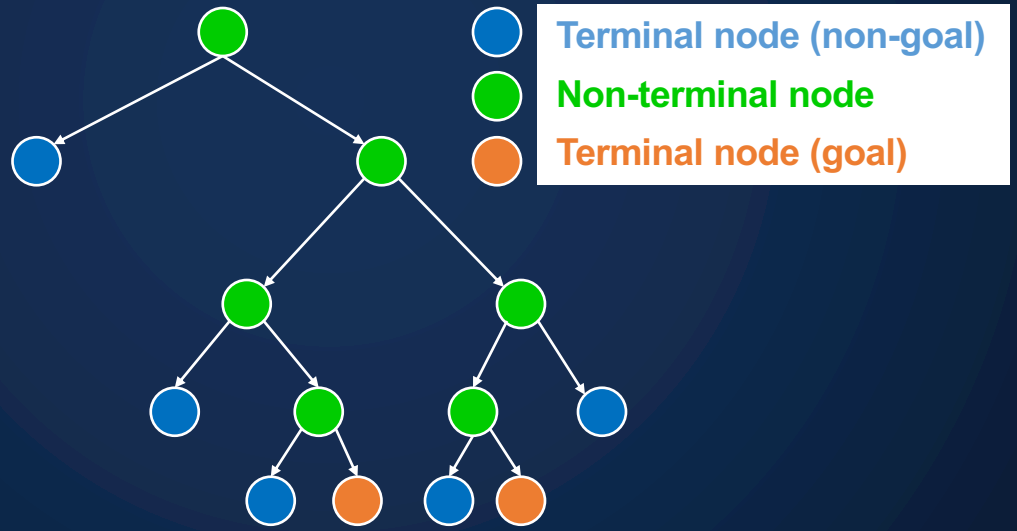
Tasks Trees: Search example

- **Search problems are often:**
  - **Computationally expensive**
  - **Have very different parallelization strategies than physical simulations.**
  - **Require dynamic load balancing**

- **Examples:**
  - **Chess and other games (N-queens)**
  - **Optimal layout of VLSI chips**
  - **Robot motion planning**
  - **Computing a (Gröbner) basis for set of polynomials**
  - **Constructing phylogeny tree from set of genes**

Example (Game) Tree Search

- Used in single and multi-player games
- Tree unfolds dynamically—structure, # nodes, unknown

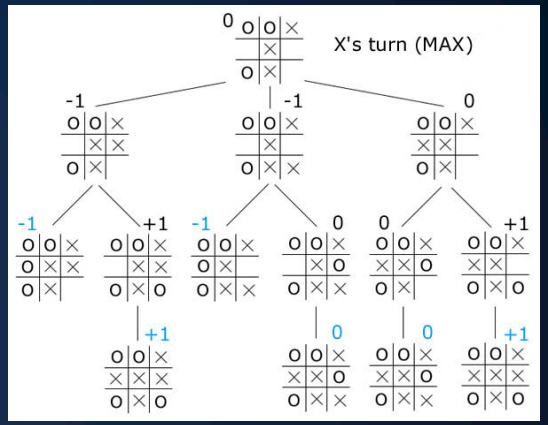


Depth vs Breadth First Search (Review)

Same as on graph, but no need to mark nodes

- **DFS with Explicit Stack – little parallelism**
  - Put root into Stack (LIFO)
  - While Stack not empty
    - Remove top of stack
    - If found goal? → return success
    - Else push child nodes on stack
- **BFS with Explicit Queue – lots of parallelism (depending on graph)**
  - Put root into Queue (FIFO)
  - While Queue not empty
    - Remove front of queue
    - If found goal? → return success
    - Else enqueue child nodes onto the end of the queue

Sequential Search Algorithms

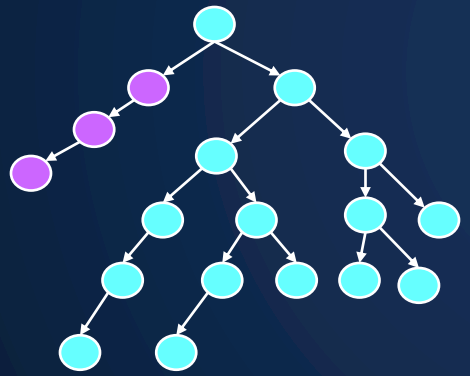


Tree search approaches

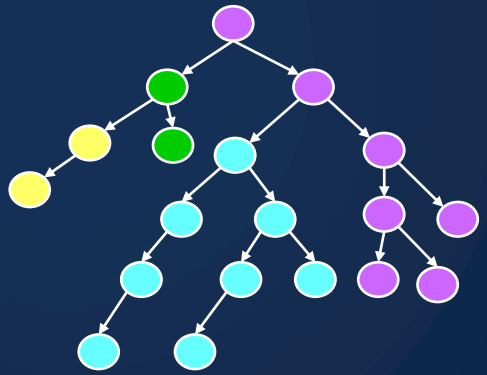
- **Simple backtracking**
  - Search to bottom, backing if necessary
- **Branch-and-bound**
  - Keep track of best solution so far (“bound”)
  - Cut off sub-trees that are guaranteed to be worse than bound
- **Iterative Deepening (“in between” DFS and BFS)**
  - Choose a bound d on search depth, and use DFS up to depth d
  - If no solution is found, increase d and start again
  - Can use an estimate of cost-to-solution to get bound on d

Parallel Search

- **Consider backtracking search**
- **Try static load balancing: spawn each new task on an idle processor, until all have a subtree**



Load balance on 2 processors



Load balance on 4 processors

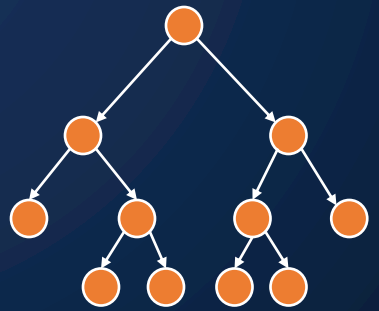
Clearly a bad idea…unknown task count. And they arrive 'sequentially.'

Theoretical Results

Simple randomized algorithms are optimal with high probability

- Generalizations for independent tasks
  - "Throw n balls into n random bins": Θ (log n / log log n) in fullest bin
  - Throw d times and pick the emptiest bin: log log n / log d [Azar]
  - Extension to parallel throwing [Adler et all 95]
  - Shows p log p tasks leads to "good" balance

- Karp and Zhang show this for a tree of unit cost tasks
  - Parent must be done before children
  - Tree unfolds at runtime
  - Task number/priorities not known a priori
  - Children "pushed" to random processors



Theoretical Results (2)

Randomized algorithms optimal with high probability

- Blumofe and Leiserson [94] show this for a fixed task tree of variable cost tasks
  - Task pulling (stealing), when a processor becomes idle, it steals from random processor
  - Good for locality, balances slowly
  - Also have (loose) bounds on the total memory required
  - Used in Cilk
- Chakrabarti et al [94] show this for a dynamic tree of variable cost tasks
  - Uses randomized pushing
  - Quickly load balances, worse for locality
  - Works for branch and bound, i.e. tree structure can depend on execution order

Randomized work pushing

- For sufficiently large compute : communicate ratios randomized work stealing is effective even on distributed memory (admittedly small, old Connection Machine CM5)

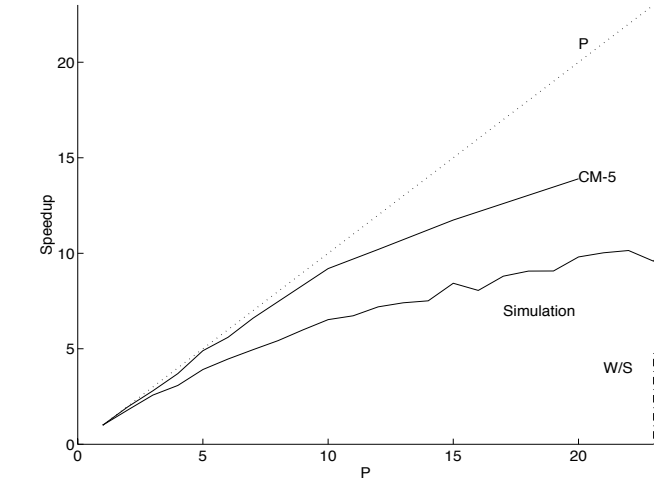


Figure 3: Performance on the Gröbner basis example. Speedups were averaged over 20 runs.

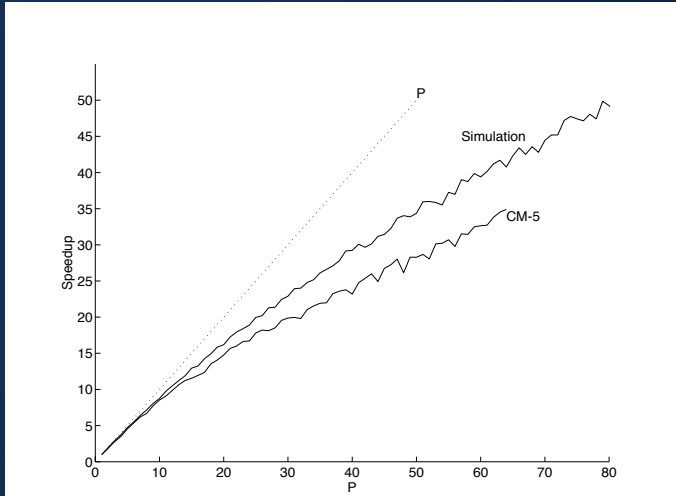


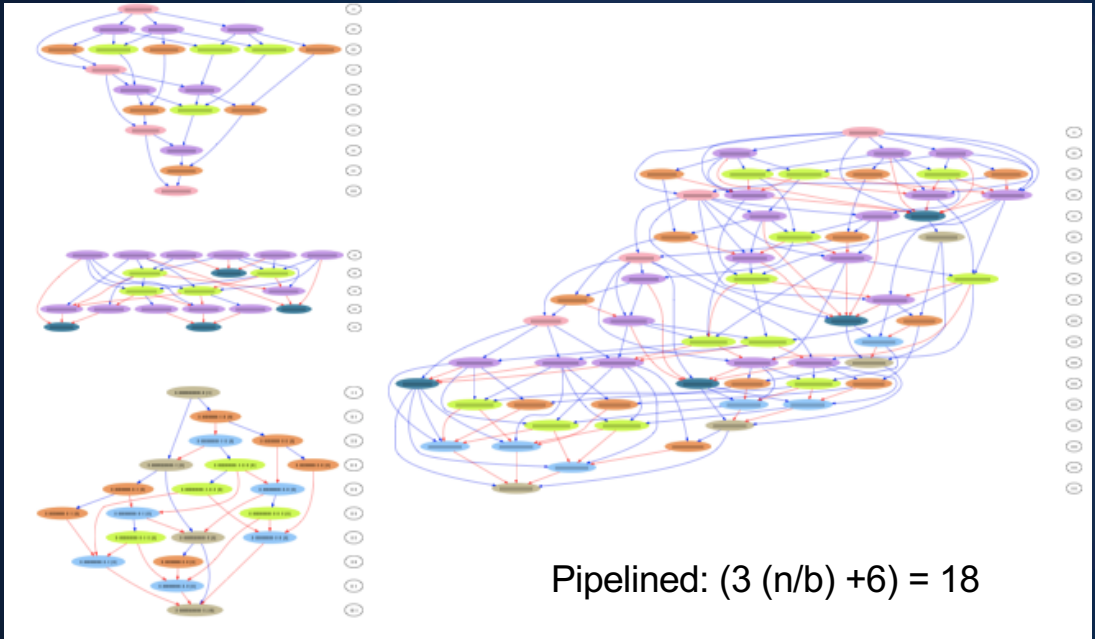
Figure 4: Performance on the eigenvalue example. Speedups were averaged over 20 runs.

Chakrabarti, Ranade, Yelick 1994

DAG Scheduling software

- DAGuE (U. Tennessee)
  - Library developed to support (originally) dense linear algebra
- SMPss (Barcelona)
  - Compiler based; Data usage expressed via pragmas; Proposal to be in OpenMP; Recently added GPU support
- StarPU (INRIA)
  - Library based; GPU support; Distributed data management; Codelets=tasks (map CPU, GPU versions)
- OpenMP4.0 ➔ GCC 4.9
  - See openmp.org
- Other tools (e.g., fork-join graphs only)
  - Cilk, Intel Threaded Building Blocks (TBB), Microsoft CCR, SuperGlue and DuctTEiP (Uppsala), …

DAG scheduling, e.g., Cholesky



POTRF+TRTRI+LAUUM:
- span = (7 (n/b) -3) = 25
- here (n/b=4)

Cholesky Factorization alone:
- 3 (n/b) -2

Source: Julien Langou: ICL presentation 2011/02/04

Scalability of DAG Schedulers

- How many tasks are there in DAG for dense linear algebra operation on an n x n matrix with b x b blocks?

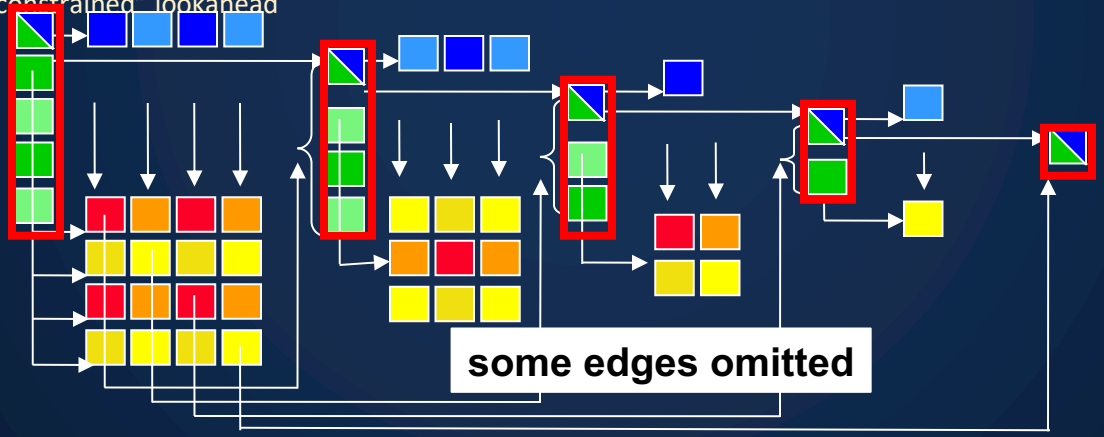
  $O((n/b)^3) = 1M$, for n=10,000 and b = 100

- Creating, scheduling entire DAG does not scale
- PLASMA: static scheduling of entire DAG
- QUARK: dynamic scheduling of "frontier" of DAG at any one time
- DAGuE: Symbolic interpretation of the DAG

Dongarra et al, U. Tenn

Event Driven LU in UPC

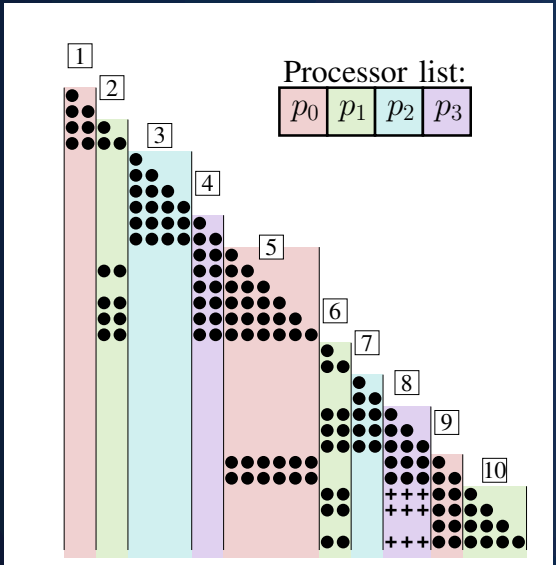
- DAG Scheduling before it's time
- Assignment of work is static; schedule is dynamic
- Ordering needs to be imposed on the schedule
  - Critical path operation: Panel Factorization
- General issue: dynamic scheduling in partitioned memory
  - Can deadlock in memory allocation
  - "memory constrained" lookahead

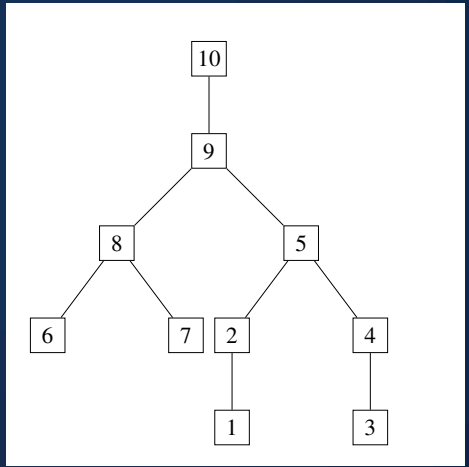


"Multithreading and 1-sided communication in LU…" Husbands and Yelick, SC07.

symPACK: Sparse Cholesky

- Sparse Cholesky using fan-both algorithm in UPC++
  - Uses asyncs with dependencies





Matthias Jacquelin, Yili Zheng, Esmond Ng, Katherine Yelick

symPACK: Sparse Cholesky

- Scalability of symPACK on Cray XC30 (Edison)

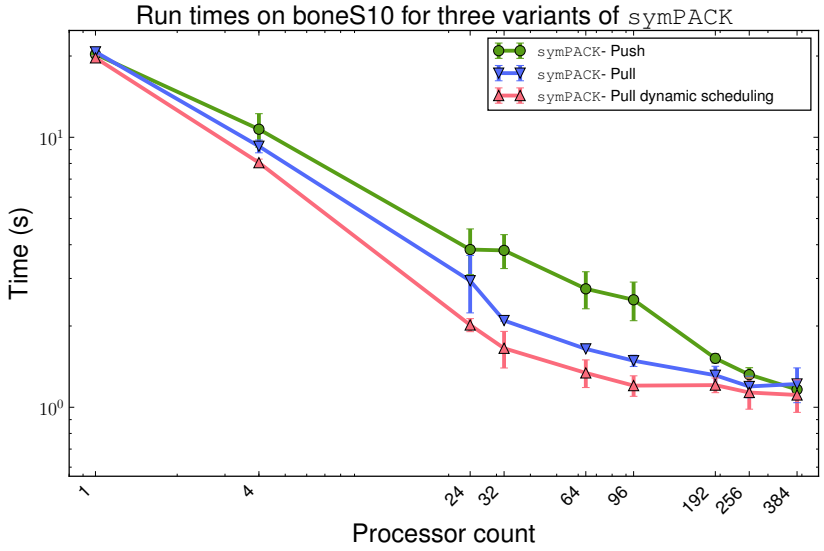
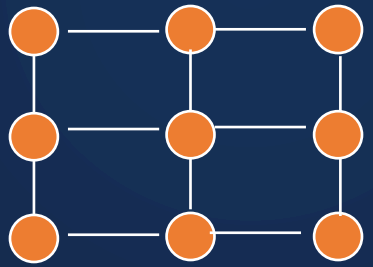


Figure 7: Impact of communication strategy and scheduling on symPACK performance

Matthias Jacquelin, Yili Zheng, Esmond Ng, Katherine Yelick

- Comparable or better than best solvers (evaluation in progress)
- Notoriously hard parallelism problem

Case 3: Tasks with Communication



Task Communication Spectrum

- **Static: Regular (or none)**

Nearest neighbor on mesh, dense matrices, regular mesh, FFTs, direct n-body (all-to-all) collectives (hard to scale; easy to schedule)

- **Semi-Static: Communication pattern can be pre-computed**

Sparse direct linear algebra solvers, e.g., LU, Cholesky, AMR, Tree-structured n-body

Can pre-arrange send/receive pairs

- **Dynamic: Random access – pattern is not known in advance and does not repeat**

Search, Discrete events, Sparse updates, histogram, hash tables

Charm++
Load balancing based on Over-decomposition

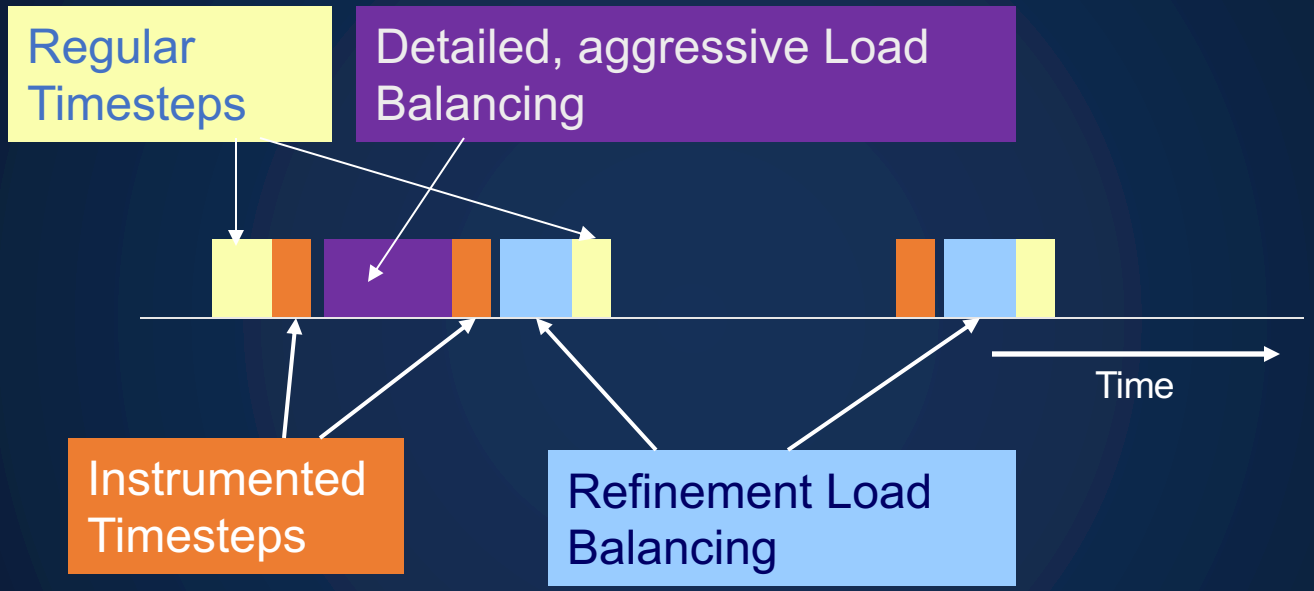
- Context: "Iterative Applications"
  - Repeatedly execute similar set of tasks
- Idea: decompose work/data into chunks (*chares*), units for load balance
- How to predict the computational load and communication? Could be:
  - user-provided
  - simple metrics (e.g. data size)
  - *principle of persistence*
    - Performance changes slowly, can rebalance occasionally
- Software, documentation at charm.cs.uiuc.edu
  - Many applications: NAMD, LeanMD, OpenAtom, ChaNGa, ...

Source: Laxmikant Kale

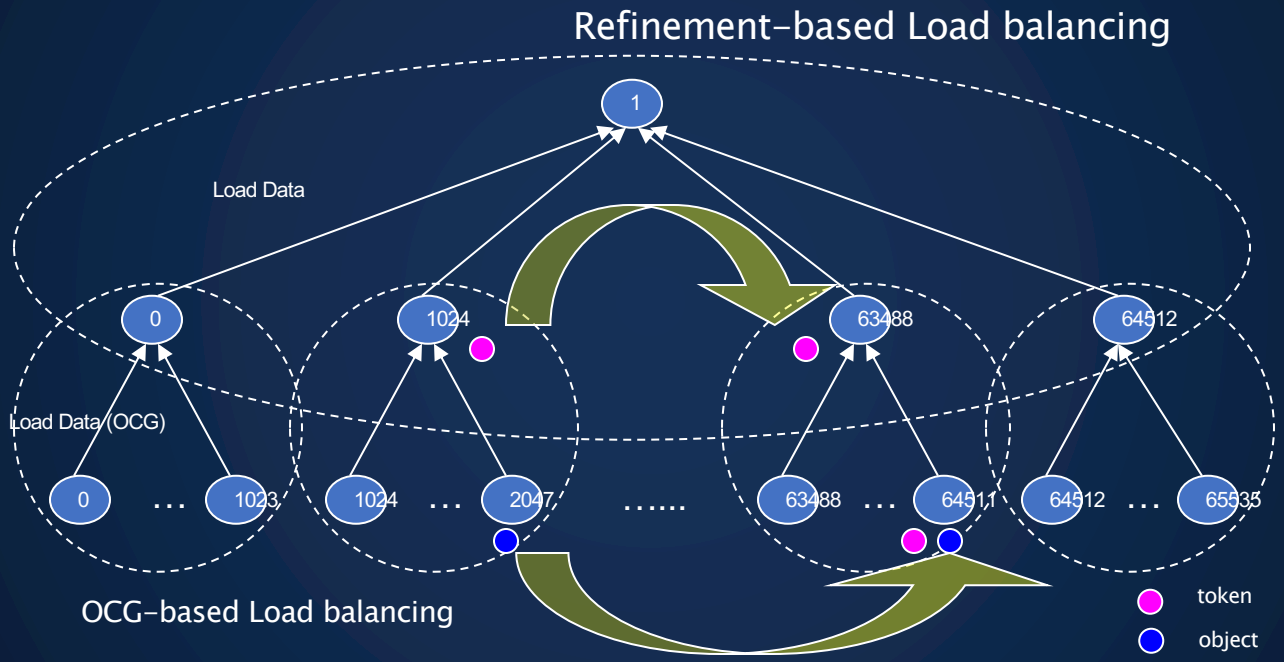
Load Balancing in Charm++

- Principle of persistence (A Heuristic)
  - *Communication patterns and computational loads tend to persist*
  - In spite of dynamic behavior
    - Abrupt but infrequent changes
    - Slow and small changes
- Measurement based load balancing
  - Charm++ measures load and communication
  - Periodically make new decisions, and migrates objects
- Provides a suite of plug-in strategies and user-defined ones
  - Meta-balancer decides how often to balance, and strategy

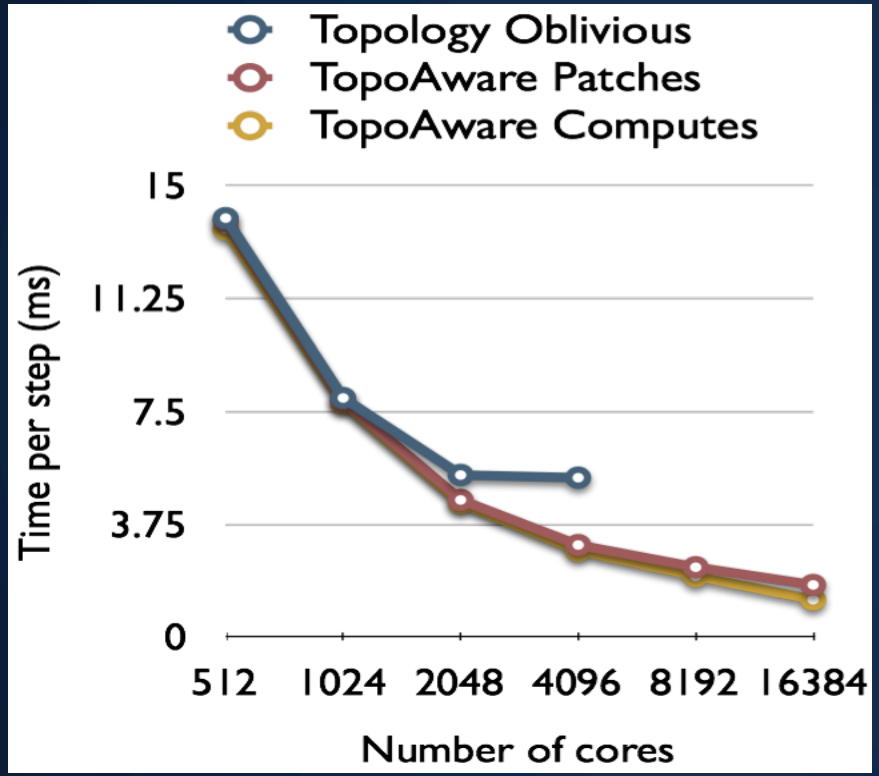
Load Balancing Steps



Charm++ Hierarchical Load Balancer



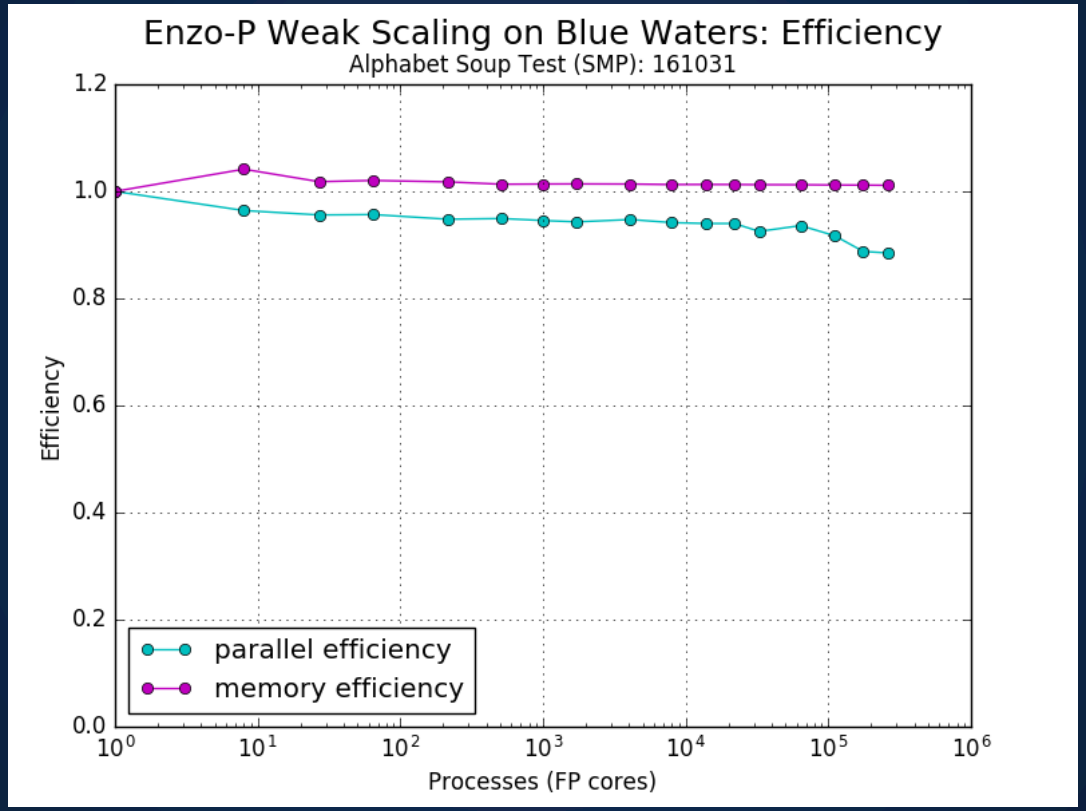
Efficacy of Topology aware load balancing



NAMD biomolecular simulation running on BG/P

Source: Laxmikant Kale, UIUC

Enzo astrophysics code using CHARM++



Source: Mike Norman, UCSD

Summary and Take-Home Messages

- There is often a trade-off between locality and load balance
- Many algorithms, papers, & software for load balancing
- Key to understanding how and what to use means understanding your application domain and their target
  - Shared vs. distributed memory machines
  - Dependencies among tasks, tasks cost, communication
  - Locality oblivious vs locality "encouraged" vs locality optimized
    Computational intensity: ratio of computation to data movement cost
  - When you know information is key (static, semi, dynamic)
- Will future architectures require dynamic load balancing?