Load Balancing

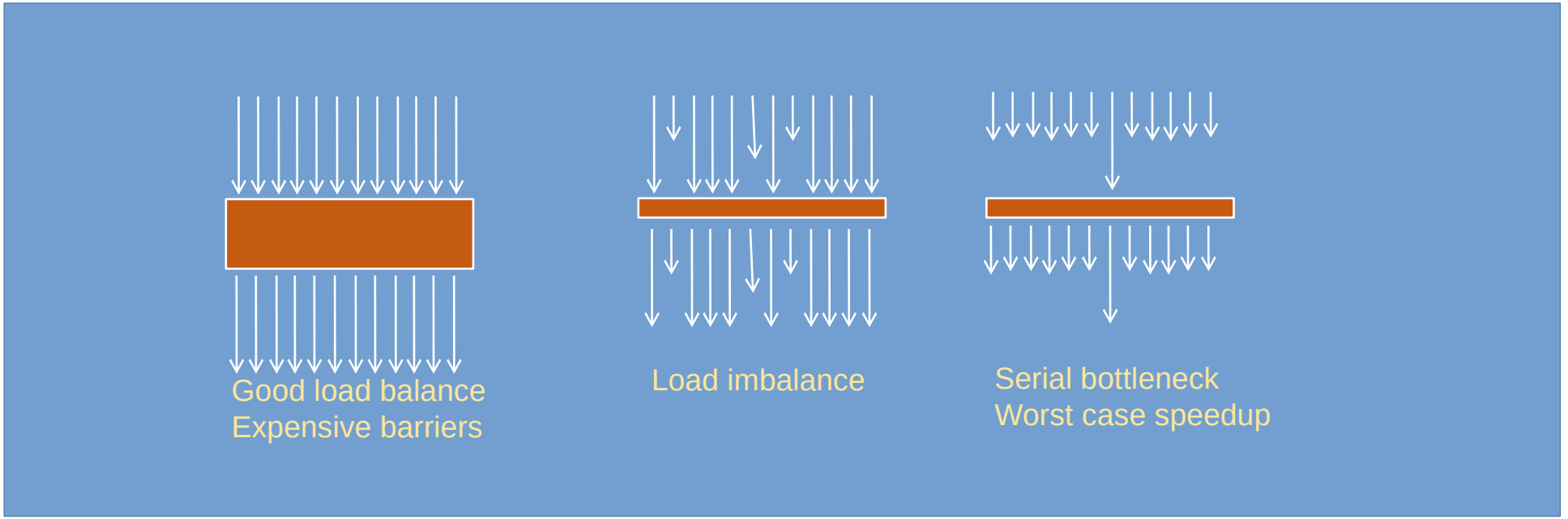
- Causes of load imbalance
- Load Balancing Approaches
  - Independent tasks
  - Tasks with dependencies
  - Tasks with communication
- Task Scheduling
  - Static
  - Dynamic
  - Randomization, Diffusion
  - DAG Scheduling

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)



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

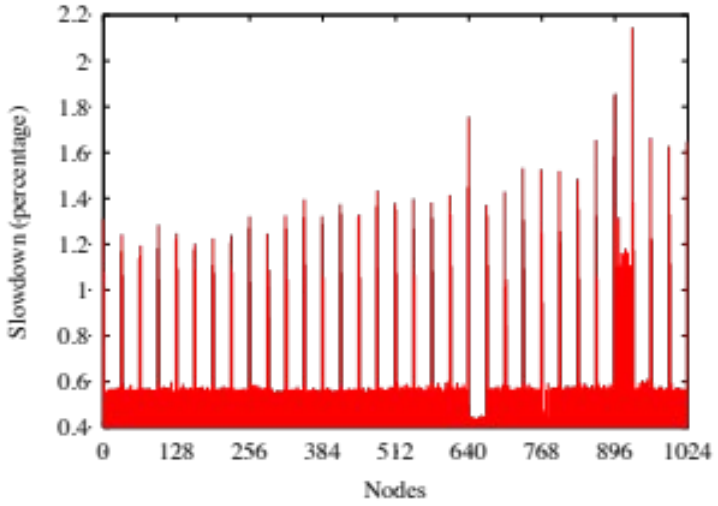
  (max – average) / average
- “Spin locks” look like useful work (but obviously are not)

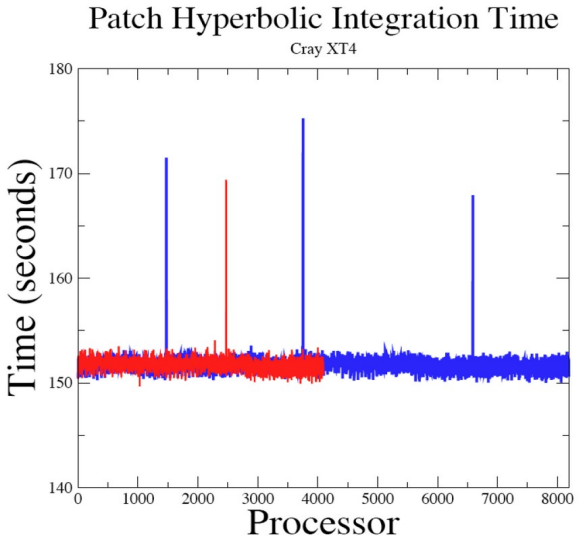
Sources of Load Imbalance

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

System performance variability

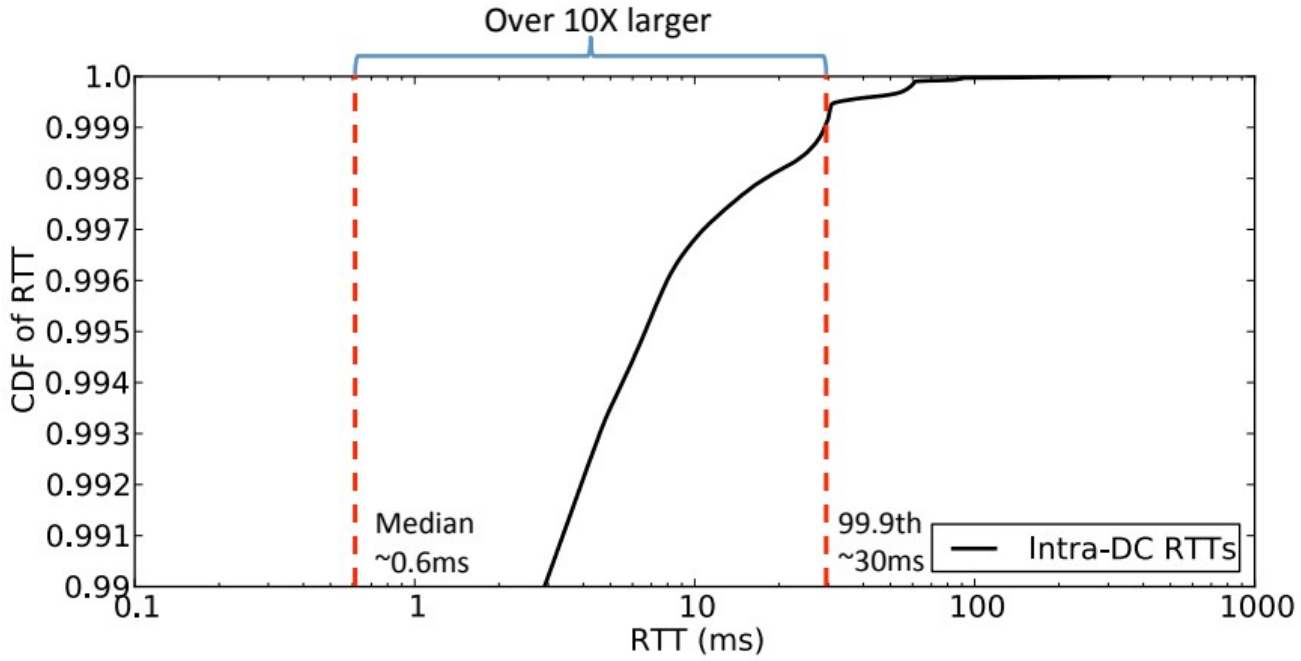
- Load imbalance isn't always an application level problem: Performance variability can come from hardware or operating system effects





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

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



Load Balancing Approaches

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

Load balancing *independent tasks*

- Loop over a set of independent computations

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

  … loop body ...

  }

  - *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 (due to 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, irregular boundaries, subgrid physics, unpredictable machines

Bin-packing for statically known costs is hard … NP-hard

- 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)
- So good load balance is done approximately

Dynamic Scheduling

- What if work estimates are unknown?
  - Until mid-execution of an iteration / chunk
  - E.g., Mandelbrot, game trees
- Dynamic scheduling:
  - Self Scheduling
  - Chunk Scheduling
  - Guided Self Scheduling

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)
  - How to choose K?
    - 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 = ceiling(R_i/p) where R_i is the # of tasks remaining and p is the number of processors

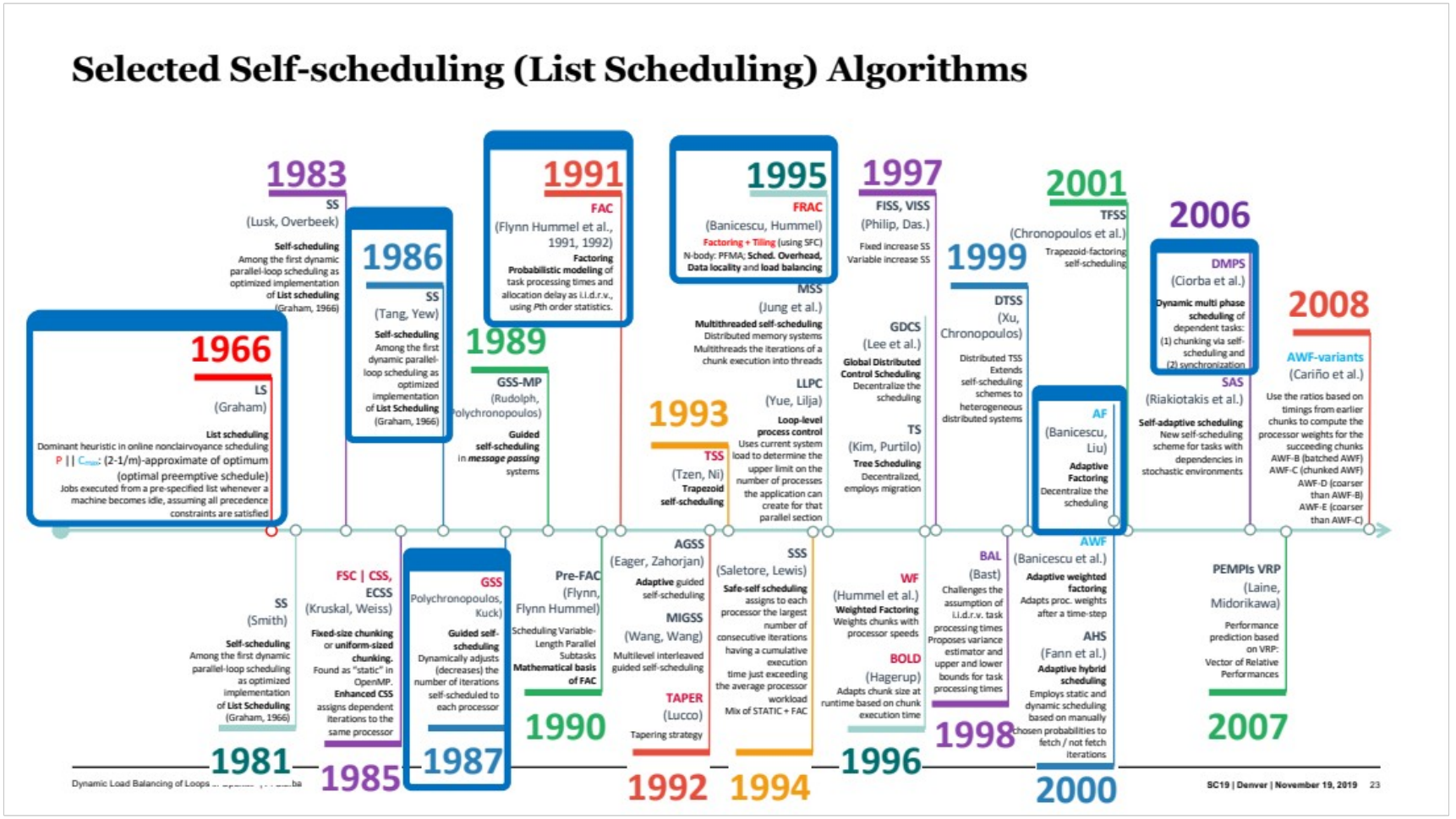
Privatizing the Queue

- Avoid bottleneck of a shared queue
- Use a “distributed” queue (1 per processor)
- But what if the queue is empty?
  - Then steal from another processor’s queue, which one?
    - A different one each time
    - Neighbors – good for locality
    - Far away – good for load balance
  - How many to steal (see chunking)?
  - Which jobs to steal?

Balls-in-bins: Randomization

- 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

Self scheduling Algorithms



Load Balancing *Tasks with Dependencies* Trees, Graphs (DAGs)

Easy: Set of ready tasks	Matrix multiply, domain decomposition (loops over space)
Medium: Tasks have known relationship (task graph)	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

Task 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

Depth vs Breadth First Search (Review)

- 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

- 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 Algorithms

- Consider backtracking search
- Try static load balancing: spawn each new task on an idle processor, until all have a subtree
- Clearly a bad idea…unknown task count. And they arrive ‘sequentially.’

Theoretical random results

Simple randomized algorithms are optimal with high probability

- Generalizations for independent tasks
  - “Throw n balls into n random bins”: O(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
- For a tree of unit cost tasks [Karp, Zhang]
  - Parent must be done before children
  - Tree unfolds at runtime
  - Task number/priorities not known a priori
  - Children “pushed” to random processors

More theoretical results

- 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
- 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 Load Balancing

- Want to avoid bottlenecks of shared queue
  - Especially in distributed memory (but even in shared)
  - So Self-scheduling, Chunked Self Scheduling, Guided Self Scheduling are not good
- Use distributed queues with stealing / sharing
- How to select processor?
  - Asynchronous or global round robin
  - Randomize pushing: balances quickly, may lose spatial locality
  - Randomized pulling: balances slowly, preserves locality as much as possible

Greedy Scheduling

Do as much as possible on every step.

- Definition: A thread is ready if all its predecessors have Executed.
- Complete step
  - ≥ P threads ready.
  - Run any P.
- Incomplete step
  - < P threads ready.
  - Run all of them.

Distributed Task Queues

- The obvious extension of task queue to distributed memory is:
  - a distributed task queue (or “bag”), i.e., one per processor
  - Idle processors can “pull” work, or busy processors “push” work
- When are “distributed” queues a good idea?
  - Distributed memory multiprocessors
  - Often on shared memory to avoid synchronization contention
  - Locality between tasks is not (very) important
  - The costs of tasks and/or number is not known in advance
- Side note on terminology:
  - Queue: First-In-First-Out (FIFO)
  - Stack: Last-In-First-Out (LIFO)
  - Bag: Arbitrary-Out

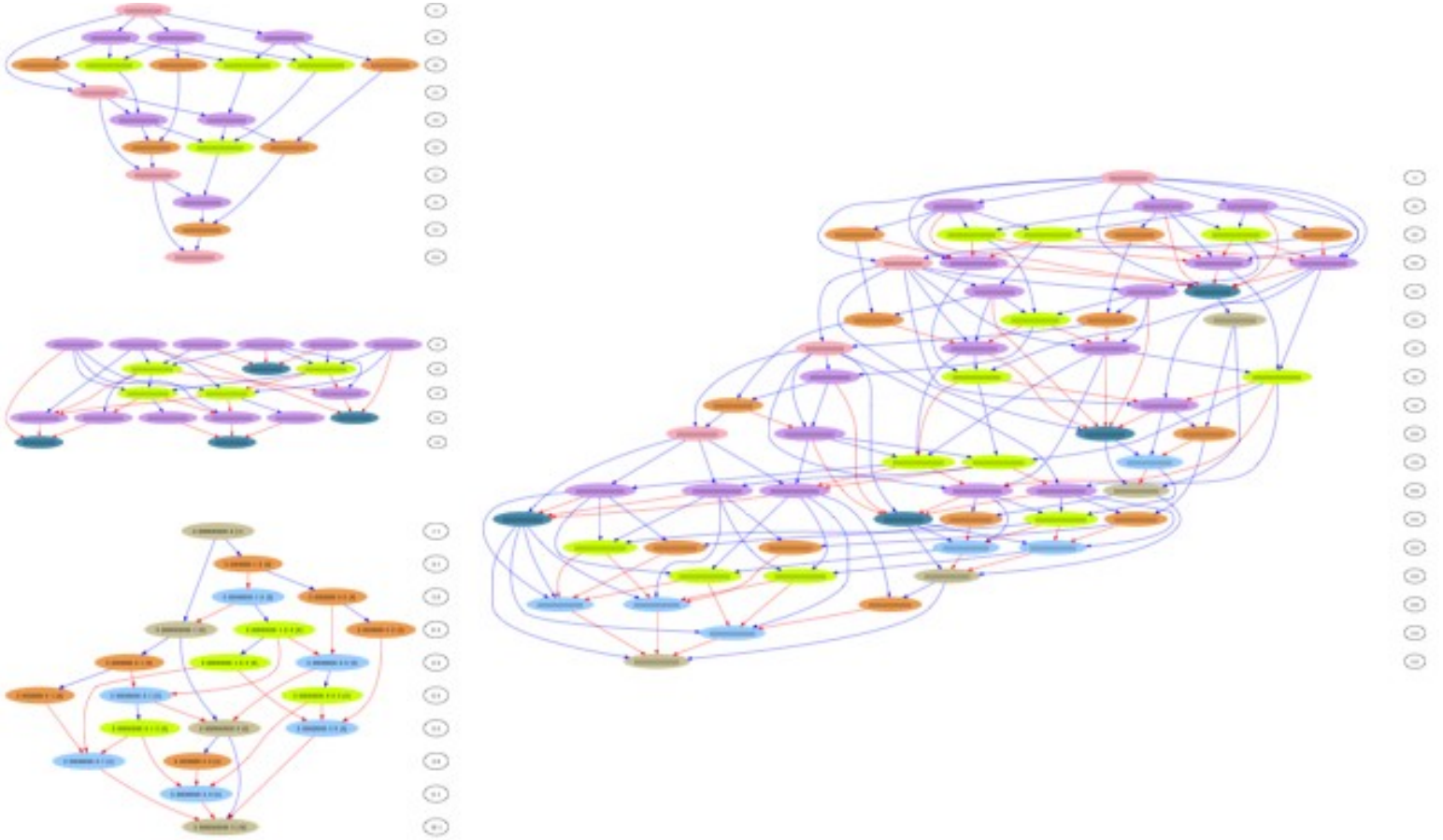
Diffusion-Based Load Balancing

- In the randomized schemes, the machine is treated as fully-connected.
- Diffusion-based load balancing takes topology into account
  - Send some extra work to a few nearby processors: Average work with nearby neighbors
  - Locality properties better than choosing random processor
  - Load balancing somewhat slower than randomized
  - Cost of tasks must be known at creation time
  - No dependencies between tasks

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

DAG Scheduler



How to Select a Donor/Acceptor Processor

- Independent round robin (common bug: all start looking at p0)
  - Each processor k, keeps a variable "$target_k$"
  - When a processor runs out of work, requests work from $target_k$
  - Set $target_k$ = ($target_k$ +1) mod procs
- Global round robin
  - Proc 0 keeps a single variable "target"
  - When a processor needs work, gets target, requests work from target
  - Proc 0 sets target = (target + 1) mod procs
- Random stealing
  - When a processor needs work, select a random processor and request work from it
- Random pushing
  - When a processor has too much work (at least two tasks), push tasks to a random processor

Load Balancing
Tasks with communication

Static	Regular meshes, dense matrices, direct n-body, FFT
Semi-Static: Communication pattern can be pre-computed	Adaptive and unstructured meshes, sparse matrices, tree-based n-body, particle-mesh methods
Dynamic: Random access – pattern is not known in advance and does not repeat	Search, irregular boundaries, subgrid physics, unpredictable machines, hashing

Load balancing based on Over-decomposition

- Context: “Iterative Applications”
- Repeatedly execute similar set of tasks
- Idea: decompose work/data into chunks, units for load balance
- How to predict the computational load and communication?
  - user-provided
  - simple metrics (e.g. data size)
  - principle of persistence
    - Performance changes slowly, can rebalance occasionally
- Implemented in charm++ (UIUC)

Summary

- 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?