CS 267: Applications of Parallel Computers

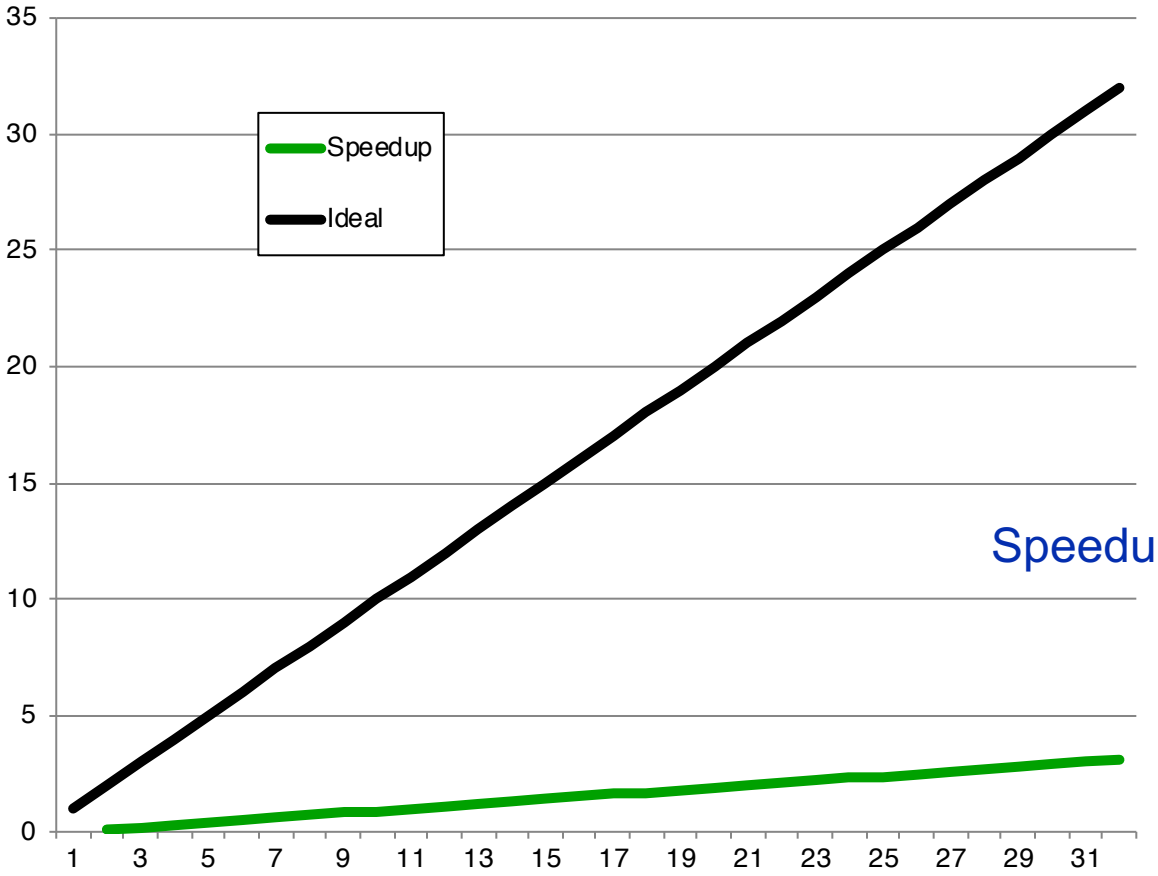
Dynamic Load Balancing

Kathy Yelick

https://sites.google.com/lbl.gov/cs267-spring-2018/

First, some speedup flash cards

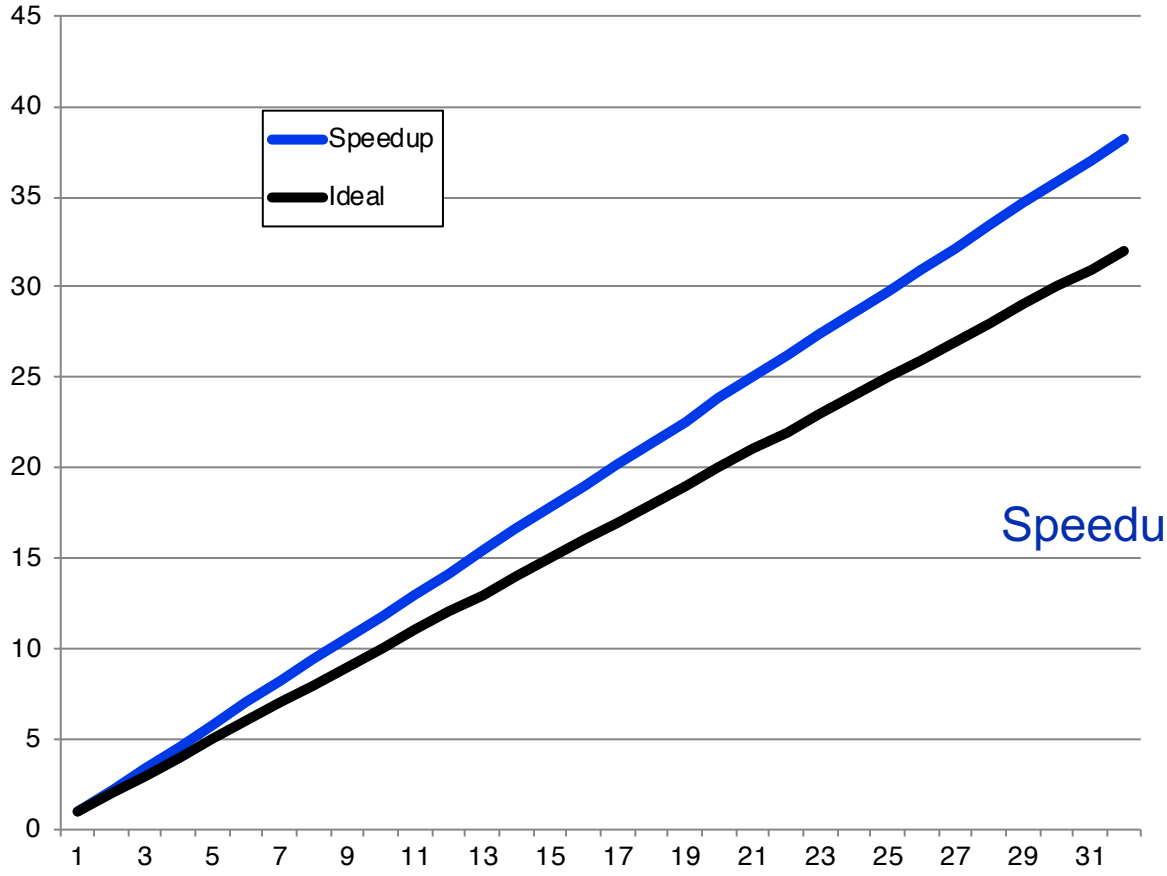
Problem #1: Perfect scaling, but slow



Speedup = T_1 / T_P

- Why would a speedup curve look like this?

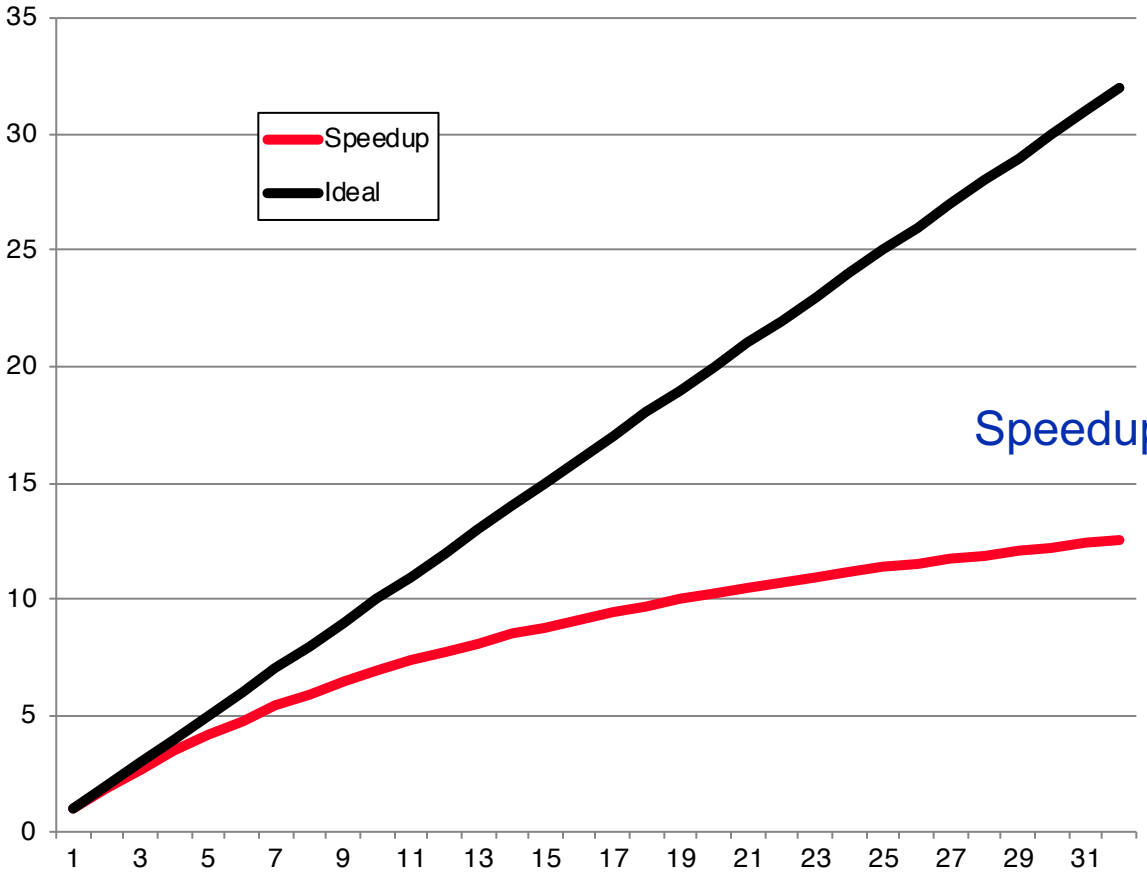
Problem #2: Superlinear speedup



Speedup = T_1 / T_P

- Why would a speedup curve look like this?

Problem #3: Poor scaling



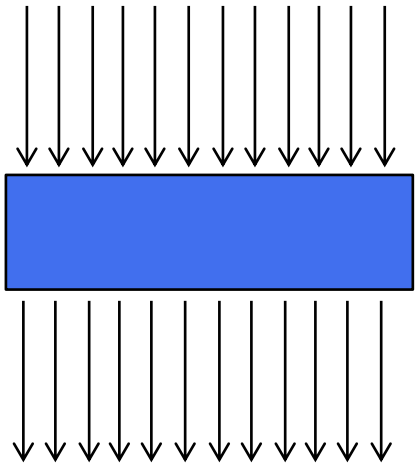
Speedup = T_1 / T_P

- Why would a speedup curve look like this?

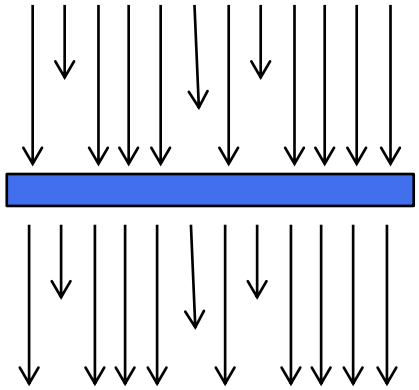
Causes of poor scaling

- High communication cost
- High synchronization overhead
  - Processors waiting in barriers or locks
- Idle processors (aka load imbalance)
  - Load imbalance
  - Lack of parallelism (load cannot be balanced)
  - Serial bottleneck (Amdahl's law)

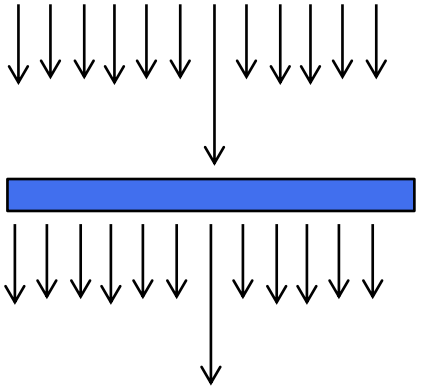
It can be hard to distinguish between these, especially when using collective communication



Good load balance
Expensive barriers



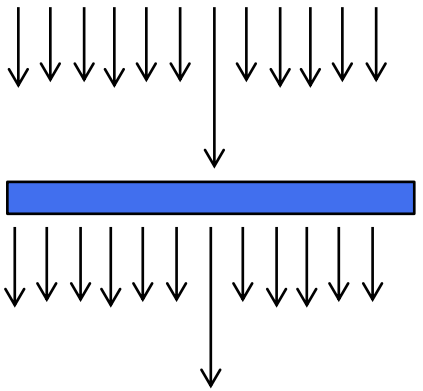
Load imbalance



Serial bottleneck
Worst case speedup

Measuring Load Imbalance

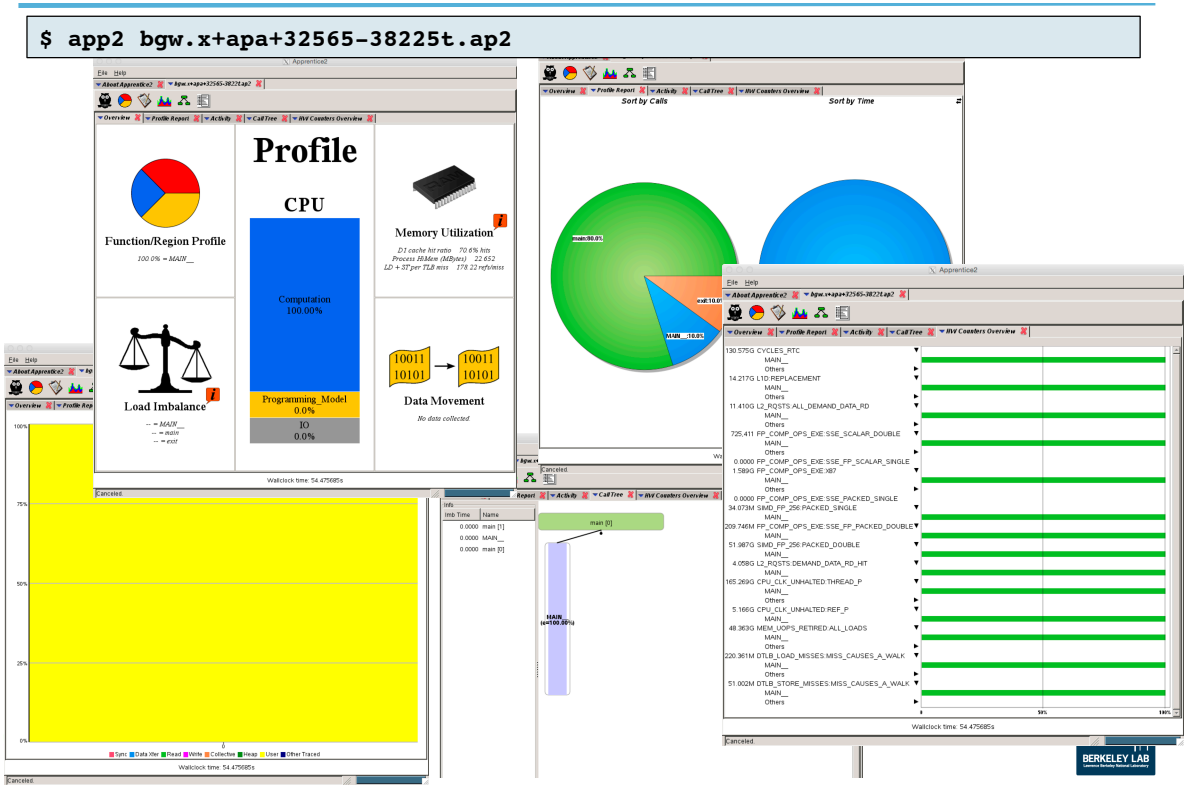
- **Also hard to separate load imbalance from high synchronization overhead**
- **Basic measurement: timers around barriers**
  - Don't average! Need to look at all values or histogram
  - Especially subtle if not bulk-synchronous
  - "Spin locks" can make synchronization look like useful work
  - Imbalance may be caused by hardware (caches, dynamic clocks,…)



- **Lack of parallelism is inherent load imbalance – can't be fixed with better load balancing**

Measuring Load Imbalance

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

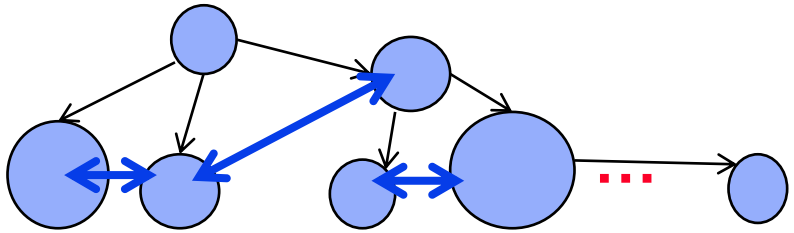


Beware of Heisenbugs in performance.

http://www.nersc.gov/users/software/performance-and-debugging-tools/

Overview of lecture

- Motivation for Load Balancing
- Spectrum of difficulty
  - Tasks cost spectrum
  - Tasks dependence spectrum
  - Tasks communication (locality) spectrum

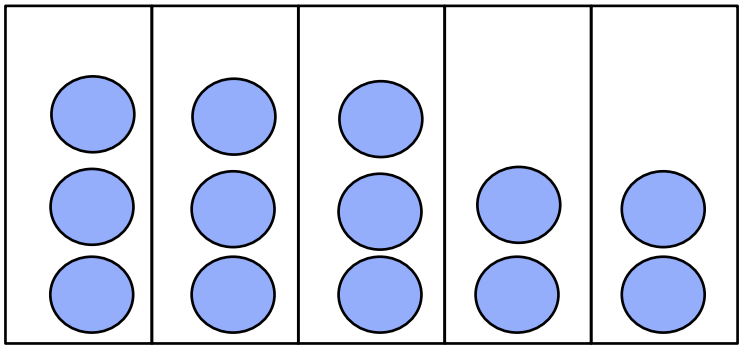


- Notes:
  - Each of these factors (cost/count, dependencies, communication) are independent
  - Load balancing: where (which processor) to execute
  - Scheduling: when to execute

Balls-in-bins – Using Randomization

- Load balancing is often a balls-and-bins problem
  - Tasks are balls
  - Processors are bins
- As evenly as possible

  divide balls between bins

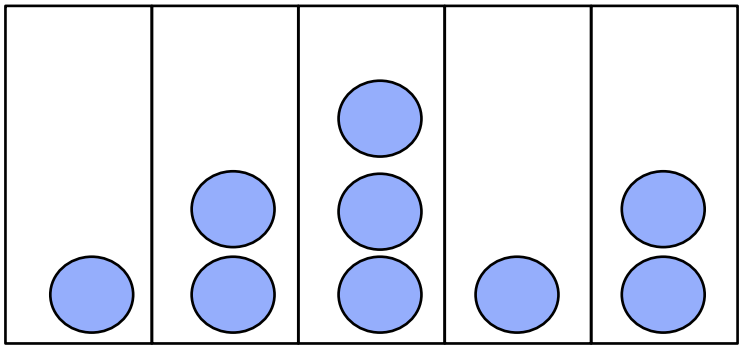


P = 5 (number of bins)

- What if we don't know how many balls there will be?
  - Can always put ball in least loaded bin (if we know what that is – requires global information) and get load [n/p]
- How bad is the load imbalance if we throw them into bins randomly?

Balls-in-bins – Using Randomization

- Classic balls-and-bins result [Kotz '97, Kolchin '98]
  - Tasks are balls
  - Processors are bins
- Can prove properties "with high probability" using randomization

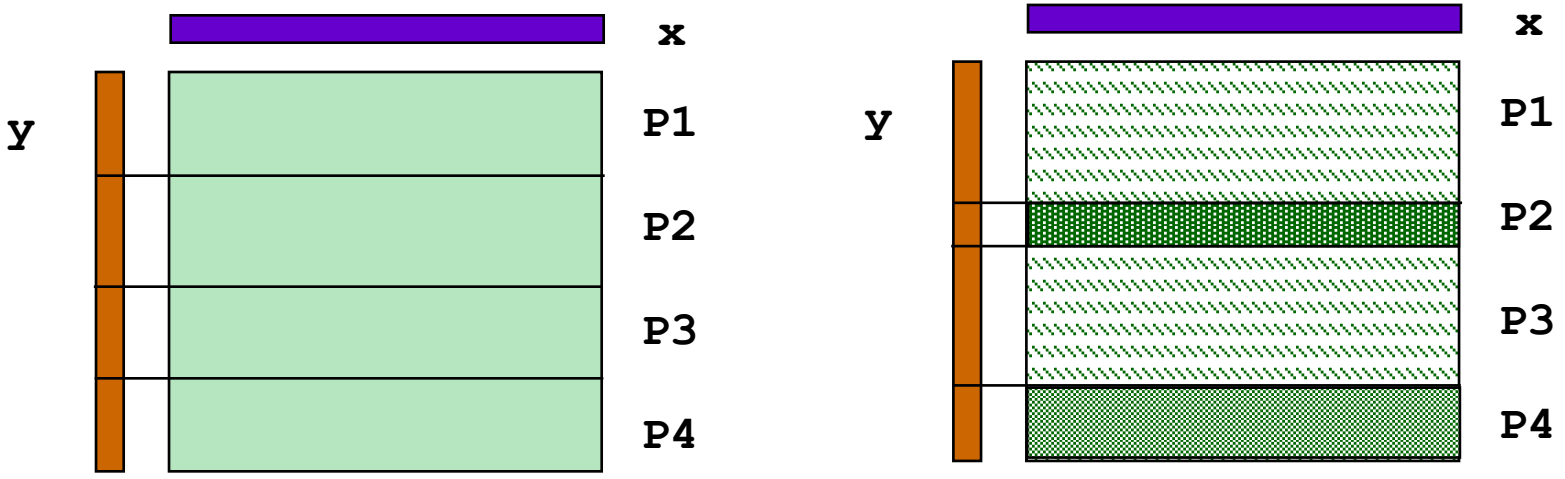


P = 5 (number of bins)

- Given n balls thrown randomly into p bins: With high probability, the maximum load in any bin is:

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

Matrix Vector Multiplication – Parallel by Rows



- Easy: Dense matrix-vector multiply
  - All rows have equal cost
- Hard: Sparse matrix-vector multiply
  - Each row may have different cost
  - If you know pattern in advance (e.g., iterative solver), divide rows by equal nonzeros, not equal number of rows
- Harder: Nonzero pattern not known, no time to estimate

Load balancing for independent tasks

- Loop over a set of independent computations

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

}
```

Easy (offline)
Statements and loops with fixed bounds, but no conditionals

Hard (offline / runtime)
Switch where each branch has "known" work, e.g., block-structure AMR

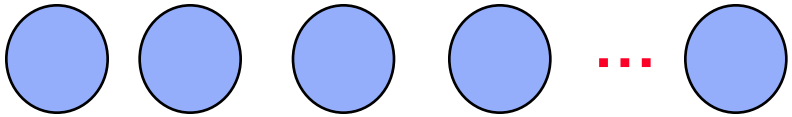
Harder (online)
Conditionals, computed loop bounds, etc.

- None of these have communication / dependencies between iterations

Task cost variability (from the application)

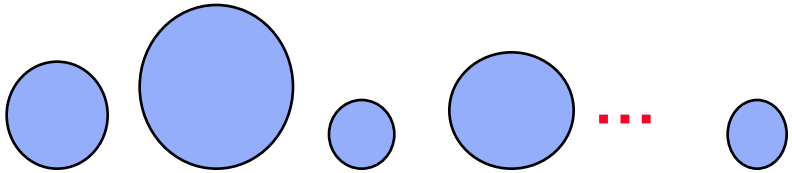
- **Easy: Equal costs, fixed count**

Regular meshes, dense matrices, direct n-body



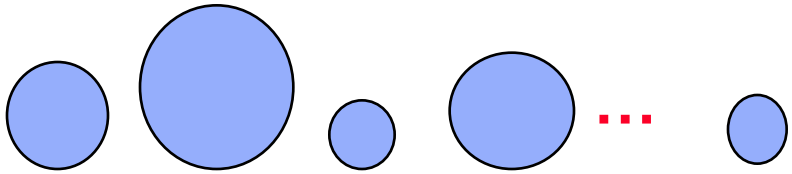
- **Hard: Tasks have different but estimable times, fixed count**

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



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

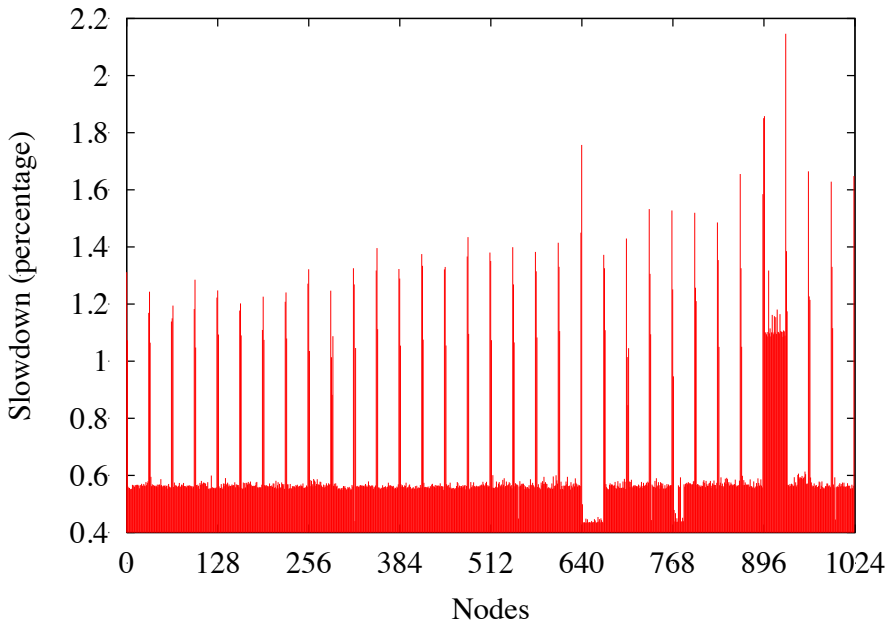
Search (UTS), irregular boundaries, subgrid physics, unpredictable machines



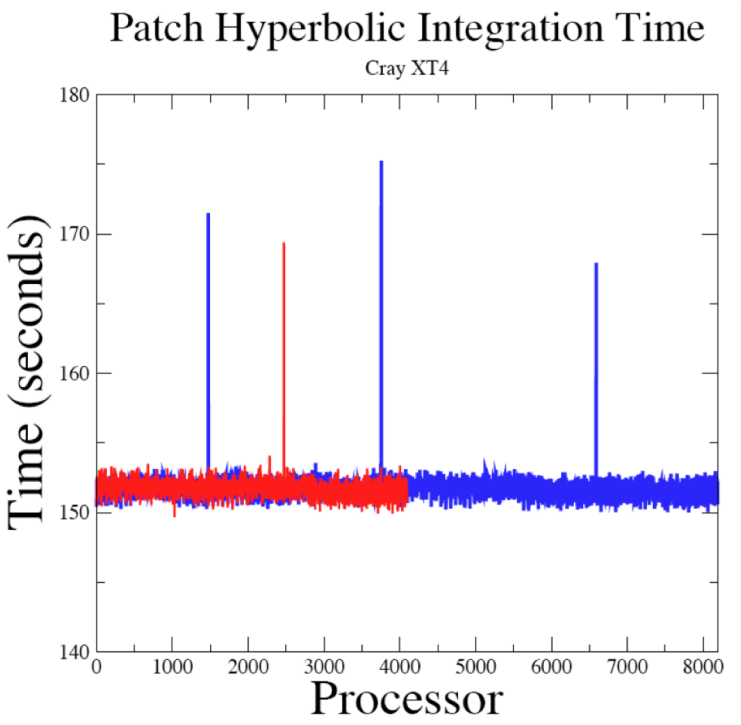
Note that good load balancing can't fix lack of parallelism.

System performance variability

Load imbalance isn't always an application level problem



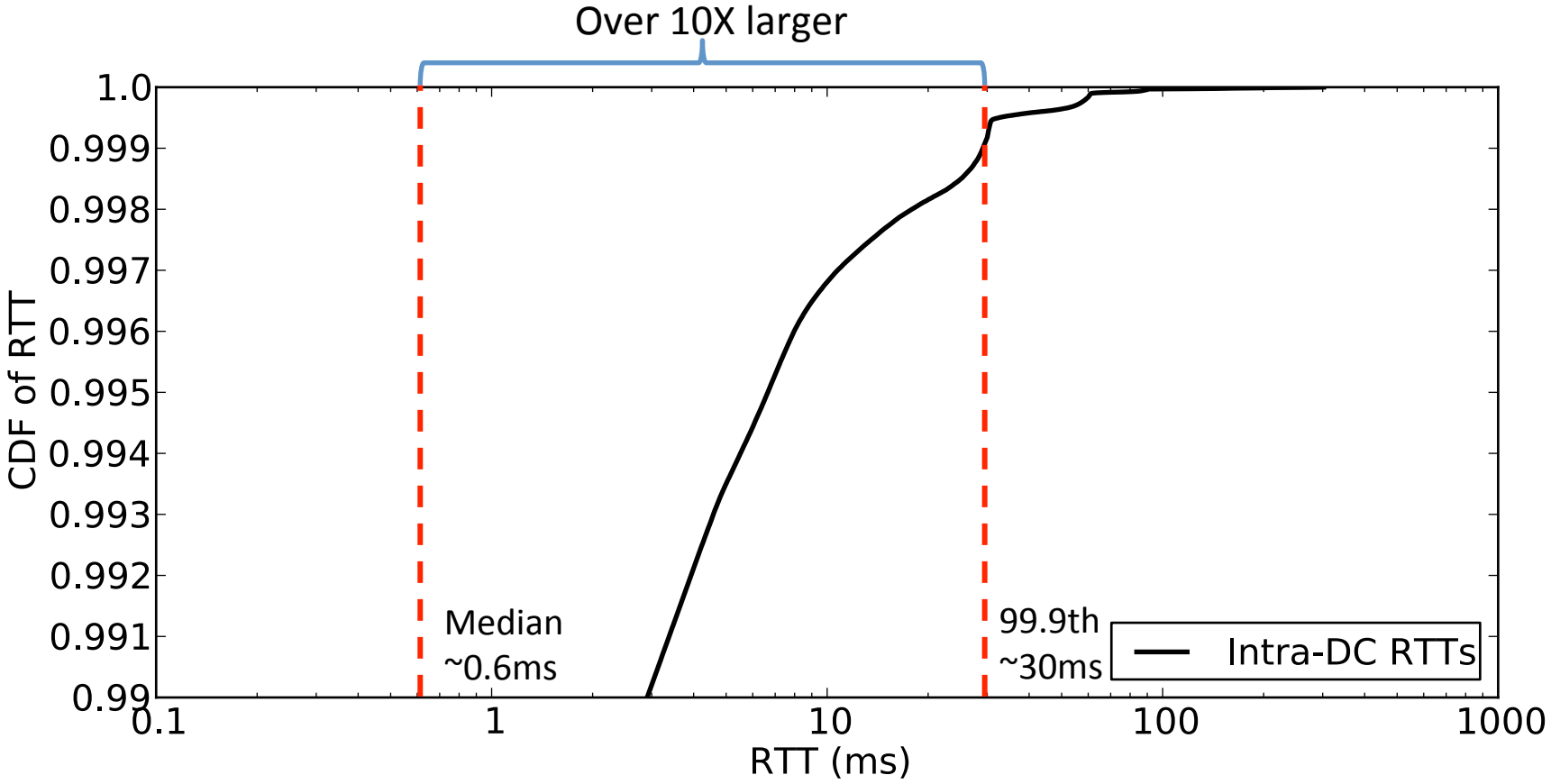
Petrini, Kerbyson, Pakin, "Case of Missing Supercomputer Performance," SC03



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

- Performance variability can come from hardware or operating system effects

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

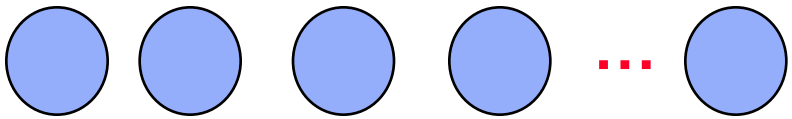


- Variability of node performance (Round-Trip-Time) in the cloud

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

Static scheduling is common

- **Easy: Equal size tasks**



Regular meshes, dense matrices, direct n-body

- Given a set of n tasks to schedule on p processor
- Given each processor n / p tasks (evenly distribute extras)
- Assumes n is known in advance (not necessarily compile time, but before the loop starts executing at runtime)

```
for (i = 0, i < n, i++) {
    Loop body B
}
```

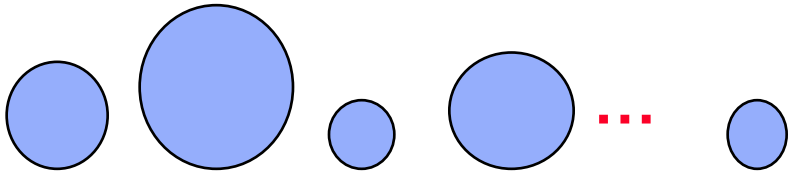
B has no cross-loop dependencies

B does not modify i or n

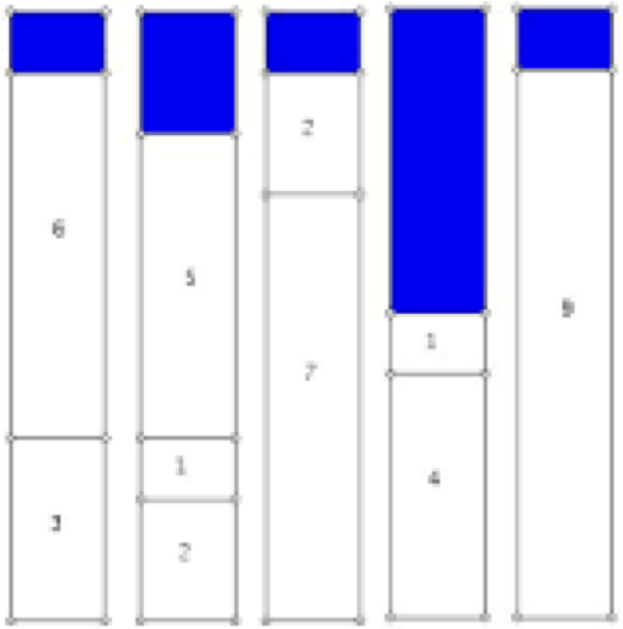
Bin-packing for statically known costs is hard

- **Hard: Tasks have different but estimable times; known count**

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



Source: Wikipedia

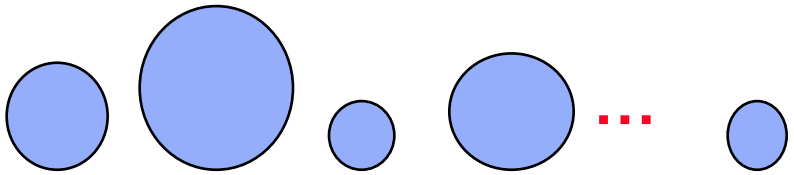


- When memory is limited (bins have fixed size) this is the classic bin-packing problem
- NP-hard (only exponential time algorithms are known)

- So in principle at runtime one can calculate a good load balance (approximate rather than optimal)

Application performance variability

- **Harder: Count or cost are not known until mid-execution**



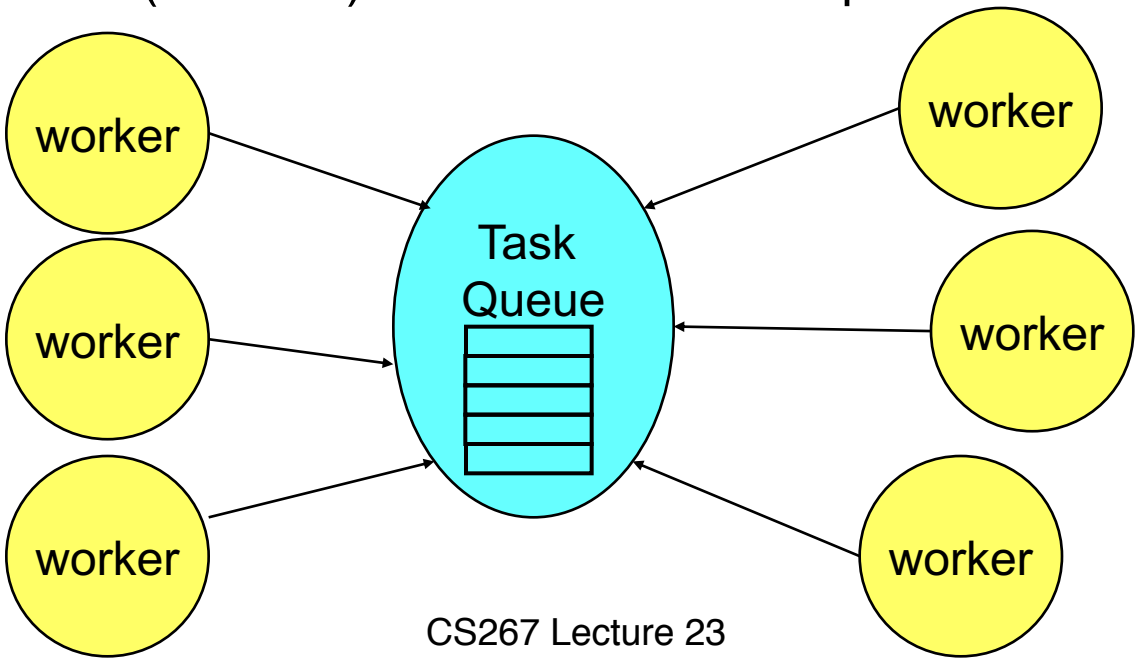
Irregular boundaries, subgrid physics, unpredictable machines

Anything with iteration-dependent branches

Self scheduling [Tang and Yew, ICPP 1986]:

Centralized queue of tasks

Processors (workers) take / add tasks to queue



Self-scheduling variations

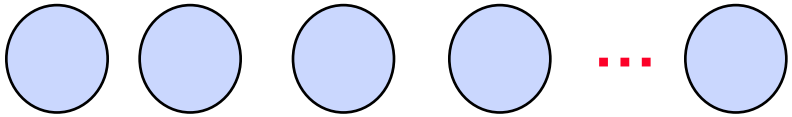
- **Self scheduling** [Tang and Yew, ICPP 1986]**:**
  - Individual tasks are scheduled
- **Chunked scheduling for size K** [Kruskal and Weiss 1985]
  - Fixed task count, cost distribution is IFR (increasing failure rate)
- **Guided scheduling** [Kuck and Polychronopolous 1987]
  - The chunk size K_i at the ith access to the task pool is given by

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

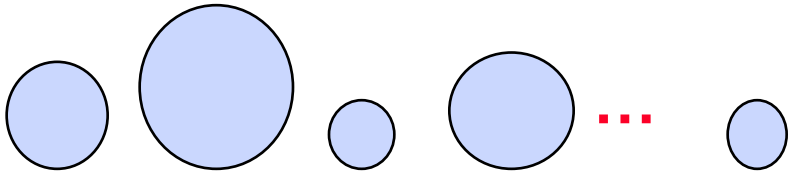
  - where R_i is the total number of tasks remaining and
  - p is the number of processors
- **Tapering** [Lucco 1994]
  - Change chunk size based on cost variance, estimated from history
- **Weighted Factoring** [Hummel, Schmit, Uma, Wein 1996]
  - For heterogeneous processors, weight chunk by speed

Summarize: Task count and cost

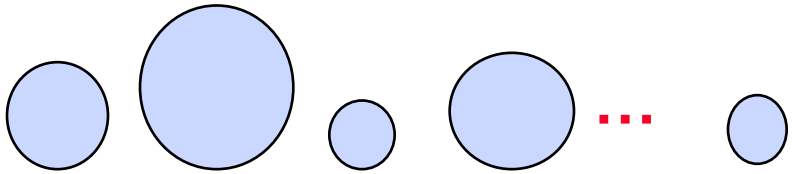
- **Static:** Equal size tasks, count known



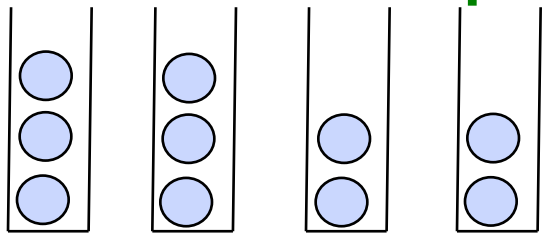
- **Semi-Static:** Task costs, count known in advance at runtime



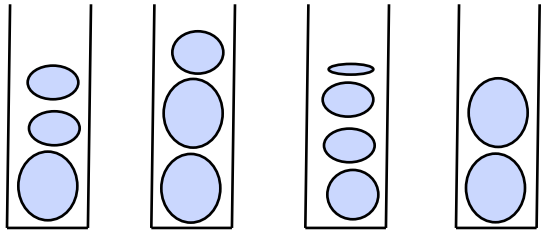
- **Dynamic:** Task costs or count not known until mid-execution



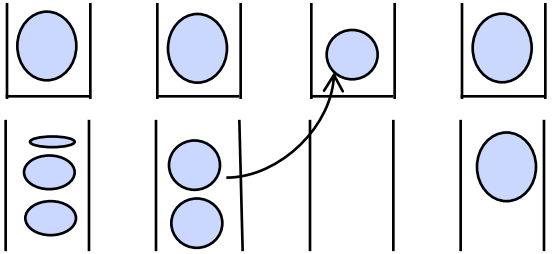
Branch-free loops



Adaptive meshes



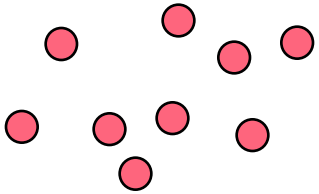
Branchy loops



Dependent sets of tasks (task trees, graphs, etc.)

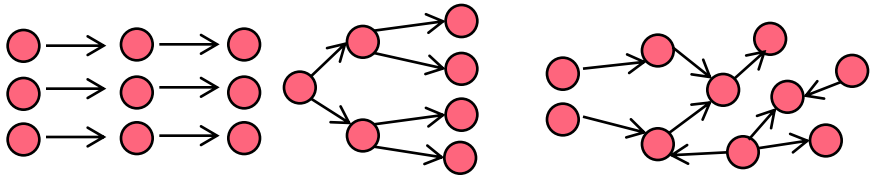
Task Dependency Spectrum

- **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: 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

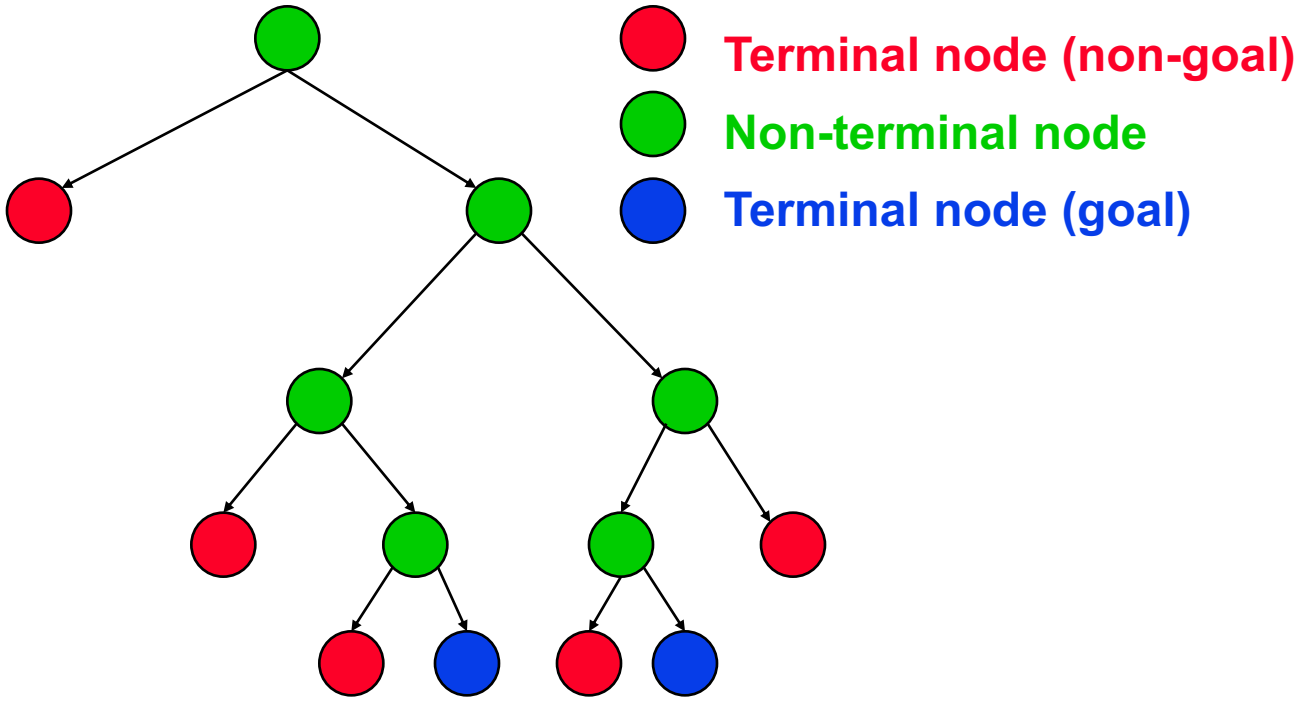
Tasks Trees: Search example

- **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 Problem: Tree Search

- In Tree Search the tree unfolds dynamically
- Number of tasks is probably unknown
- Structure of tree is unknown (if it were balanced, or some known pattern, this would be easy)



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**

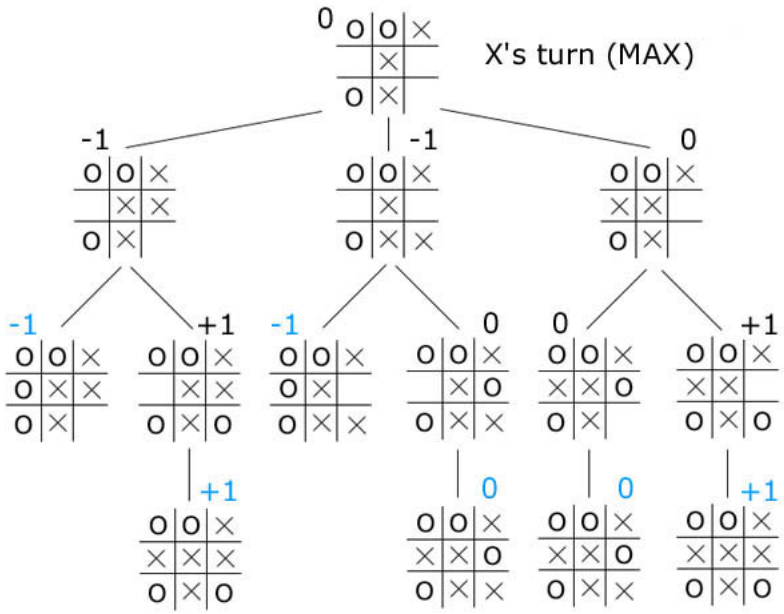
Same as on graph, but no need to mark nodes

- **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

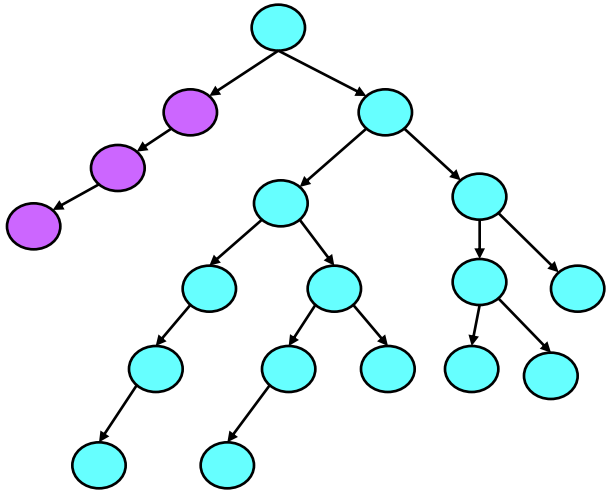
- **Simple backtracking**
  - Search to bottom, backing up to last choice if necessary
- **Branch-and-bound**
  - Keep track of best solution so far ("bound")



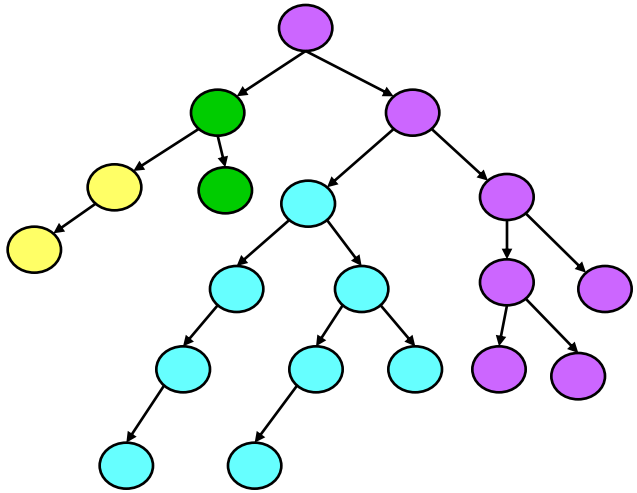
  - Cut off sub-trees that are guaranteed to be worse than bound
- **Iterative Deepening ("in between" DFS and BFS)**
  - Choose a bound d on search depth, and use DFS up to depth d
  - If no solution is found, increase d and start again
  - Can use an estimate of cost-to-solution to get bound on d

Parallel Search

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



Load balance on 2 processors



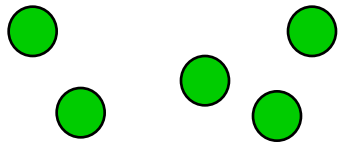
Load balance on 4 processors

Clearly a bad idea…variation of known costs but unknown count. And they arrive 'sequentially.'

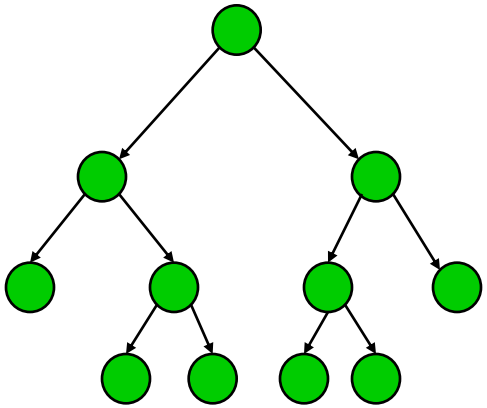
Theoretical Results (1)

Main result: Simple randomized algorithms are optimal with high probability

- Others show this for independent, equal sized 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 (equal size) 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)

Main result: Simple randomized algorithms are optimal with high probability

- **Blumofe and Leiserson [94] show this for a fixed task tree of variable cost tasks**
  - **their algorithm uses task pulling (stealing) instead of pushing, which is good for locality**
  - **I.e., when a processor becomes idle, it steals from a random processor**
  - **also have (loose) bounds on the total memory required**
  - **Used in Cilk**
  - **"better to receive than to give"**
- **Chakrabarti et al [94] show this for a dynamic tree of variable cost tasks**
  - **works for branch and bound, i.e. tree structure can depend on execution order**
  - **uses randomized pushing of tasks instead of pulling, so worse locality**

Distributed Task Queues

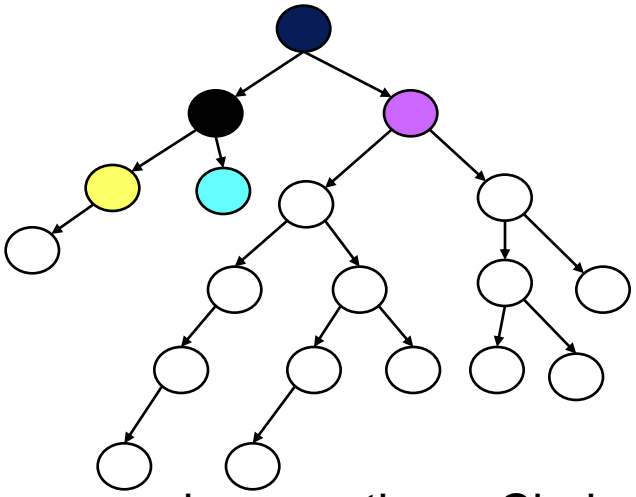
- The obvious extension of task queue to distributed memory is:
  - a distributed task queue (or "bag"), i.e., one per processor
  - Idle processors can "pull" work, or busy processors "push" work
- When are "distributed" queues a good idea?
  - Distributed memory multiprocessors
  - Often on shared 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

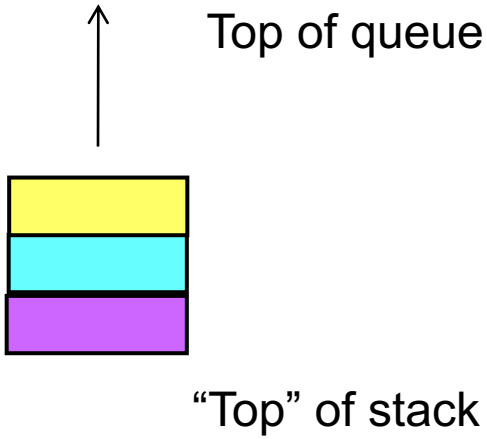
How to Select a Donor/Acceptor Processor

- **Basic techniques:**
  1. **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 $target_k = (target_k + 1) \bmod procs$**
  2. **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**
  3. **Random stealing**
     - **When a processor needs work, select a random processor and request work from it**
  4. **Random pushing**
     - **When a processor has too much work (at least two tasks), push tasks to a random processor**
- **Termination detection is nontrivial.**

Pragmatics of distributed work stealing

- **As in self-scheduling, may move a chunk, but no global information about task count (even dynamically)**
- **Which task(s)**
  - **Assume tasks are pushed to the bottom of the local queue**
  - **Execute from the bottom (most recent)**
  - **Send tasks from the top (oldest)**
  - **May be able to do better with information about task costs**

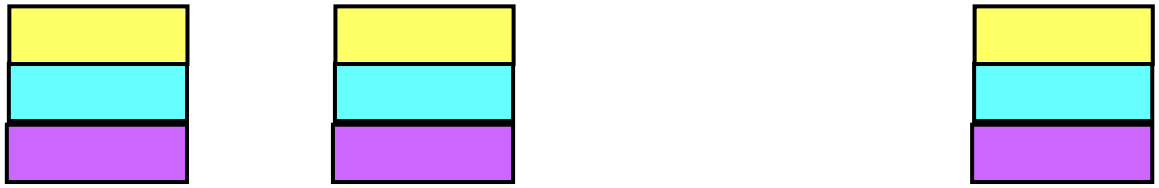




E.g., Multipol library and many others, Chakrabarti et al 1995, Chih-Po Wen PhD 1995

Randomized load balancing

- Want to avoid bottlenecks of shared queue
- Especially in distributed memory (but even in shared)
- So Self-scheduling, Chunked SS, GSS 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

Randomized work pushing

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

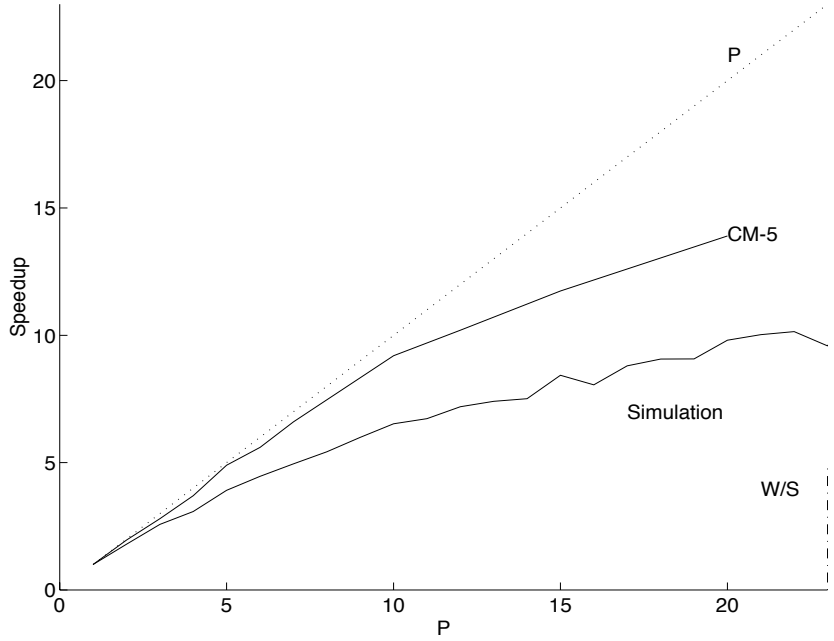


Figure 3: Performance on the Gröbner basis example. Speedups were averaged over 20 runs.

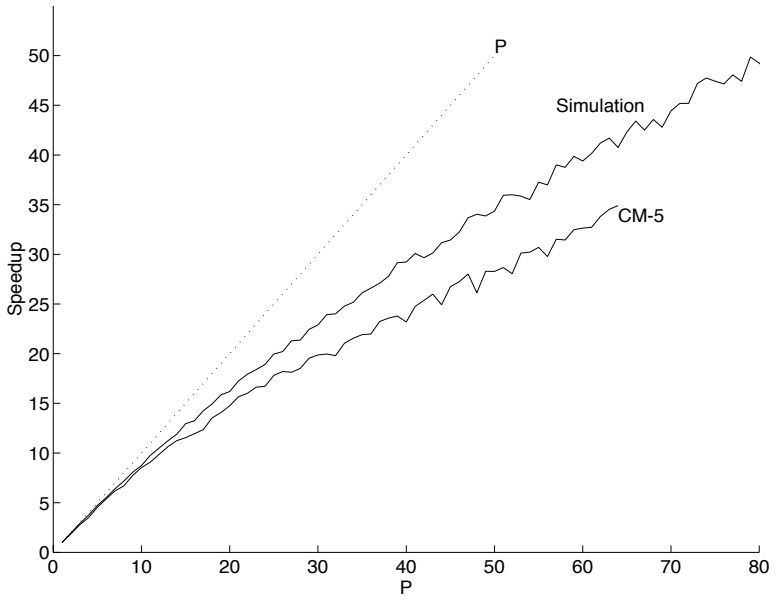


Figure 4: Performance on the eigenvalue example. Speedups were averaged over 20 runs.

Chakrabarti, Ranade, Yelick 1994

Cilk: A Language with Built-in Load balancing

A C language for programming dynamic multithreaded applications on shared-memory multiprocessors.

CILK (Leiserson et al) (supertech.lcs.mit.edu/cilk)

- **Created startup company called CilkArts**
- **Acquired by Intel, being reinvigorated..**

Example applications:

- virus shell assembly
- graphics rendering
- *n*-body simulation
- heuristic search
- dense and sparse matrix computations
- friction-stir welding simulation
- artificial evolution

Fibonacci Example: Creating Parallelism

```
int fib (int n) {
if (n<2) return (n);
  else {
    int x,y;
    x = fib(n-1);
    y = fib(n-2);
    return (x+y);
  }
}
```

C elision

Cilk code

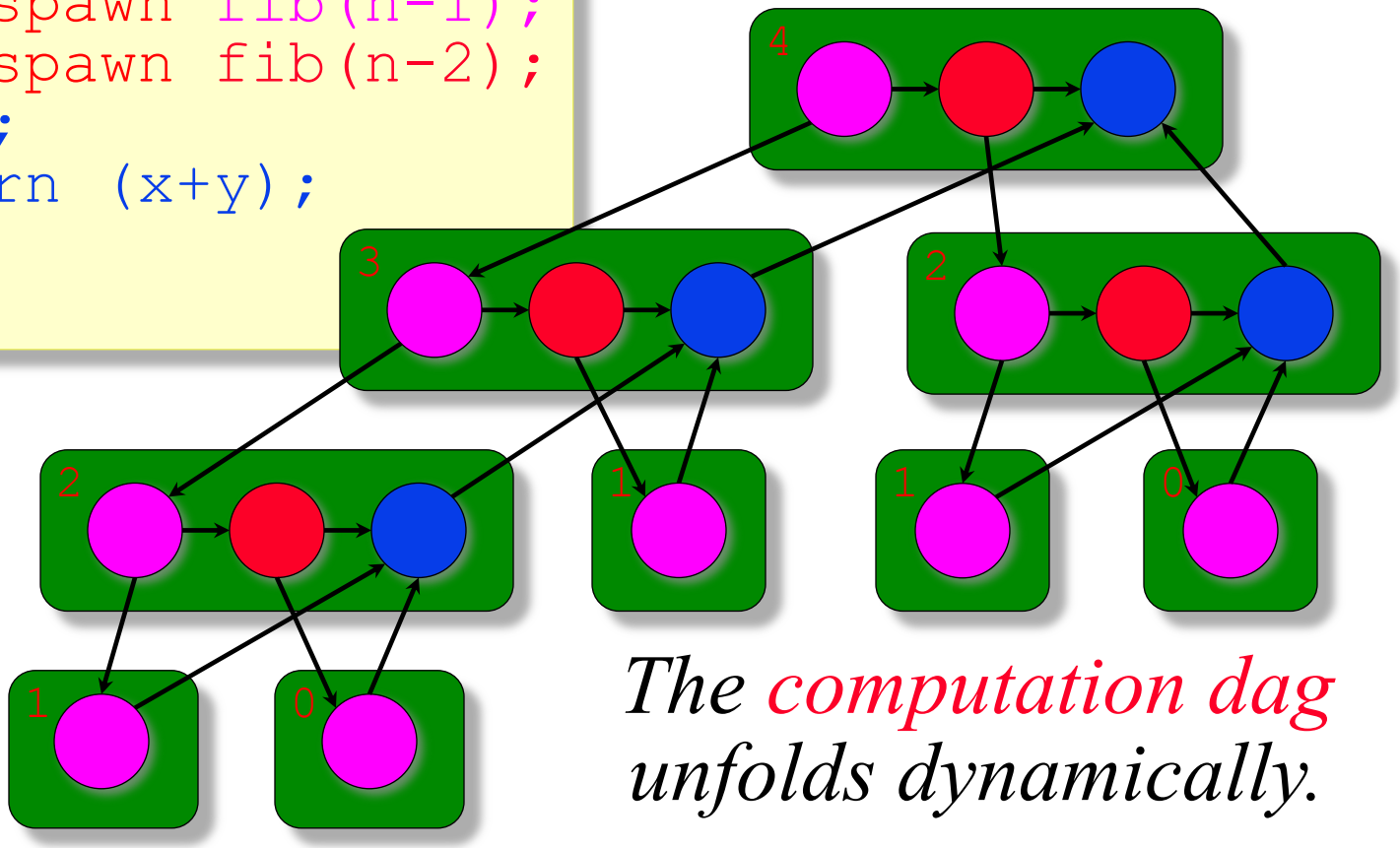
```
cilk int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x+y);
  }
}
```

Cilk is a *faithful* extension of C. A Cilk program's *serial elision* is always a legal implementation of Cilk semantics. Cilk provides *no* new data types.

Dynamic Multithreading

```
cilk int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x+y);
  }
}
```

Example: `fib(4)`



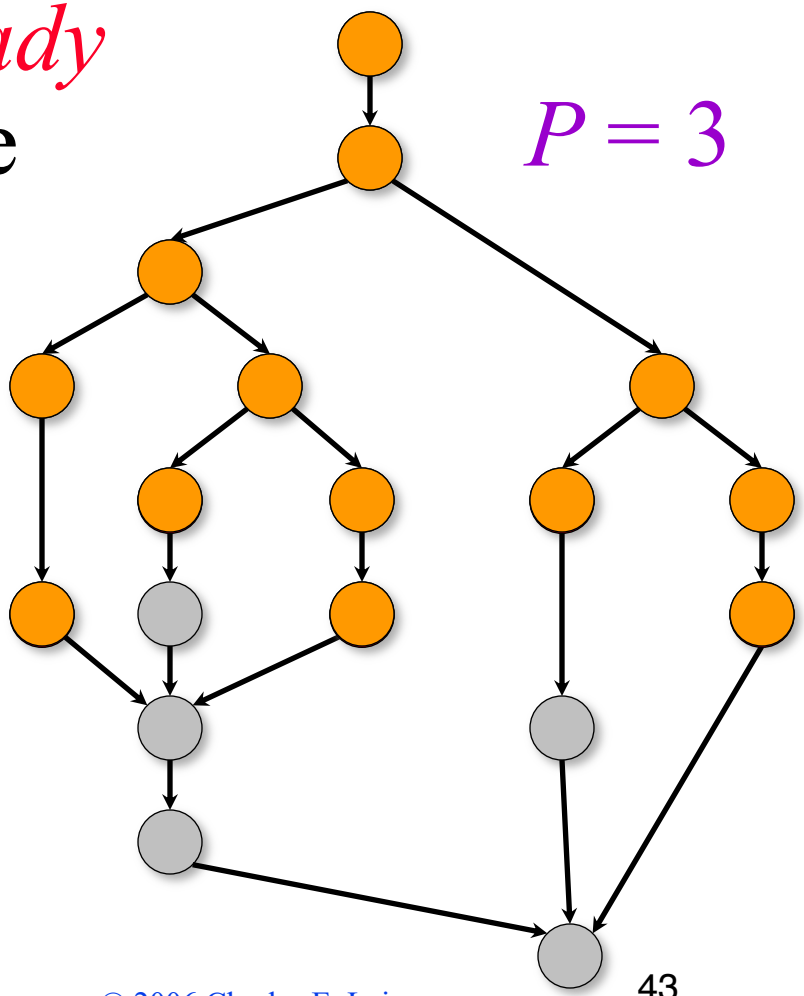
processors are virtualized

The computation dag unfolds dynamically.

Greedy Scheduling

IDEA: Do as much as possible on every step.

Definition: A thread is *ready* if all its predecessors have *executed*.



Complete step

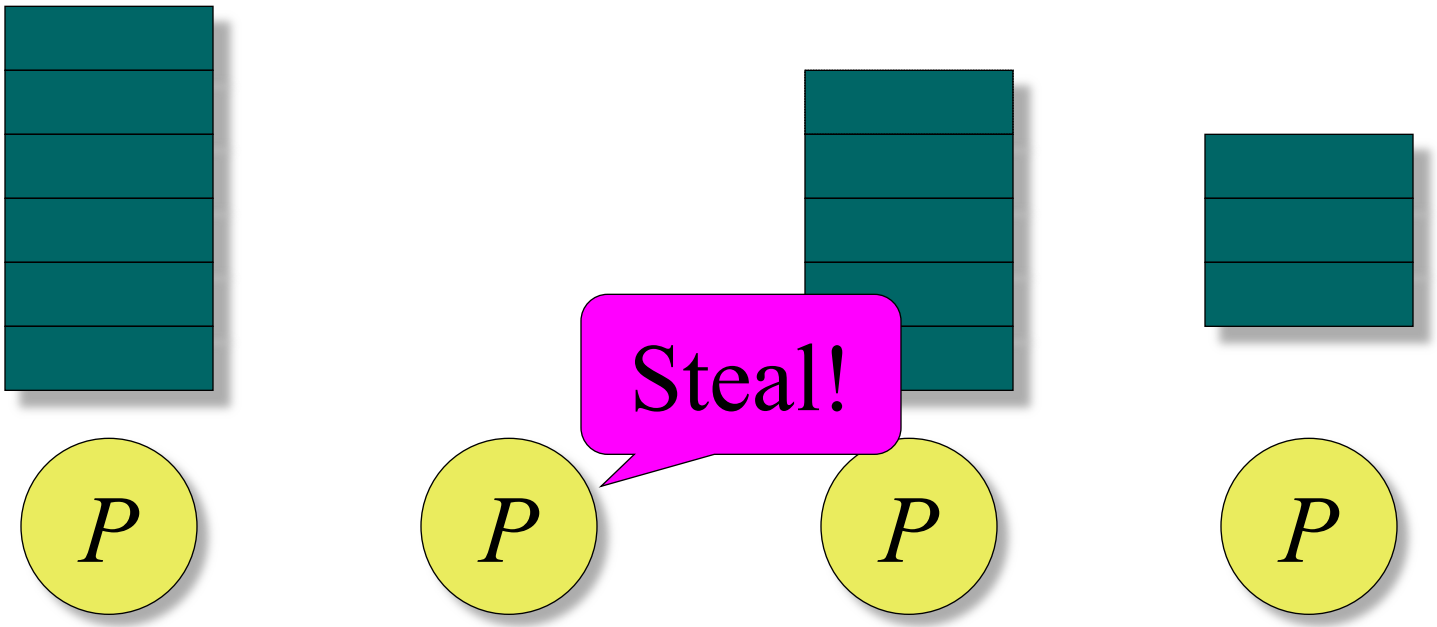
- $\geq P$ threads ready.
- Run any P.

Incomplete step

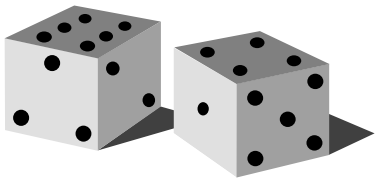
- $< P$ threads ready.
- Run all of them.

Cilk's Work-Stealing Scheduler

Each processor maintains a *work deque* of ready threads, and it manipulates the bottom of the deque like a stack.



When a processor runs out of work, it *steals* a thread from the top of a *random* victim's deque.



Performance of Work-Stealing

Theorem: Cilk's work-stealing scheduler achieves an expected running time of

$$T_P \le T_1/P + O(T_\infty)$$

on P processors.

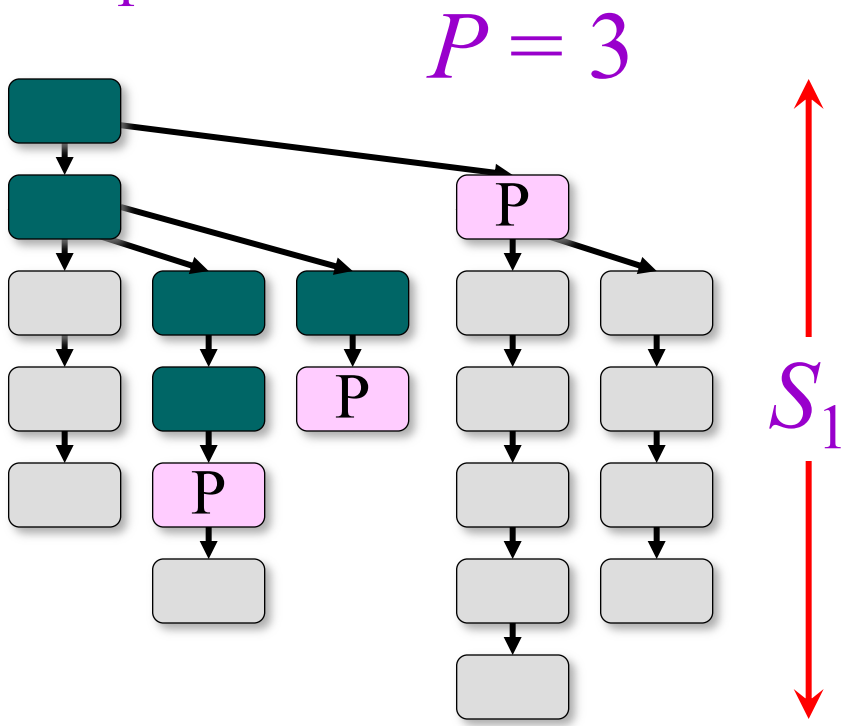
Pseudoproof. A processor is either *working* or *stealing*. The total time all processors spend working is T_1. Each steal has a $1/P$ chance of reducing the span by 1. Thus, the expected cost of all steals is $O(PT_\infty)$. Since there are P processors, the expected time is

$$(T_1 + O(PT_\infty))/P = T_1/P + O(T_\infty) \ .$$ ■

Space Bounds

Theorem. Let S_1 be the stack space required by a serial execution of a Cilk program. Then, the space required by a P-processor execution is at most $S_P = PS_1$.

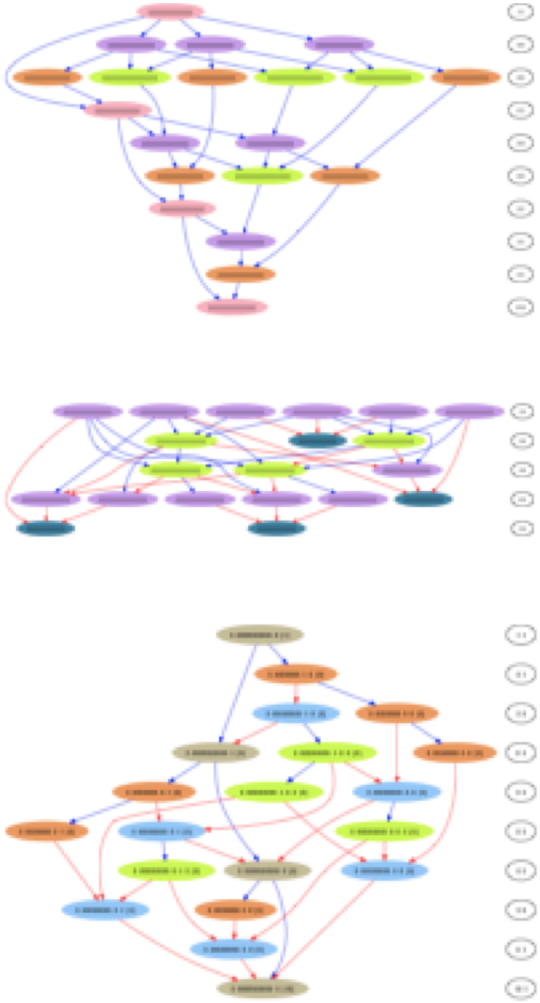
Proof (by induction). The work-stealing algorithm maintains the *busy-leaves property*: *every extant procedure frame with no extant descendents has a processor working on it.*



Diffusion-Based Load Balancing

- **In the randomized schemes, the machine is treated as fully-connected**. [Cybenko, 1989]
- **Diffusion-based load balancing takes topology into account**
  - **Send some extra work to a few nearby processors**
    - **Average work with nearby neighbors**
    - **Analogy to diffusion (Jacobi for solving Poisson equation)**
  - **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**
- **See Ghosh et al, SPAA96 for a second order diffusive load balancing algorithm**
  - **takes into account amount of work sent last time**
  - **avoids some oscillation of first order schemes**

DAG scheduling, e.g., Cholesky Inversion



Pipelined: (3 (n/b) +6) = 18

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

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), …

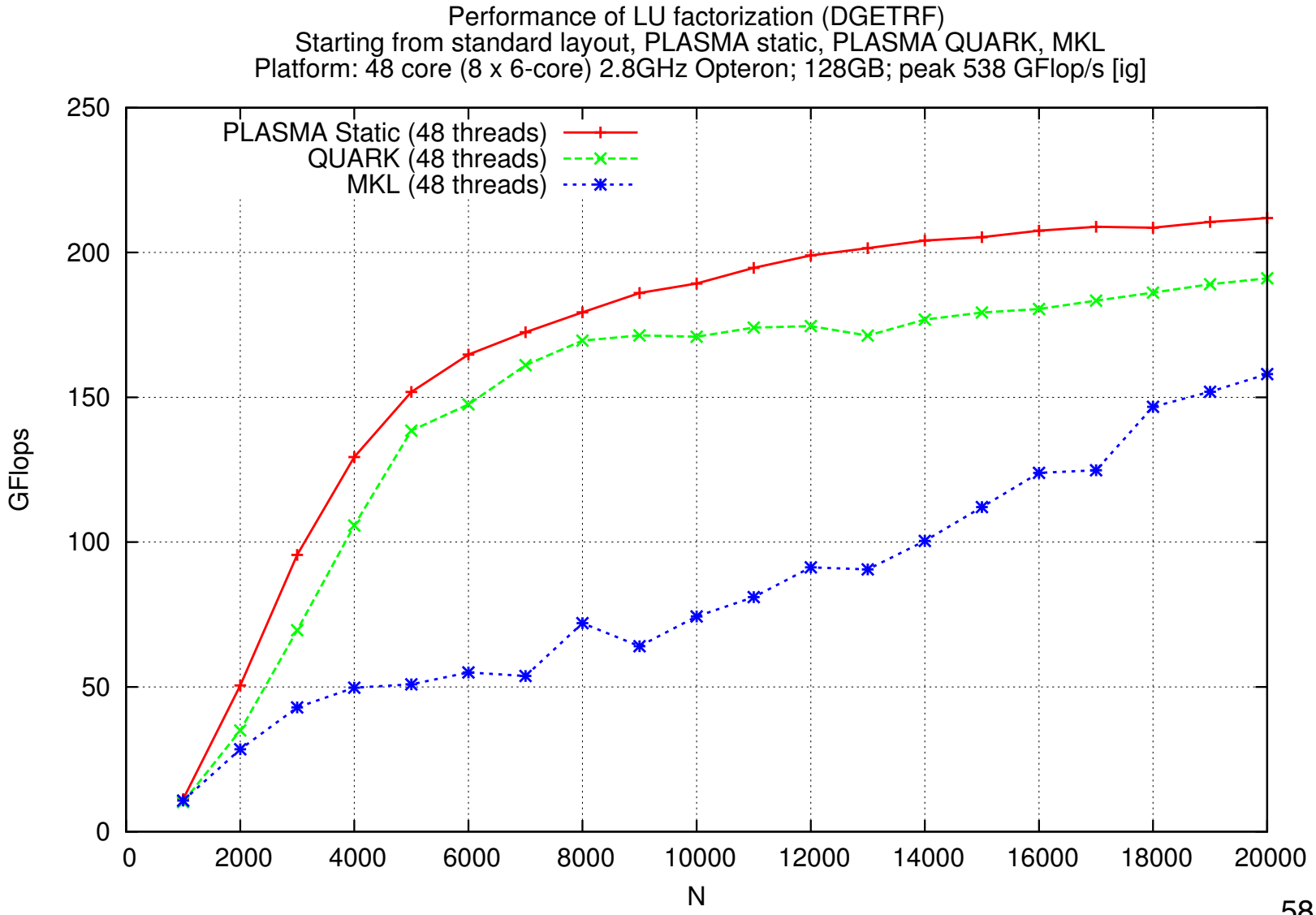
Scalability of DAG Schedulers

- 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

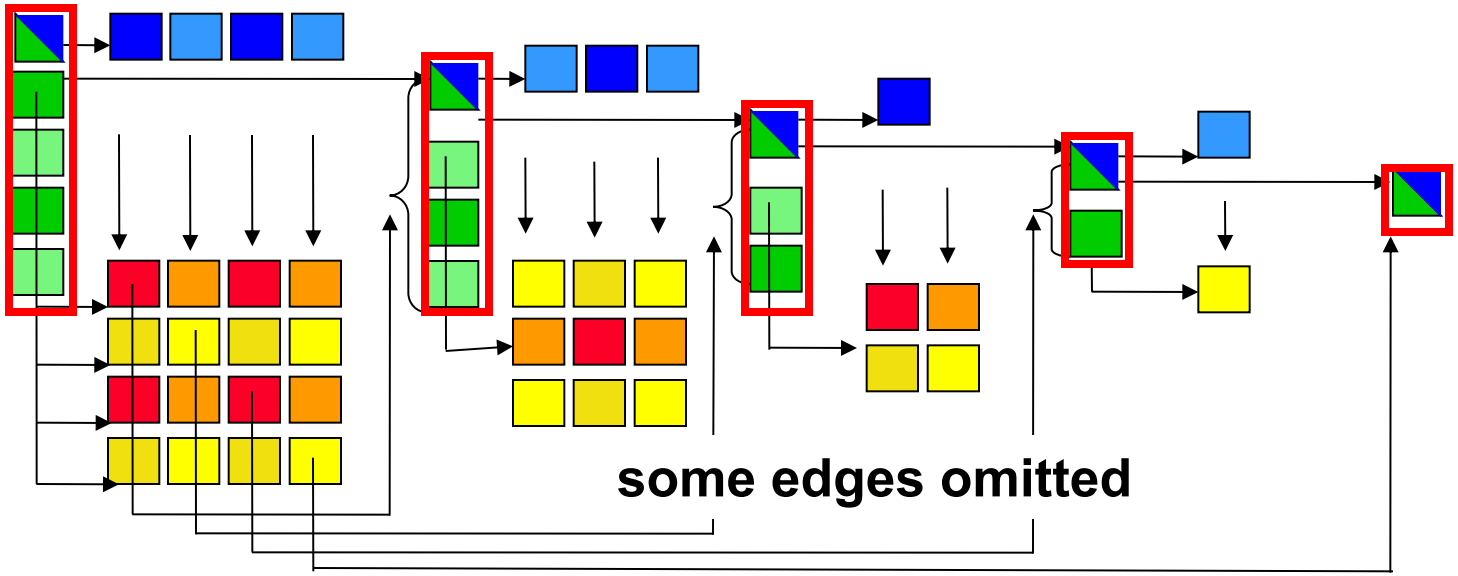
Performance – 48 core



QUARK is approx 10% less than static; MKL scales up more slowly.

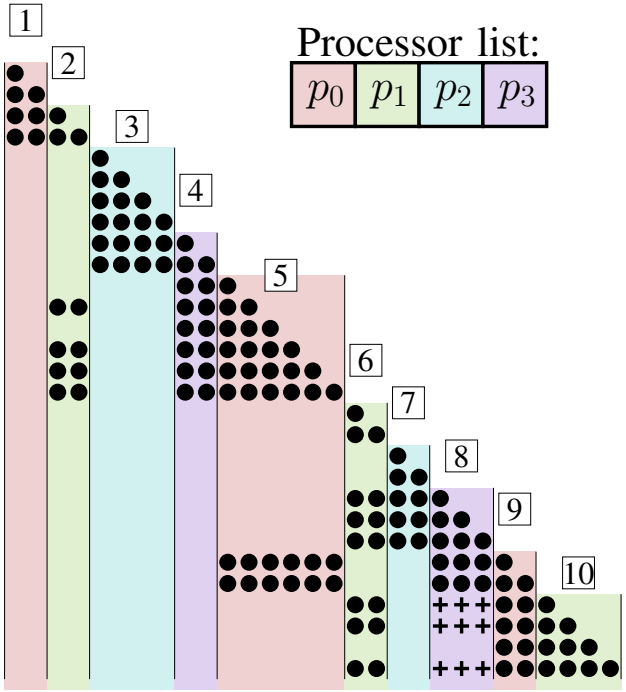
Event Driven LU in UPC

- 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

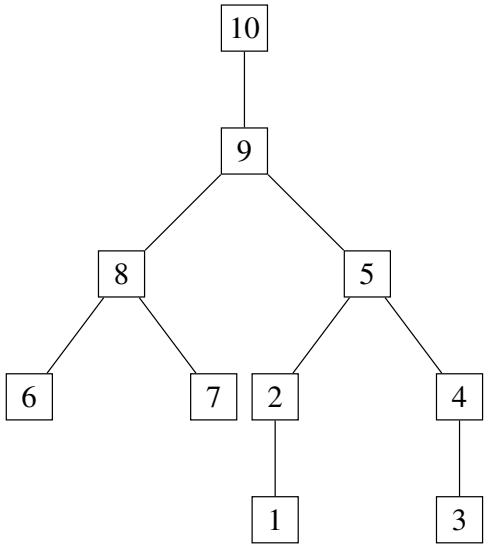


"Multithreading and 1-sided communication in LU…" Husbands and Yelick, SC07.

symPACK: Sparse Cholesky



(a) Structure of Cholesky factor $\mathbf{L}$ (



) Supernodal elimination tree of matrix $\mathbf{A}$

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

Matthias Jacquelin, Yili Zheng, Esmond Ng, Katherine Yelick

symPACK: Sparse Cholesky

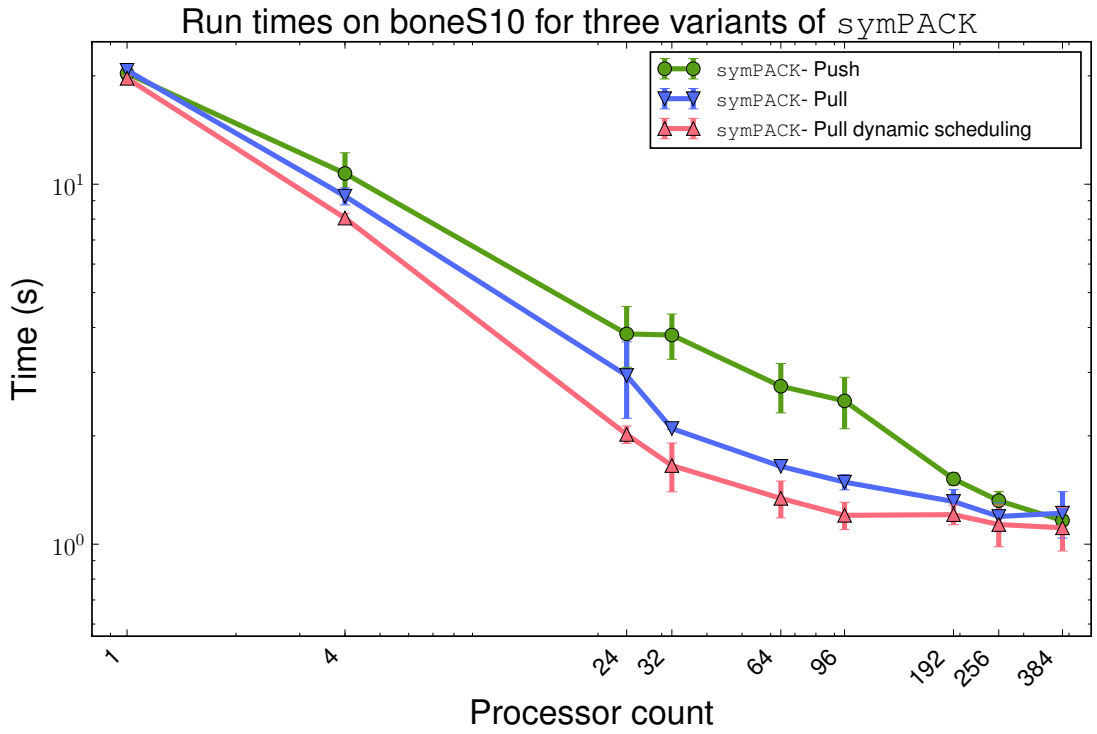


Figure 7: Impact of communication strategy and scheduling on symPACK performance

- Scalability of symPACK on Cray XC30 (Edison)
  - Comparable or better than best solvers (evaluation in progress)
  - Notoriously hard parallelism problem

Matthias Jacquelin, Yili Zheng, Esmond Ng, Katherine Yelick

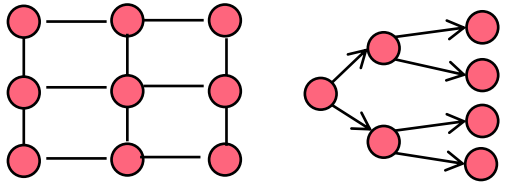
Tasks with Communication

The previous work had tasks with temporal locality
(reuse within tasks, and possibly parent/child in tree)

What about spatial locality expressed as
communication between tasks?

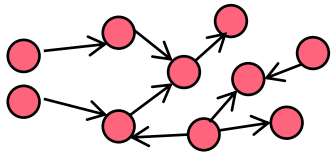
Task Communication Spectrum

- **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

Review of Graph Partitioning – static case

- Partition G(N,E) so that
  - $N = N_1 \cup \ldots \cup N_p$, with each $|N_i| \sim |N|/p$
  - As few edges connecting different N_i and N_k as possible
- If N = {tasks}, each unit cost, edge e=(i,j) means task i has to communicate with task j, then partitioning means
  - balancing the load, i.e. each $|N_i| \sim |N|/p$
  - minimizing communication volume
- Optimal graph partitioning is NP complete, so we use heuristics (see earlier lectures)
  - Spectral, Kernighan-Lin, Multilevel …
- Good software available
  - (Par)METIS, Scotch, Zoltan, …
- Speed of partitioner trades off with quality of partition
  - Better load balance costs more; may or may not be worth it
- Need to know tasks, communication pattern before starting
  - What if you don't? Can redo partitioning, but not frequently

Charm++

Load balancing based on Over-decomposition

- Context: “Iterative Applications”
  - Repeatedly execute similar set of tasks
- Idea: decompose work/data into chunks (*chares* in Charm++) , and migrate chares for balancing loads
  - Chares can be split or merged, but typically less frequently (or unnecessary in many cases)
- How to predict the computational load and communication between objects?
  - Could rely on user-provided info, or based on simple metrics
    - (e.g. number of elements)
  - Alternative: *principle of persistence*
    - Statistics change slowly, can rebalance occasionally
- Software, documentation at charm.cs.uiuc.edu
  - Many applications: NAMD, LeanMD, OpenAtom, ChaNGa, ...

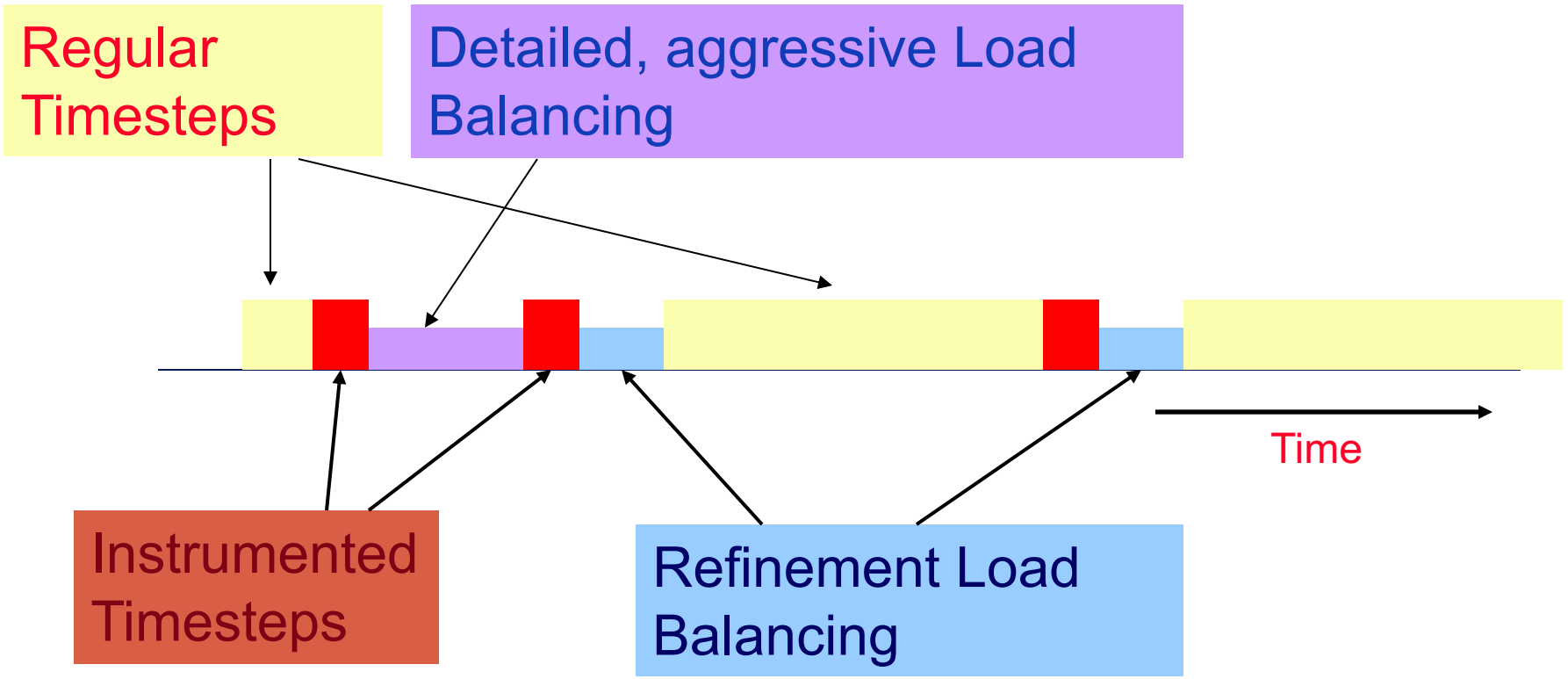
Source: Laxmikant Kale

Measurement Based Load Balancing in Charm++

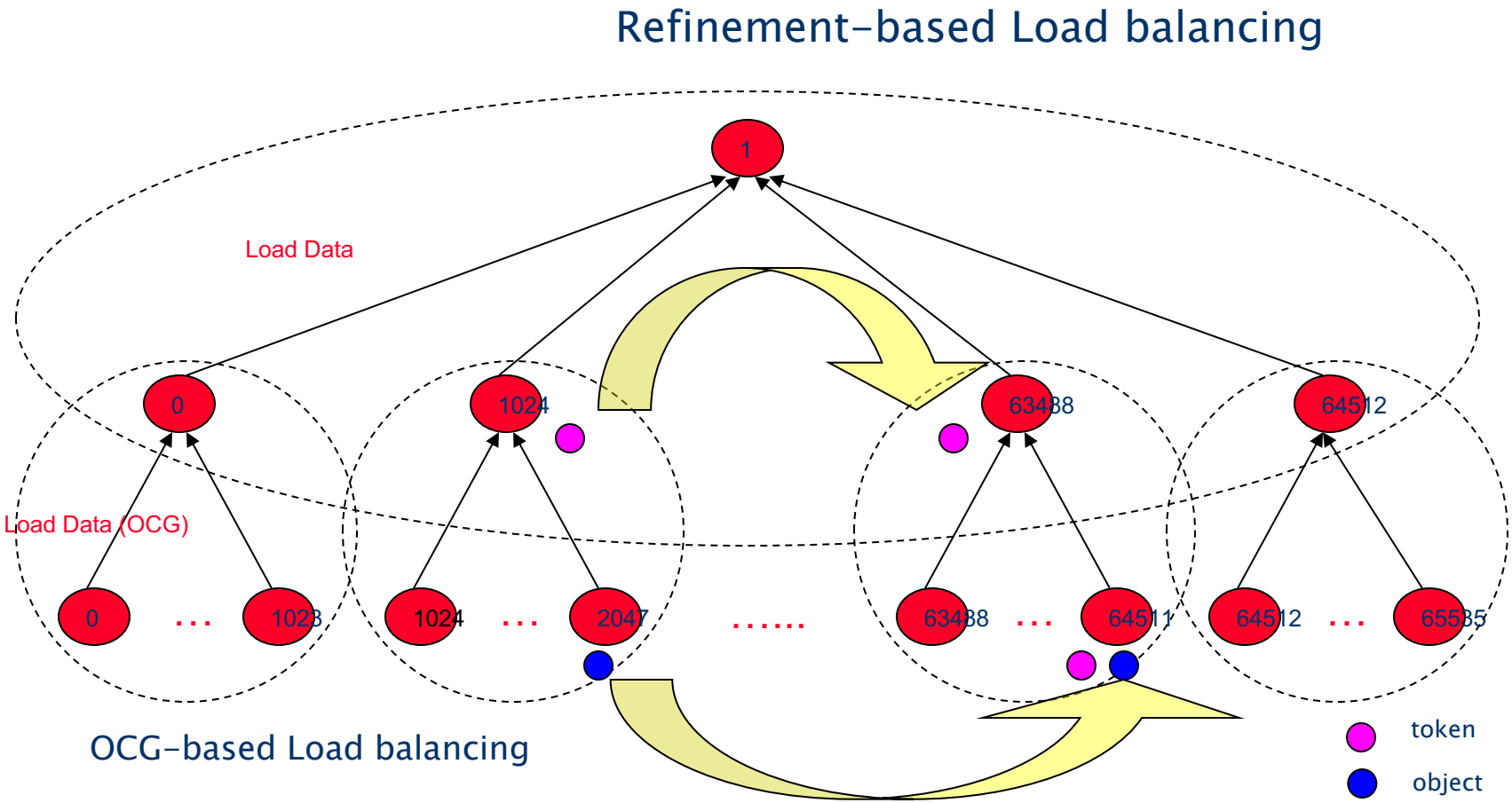
- Principle of persistence (A Heuristic)
  - *Object communication patterns and computational loads tend to persist over time, so recent past good predictor of future*
  - In spite of dynamic behavior
    - Abrupt but infrequent changes
    - Slow and small changes
  - Only a heuristic, but applies to many applications
- Measurement based load balancing
  - Runtime system (in Charm++) schedules objects and mediates communication between them, so can measure load
  - Use the instrumented data-base periodically to make new decisions, and migrate objects accordingly
- Charm++ provides a suite of strategies, and plug-in capability for user-defined ones
  - Also, a meta-balancer for deciding how often to balance, and what type of strategy to use

Source: Laxmikant Kale

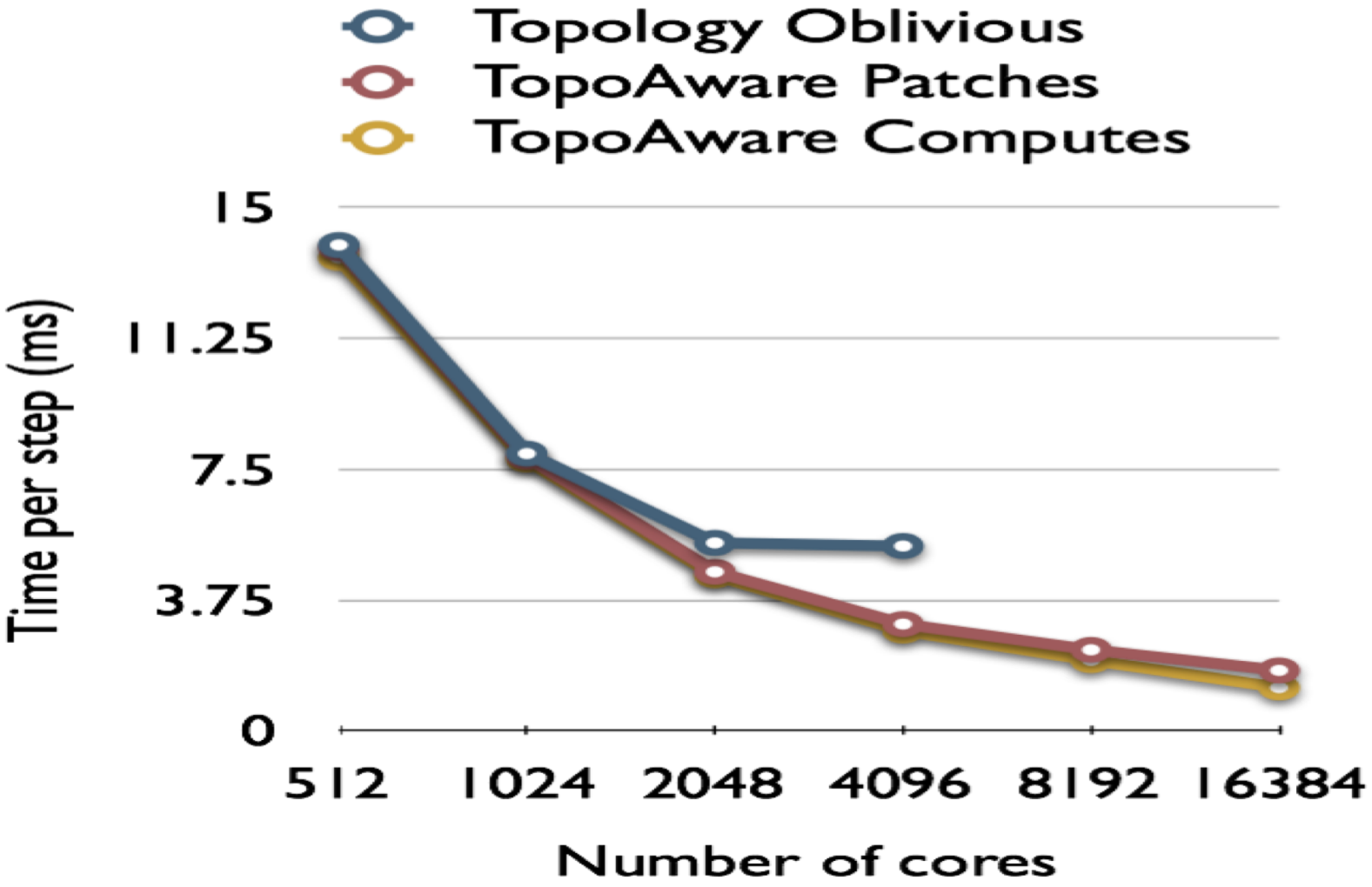
Load Balancing Steps



Charm++ Hierarchical Load Balancer Scheme



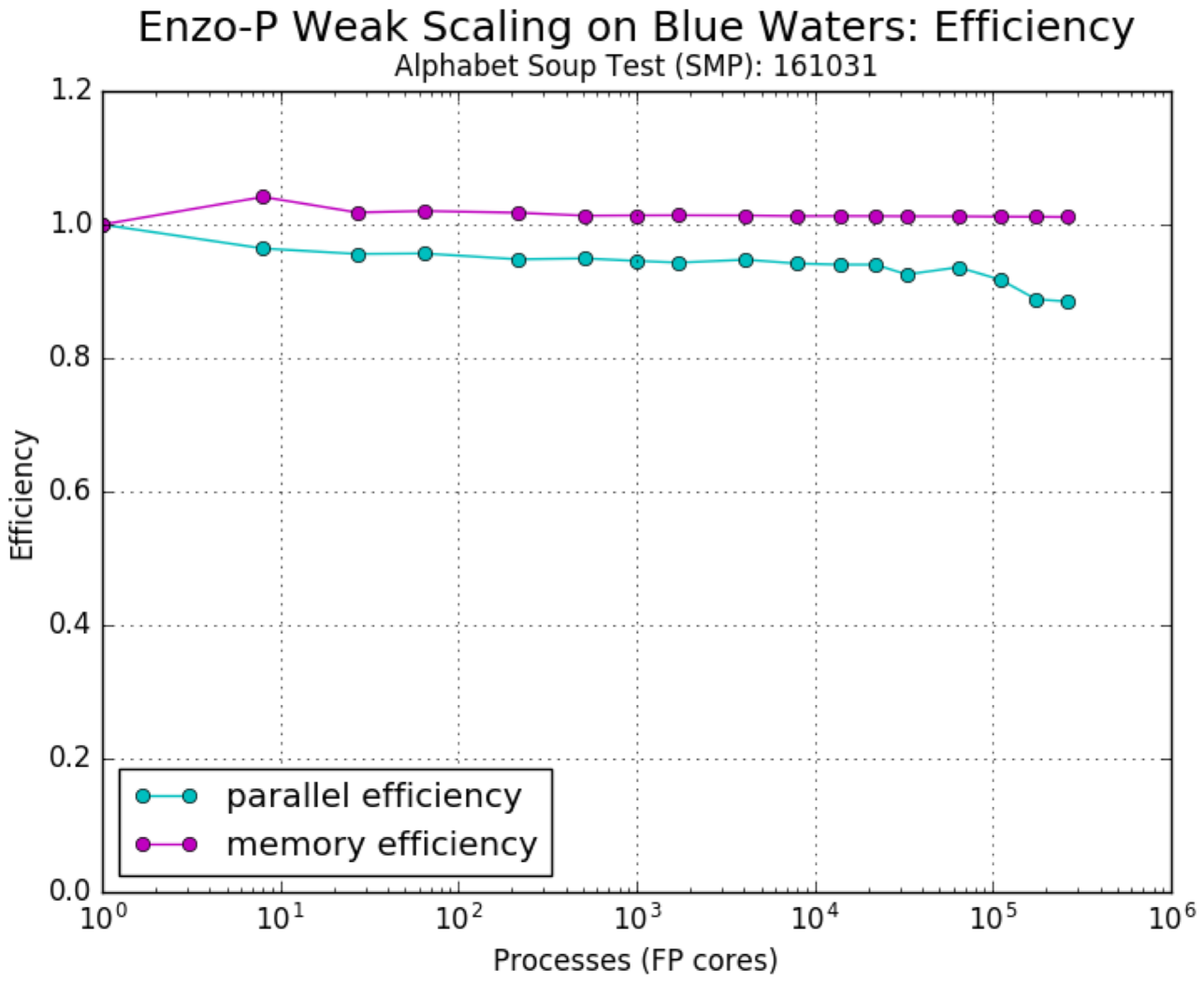
Efficacy of Topology aware load balancing



NAMD biomolecular simulation
running on BG/P

Enzo astrophysics code using CHARM++

From Mike Norman et al, UCSD



Summary and Take-Home Messages

- There is a fundamental 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)
- Open question: will future architectures lead to so much load imbalance that even "regular" problems need dynamic balancing?

Extra Slides

Spectrum of Solutions

A key question is when certain information about the load balancing problem is known.

Leads to a spectrum of solutions:

- **Static scheduling.** All information is available to scheduling algorithm, which runs before any real computation starts.
  - Off-line algorithms, eg graph partitioning, DAG scheduling
  - Still might use dynamic approach if too much information
- **Semi-static scheduling.** Information may be known at program startup, or the beginning of each timestep, or at other well-defined points. Offline algorithms may be used even though the problem is dynamic.
  - eg Kernighan-Lin, as in Zoltan
- **Dynamic scheduling.** Information is not known until mid-execution.
  - On-line algorithms – main topic today

Dynamic Load Balancing

- Motivation for dynamic load balancing
  - Search algorithms as driving example
- Centralized load balancing
  - Overview
  - Special case for scheduling independent loop iterations
  - Makes most sense in shared memory environment
  - Hard to scale to large numbers of processors
- Distributed load balancing
  - Overview – randomization often used
  - Engineering
  - Theoretical results

Load Balancing Overview

Load balancing differs with properties of the tasks

- **Tasks costs**
  - Do all tasks have equal costs?
  - If not, when are the costs known?
    - Before starting, when task created, or only when task ends
- **Task dependencies**
  - Can all tasks be run in any order (including parallel)?
  - If not, when are the dependencies known?
    - Before starting, when task created, or only when task ends
    - One task may prematurely end another task (eg search)
- **Locality (may tradeoff with load balance)**
  - Is it important for some tasks to be scheduled on the same processor (or nearby) to reduce communication cost?
  - When is the information about communication known?
- **If properties known only when tasks end**
  - Are statistics fixed, change slowly, change abruptly?

Mixed Parallelism

As another variation, consider a problem with 2 levels of parallelism

- course-grained task parallelism
  - good when many tasks, bad if few
- fine-grained data parallelism
  - good when much parallelism within a task, bad if little

Appears in:

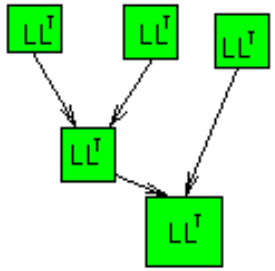
- Adaptive mesh refinement
- Discrete event simulation, e.g., circuit simulation
- Database query processing
- Sparse matrix direct solvers

How do we schedule both kinds of parallelism well?

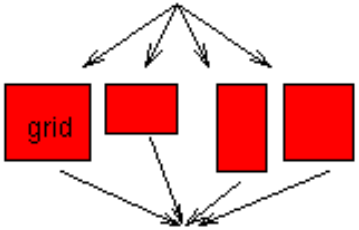
Mixed Parallelism Strategies

Many applications have course-grained task parallelism and fine-grained data parallelism

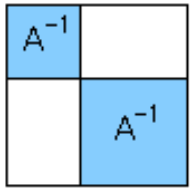
sparse cholesky



adaptive mesh refinement



sign function



blocks are data-parallel tasks within a task parallel execution

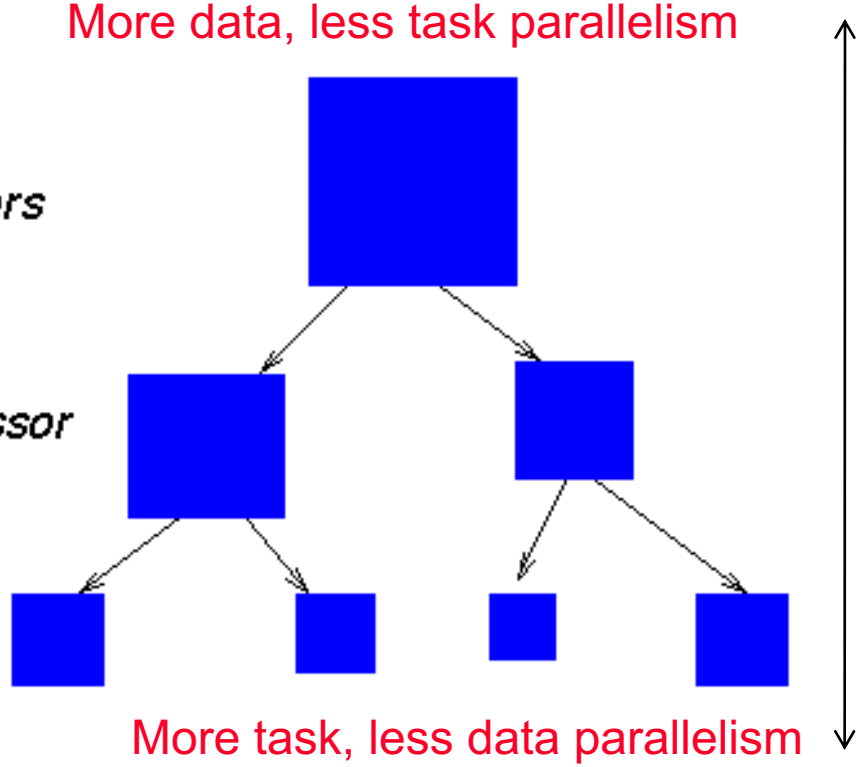
Questions:

Should the execution use only data parallelism, only task parallelism, or a mixture?

What is the relative benefit?

What is a good scheduling algorithm?

Which Strategy to Use



Pure data parallelism

spread each block over all processors

Pure task parallelism

assign each block to a single processor

Switched parallelism

at some level, go from data to task

Mixed parallelism

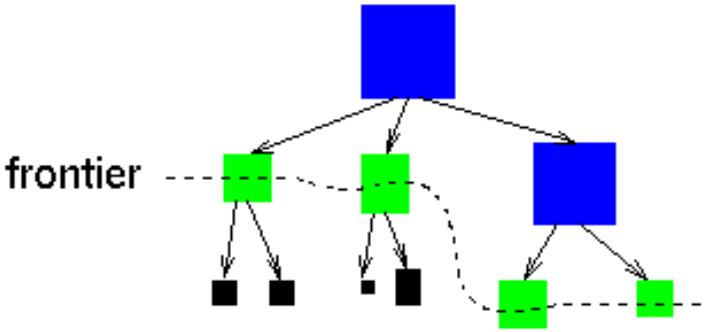
spread blocks on subsets of processors

Modeling shows that switch parallelism gets almost all the benefit of mixed.

And easier to implement

Switch Parallelism: A Special Case

A Prefix–Suffix Heuristic



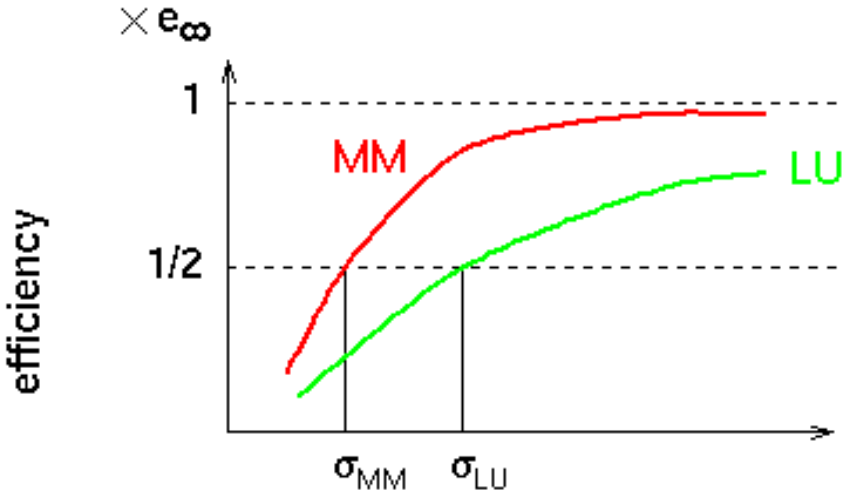
* Sort the current frontier of tasks to be executed: N1 > N2 > N3 > ... > Nl
* Assume cost(Ni, P) is known
* Restrict decision to executing
  - a prefix of the largest tasks using data parallelism
  - and the remaining suffix of tasks using task parallelism
* Compare all prefix choices in linear time

See Soumen Chakrabarti's 1996 UCB EECS PhD Thesis
See also J. Parallel & Distributed Comp, v. 47, pp 168-184, 1997

Extra Slides

Simple Performance Model for Data Parallelism

Observation: the efficiency of a data parallel algorithm depends on the problem size per processor, N/P, for sufficiently large N.

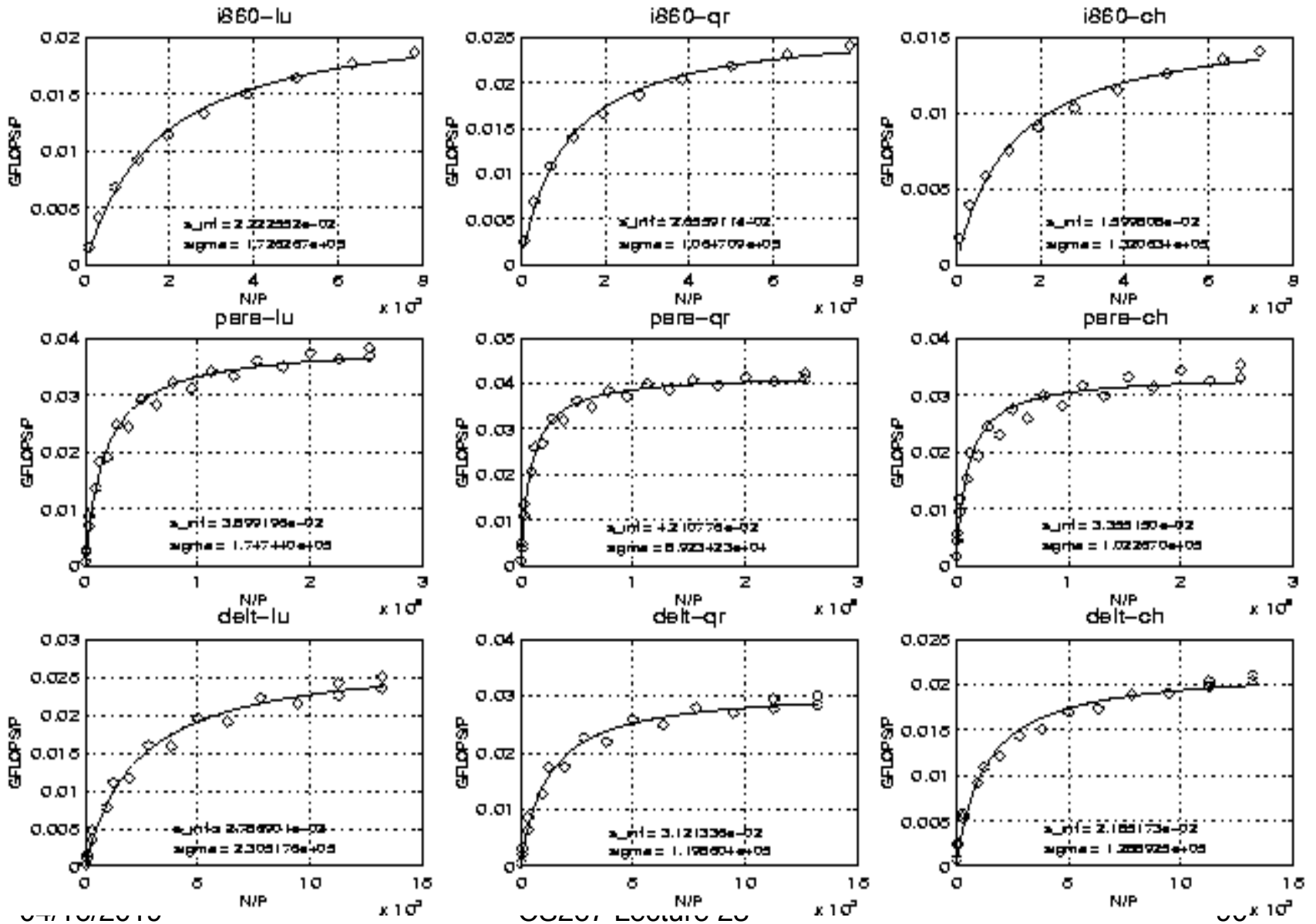


σ is the problem size at which 1/2 efficiency is obtained

$$e(N, P) = \begin{cases} 1 & \text{if } P = 1 \\ \dfrac{e_\infty}{1 + \sigma P/N} & \text{if } P > 1 \end{cases}$$

Validated against experimental data from ScaLAPACK for several algorithms

Model Validation from ScaLAPACK

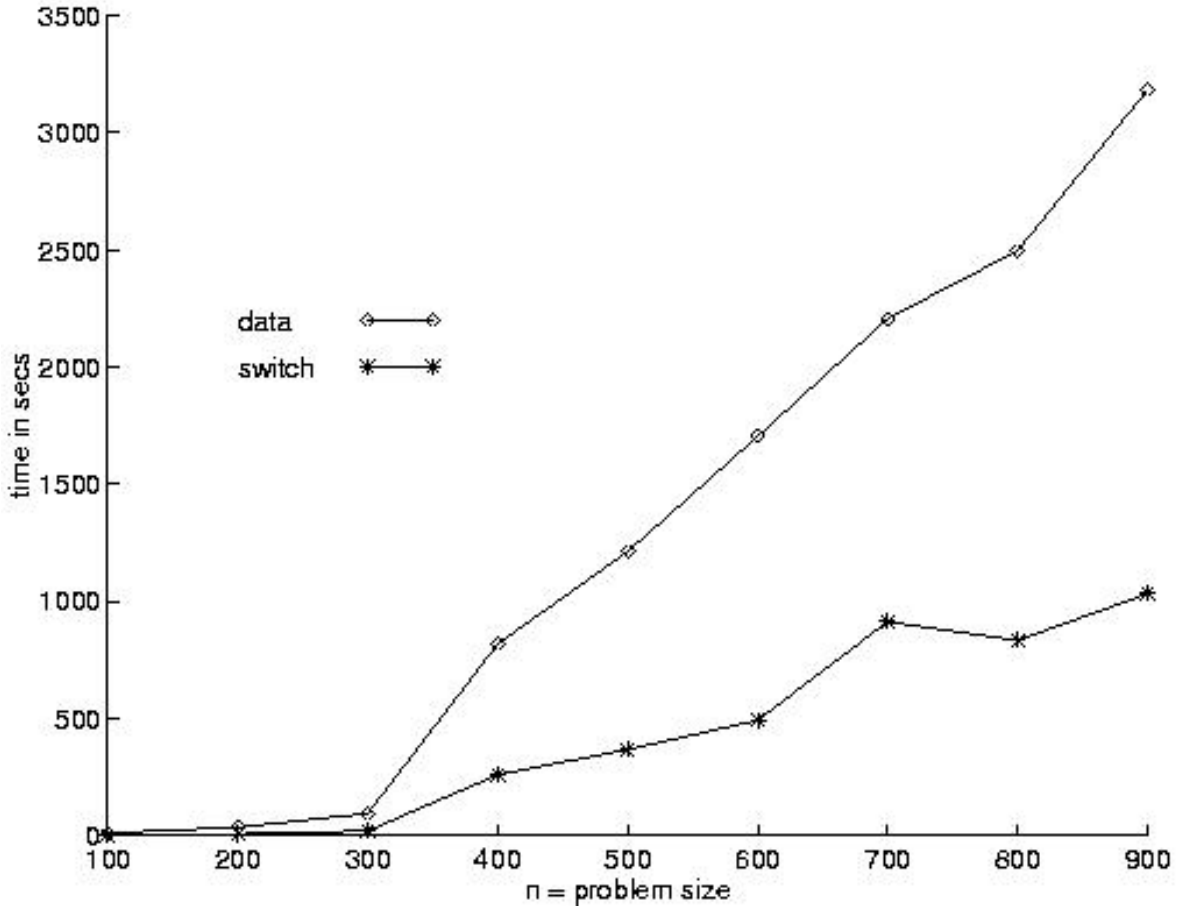


Modeling Performance

- To predict performance, make assumptions about task tree
  - complete tree with branching factor d>= 2
  - d child tasks of parent of size N are all of size N/c, c>1
  - work to do task of size N is $O(N^a)$, a>= 1
- Example: Sign function based eigenvalue routine
  - d=2, c=2 (on average), a=3
- Combine these assumptions with model of data parallelism

Actual Speed of Sign Function Eigensolver

- Starred lines are optimal mixed parallelism
- Solid lines are data parallelism
- Dashed lines are switched parallelism
- Intel Paragon, built on ScaLAPACK
- Switched parallelism worthwhile!



Values of Sigma (Problem Size for Half Peak)

The efficiency of data parallel algorithms depend on characteristics of the algorithm and the machine.

- σ is high if algorithm demands a lot of communication
- σ is high if communication cost on machine is high

Typical values for σ and P for matrix multiply on large scale machines

	CM-5	Paragon	T3D	SP1
σ	53	633	1544	4250
P	256	128	128	64
σ P	14K	81K	200K	270K

Results for LU or FFT are similar, but somewhat higher.

Best-First Search

- Rather than searching to the bottom, keep set of current states in the space
- Pick the "best" one (by some heuristic) for the next step

- Use lower bound l(x) as heuristic
  - l(x) = g(x) + h(x)
  - g(x) is the cost of reaching the current state
  - h(x) is a heuristic for the cost of reaching the goal
  - Choose h(x) to be a lower bound on actual cost

  - E.g., h(x) might be sum of number of moves for each piece in game problem to reach a solution (ignoring other pieces)

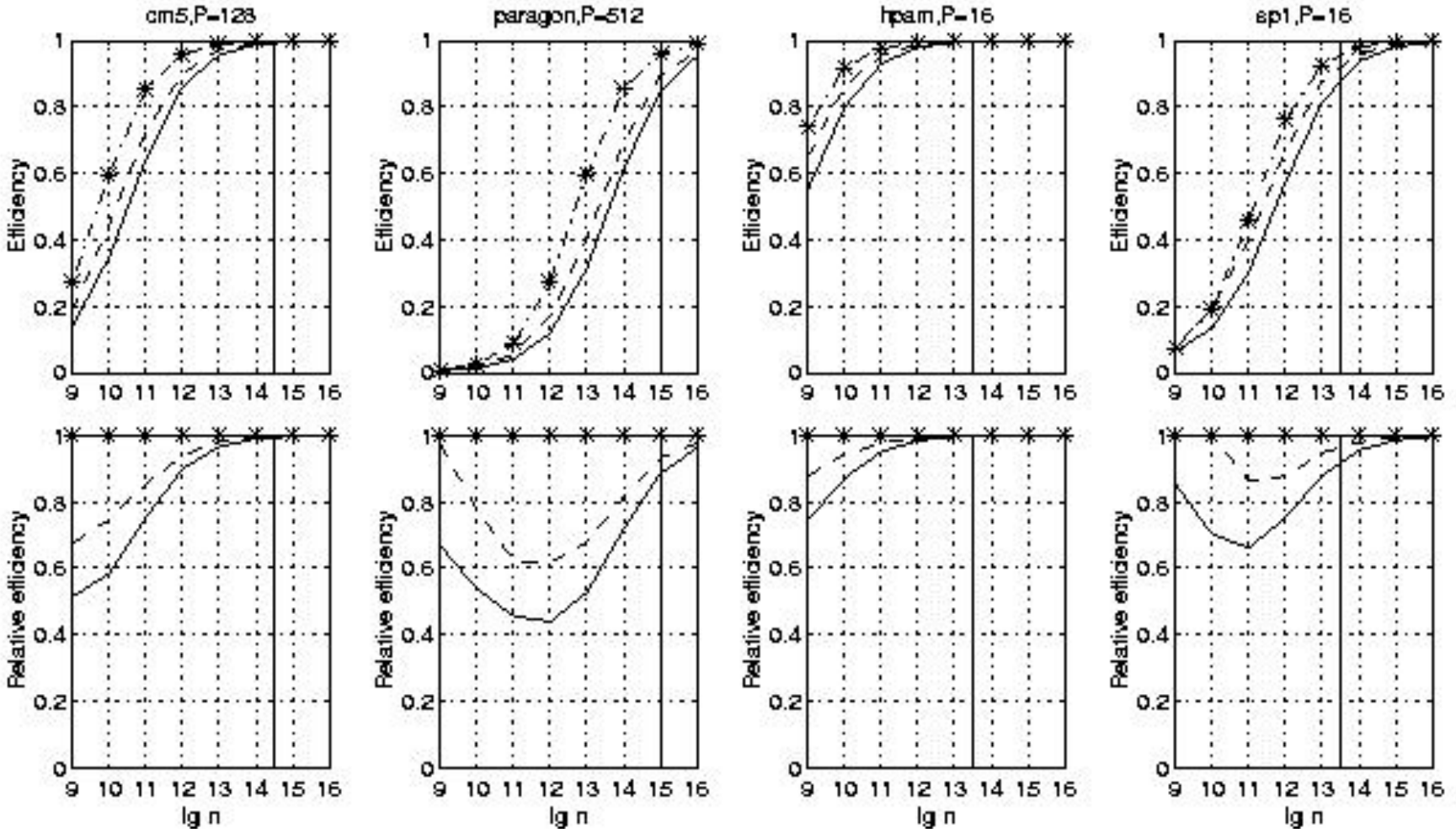
Branch and Bound Search Revisited

- The load balancing algorithms as described were for full depth-first search

- For most real problems, the search is bounded
  - Current bound (e.g., best solution so far) logically shared
  - For large-scale machines, may be replicated
  - All processors need not always agree on bounds
    - Big savings in practice
  - Trade-off between
    - Work spent updating bound
    - Time wasted search unnecessary part of the space

Simulated Efficiency of Eigensolver

- Starred lines are optimal mixed parallelism
- Solid lines are data parallelism
- Dashed lines are switched parallelism



Simulated efficiency of Sparse Cholesky

- Starred lines are optimal mixed parallelism
- Solid lines are data parallelism
- Dashed lines are switched parallelism