

---

# CS 267: Applications of Parallel Computers

## Load Balancing

Kathy Yelick

<http://www-inst.eecs.berkeley.edu/~cs267>

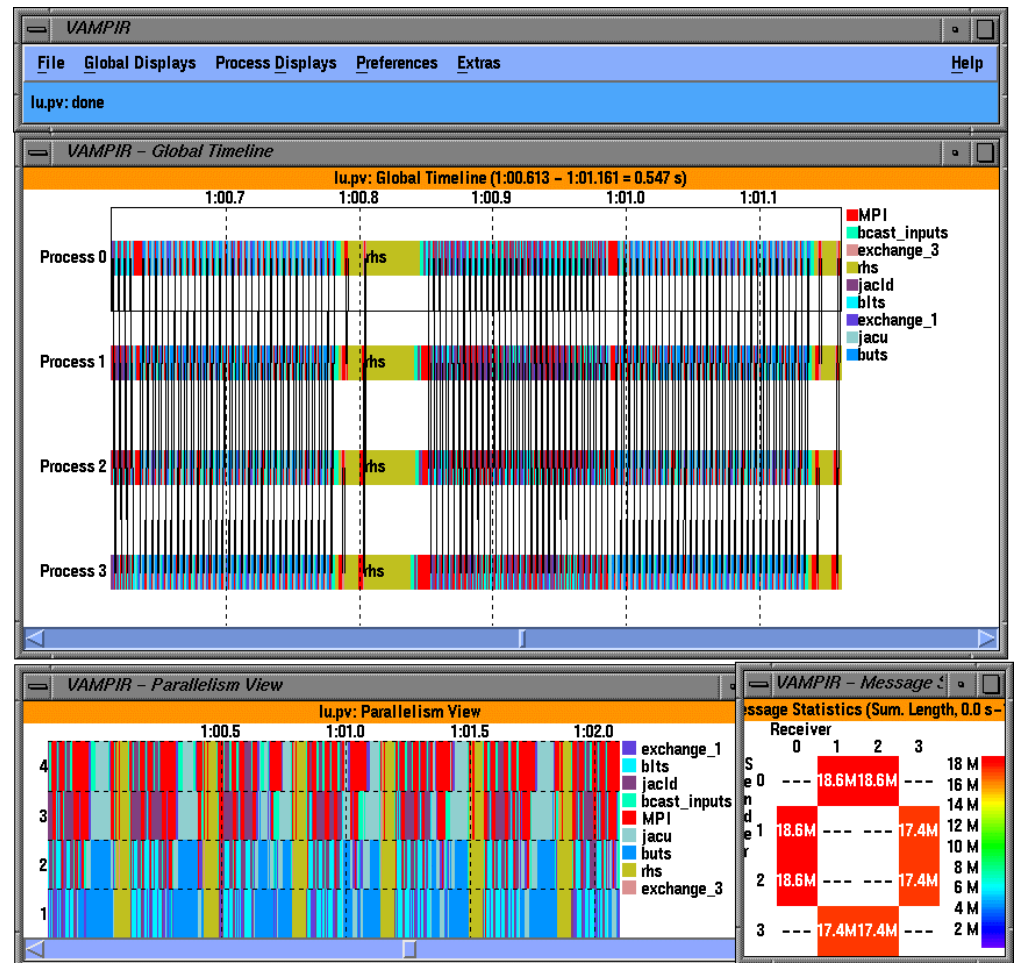
# Load Imbalance in Parallel Applications

The primary sources of inefficiency in parallel codes:

- Poor single processor performance
  - Typically in the memory system
- Too much parallelism overhead
  - Thread creation, synchronization, communication
- Load imbalance
  - Different amounts of work across processors
    - Computation and communication
  - Different speeds (or available resources) for the processors
    - Possibly due to load on the machine
- Recognizing load imbalance
  - Time spent at synchronization is high and is uneven across processors: don't take min/max/med/mean of barrier times

# Measuring Load Imbalance

- Challenges:
  - Can be hard to separate from high synch overhead
  - Especially subtle if not bulk-synchronous
  - “Spin locks” can make synchronization look like useful work
  - Note that imbalance may change over phases
  - Insufficient parallel always leads to load imbalance
  - Tools like TAU (shown) can help



# Load Balancing Overview

---

Load balancing differs with properties of the tasks (chunks of work):

- **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
- **Locality**
  - 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?

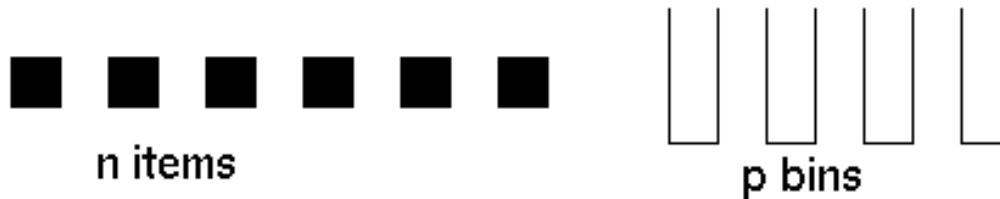
# Task Cost Spectrum

---

Schedule a set of tasks under one of the following assumptions:

**Easy:** The tasks all have equal (unit) cost.

branch-free loops



**Harder:** The tasks have different, but known, times.

sparse matrix-vector multiply



**Hardest:** The task costs unknown until after execution.

GCM, circuits

# Task Dependency Spectrum

---

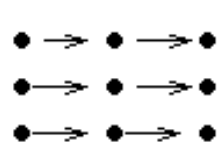
Schedule a graph of tasks under one of the following assumptions:

**Easy:** The tasks can execute in any order.



dependence  
free loops

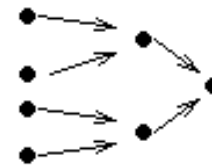
**Harder:** The tasks have a predictable structure.



wave-front



out-tree



in-tree



general dag

balanced or unbalanced

matrix

computations  
(dense, and some  
sparse, Cholesky)

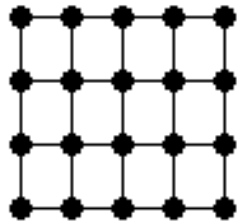
**Hardest:** The structure changes dynamically (slowly or quickly) search, sparse LU

# Task Locality Spectrum (Communication)

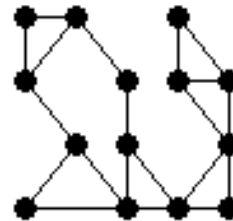
*Schedule a set of tasks under one of the following assumptions:*

*Easy:* The tasks, once created, do not communicate. **embarrassingly parallel**

*Harder:* The tasks communicate in a predictable pattern.



regular



irregular

**PDE solver**

*Hardest:* The communication pattern is unpredictable. **discrete event simulation**

# Spectrum of Solutions

---

One of the key questions 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. (offline algorithms)
- **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.
- **Dynamic scheduling.** Information is not known until mid-execution. (online algorithms)

# Dynamic Load Balancing

---

- Motivation for dynamic load balancing
  - Search algorithms
- Centralized load balancing
  - Overview
  - Special case for schedule independent loop iterations
- Distributed load balancing
  - Overview
  - Engineering
  - Theoretical results
- Load balancing in general
- Example scheduling problem: mixed parallelism
  - Demonstrate use of coarse performance models

# Search

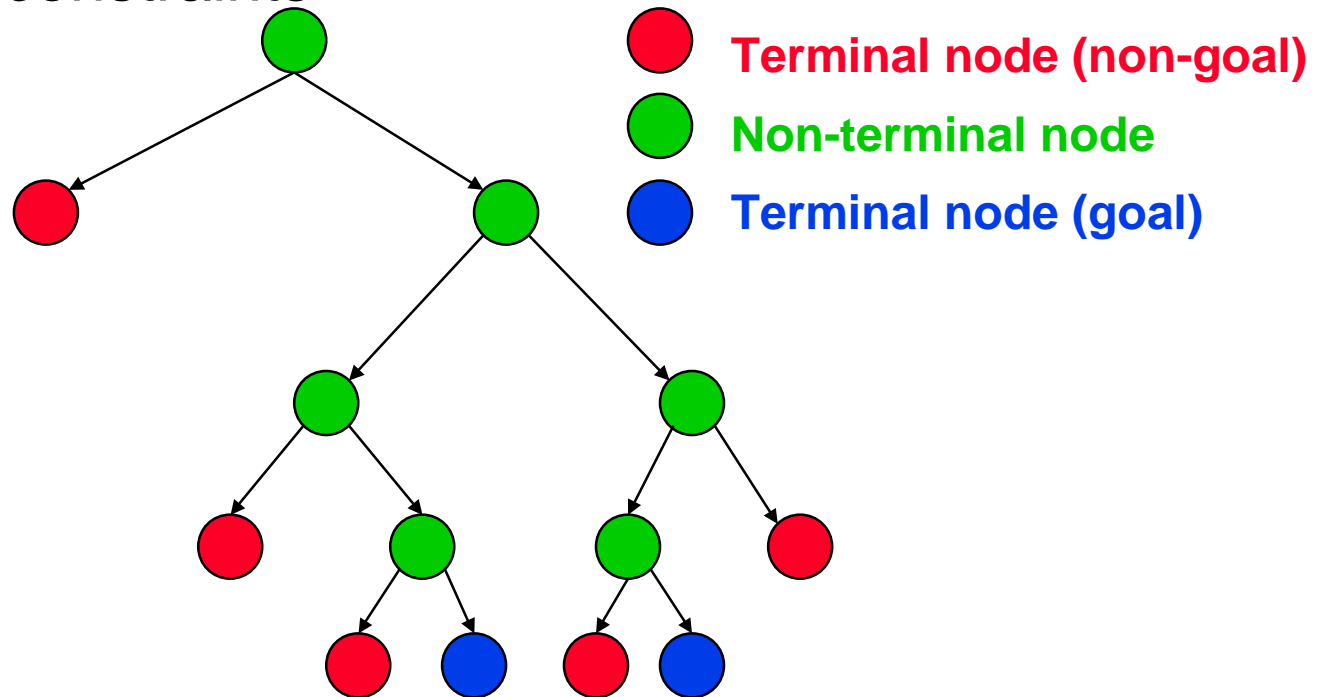
---

- Search problems are often:
  - Computationally expensive
  - Have very different parallelization strategies than physical simulations.
  - Require dynamic load balancing
  
- Examples:
  - Optimal layout of VLSI chips
  - Robot motion planning
  - Chess and other games (N-queens)
  - Speech processing
  - Eigenvalue search
  - Constructing phylogeny tree from set of genes

# Example Problem: Tree Search

---

- In Tree Search the tree unfolds dynamically
- May be a graph if there are common sub-problems along different paths
- Graphs unlike meshes which are precomputed and have no ordering constraints



# Sequential Search Algorithms

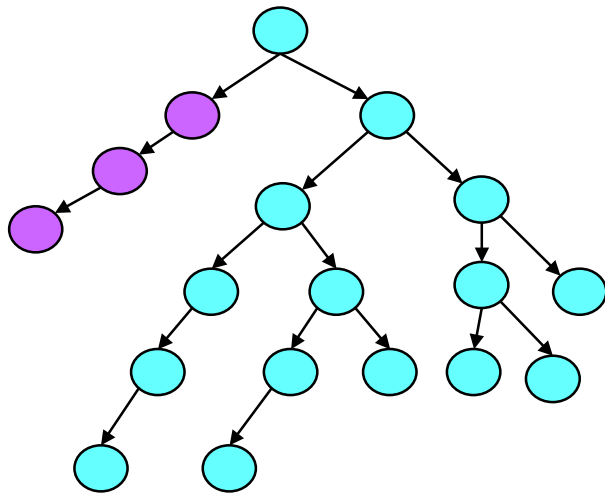
---

- Depth-first search (DFS)
  - Simple backtracking
    - Search to bottom, backing up to last choice if necessary
  - Depth-first 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
    - Choose a bound on search depth,  $d$  and use DFS up to depth  $d$
    - If no solution is found, increase  $d$  and start again
    - Iterative deepening  $A^*$  uses a lower bound estimate of cost-to-solution as the bound
- Breadth-first search (BFS)
  - Search across a given level in the tree

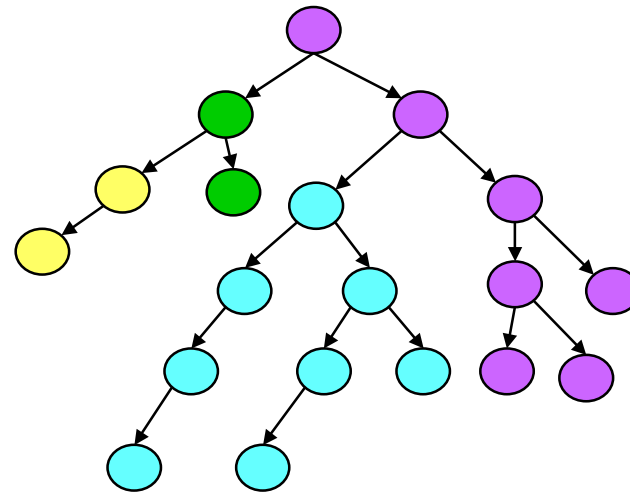
# Parallel Search

---

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



Load balance on 2 processors

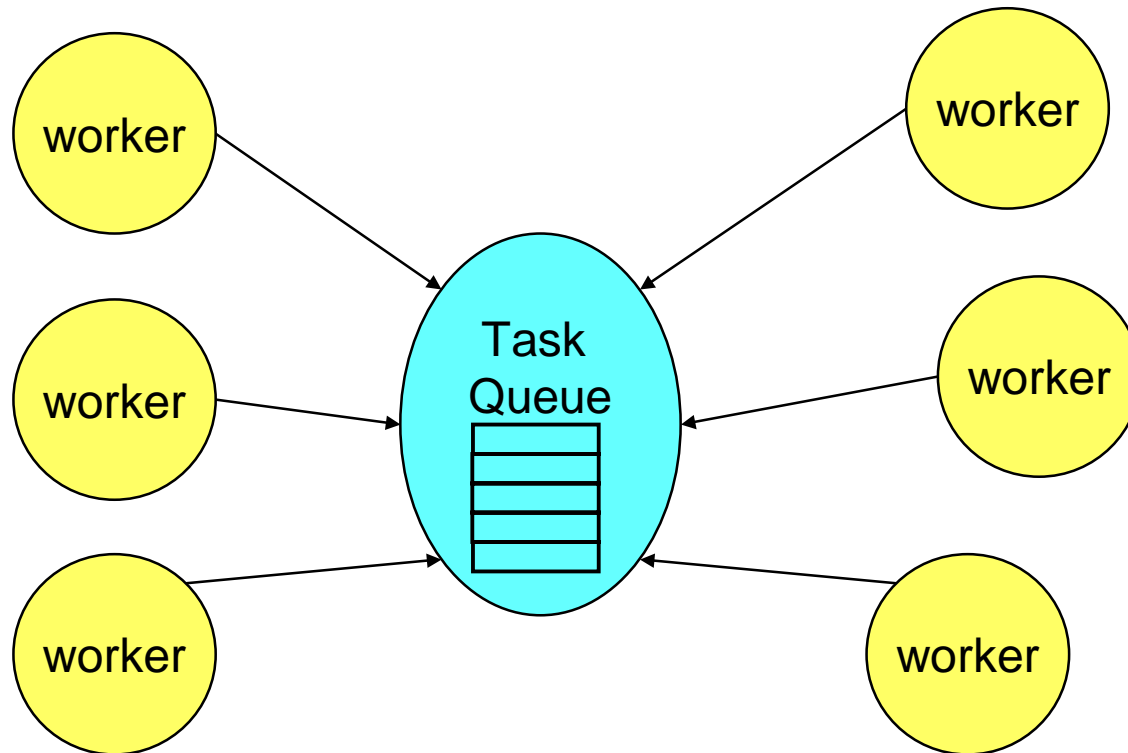


Load balance on 4 processors

# Centralized Scheduling

---

- Keep a queue of task waiting to be done
  - May be done by manager task
  - Or a shared data structure protected by locks



# Centralized Task Queue: Scheduling Loops

- When applied to loops, often called **self scheduling**:
  - Tasks may be range of loop indices to compute
  - Assumes independent iterations
  - Loop body has unpredictable time (branches) or the problem is not interesting
- Originally designed for:
  - Scheduling loops by compiler (really the runtime-system)
  - Original paper by Tang and Yew, ICPP 1986
- This is:
  - Dynamic, online scheduling algorithm
  - Good for a small number of processors (centralized)
  - Special case of task graph – independent tasks, known at once

# Variations on Self-Scheduling

---

- Typically, don't want to grab smallest unit of parallel work, e.g., a single iteration
- Instead, choose a chunk of tasks of size  $K$ .
  - If  $K$  is large, access overhead for task queue is small
  - If  $K$  is small, we are likely to have even finish times (load balance)
- Four variations:
  1. Use a fixed chunk size
  2. Guided self-scheduling
  3. Tapering
  4. Weighted Factoring
  - Note: there are other variations

## Variation 1: Fixed Chunk Size

---

- Kruskal and Weiss give a technique for computing the optimal chunk size
- Requires a lot of information about the problem characteristics
  - e.g., task costs as well as number
- Not very useful in practice.
  - Task costs must be known at loop startup time
  - E.g., in compiler, all branches be predicted based on loop indices and used for task cost estimates

## Variation 2: Guided Self-Scheduling

---

- Idea: use larger chunks at the beginning to avoid excessive overhead and smaller chunks near the end to even out the finish times.
  - The chunk size  $K_i$  at the  $i^{\text{th}}$  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
- See Polychronopolous, “Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers,” IEEE Transactions on Computers, Dec. 1987.

## Variation 3: Tapering

---

- Idea: the chunk size,  $K_i$  is a function of not only the remaining work, but also the task cost variance
  - variance is estimated using history information
  - high variance => small chunk size should be used
  - low variance => larger chunks OK
- See S. Lucco, “Adaptive Parallel Programs,” PhD Thesis, UCB, CSD-95-864, 1994.
  - Gives analysis (based on workload distribution)
  - Also gives experimental results -- tapering always works at least as well as GSS, although difference is often small

## Variation 4: Weighted Factoring

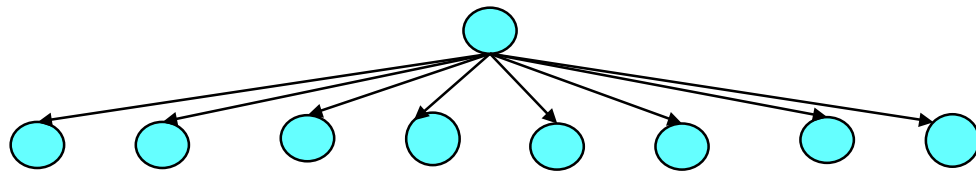
---

- Idea: similar to self-scheduling, but divide task cost by computational power of requesting node
- Useful for heterogeneous systems
- Also useful for shared resource NOWs, e.g., built using all the machines in a building
  - as with Tapering, historical information is used to predict future speed
  - “speed” may depend on the other loads currently on a given processor
- See Hummel, Schmit, Uma, and Wein, SPAA ‘96
  - includes experimental data and analysis

# When is Self-Scheduling a Good Idea?

Useful when:

- A batch (or set) of tasks without dependencies
  - can also be used with dependencies, but most analysis has only been done for task sets without dependencies



- The cost of each task is unknown
- Locality is not important
- Shared memory machine, or at least number of processors is small – centralization is OK

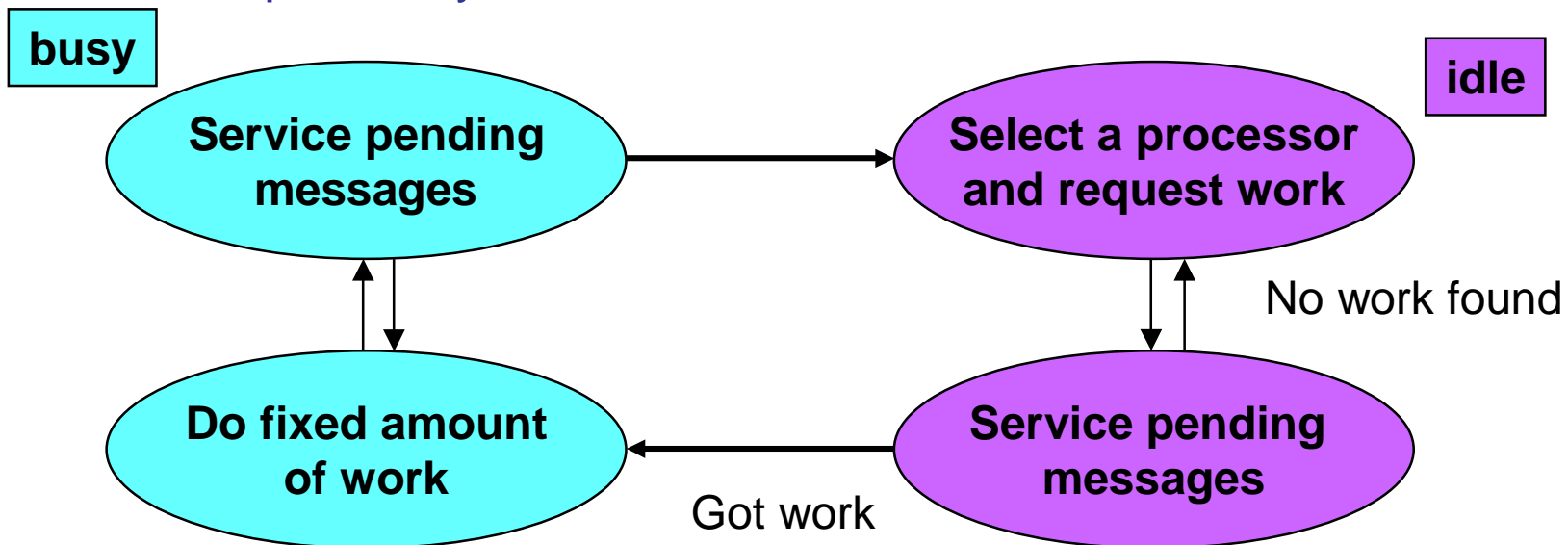
# Distributed Task Queues

---

- The obvious extension of task queue to distributed memory is:
  - a distributed task queue (or bag)
  - Doesn't appear as explicit data structure in message-passing
- When are these a good idea?
  - Distributed memory multiprocessors
  - Or, shared memory with significant synchronization overhead
  - Locality is not (very) important
  - Tasks that are:
    - known in advance, e.g., a bag of independent ones
    - dependencies exist, i.e., being computed on the fly
  - The costs of tasks is not known in advance

# Distributed Dynamic Load Balancing

- Dynamic load balancing algorithms go by other names:
  - Work stealing, work crews, hungry puppies...
- Basic idea, when applied to tree search:
  - Each processor performs search on disjoint part of tree
  - When finished, get work from a processor that is still busy
  - Requires asynchronous communication



# How to Select a Donor Processor

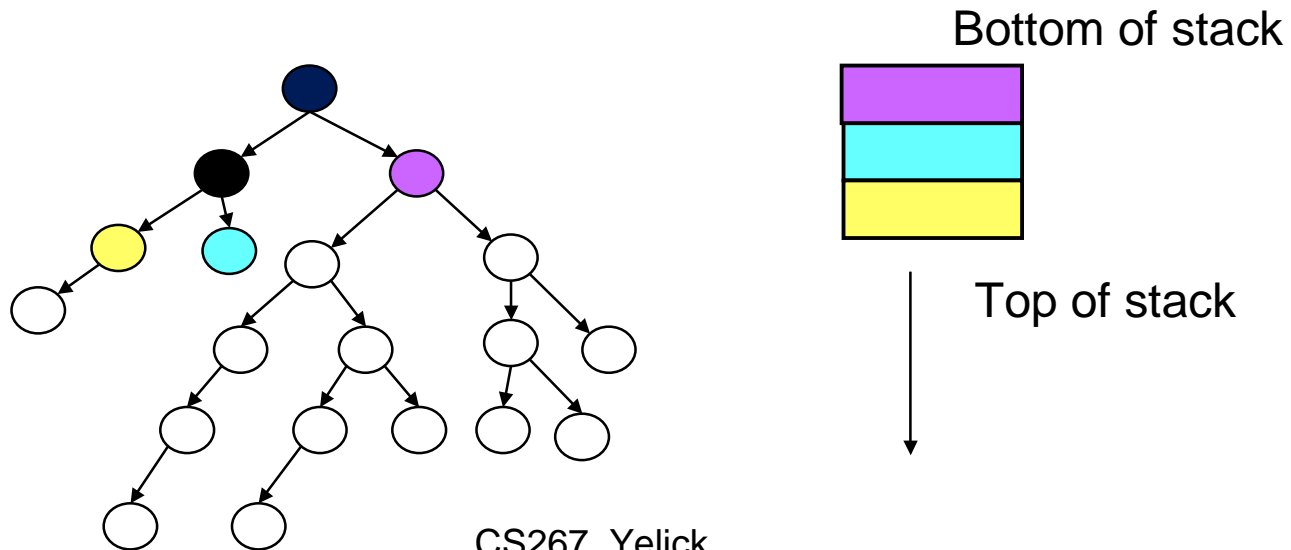
---

- Three basic techniques:
  1. Asynchronous round robin
    - Each processor  $k$ , keeps a variable  $target_k$
    - When a processor runs out of work, request from  $target_k$
    - Set  $target_k = (target_k + 1) \% procs$
  2. Global round robin
    - Processor 0 keeps a single variable  $target$
    - When a processor needs work, get  $target$ , a request from  $target$
    - P0 increments  $(mod\ procs)$  with each access to  $target$
  3. Random polling/stealing
    - When a processor needs work, select a random processor and request work from it
- All three used repeatedly if no work is found

# How to Split Work

---

- First parameter is number of tasks to split
  - Related to the self-scheduling variations, but total number of tasks is now unknown
- Second question is which one(s)
  - Send tasks near the bottom of the stack (oldest)
  - Execute from the top (most recent)
  - May be able to do better with information about task costs



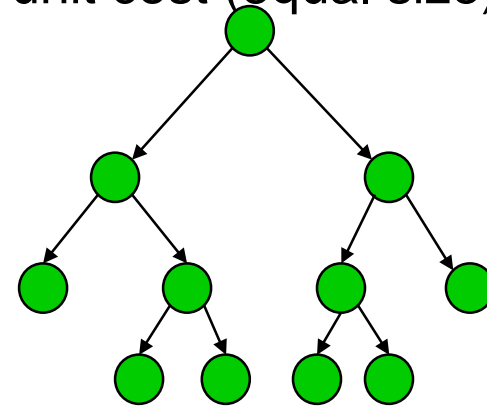
# Theoretical Results (1)

---

Main result: A simple randomized algorithm is optimal with high probability

- Karp and Zhang [88] 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



- Show this for independent, equal sized tasks

- “Throw balls into random bins”:  $\Theta ( \log n / \log \log n )$  in largest bin
- Throw  $d$  times and pick the smallest bin:  $\log \log n / \log d = \Theta (1)$  [Azar]
- Extension to parallel throwing [Adler et al 95]
- Shows  $p \log p$  tasks leads to “good” balance

## Theoretical Results (2)

---

Main result: A simple randomized algorithm is 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
- 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
- Open problem: does task pulling provably work well for dynamic trees?

## Distributed Task Queue References

---

- Introduction to Parallel Computing by Kumar et al (text)
- Multipol library (See C.-P. Wen, UCB PhD, 1996.)
  - Part of Multipol ([www.cs.berkeley.edu/projects/multipol](http://www.cs.berkeley.edu/projects/multipol))
  - Try to push tasks with high ratio of cost to compute/cost to push
    - Ex: for matmul, ratio =  $2n^3 \text{ cost(flop)} / 2n^2 \text{ cost(send a word)}$
- Goldstein, Rogers, Grunwald, and others (independent work) have all shown
  - advantages of integrating into the language framework
  - very lightweight thread creation
- CILK (Leiserson et al) ([supertech.lcs.mit.edu/cilk](http://supertech.lcs.mit.edu/cilk))

# Diffusion-Based Load Balancing

---

- In the randomized schemes, the machine is treated as fully-connected.
- Diffusion-based load balancing takes topology into account
  - Locality properties better than prior work
  - Load balancing somewhat slower than randomized
  - Cost of tasks must be known at creation time
  - No dependencies between tasks

## Diffusion-based load balancing

---

- The machine is modeled as a graph
- At each step, we compute the **weight** of task remaining on each processor
  - This is simply the number if they are unit cost tasks
- Each processor compares its weight with its neighbors and performs some averaging
  - Markov chain analysis
- 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
- Note: locality is still not a major concern, although balancing with neighbors may be better than random

## Mixed Parallelism: Digression

---

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

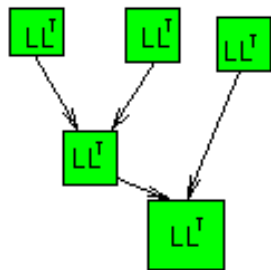
Appears in:

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

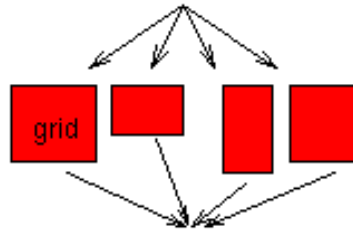
# Mixed Parallelism Strategies

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

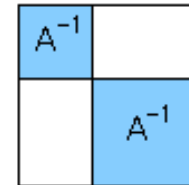
*sparse cholesky*



*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?

## Approach:

Use modeling, validated by experiments to predict performance

# Which Strategy to Use

---

Pure data parallelism

*spread each block over all processors*

Pure task parallelism

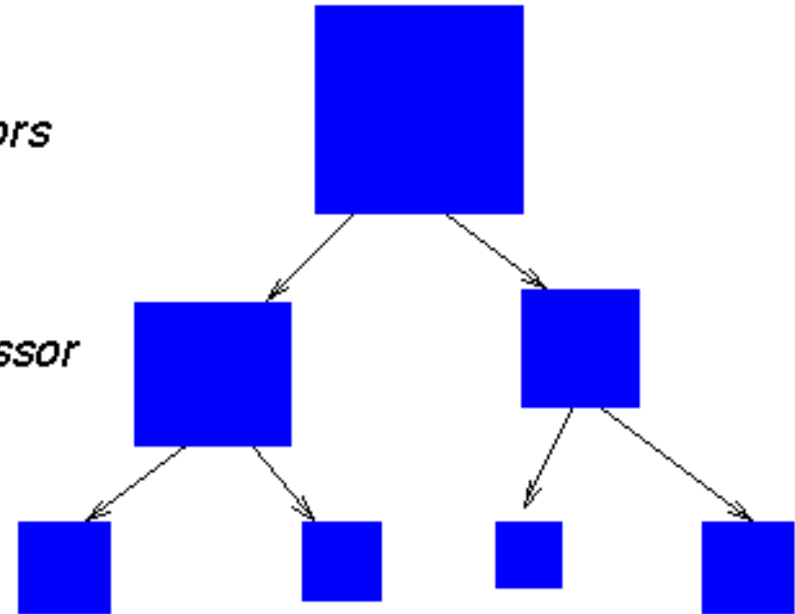
*assign each block to a single processor*

Switched parallelism

*at some level, go from data to task*

Mixed parallelism

*spread blocks on subsets of processors*



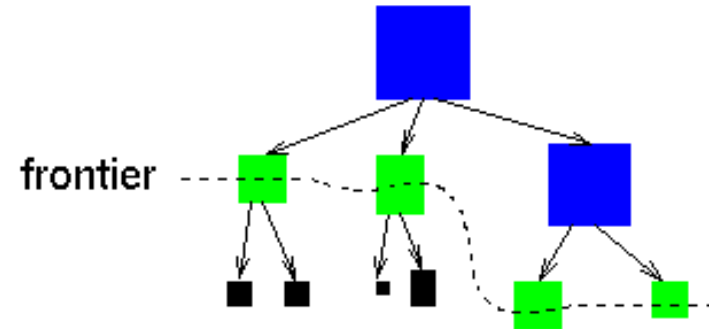
Modeling shows that switch parallelism gets almost all the benefit of mixed.

# Switch Parallelism: A Special Case

---

## A Prefix-Suffix Heuristic

- \* Sort the current frontier of tasks to be executed:  $N_1 > N_2 > N_3 > \dots > N_I$
- \* Assume  $\text{cost}(N_i, P)$  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



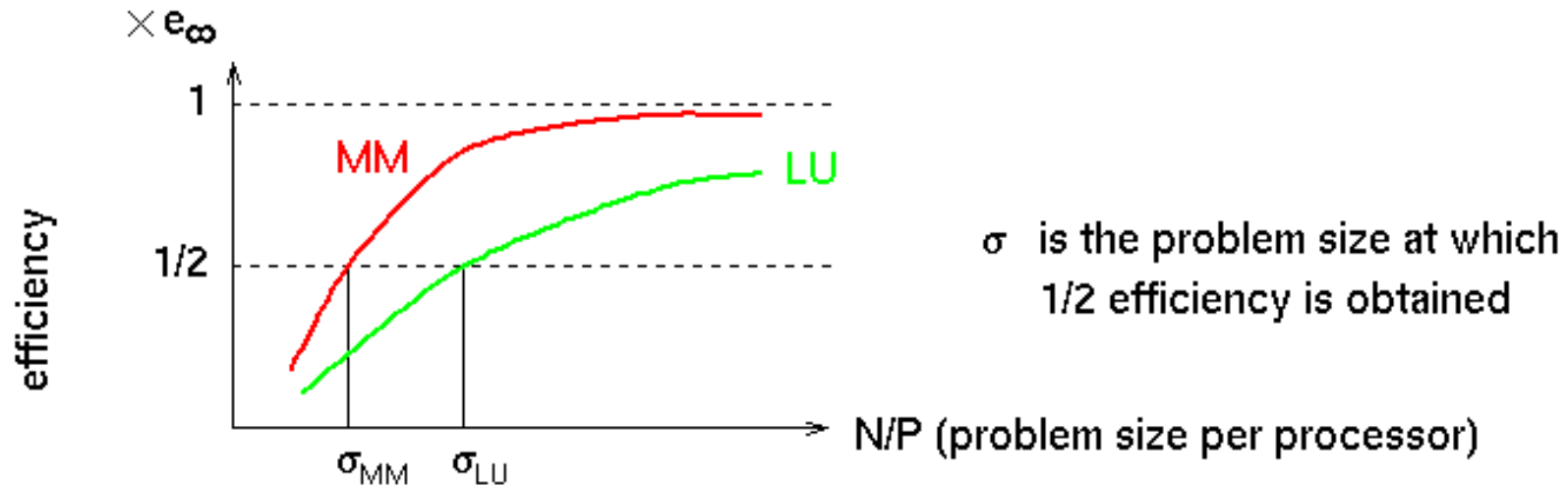
## Notes:

Sorting is unnecessary if all tasks have the same size

The decision to run something in data or task models is not simply a function of the task size/cost

# Simple Performance Model for Data Parallelism

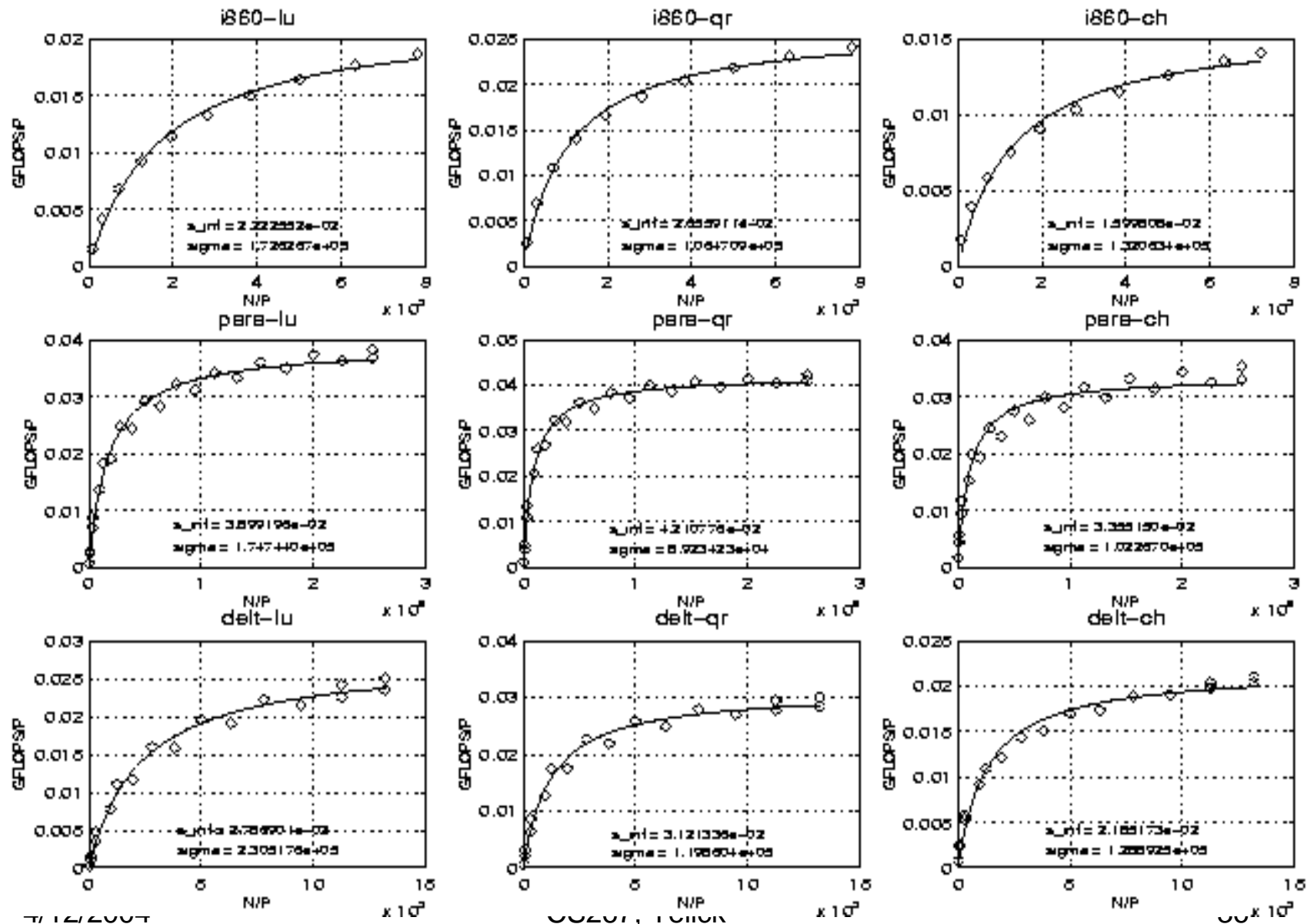
Observation: the efficiency of a data parallel algorithm depends on the problem size per processor,  $N/P$ , for sufficiently large  $N$ .



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

Validated against experimental data from ScaLAPACK for several algorithms

# Model Validation from ScaLAPACK



# Values of Sigma (Problem Size for Half Peak)

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

$\sigma$  is high if algorithm demands a lot of communication

$\sigma$  is high if communication cost on machine is high

Typical values for  $\sigma$  and P for matrix multiply on large scale machines

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

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

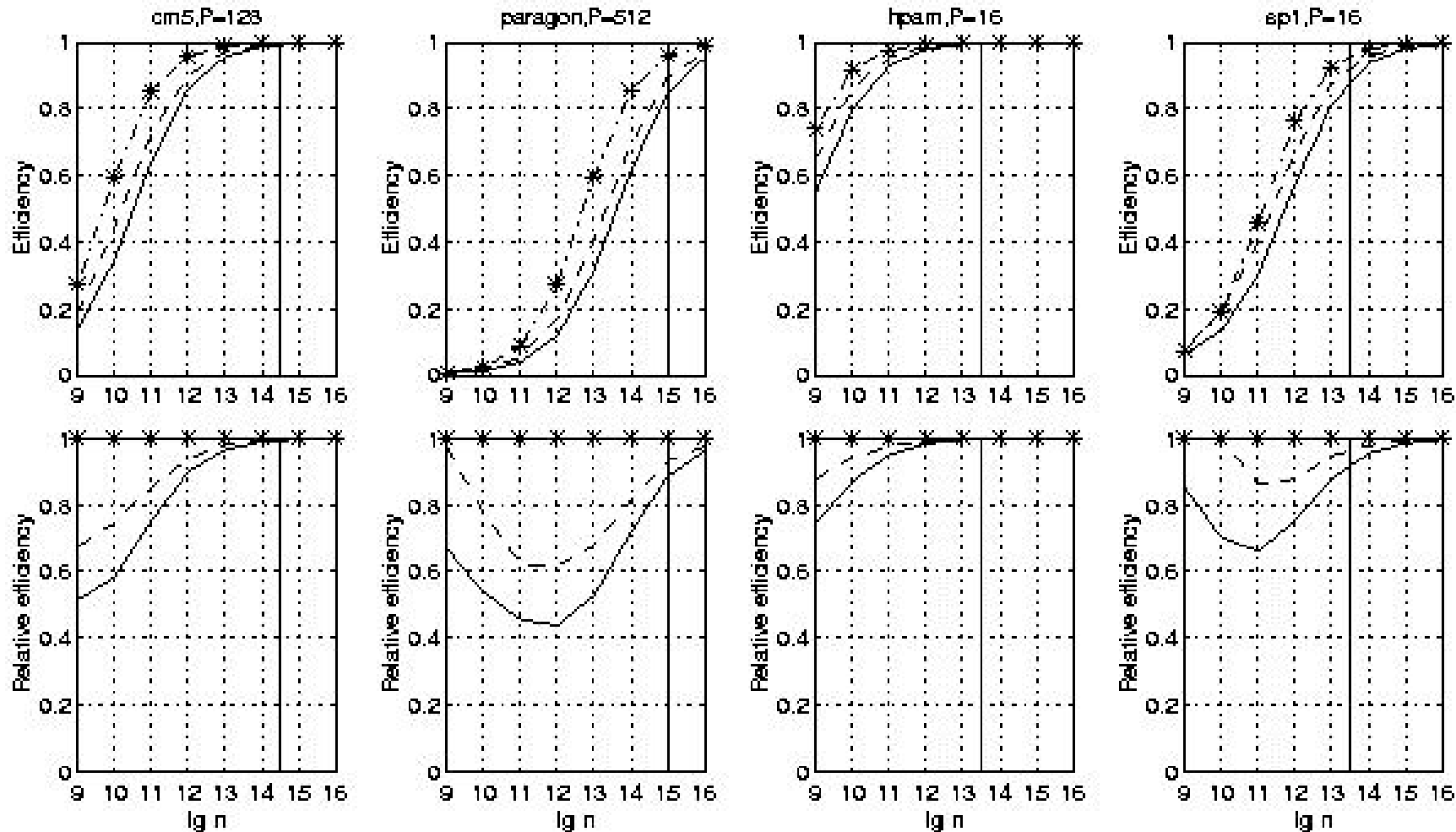
# Modeling Performance

---

- To predict performance, make assumptions about task tree
  - complete tree with branching factor  $d \geq 2$
  - $d$  child tasks of parent of size  $N$  are all of size  $N/c$ ,  $c > 1$
  - work to do task of size  $N$  is  $O(N^a)$ ,  $a \geq 1$
- Example: Sign function based eigenvalue routine
  - $d=2$ ,  $c=4$  (on average),  $a=1.5$
- Example: Sparse Cholesky on 2D mesh
  - $d=4$ ,  $c=4$ ,  $a=1.5$
- Combine these assumptions with model of data parallelism

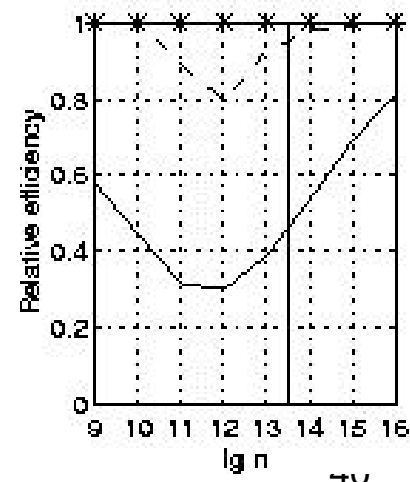
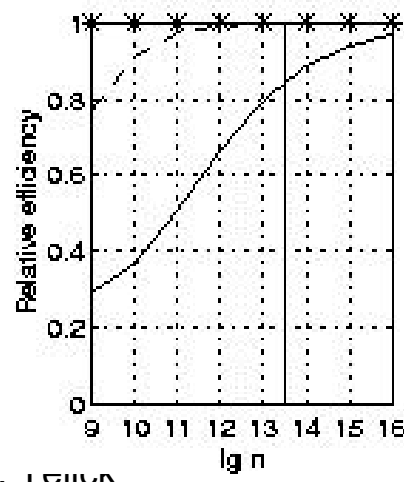
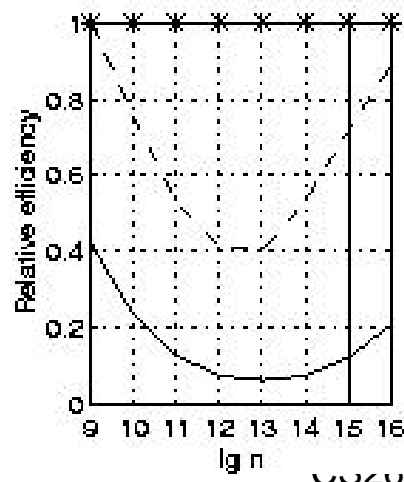
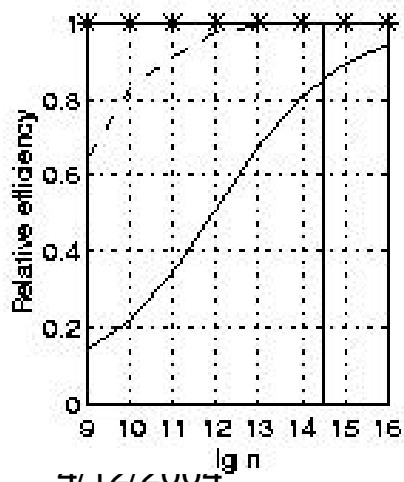
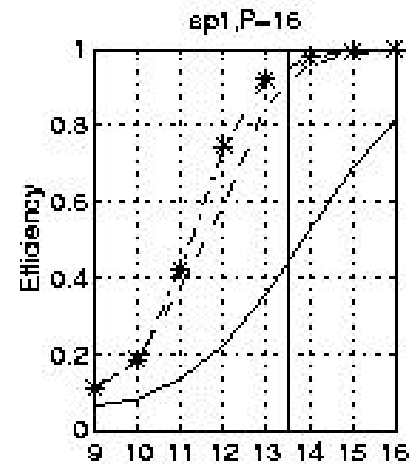
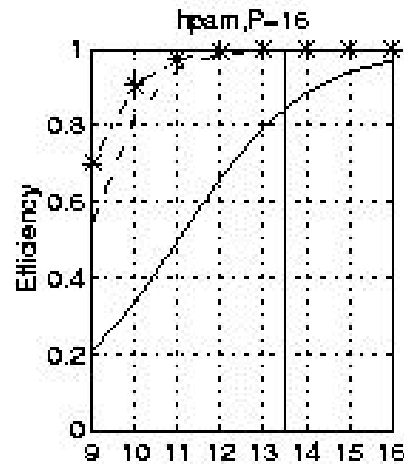
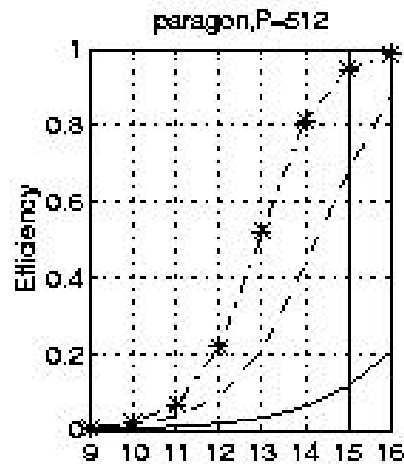
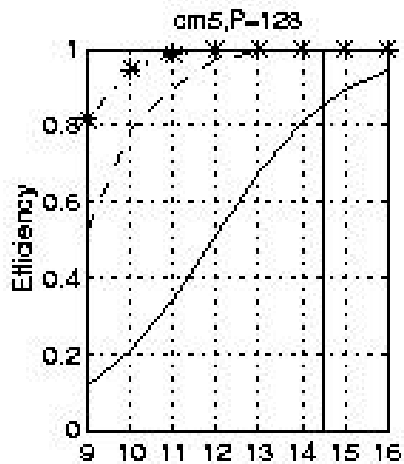
# Simulated Efficiency of Eigensolver

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



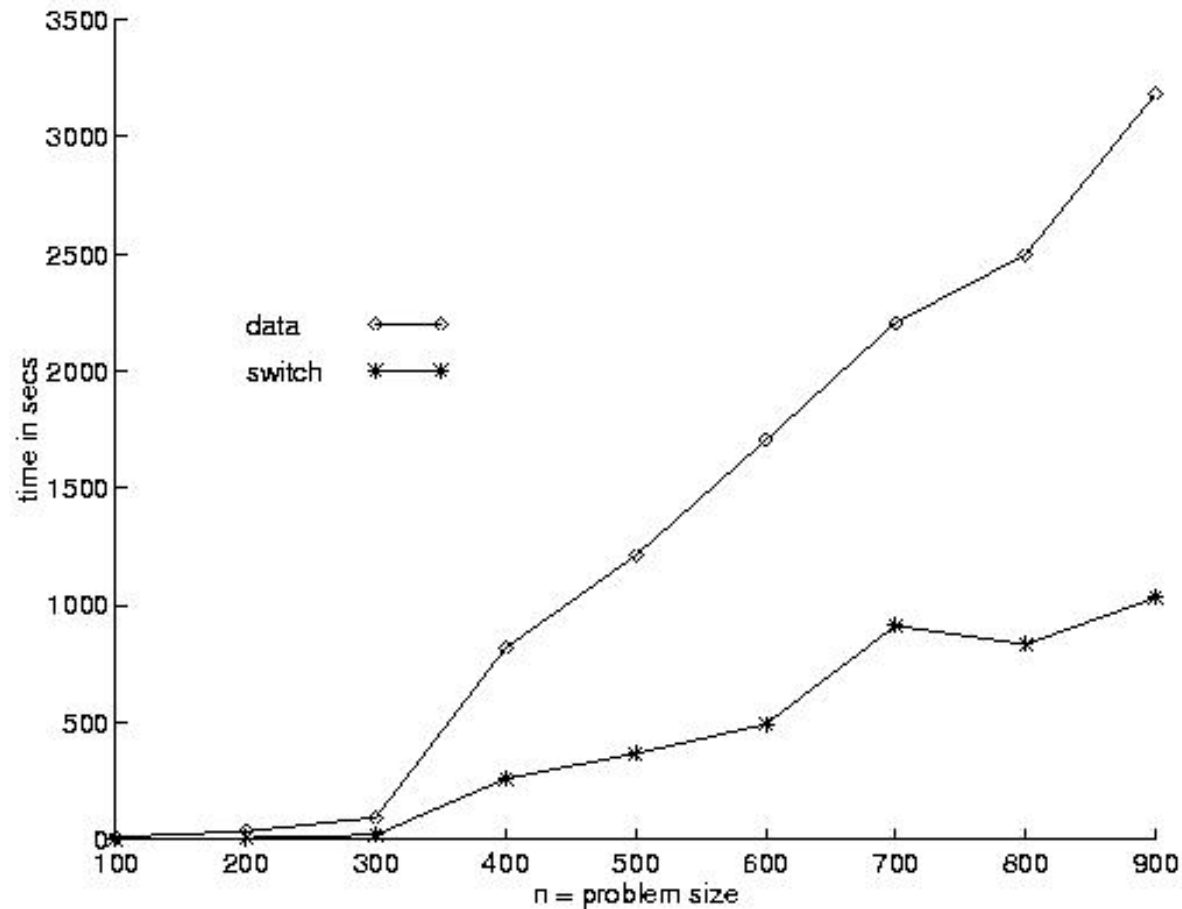
# Simulated efficiency of Sparse Cholesky

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



# 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!



# Summary

---

- Dynamic load balancing using task queues
  - Requires problems with little data (low locality requirements)
  - Good when load is very unbalanced
    - Dynamically unfolding task graph
    - Unpredictable running time for tasks
- For the opposite kinds of problems, use graph partitioning
  - Have static or slowly changing load
  - Information (estimates available) on
    - Tasks costs
    - Tasks dependencies
    - Communication costs
- More on graph partitioning next time

# Small Example

---

- The 0/1 integer-linear-programming problem

- Given integer matrices/vectors as follows:

- an  $n \times m$  matrix  $A$ ,
- an  $m$ -element vector  $b$ , and
- an  $n$ -element vector  $c$

- Find

- $n$ -element vector  $x$  whose elements are 0 or 1
- Satisfies the constraint:  $A \cdot x \geq b$
- The function:  $f(x) = c \cdot x$  should be minimized

- E.g.,

$$A = \begin{pmatrix} 5 & 2 & 1 & 2 \\ 1 & -1 & -1 & 2 \\ 3 & 1 & 1 & 3 \end{pmatrix} \quad b = \begin{pmatrix} 8 \\ 2 \\ 5 \end{pmatrix} \quad c = \begin{pmatrix} 2 \\ 1 \\ -1 \\ -2 \end{pmatrix}$$

- $5x_1 + 2x_2 + x_3 + 2x_4 \geq 8$  (and 2 others inequalities)
- Minimize:  $2x_1 + x_2 - x_3 - 2x_4$  Note:  $2^4$  possible values for  $x$

# Discrete Optimizations Problems in General

- A discrete optimization problem  $(S, f)$ 
  - $S$  is a set of **feasible solutions** that satisfy given constraints.  $S$  is finite or countably infinite.
  - $f$  is the **cost function** that maps each element of  $S$  onto the set of real numbers  $R$ .
- The objective of a discrete optimization problem (DOP) is to find a feasible solution  $x_{\text{opt}}$ , such that  $f(x_{\text{opt}}) \leq f(x)$  for all  $x$  in  $S$ .
- Discrete optimizations problems are NP-complete, so only exponential solutions are known
  - Parallelism gives only a constant speedup
  - Need to focus on average case behavior

# 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