
CS 267: Applications of Parallel Computers

Load Balancing

Horst Simon

<http://www.cs.berkeley.edu/~skamil/cs267/>

Outline

- **Motivation for Load Balancing**
- **Recall graph partitioning as load balancing technique**
- **Overview of load balancing problems, as determined by**
 - **Task costs**
 - **Task dependencies**
 - **Locality needs**
- **Spectrum of solutions**
 - **Static - all information available before starting**
 - **Semi-Static - some info before starting**
 - **Dynamic - little or no info before starting**
- **Survey of solutions**
 - **How each one works**
 - **Theoretical bounds, if any**
 - **When to use it**

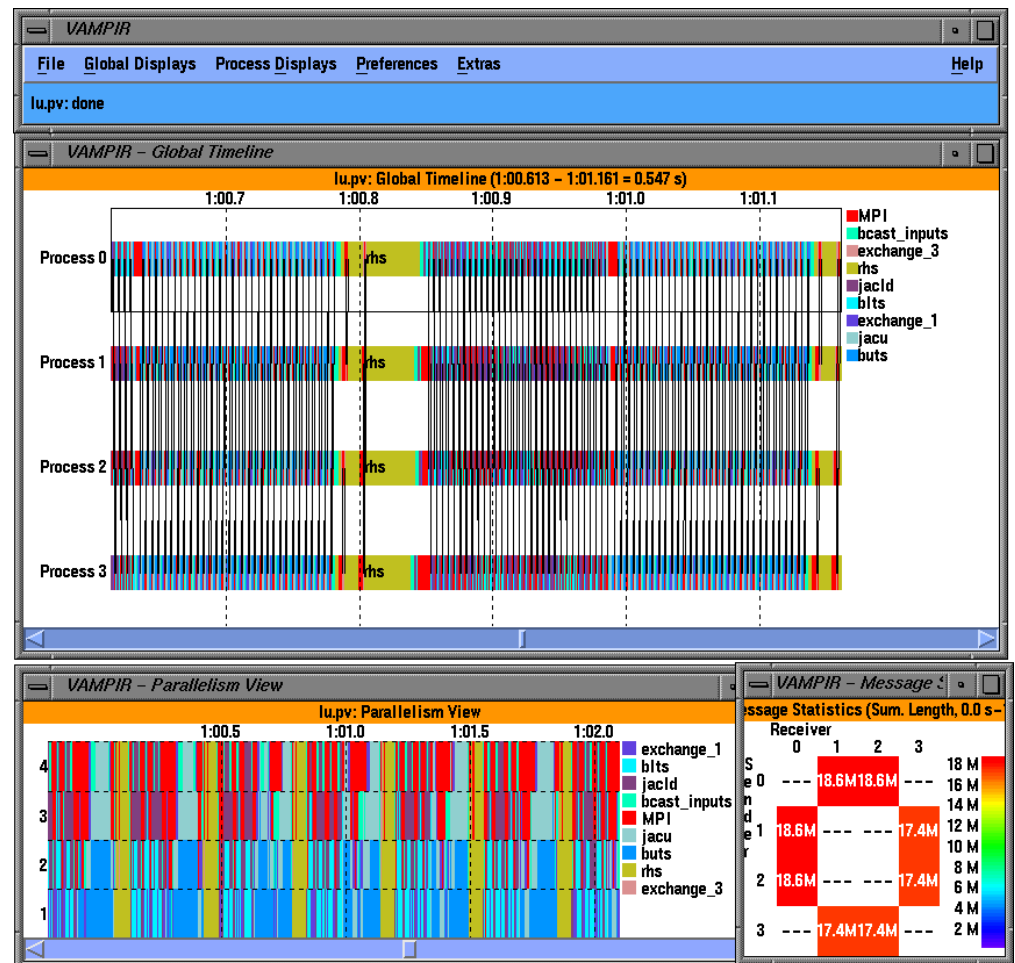
Load Imbalance in Parallel Applications

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
- **How to recognizing load imbalance**
 - Time spent at synchronization is high and is uneven across processors, but not always so simple ...

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 parallelism always leads to load imbalance
 - Tools like TAU can help (acts.nerisc.gov)



Review of Graph Partitioning

- **Partition $G(N,E)$ so that**
 - $N = N_1 \cup \dots \cup N_p$, with each $|N_i| \sim |N|/p$
 - As few edges connecting different N_i and N_k as possible
- **If $N = \{\text{tasks}\}$, 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
- **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?

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, search

Task Dependency Spectrum

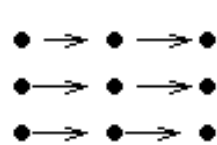
Schedule a graph of tasks under one of the following assumptions:

Easy: The tasks can execute in any order.



dependence
free loops

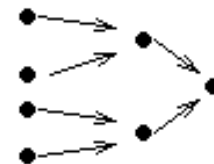
Harder: The tasks have a predictable structure.



wave-front



out-tree



in-tree

balanced or unbalanced



general dag

matrix

computations
(dense, and some
sparse, Cholesky)

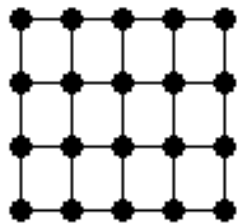
Hardest: The structure changes dynamically (slowly or quickly) search, sparse LU

Task Locality Spectrum (Communication)

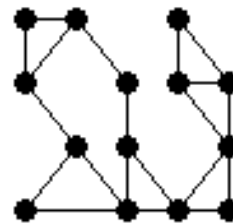
Schedule a set of tasks under one of the following assumptions:

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

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, e.g. graph partitioning**
- **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.
 - **e.g. Kernighan-Lin**
- **Dynamic scheduling.** Information is not known until mid-execution.
 - **On-line algorithms**

Dynamic Load Balancing

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

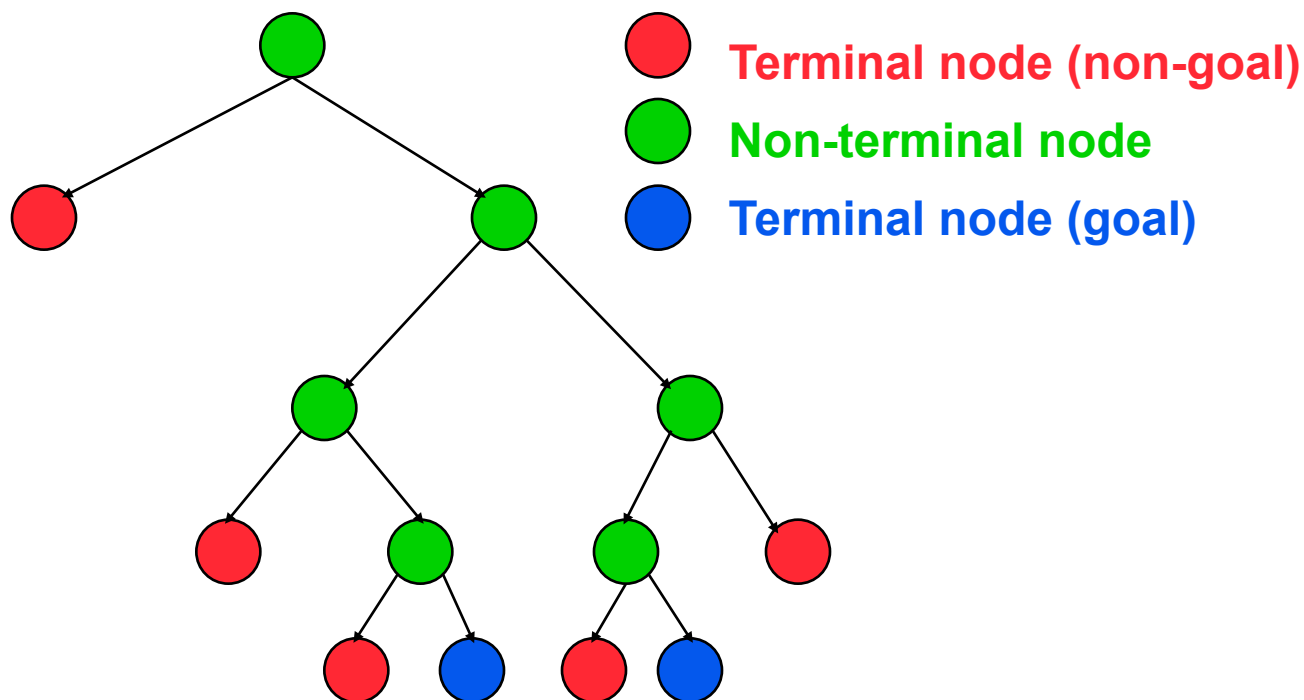
Search

- **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
 - 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
- Tree graph cannot be precomputed and has ordering constraints (unlike graphs or meshes in previous lectures)



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 uses a lower bound estimate of cost-to-solution as the bound
- **Breadth-first search (BFS)**
 - Search across a given level in the tree

Depth vs Breadth First Search

- **DFS with Explicit Stack**

- **Put root into Stack**

- Stack is data structure where items added to and removed from the top only

- **While Stack not empty**

- If node on top of Stack satisfies goal of search, return result, else
 - Mark node on top of Stack as “searched”
 - If top of Stack has an unsearched child, put child on top of Stack, else remove top of Stack

- **BFS with Explicit Queue**

- **Put root into Queue**

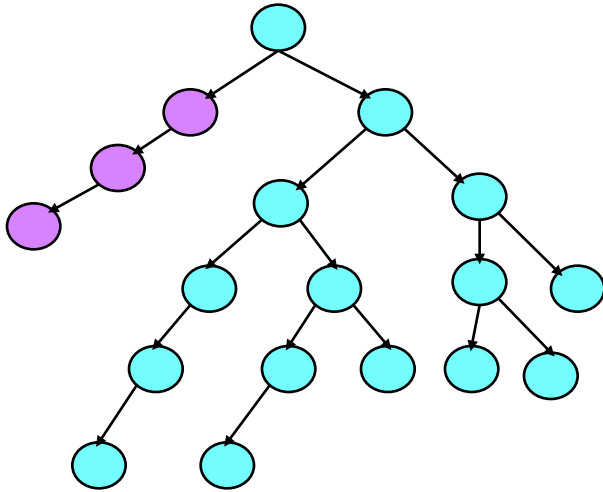
- Queue is data structure where items added to end, removed from front

- **While Queue not empty**

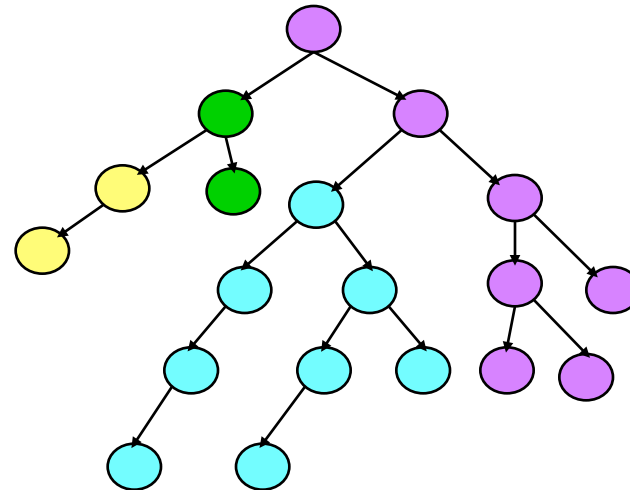
- If node at front of Queue satisfies goal of search, return result, else
 - Mark node at front of Queue as “searched”
 - If node at front of Queue has any unsearched children, put them all at end of Queue
 - Remove node at front from Queue

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

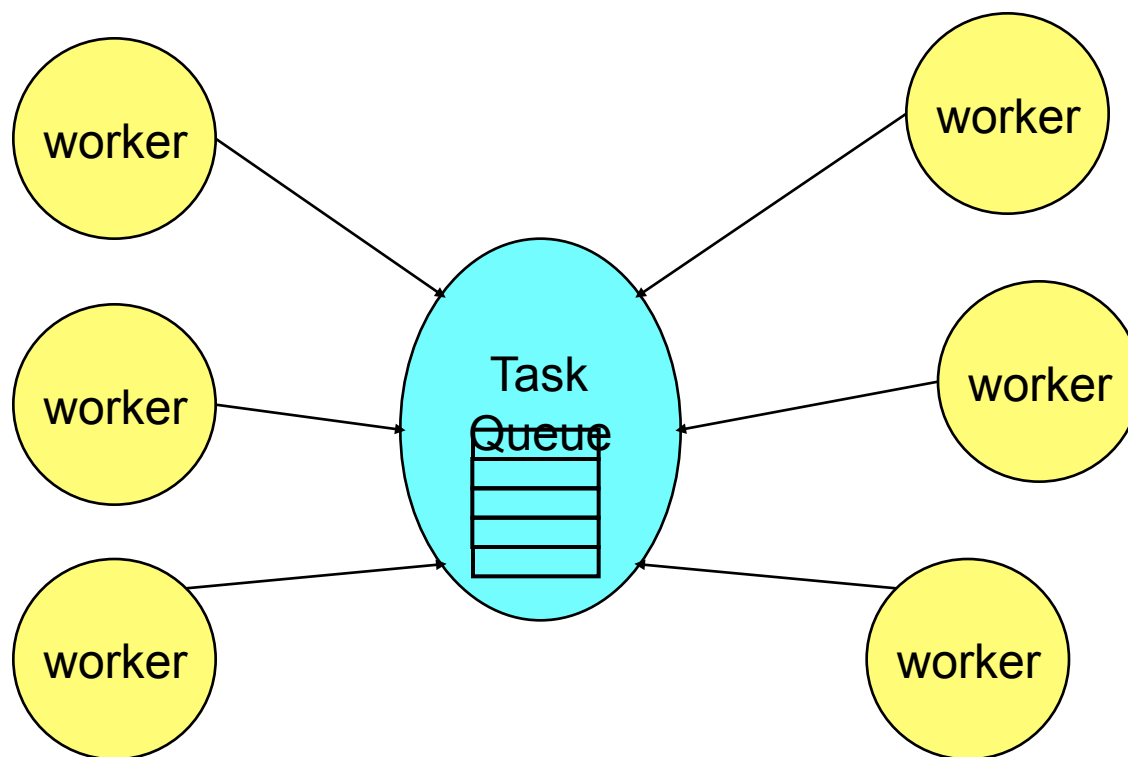


Load balance on 4 processors

- **We can and should do better than this ...**

Centralized Scheduling

- **Keep a queue of tasks 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 (or 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
 - Too much contention at shared queue
- 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)
- (at least) Four Variations:
 1. Use a fixed chunk size
 2. Guided self-scheduling
 3. Tapering
 4. Weighted Factoring

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^{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 \Rightarrow small chunk size should be used
 - low variance \Rightarrow 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

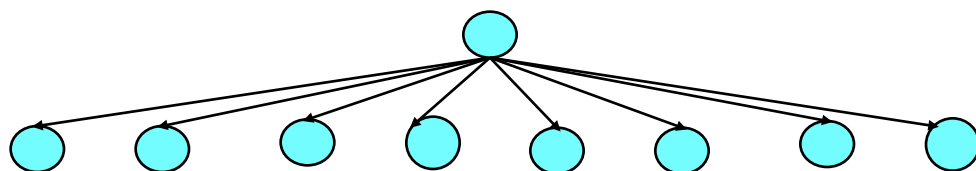
Variation 4: Weighted Factoring

- **Idea: similar to self-scheduling, but divide task cost by computational power of requesting node**
- **Useful for heterogeneous systems**
- **Also useful for shared resource clusters, 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



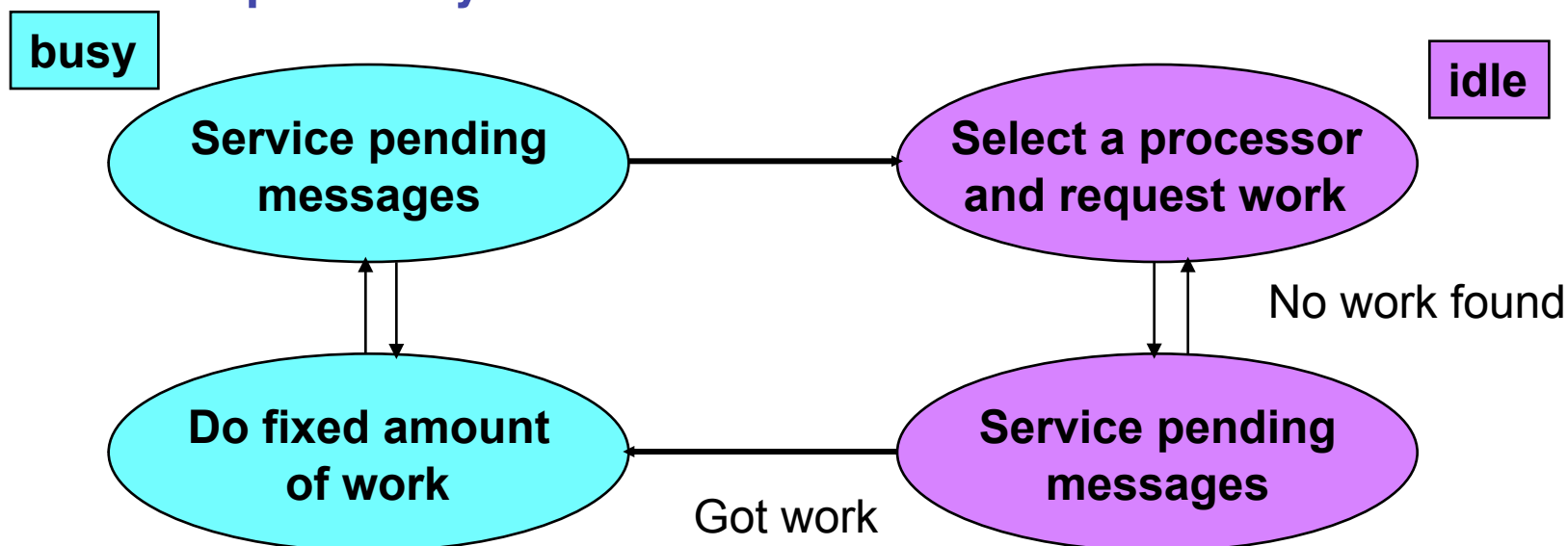
- **The cost of each task is unknown**
- **Locality is not important**
- **Shared memory machine, or at least number of processors is small – centralization is OK**

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
 - Idle processors can “pull” work, or busy processors “push” work
- **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, ...
- **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

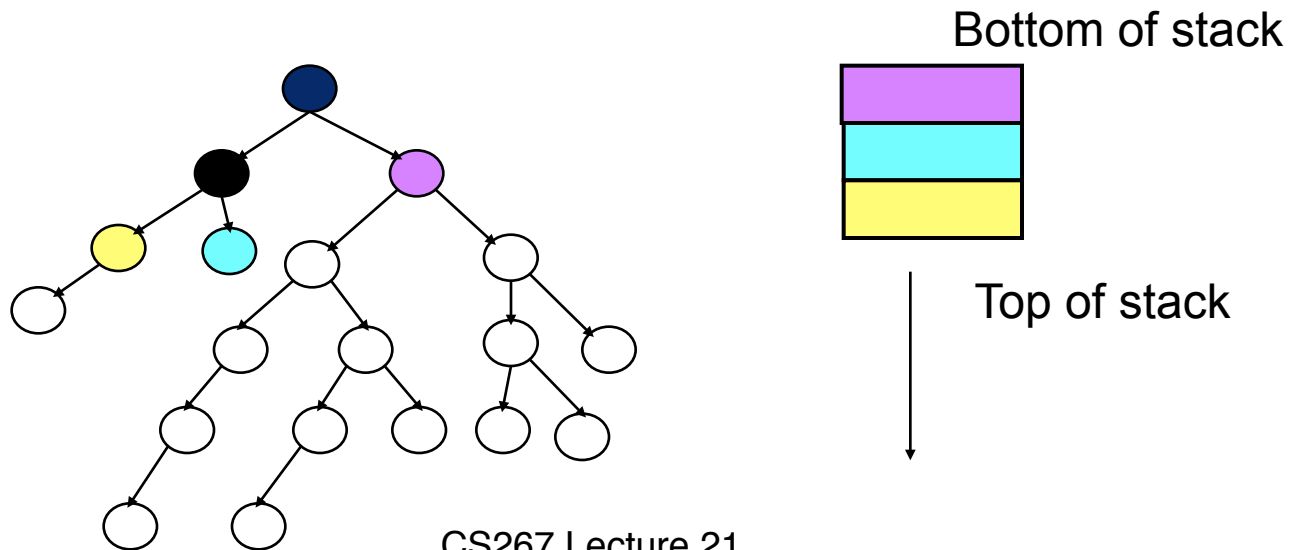


How to Select a Donor Processor

- **Three basic techniques:**
 1. **Asynchronous round robin**
 - Each processor k , keeps a variable “target $_k$ ”
 - When a processor runs out of work, requests work from target $_k$
 - Set $\text{target}_k = (\text{target}_k + 1) \bmod \text{procs}$
 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 $\text{target} = (\text{target} + 1) \bmod \text{procs}$
 3. **Random polling/stealing**
 - When a processor needs work, select a random processor and request work from it
- **Repeat 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

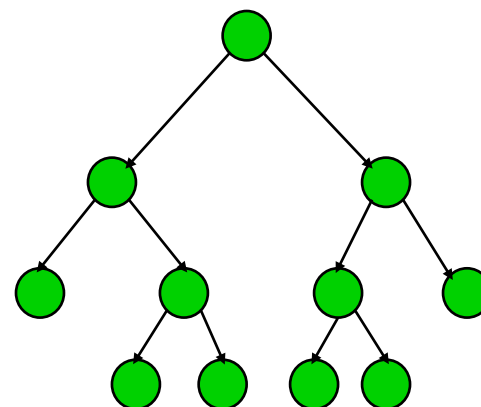


Theoretical Results (1)

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**
- **Multipol library (See C.-P. Wen, UCB PhD, 1996.)**
 - **Part of Multipol (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)**
 - **Space bound on task stealing**
- **X10 from IBM**

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
 - Analysis using Markov chains
- 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

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

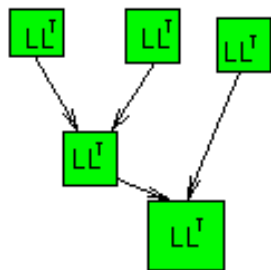
Appears in:

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

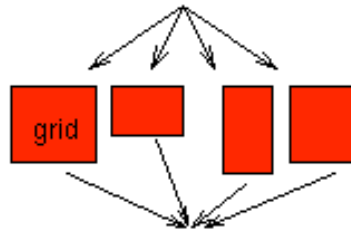
Mixed Parallelism Strategies

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

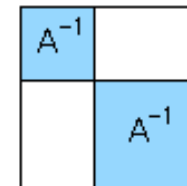
sparse cholesky



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?

Which Strategy to Use

Pure data parallelism

spread each block over all processors

Pure task parallelism

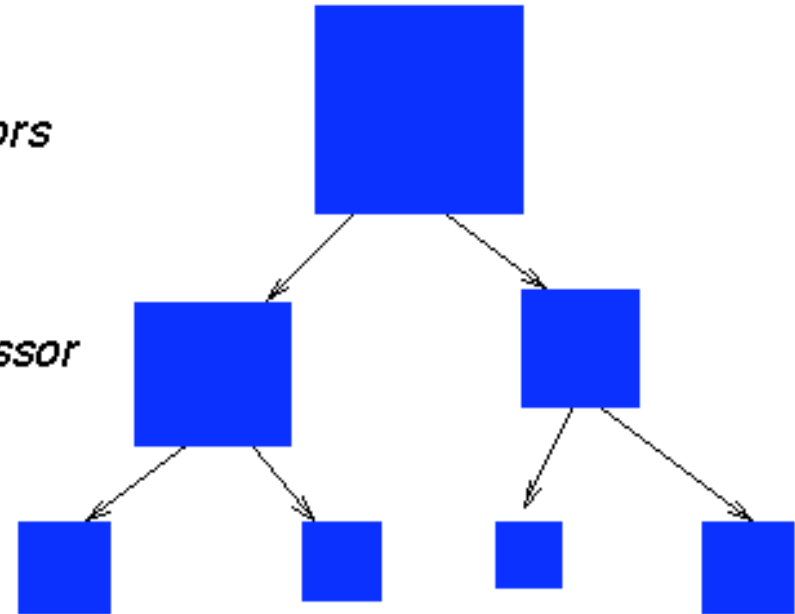
assign each block to a single processor

Switched parallelism

at some level, go from data to task

Mixed parallelism

spread blocks on subsets of processors



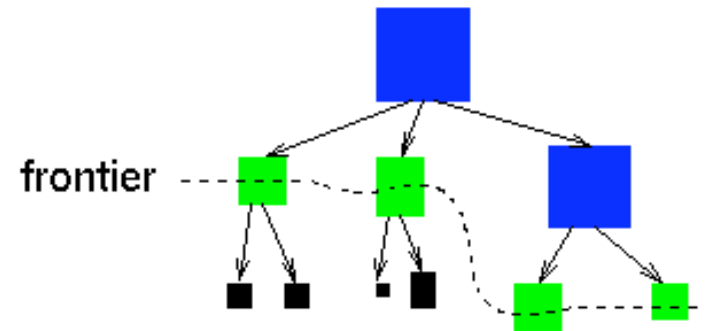
Modeling shows that switch parallelism gets almost all the benefit of mixed.

And easier to implement

Switch Parallelism: A Special Case

A Prefix-Suffix Heuristic

- * Sort the current frontier of tasks to be executed: $N_1 > N_2 > N_3 > \dots > N_I$
- * 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



List of CS267 Projects

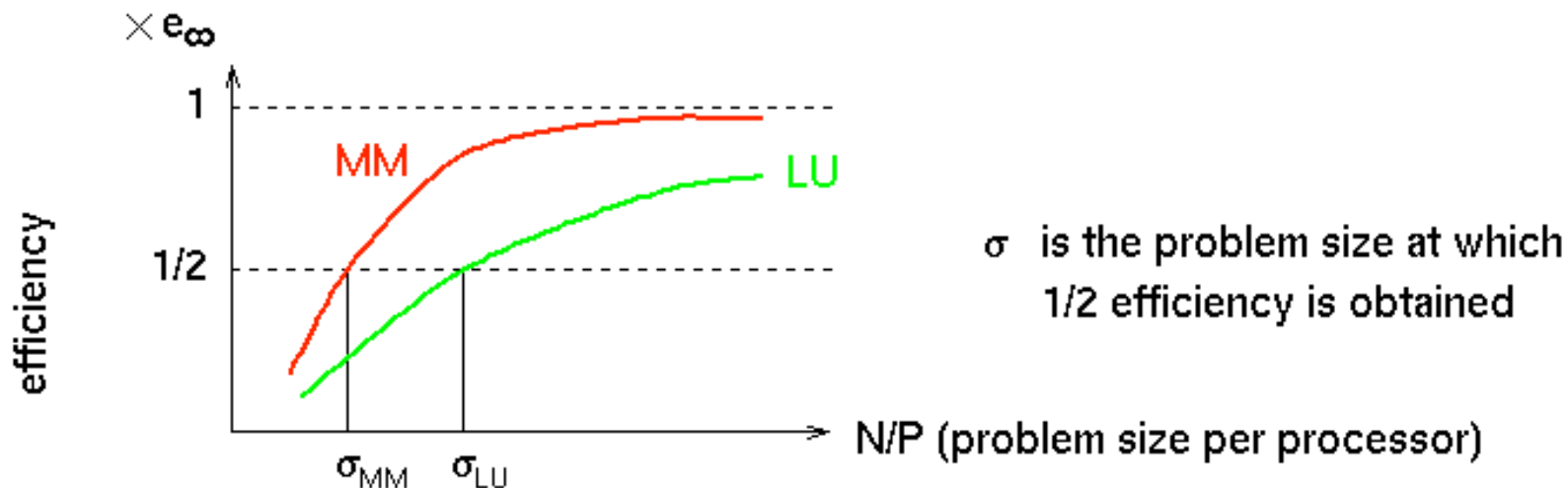
Proposed projects

- Jackie Leung, Isaac Liu, Jia Zou, Parallel Code Generation Framework for Synchronous Dataflow [NERSC systems]
- Bor-Yiing Su, (an 2 CS294 partners), Content-Based Image Retrieval on Parallel Map-Reduce [Nvidia GeForce 8800]
- Dan Bui, Evaluation of SSYTRD on NVIDIA GPU

Extra Slides

Simple Performance Model for Data Parallelism

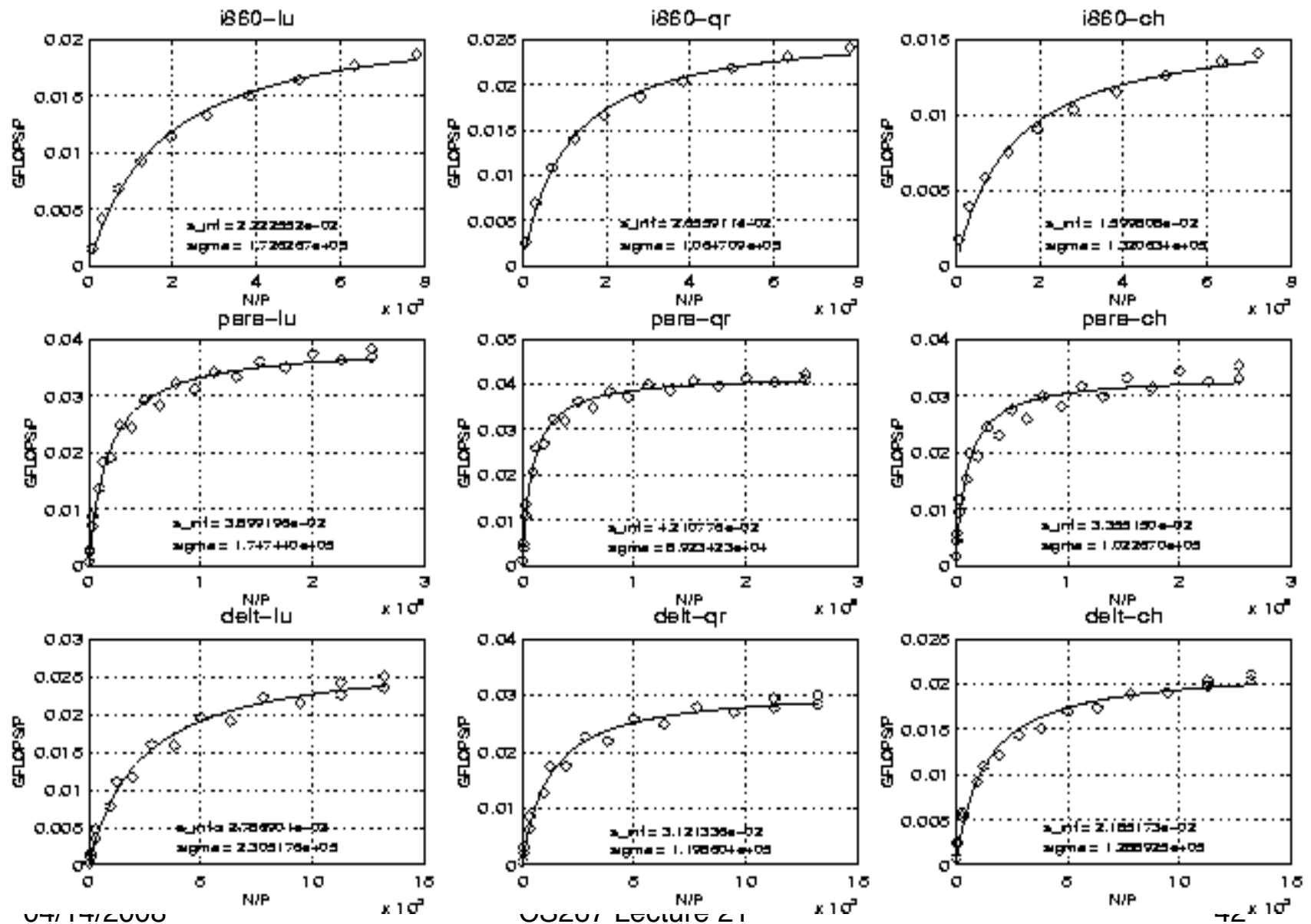
Observation: the efficiency of a data parallel algorithm depends on the problem size per processor, N/P , for sufficiently large N .



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

Validated against experimental data from ScaLAPACK for several algorithms

Model Validation from ScaLAPACK

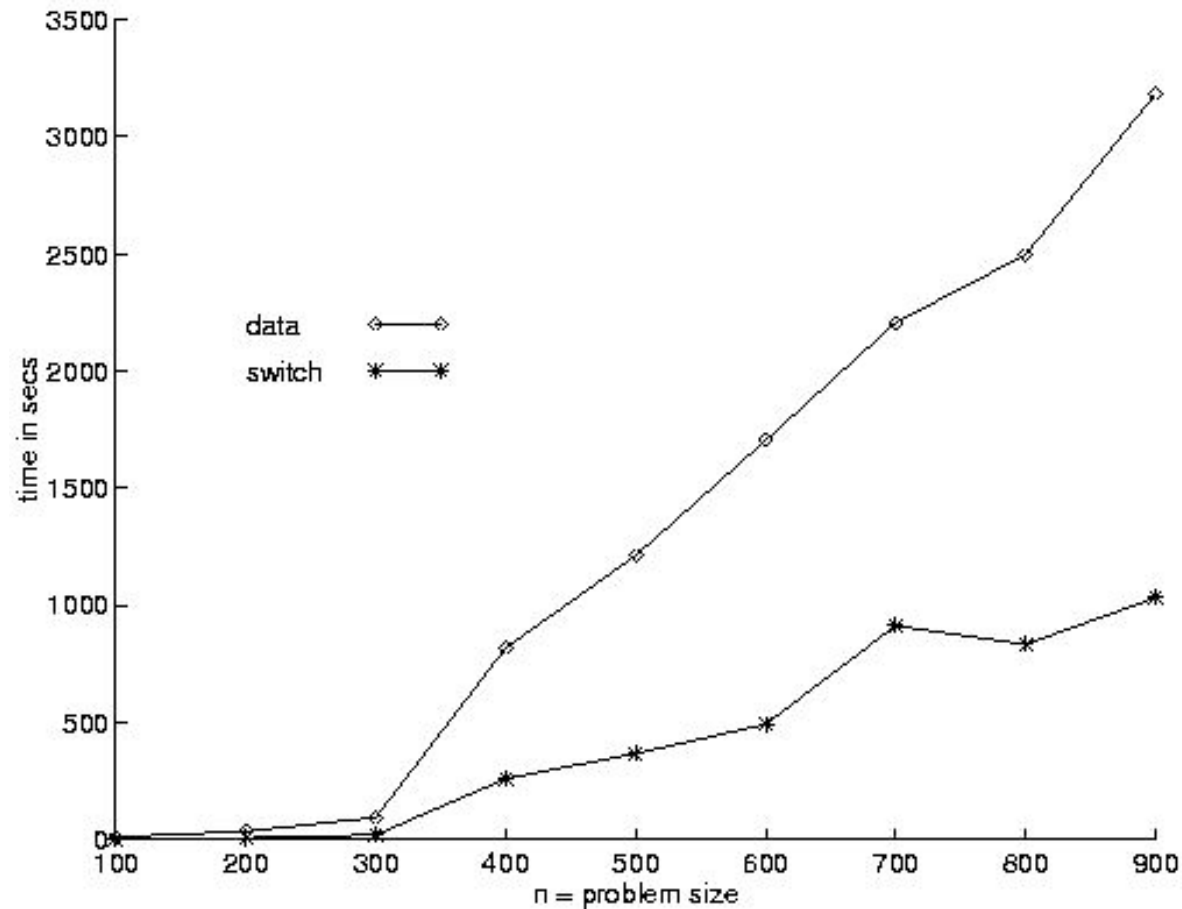


Modeling Performance

- **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**
 - $d=2$, $c=2$ (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
σP	14K	81K	200K	270K

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

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_{opt} , such that $f(x_{opt}) \leq f(x)$ 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

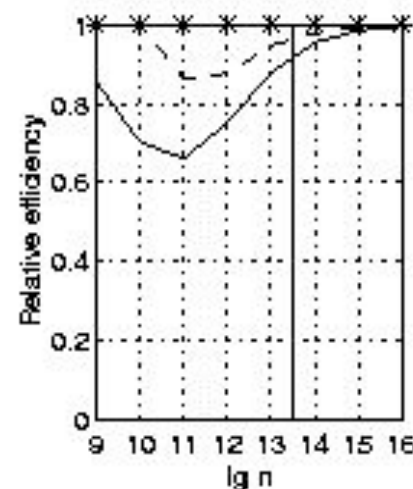
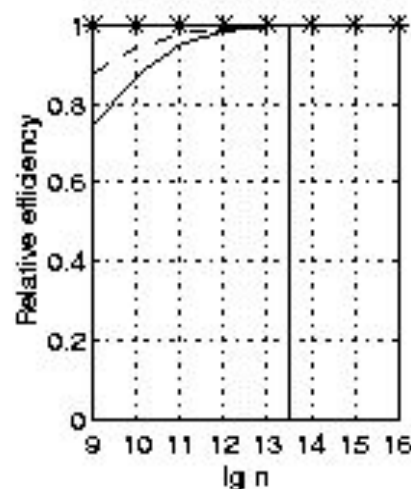
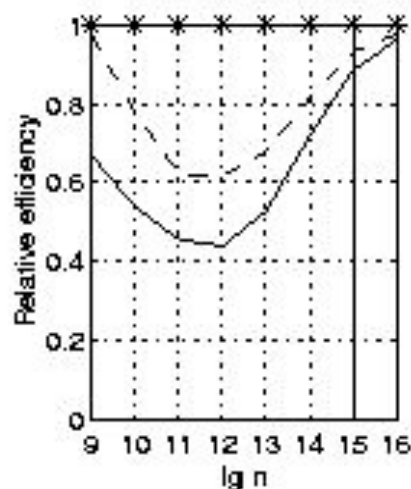
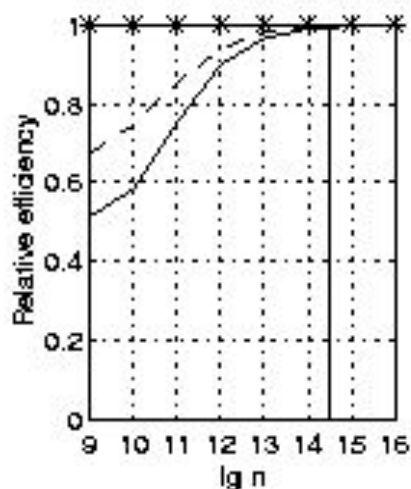
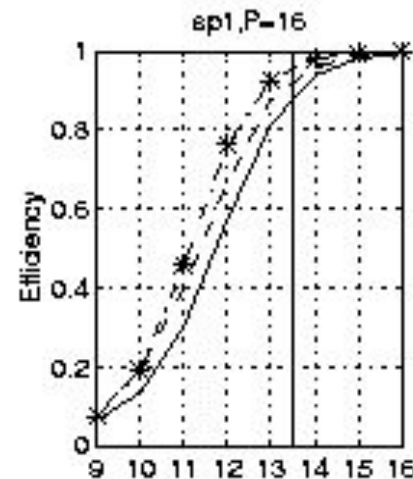
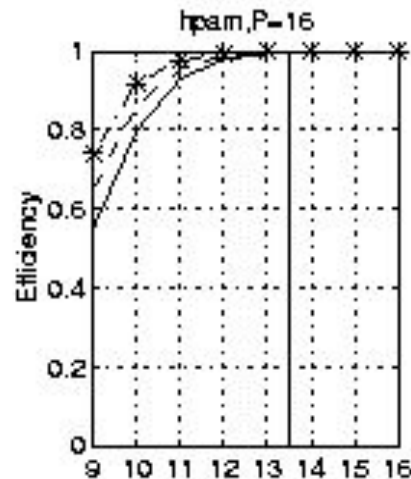
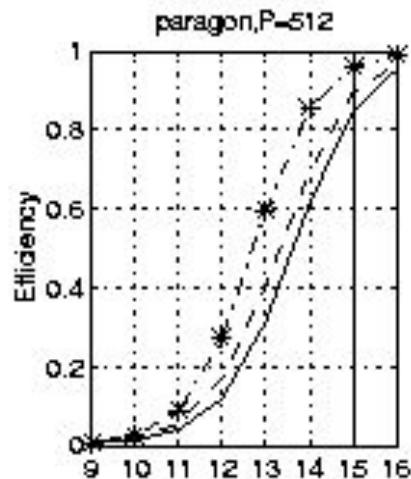
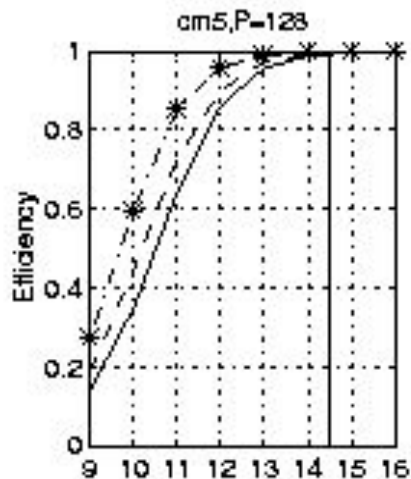
- Rather than searching to the bottom, keep set of current states in the space
- Pick the “best” one (by some heuristic) for the next step
- Use lower bound $l(x)$ as heuristic
 - $l(x) = g(x) + h(x)$
 - $g(x)$ is the cost of reaching the current state
 - $h(x)$ is a heuristic for the cost of reaching 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

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

