Load balancing

Prof. Richard Vuduc

Georgia Institute of Technology

CSE/CS 8803 PNA: Parallel Numerical Algorithms

[L.26] Thursday, April 17, 2008

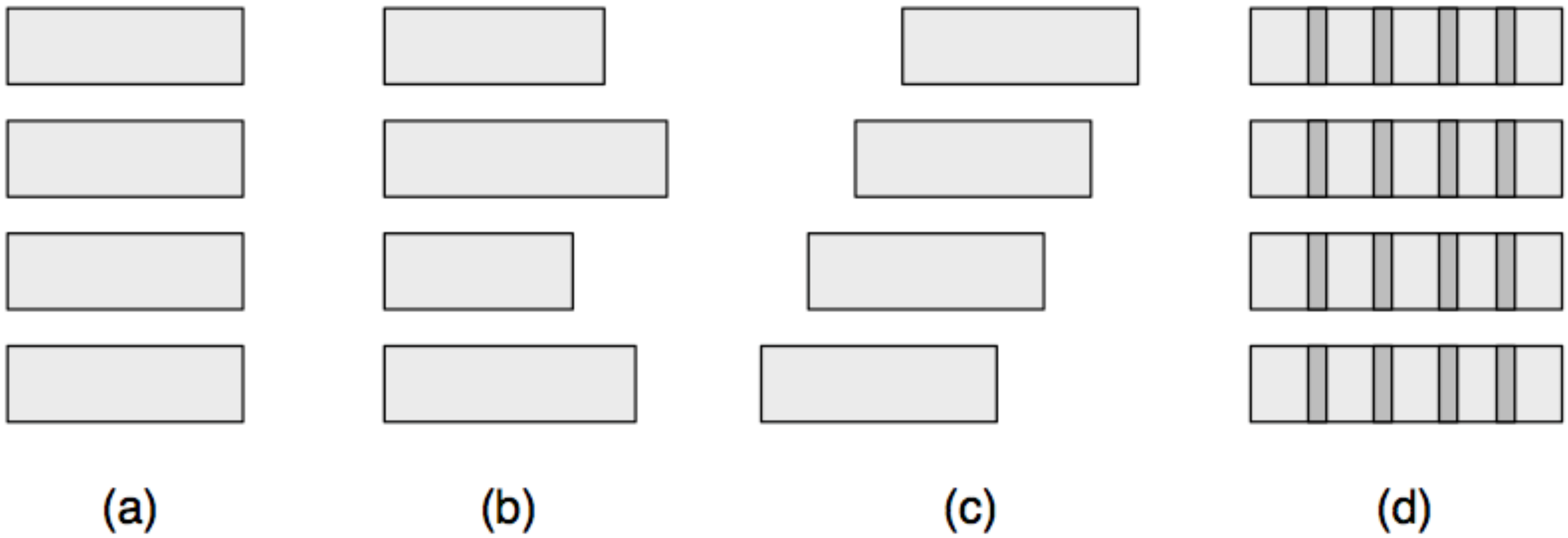
Today's sources

- CS 194/267 at UCB (Yelick/Demmel)
- "Intro to parallel computing" by Grama, Gupta, Karypis, & Kumar

Sources of inefficiency in parallel programs

- Poor single processor performance; *e.g.*, memory system
- Overheads; *e.g.*, thread creation, synchronization, communication
- Load imbalance
  - Unbalanced work / processor
  - Heterogeneous processors and/or other resources

Parallel efficiency: 4 scenarios

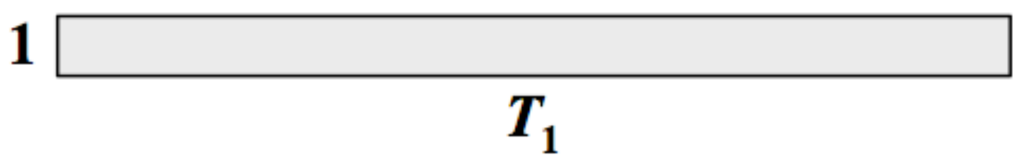


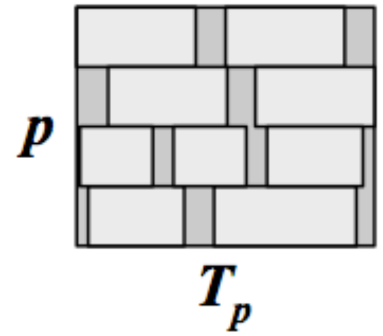
Consider **load balance**, **concurrency**, and **overhead**

Recognizing inefficiency

- **Cost** = (no. procs) * (execution time)

$$C_1 \equiv T_1 \qquad C_p \equiv p \cdot T_p = \frac{W_p}{V\left(\frac{M}{p}\right)}$$





Tools: VAMPIR, ParaProf (TAU), Paradyn, HPCToolkit (serial) …

Sources of “irregular” parallelism

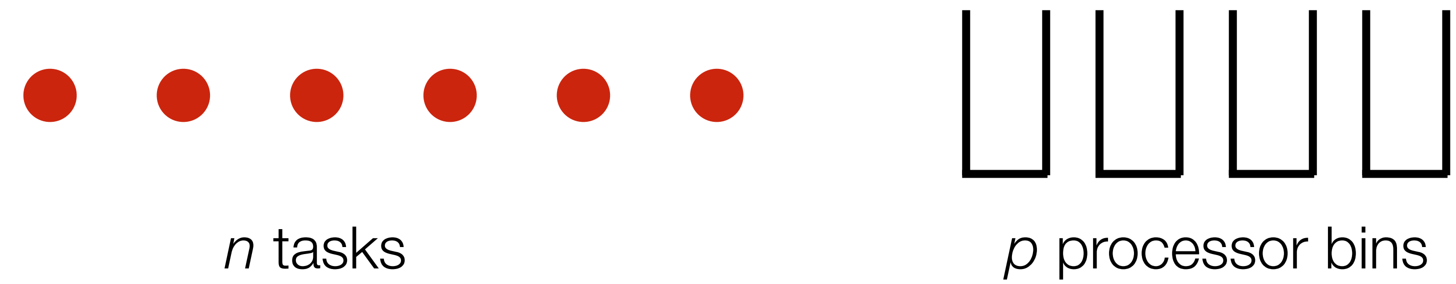
- Hierarchical parallelism, *e.g.*, adaptive mesh refinement
- Divide-and-conquer parallelism, *e.g.*, sorting
- Branch-and-bound search
  - Example: Game tree search
  - Challenge: Work depends on computed values
- Discrete-event simulation

Major issues in load balancing

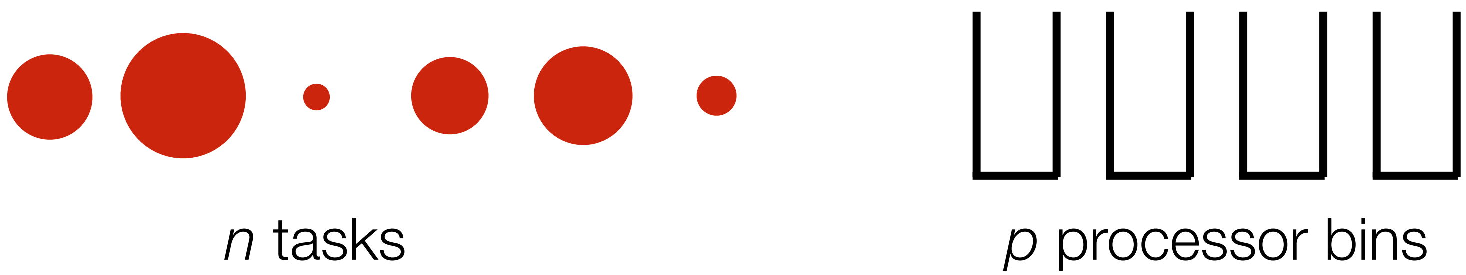
- **Task costs**: How much?
- **Dependencies**: How to sequence tasks?
- **Locality**: How does data or information flow?
- **Heterogeneity**: Do processors operate at same or different speeds?
- Common question: **When** is information known?
- Answers ⇒ Spectrum of load balancing techniques

Task costs

Easy: **Equal** costs



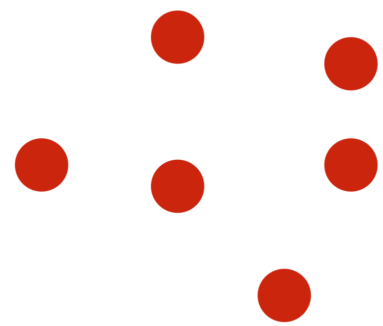
Harder: **Different, but known** costs.



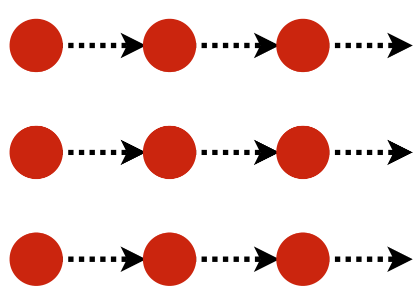
Hardest: **Unknown** costs.

Dependencies

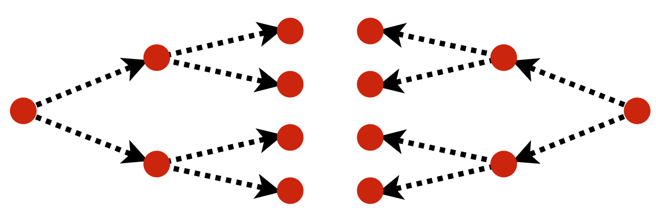
Easy: **None**



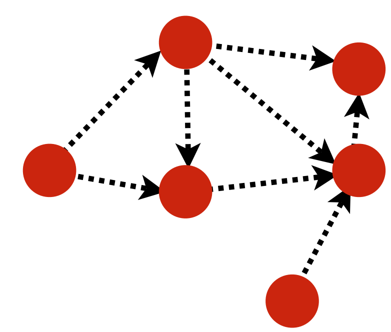
Harder: **Predictable** structure.



Wave-front



Trees
(balanced or unbalanced)

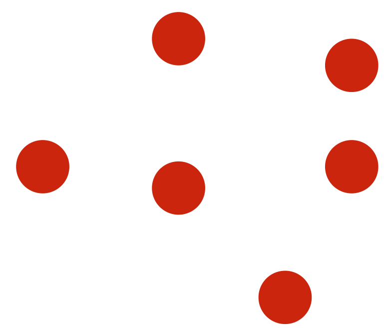


General DAG

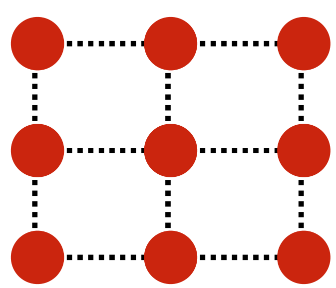
Hardest: **Dynamically evolving** structure.

Locality (communication)

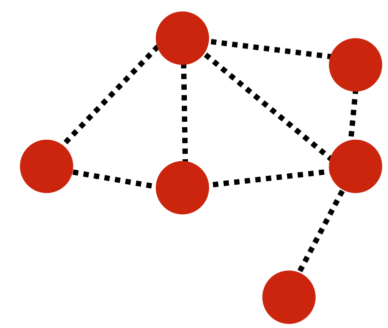
Easy: **No communication**



Harder: **Predictable** communication pattern.



Regular



Irregular

Hardest: **Unpredictable** pattern.

When information known ⇒ spectrum of scheduling solutions

- **Static**: Everything known in advance ⇒ off-line algorithms
- **Semi-static**
  - Information known at well-defined points, *e.g.*, start-up, start of time-step
  - ⇒ Off-line algorithm between major steps
- **Dynamic**
  - Information known in mid-execution
  - ⇒ On-line algorithms

Dynamic load balancing

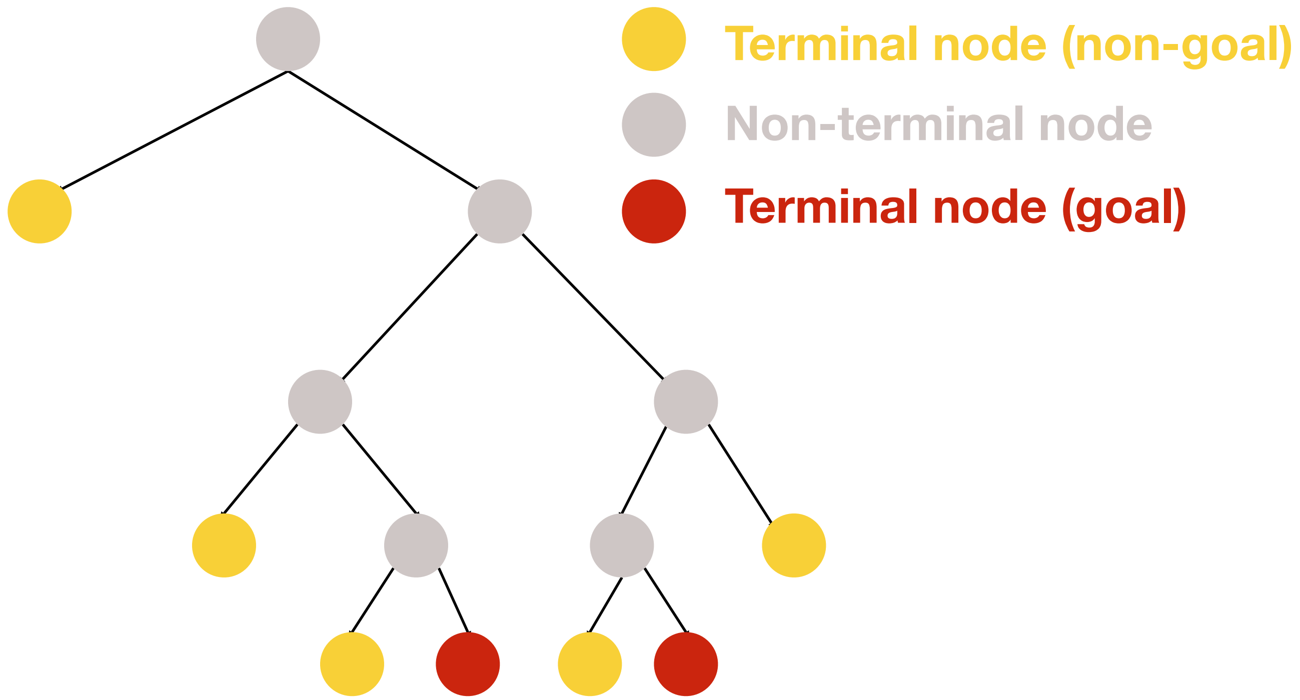
- Motivating example: Search algorithms
- Techniques: Centralized vs. distributed

Motivating example: Search problems

- Optimal layout of VLSI chips
- Robot motion planning
- Chess and other games
- Constructing a phylogeny tree from a set of genes

Example: Tree search

- Search tree unfolds dynamically
- May be a graph if there are common sub-problems

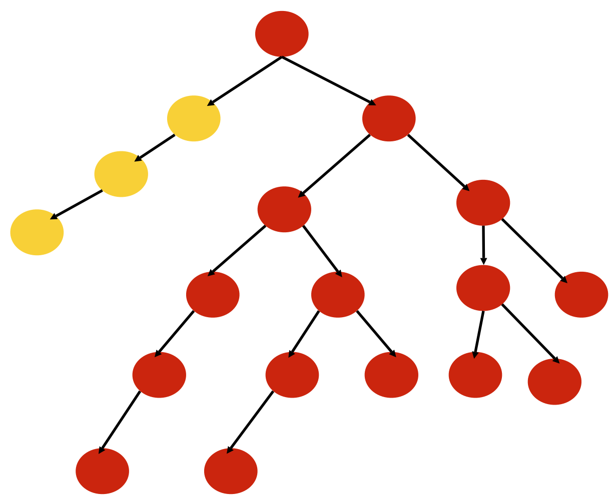


Search algorithms

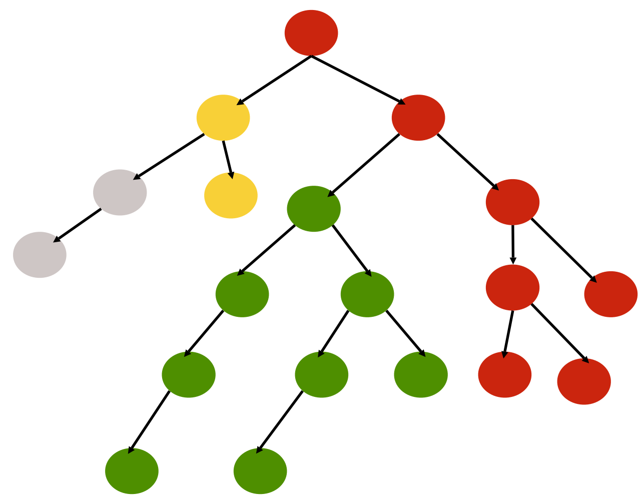
- Depth-first search
  - Simple back-tracking
  - Branch-and-bound
    - Track best solution so far (“bound”)
    - Prune subtrees guaranteed to be worse than bound
  - Iterative deepening: DFS w/ bounded depth; repeatedly increase bound
- Breadth-first search

Parallel search example: Simple back-tracking DFS

- A static approach: Spawn each new task on an idle processor



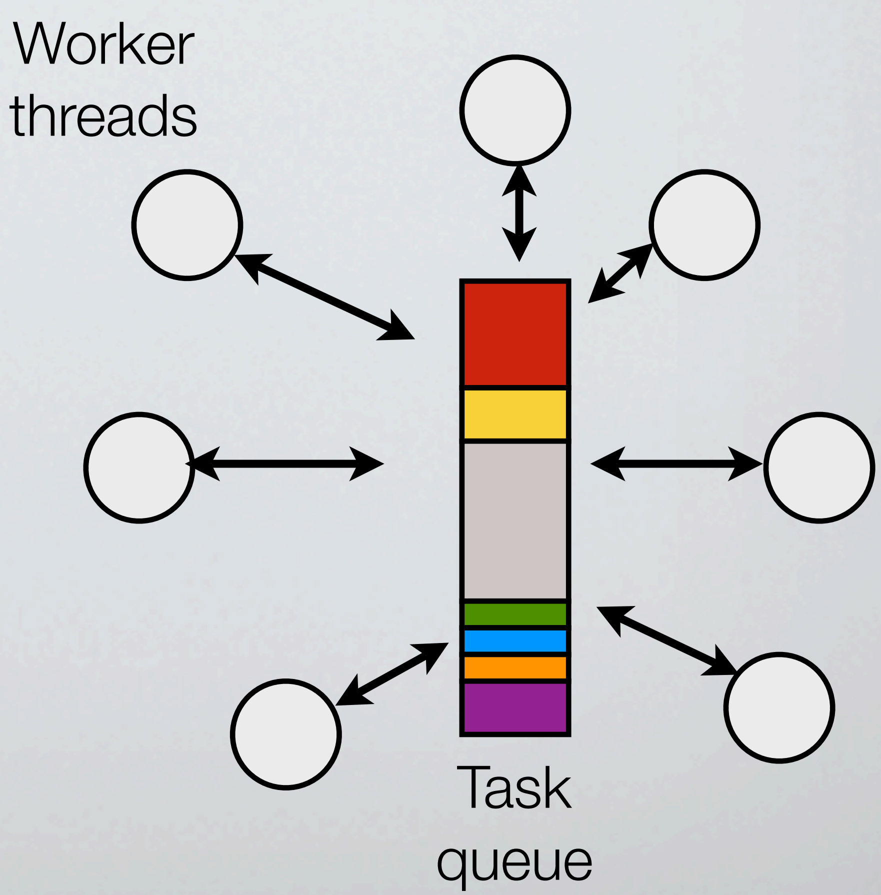
2 processors



4 processors

Centralized scheduling

- Maintain shared task queue
  - Dynamic, on-line approach
  - Good for small no. of workers
  - Independent tasks, known
- For loops: **Self-scheduling**
  - Task = subset of iterations
  - Loop body has unpredictable time
  - Tang & Yew (ICPP ’86)



Self-scheduling trade-off

- Unit of work to grab: balance vs. contention
- Some variations:
  - Grab fixed size chunk
  - Guided self-scheduling
  - Tapering
  - Weighted factoring

Variation 1: Fixed chunk size

- Kruskal and Weiss (1985) give a model for computing optimal chunk size
  - Independent subtasks
  - Assumed distributions of running time for each subtask (*e.g.*, IFR)
  - Overhead for extracting task, also random
- Limitations
  - Must know distributions
  - However, ‘n / p’ does OK (~ .8 optimal for large n/p)
- Ref: “Allocating independent subtasks on parallel processors”

Variation 2: Guided self-scheduling

- Idea
  - Large chunks at first to avoid overhead
  - Small chunks near the end to even-out finish times
  - Chunk size $K_i = ceil(R_i / p)$, R_i = # of remaining tasks
- Polychronopoulos & Kuck (1987): "Guided self-scheduling: A practical scheduling scheme for parallel supercomputers"

Variation 3: Tapering

- Idea
  - Chunk size $K_i = f(R_i;\ \mu, \sigma)$
  - (μ, σ) estimated using history
  - High-variance ⇒ small chunk size
  - Low-variance ⇒ larger chunks OK
- S. Lucco (1994), “Adaptive parallel programs.” PhD Thesis.
  - Better than guided self-scheduling, at least by a little

$$\begin{aligned} \kappa &= \text{min. chunk size} \\ h &= \text{selection overhead} \\ \Longrightarrow K_i &= f\left(\frac{\sigma}{\mu}, \kappa, \frac{R_i}{p}, h\right) \end{aligned}$$

Variation 4: Weighted factoring

- What if hardware is heterogeneous?
- Idea: Divide task cost by computational power of requesting node
- Ref: Hummel, Schmit, Uma, Wein (1996). “Load-sharing in heterogeneous systems using weighted factoring.” In *SPAA*

When self-scheduling is useful

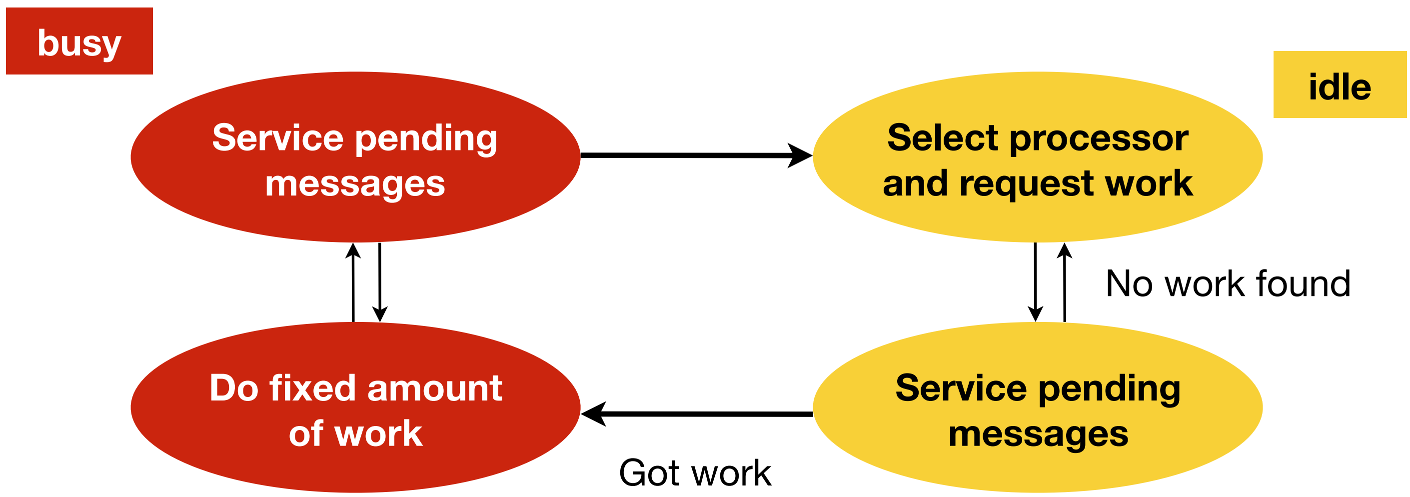
- Task cost unknown
- Locality not important
- Shared memory or “small” numbers of processors
- Tasks without dependencies; can use with, but most analysis ignores this

Distributed task queues

- Extending approach for distributed memory
  - Shared task queue → distributed task queue, or “bag”
  - Idle processors “pull” work, busy processors “push” work
- When to use?
  - Distributed memory, or shared memory with high sync overhead, small tasks
  - Locality not important
  - Tasks known in advance; dependencies computed on-the-fly
  - Cost of tasks not known in advance

Distributed dynamic load balancing

- For a tree search
  - Processors search disjoint parts of the tree
  - Busy and idle processors exchange work
  - Communicate asynchronously

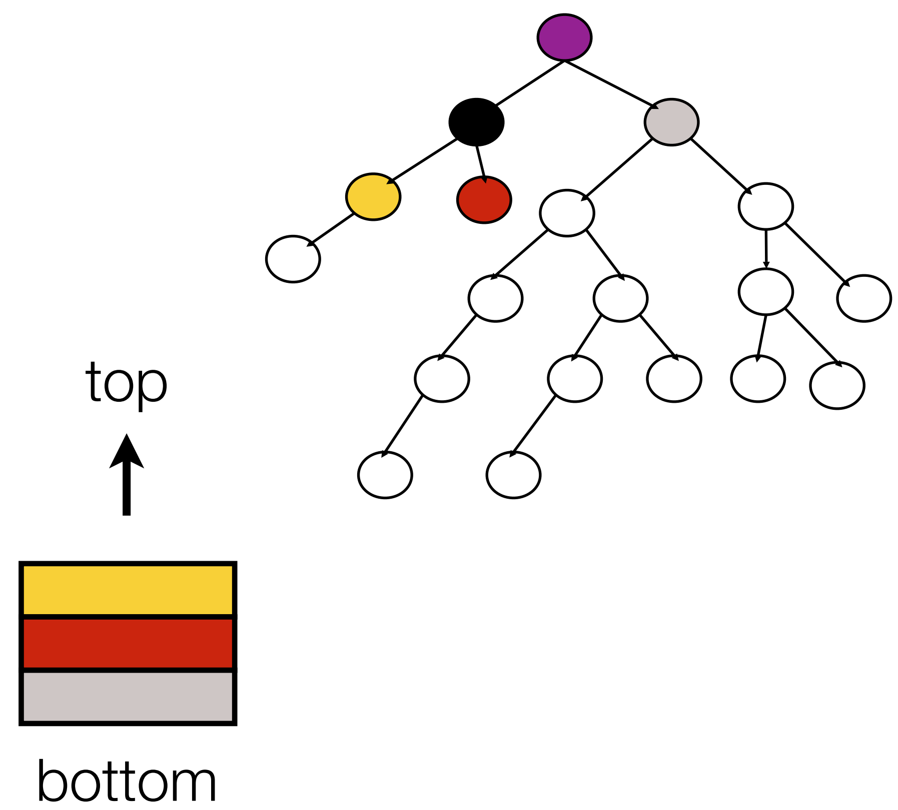


Selecting a donor processor: Basic techniques

- Asynchronous round-robin
  - Each processor k maintains $target_k$
  - When out of work, request from $target_k$ and update $target_k$
- Global round robin: Proc 0 maintains global “target” for all procs
- Random polling/stealing

How to split work?

- How many tasks to split off?
    - Total tasks unknown, unlike self-scheduling case
- Which tasks?
    - Send oldest tasks (stack bottom)
    - Execute most recent (top)
    - Other strategies?



A general analysis of parallel DFS

- Let *w* = work at some processor
  - Split into two parts:

$$0 < \rho < 1 : \qquad \rho \cdot w$$
$$(1 - \rho) \cdot w$$

  - Then:

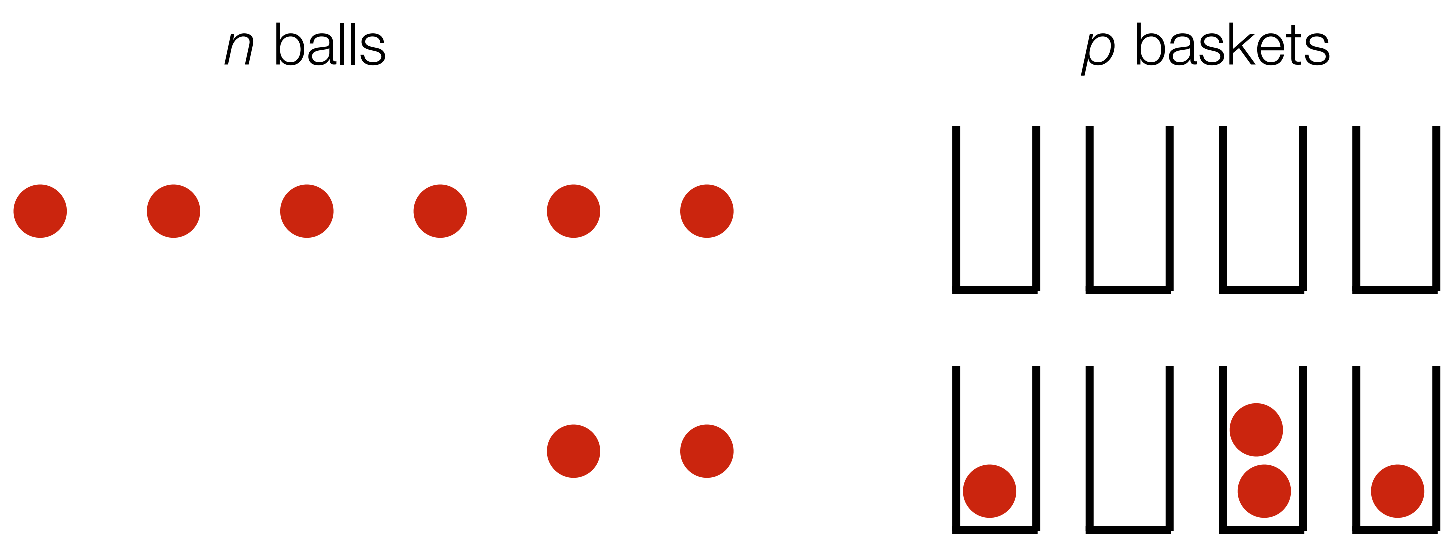
$$\exists\ \phi : \qquad 0 < \phi \leq \frac{1}{2}$$
$$\phi \cdot w < \rho \cdot w$$
$$\phi \cdot w < (1 - \rho) \cdot w$$

Each partition has at least ϕw work, or at most (1-ϕ)w.

A general analysis of parallel DFS

- If processor P_i initially has work w_i and receives request from P_j:
  - After splitting, P_i & P_j have at most $(1-\phi)w_i$ work.
- For some load balancing strategy, let $V(p)$ = no. of work requests after which each processor has received at least 1 work request [$\Rightarrow V(p) \geq p$]
- Initially, P_0 has W units of work, and all others have no work
- After $V(p)$ requests, max work < $(1-\phi)*W$
- After $2*V(p)$ requests, max work < $(1-\phi)^2*W$
- $\Rightarrow$ Total number of requests = $O\left(V\left(p\right)\log W\right)$

Computing $V(p)$ for random polling



- Consider randomly throwing balls into bins
- $V(p)$ = average number of trials needed to get at least 1 ball in each basket
- What is $V(p)$?

A general analysis of parallel DFS: Isoefficiency

- Asynchronous round-robin:

$$V(p) = O(p^2) \quad \Longrightarrow \quad W = O(p^2 \log p)$$

- Global round-robin:

$$W = O(p^2 \log p)$$

- Random:

$$W = O(p \log^2 p)$$

Theory: Randomized algorithm is optimal with high probability

- Karp & Zhang (1988) prove for tree with **equal-cost tasks**
  - “A randomized parallel branch-and-bound procedure” (JACM)
  - Parents must complete before children
  - Tree unfolds at run-time
  - Task number/priorities not known *a priori*
  - Children “pushed” to random processors

Theory: Randomized algorithm is optimal with high probability

- Blumofe & Leiserson (1994) prove for fixed task tree with **variable cost** tasks
  - Idea: **Work-stealing** – idle task pulls ("steals"), instead of pushing
  - Also bound total memory required
  - "Scheduling multithreaded computations by work stealing"
- Chakrabarti, Ranade, Yelick (1994) show for **dynamic tree** w/ variable tasks
  - Pushes instead of pulling ⇒ possibly worse locality
  - "Randomized load-balancing for tree-structured computation"

Diffusion-based load balancing

- Randomized schemes treat machine as fully connected
- **Diffusion-based** balancing accounts for topology
  - Better locality
  - “Slower”
  - Cost of tasks assumed known at creation time
  - No dependencies between tasks

Diffusion-based load balancing

- Model machine as graph
- At each step, compute weight of tasks remaining on each processor
- Each processor compares weight with neighbors and “averages”
- See: Ghosh, Muthukrishnan, Schultz (1996): “First- and second-order diffusive methods for rapid, coarse, distributed load balancing” (SPAA)

Summary

- Unpredictable loads → online algorithms
- Fixed set of tasks with unknown costs → self-scheduling
- Dynamically unfolding set of tasks → work stealing

- Other scenarios: What if…
  - locality is of paramount importance?
  - task graph is known in advance?

Administrivia

Final stretch…

- Project checkpoints due already

Locality considerations

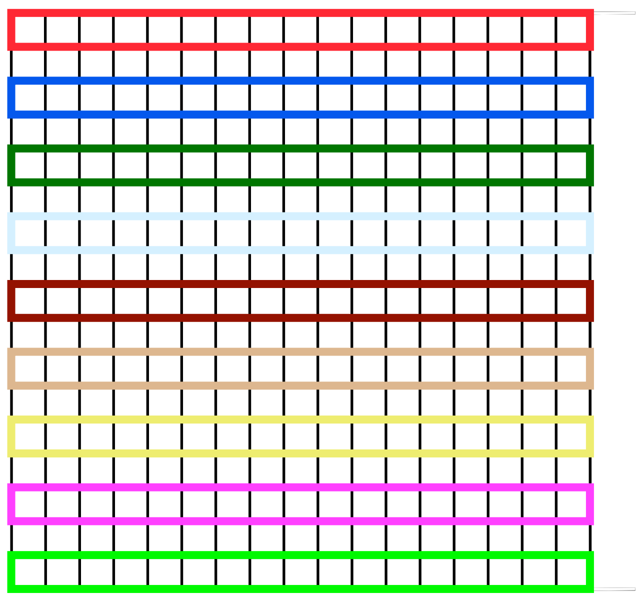
What if locality is important?

- Example scenarios
  - Bag of tasks that need to communicate
  - Arbitrary task graph, where tasks share data
- Need to run tasks on same or “nearby” processor

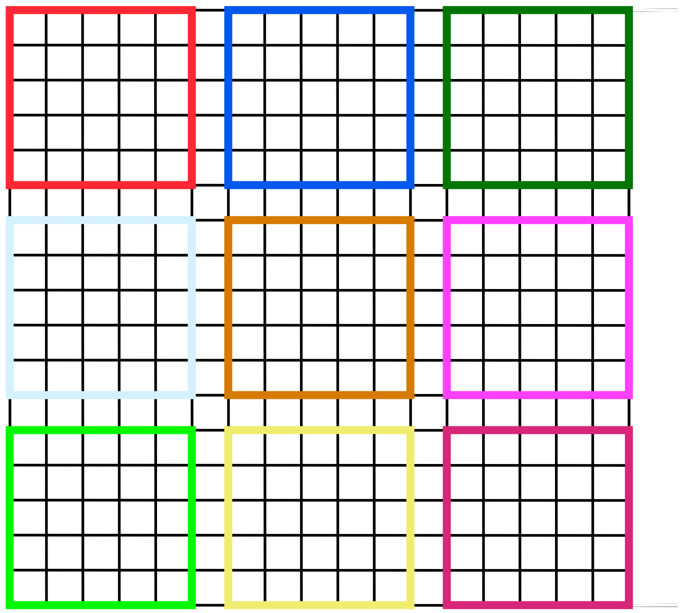
Stencil computation on a regular mesh

- Load balancing → equally sized partitions
- Locality → Minimize perimeter to minimize processor edge-crossings

$$n \times (p-1)$$



$$2 \times n \times (\sqrt{p}-1)$$



“In conclusion…”

Ideas apply broadly

- Physical sciences, *e.g.*,
  - Plasmas
  - Molecular dynamics
  - Electron-beam lithography device simulation
  - Fluid dynamics
- “Generalized” n-body problems: Talk to your classmate, Ryan Riegel

Backup slides