Introduction to Parallel and Distributed Computing

Selected Parallel Algorithms
Discrete Search

Lecture 11, Spring 2024

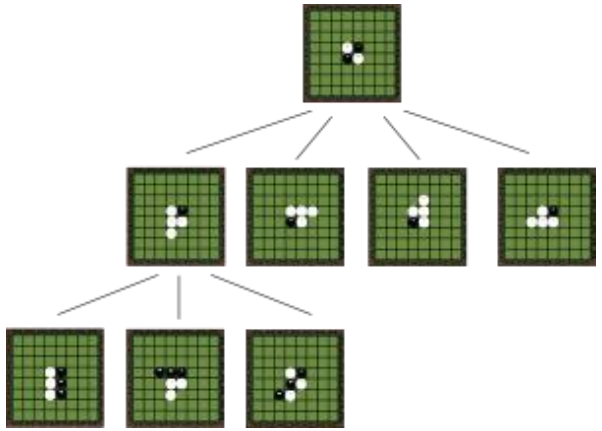
Instructor: 罗国杰

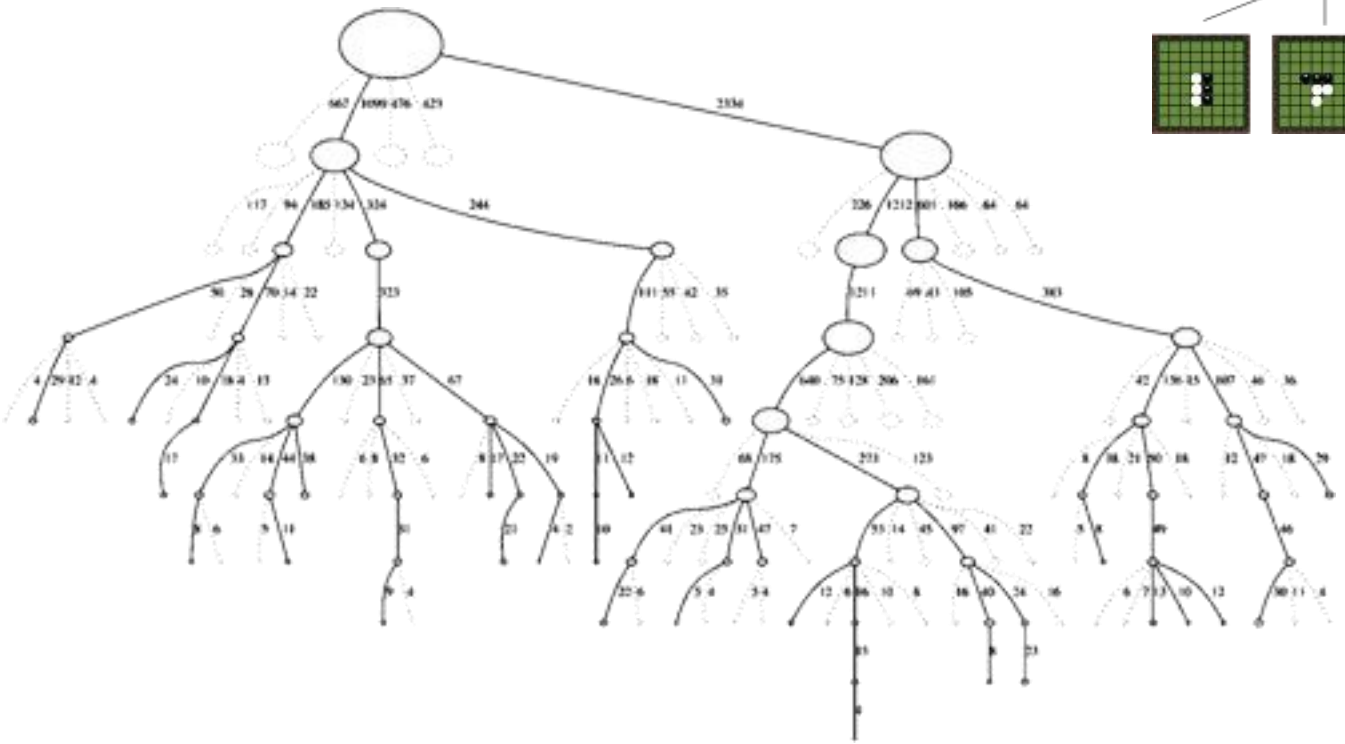
gluo@pku.edu.cn

Outline

- **Parallel discrete search**
  - Depth-first search
    - (Distribute/request work in a stack)
    - Work splitting
    - Load balancing
  - Best-first search
    - (Distribute/request work in a priority queue)
    - Communicating the lower bound value
  - Speedup anomalies

Example of Discrete Search



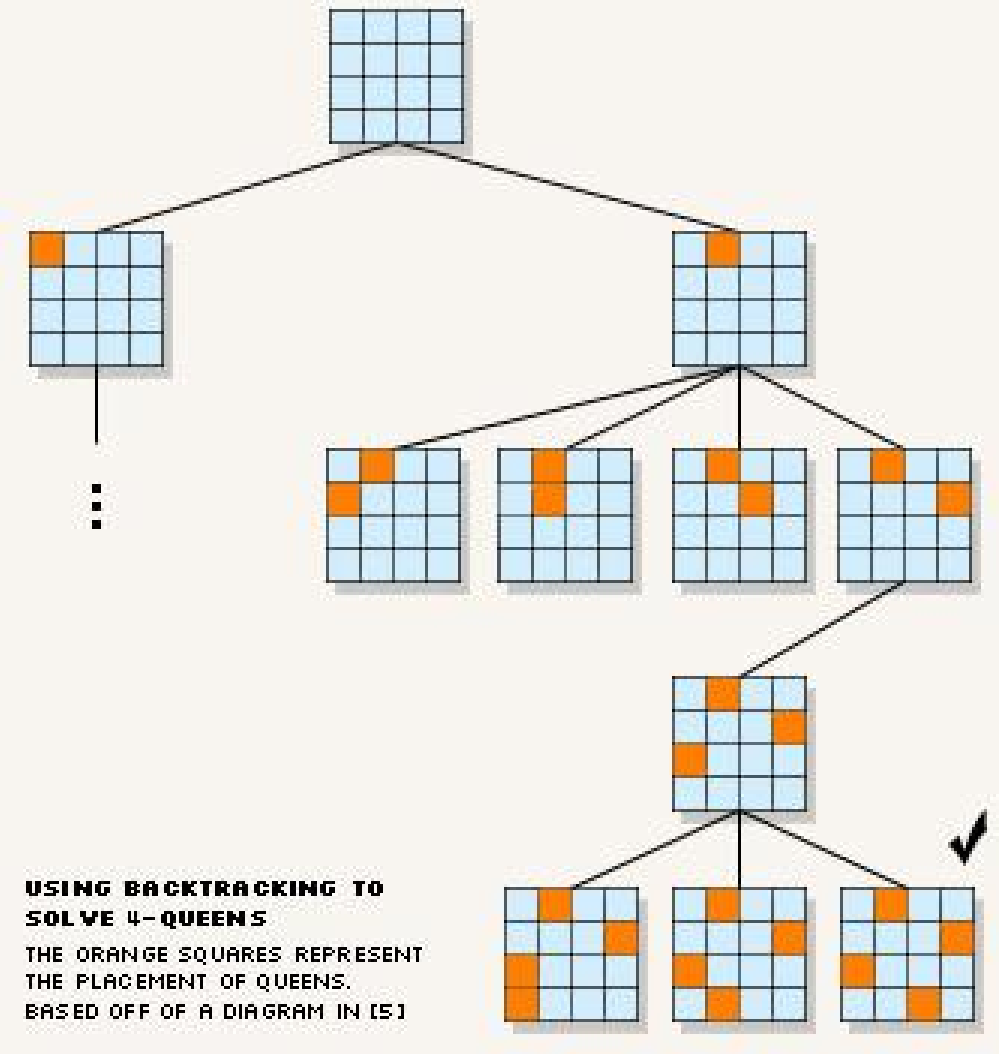


Depth-First Search

- **Uses depth-first search to consider alternative solutions to a combinatorial search problem**
- **Recursive algorithm**
- **Backtrack occurs when**
  - A node has no children ("dead end")
  - All of a node's children have been explored

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

DFS Example: 4-Queen & State Space Tree



Source: http://keiloassociates.com/res/Java-Backtracking-Example

Time and Space Complexity

- **Suppose average branching factor in state space tree is b**
  - Branching factor: the (average) number of children at each node
- **Searching a tree of depth k requires examining**

$$1+b+b^2+\cdots+b^k=\frac{b^{k+1}-b}{b-1}+1=\theta(b^k)$$

nodes in the worst case (exponential time)

- **Amount of memory required is $\Theta(k)$**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Parallel DFS: Motivation

- **Discrete optimizations are usually NP-hard problems. Does parallelism really help much?**
  - For many problems, the average-case runtime is polynomial.
  - Often, we can find suboptimal solutions in polynomial time.
  - Many problems have smaller state spaces but require real-time solutions.
  - For some other problems, an improvement in objective function is highly desirable, irrespective of time.

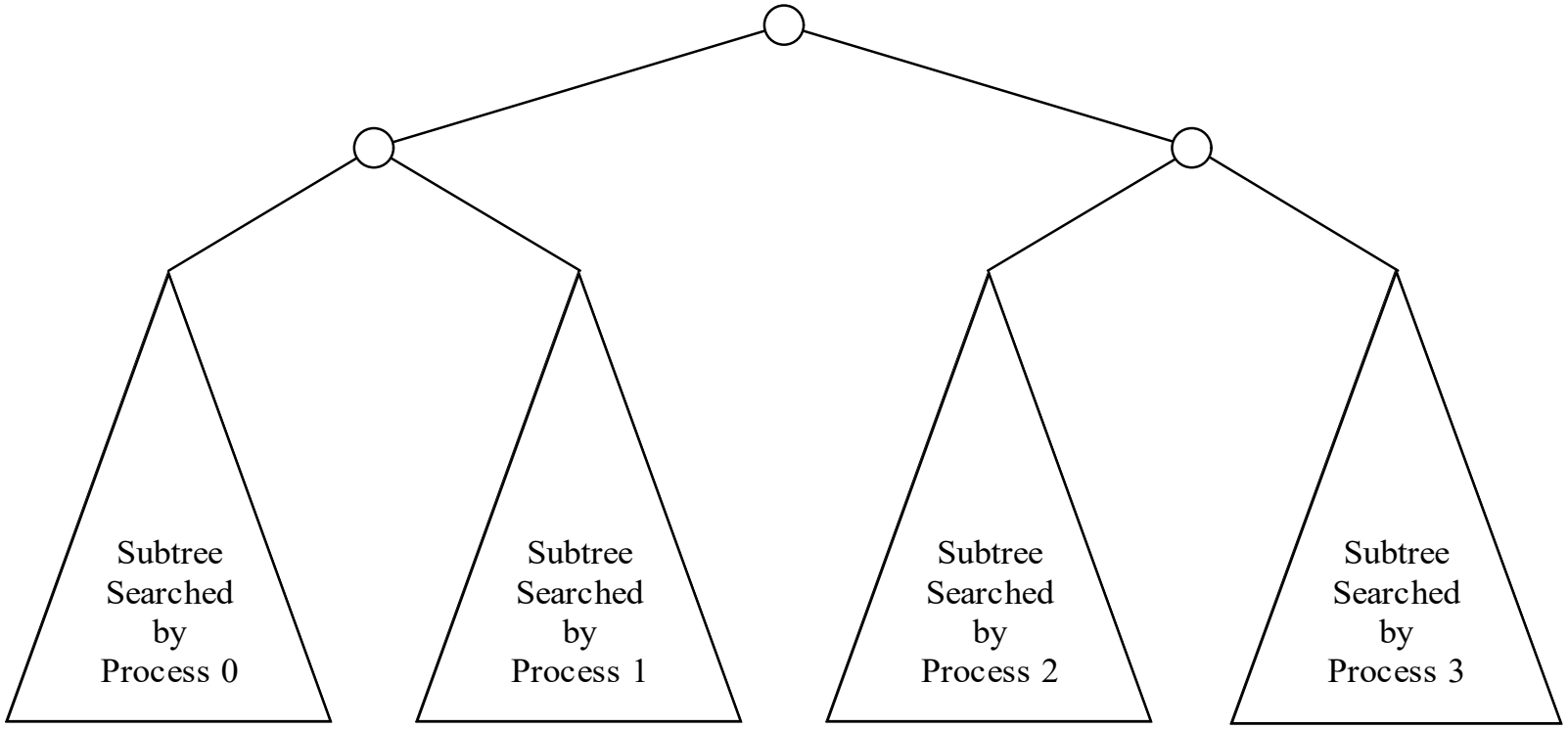
A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Parallel Depth-First Search

- **First strategy: give each processor a subtree**
- **Suppose** $p = b^k$
  - A process searches all nodes to depth k
  - It then explores only one of subtrees rooted at level k
  - If d (depth of search) $> 2k$, time required by each process to traverse first k levels of state space tree inconsequential

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Parallel Backtrack when $p = b^k$



Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

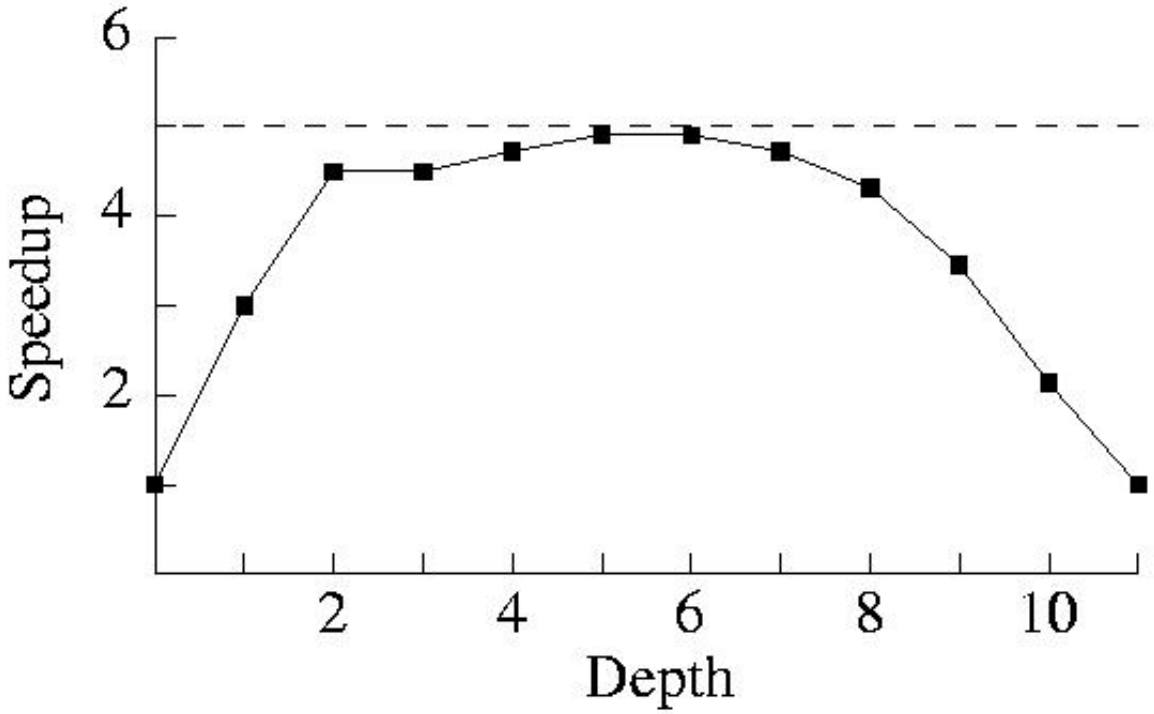
What If $p \neq b^k$?

- **A process can perform sequential search to level m of state space tree**
- **Each process explores its share of the subtrees rooted by nodes at level m**
  - As m increases, there are more subtrees to divide among processes, which can make workloads more balanced
  - Increasing m also increases number of redundant computations

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Maximum Speedup when $p \neq b^k$

In this example 5 processors are exploring a state space tree with branching factor 3 and depth 10.



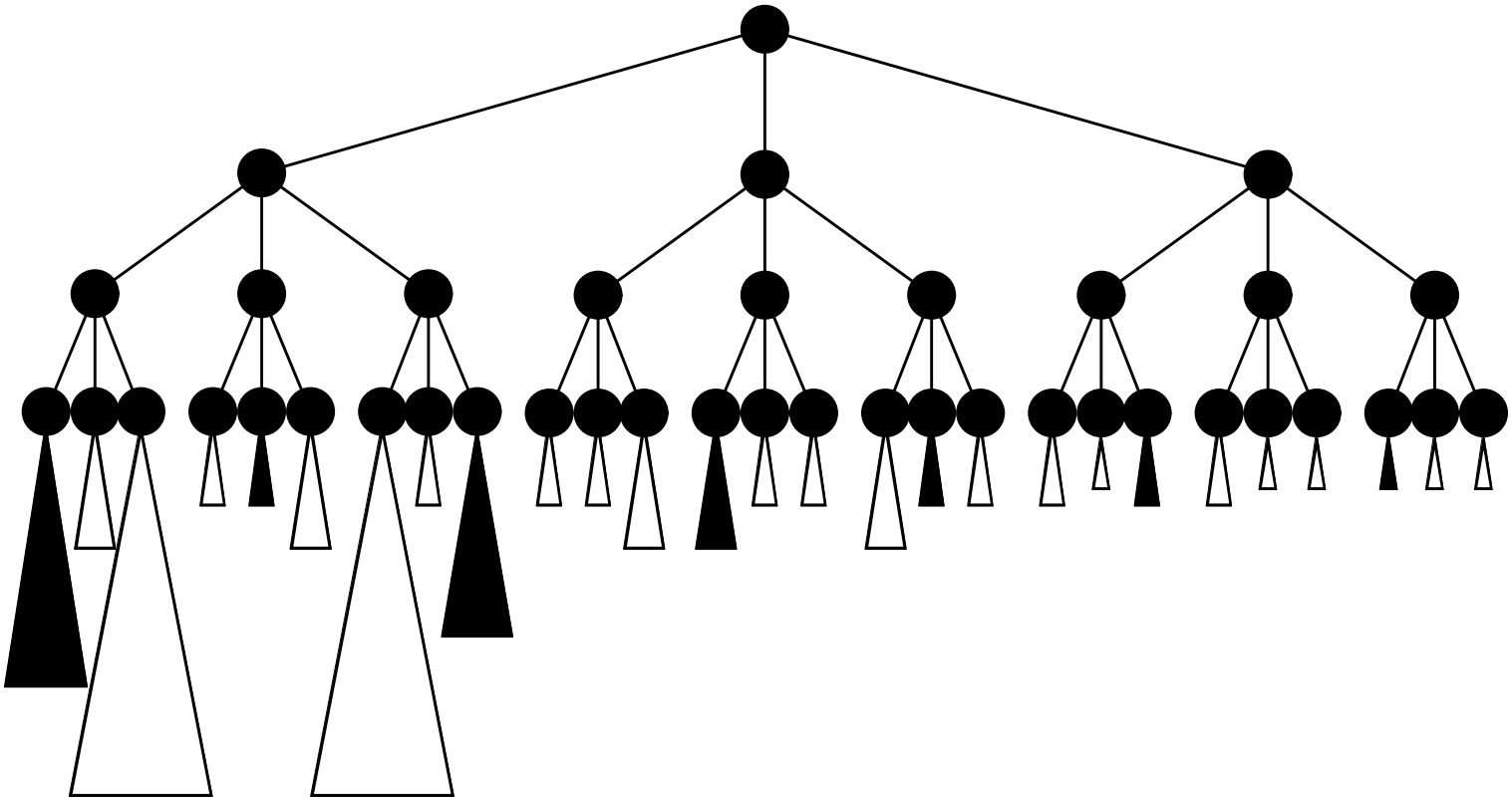
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Disadvantage of Allocating One Subtree per Process

- In most cases state space tree is not balanced
- Example: in N-queen problem, some position choices lead to dead ends quicker than others
- Alternative: make sequential search go deeper, so that each process handles many subtrees (cyclic allocation)

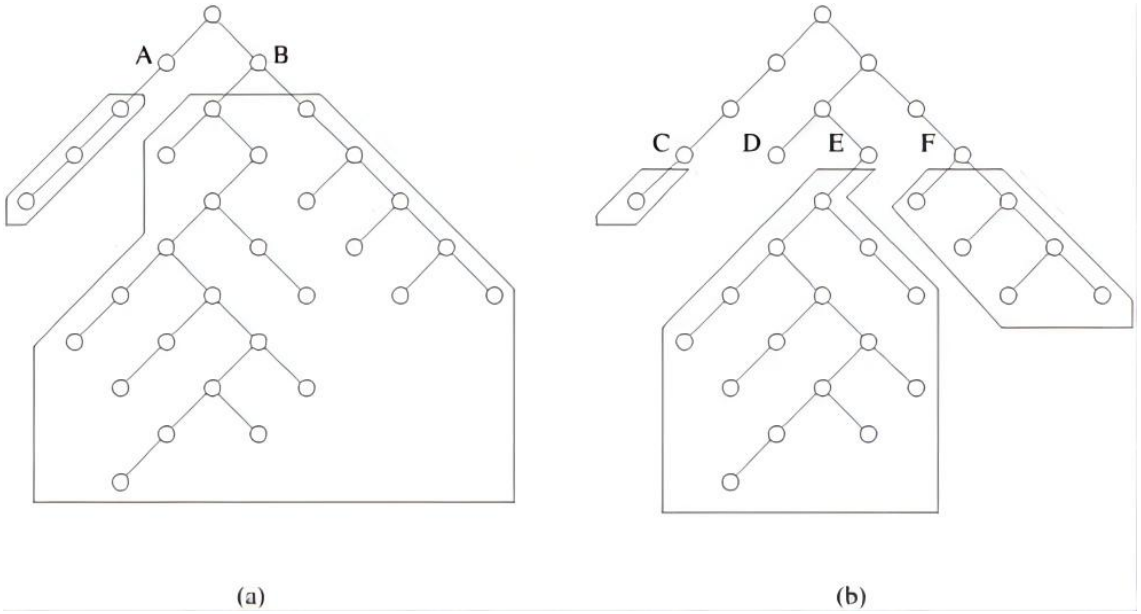
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Allocating Many Subtrees per Process



Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Parallel Depth-First Search



The unstructured nature of tree search and the imbalance resulting from static partitioning.

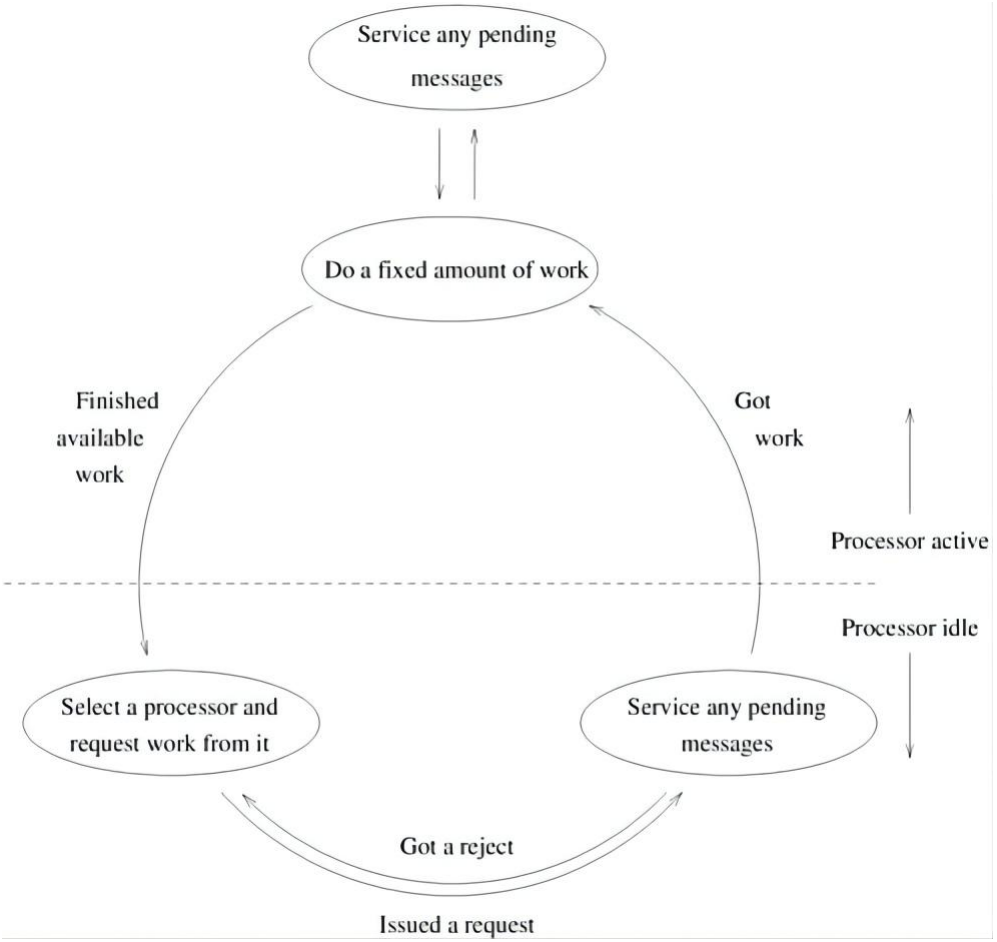
A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Parallel Depth-First Search

- **How is the search space partitioned across processors?**
  - Different subtrees can be searched concurrently.
  - However, subtrees can be very different in size.
  - It is difficult to estimate the size of a subtree rooted at a node.
- **Dynamic load balancing is required.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Parallel DFS: Dynamic Load Balancing



A generic scheme for dynamic load balancing.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Parallel DFS: Dynamic Load Balancing

- **The entire space is assigned to one processor to begin with.**
- **When a processor runs out of work, it gets more work from another processor.**
  - Message passing machines: work requests and responses
  - Shared address space machines: locking and extracting work
- **Unexplored states can be conveniently stored as local stacks at processors.**
- **On reaching final state at a processor, all processors terminate.**

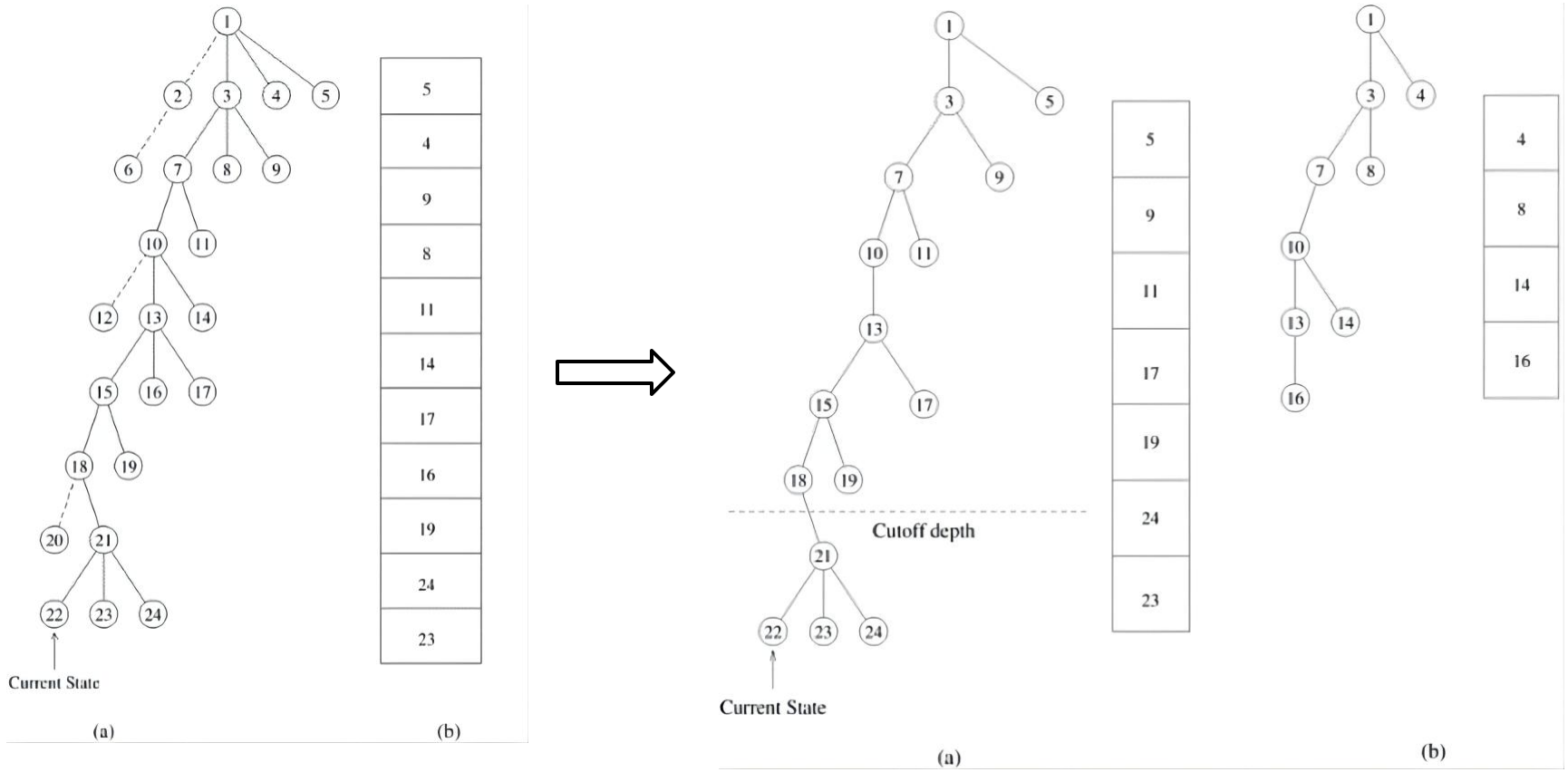
A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Parameters in Parallel DFS: Work Splitting

- **Terminologies**
  - *Donor* process: the process that sends work
  - *Recipient* process: the process that requests/receives work
  - *Half-split*: ideally, the stack is split into two equal pieces such that the search space of each stack is the same
  - *Cutoff depth*: to avoid sending very small amounts of work, nodes beyond a specified stack depth are not given away

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Parameters in Parallel DFS: Work Splitting



Splitting the DFS tree:
the two subtrees along with their stack representations.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Parameters in Parallel DFS: Work Splitting

- **Some possible strategies**
  1. Send nodes near the bottom of the stack
     - Works well with uniform search space; has low splitting cost
  2. Send nodes near the cutoff depth
     - Performs better with a strong heuristic (tries to distribute the parts of the search space likely to contain a solution)
  3. Send half the nodes between the bottom and the cutoff depth
     - Works well with uniform and irregular search space

Load-Balancing Schemes

- **Who do you request work from? Note that we would like to distribute work requests evenly, in a global sense.**
- **Asynchronous round robin (ARR)**
  - Each process maintains a counter and makes requests in a round-robin fashion.
- **Global round robin (GRR)**
  - The system maintains a global counter and requests are made in a round-robin fashion, globally.
- **Random polling (RP)**
  - Request a randomly selected process for work.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Analyzing DFS

- **We can't compute, analytically, the serial work (wall-clock time) W or parallel time W_p in terms of input size n**
- **Instead, we quantify total overhead T_o in terms of W to compute scalability.**
  - $T_O = pW_p - W$
  - Overhead is due to
    - Communication (requesting and sending work)
    - Idle time (waiting for work)
    - Termination detection
    - Contention for shared resources (e.g., the global counter in GRR)
    - Search overhead factor
  - For dynamic load balancing, idling time is subsumed by communication.
- **We must quantify the total number of requests in the system.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Search Overhead Factor

- **The amount of work done by serial and parallel formulations of search algorithms is often different.**
- **Let W be serial work and pW_P be parallel work. Search overhead factor s is defined as pW_P/W.**
- **Upper bound on speedup is $p \times 1/s$.**
  - ATTENTION: $W/W_P < 1$ is possible (speedup anomalies)

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Analyzing DFS: Assumptions

- **Search overhead factor = one**
- **Work at any processor can be partitioned into independent pieces as long as its size exceeds a threshold ε.**
- **A reasonable work-splitting mechanism is available.**
  - If work w at a processor is split into two parts ψw and $(1-\psi)w$, there exists an arbitrarily small constant α $(0 < \alpha \leq 0.5)$, such that $\psi w > \alpha w$ and $(1-\psi)w > \alpha w$.
  - The constant α sets a lower bound on the load imbalance from work splitting.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Analyzing DFS

- **If processor P_i initially had work w_i, after a single request by processor P_j and split, neither P_i nor P_j have more than $(1-\alpha)w_i$ work.**
- **For each load balancing strategy, we define $V(P)$ as the total number of work requests after which each processor receives at least one work request (note that $V(p) \geq p$).**
- **Assume that the largest piece of work at any point is W.**
- **After $V(p)$ requests, the maximum work remaining at any processor is less than $(1-\alpha)W$; after $2V(p)$ requests, it is less than $(1-\alpha)^2W$; …**
- **After $(\log_{1/1(1-\alpha)}(W/\varepsilon))V(p)$ requests, the maximum work remaining at any processor is below a threshold value ε.**
- **The total number of work requests is $O(V(p) \log W)$.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Analyzing DFS

- **If t_{comm} is the time required to communicate a piece of work, then the communication overhead T_O is**

$$T_O = t_{comm} V(p) \log W$$

The corresponding efficiency E is given by

$$E = \frac{1}{1 + T_o/W}$$

$$= \frac{1}{1 + (t_{comm} V(p) \log W)/W}$$

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Analyzing DFS: for Various Schemes

- **Asynchronous Round Robin**
  - $V(p) = O(p^2)$ in the *worst case*.
- **Global Round Robin**
  - $V(p) = p$.
- **Random Polling**
  - Worst case $V(p)$ is unbounded.
  - We do *average case* analysis.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

for Random Polling

- **Let $F(i,p)$ represent a state in which i of the processors have been requested, and $p–i$ have not.**
- **Let $f(i,p)$ denote the average number of trials needed to change from state $F(i,p)$ to $F(p,p)$ ($V(p) = f(0,p)$).**

$$f(i,p) = \frac{i}{p}(1 + f(i,p)) + \frac{p-i}{p}(1 + f(i+1,p)),$$

$$\frac{p-i}{p} f(i,p) = 1 + \frac{p-i}{p} f(i+1,p),$$

$$f(i,p) = \frac{p}{p-i} + f(i+1,p).$$

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

for Random Polling

- **We have:**

$$f(0,p) = p \times \sum_{i=0}^{p-1} \frac{1}{p-i},$$

$$= p \times \sum_{i=1}^{p} \frac{1}{i},$$

$$= p \times H_p,$$

H_p ← harmonic number

As p becomes large, $H_p \simeq 1.69 \ln p$. Thus, $V(p) = O(p \log p)$.

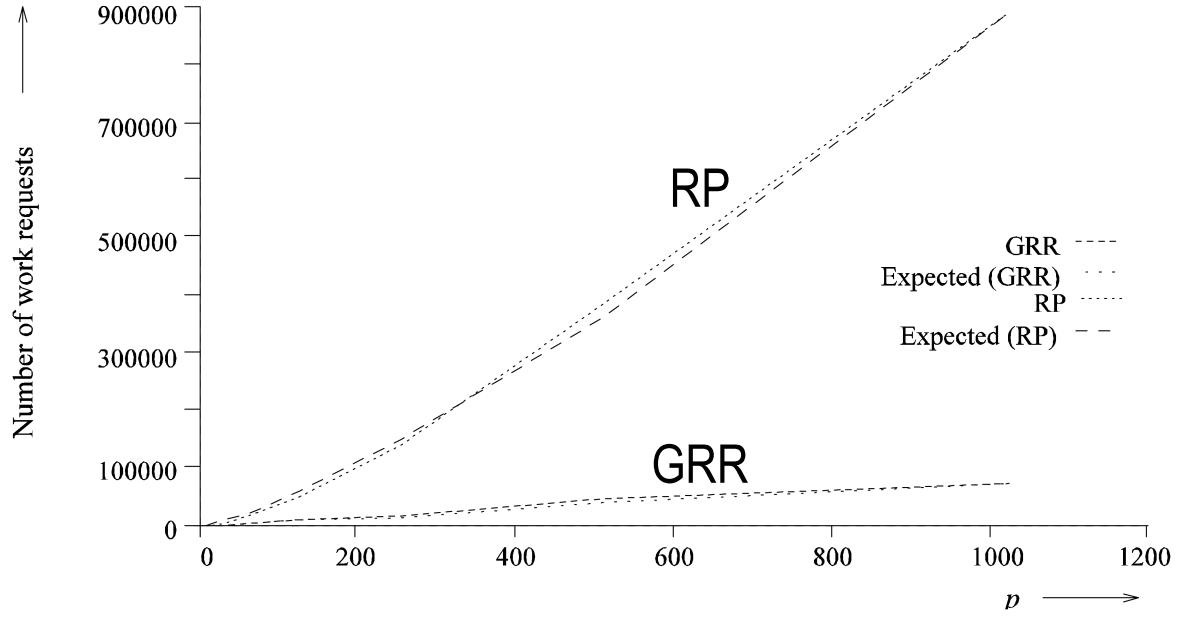
A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Analysis of Load-Balancing Schemes: Conclusions

- **Asynchronous round robin has poor performance because it makes a large number of work requests.**
- **Global round robin has poor performance because of contention at counter, although it makes the least number of requests.**
- **Random polling strikes a desirable compromise.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

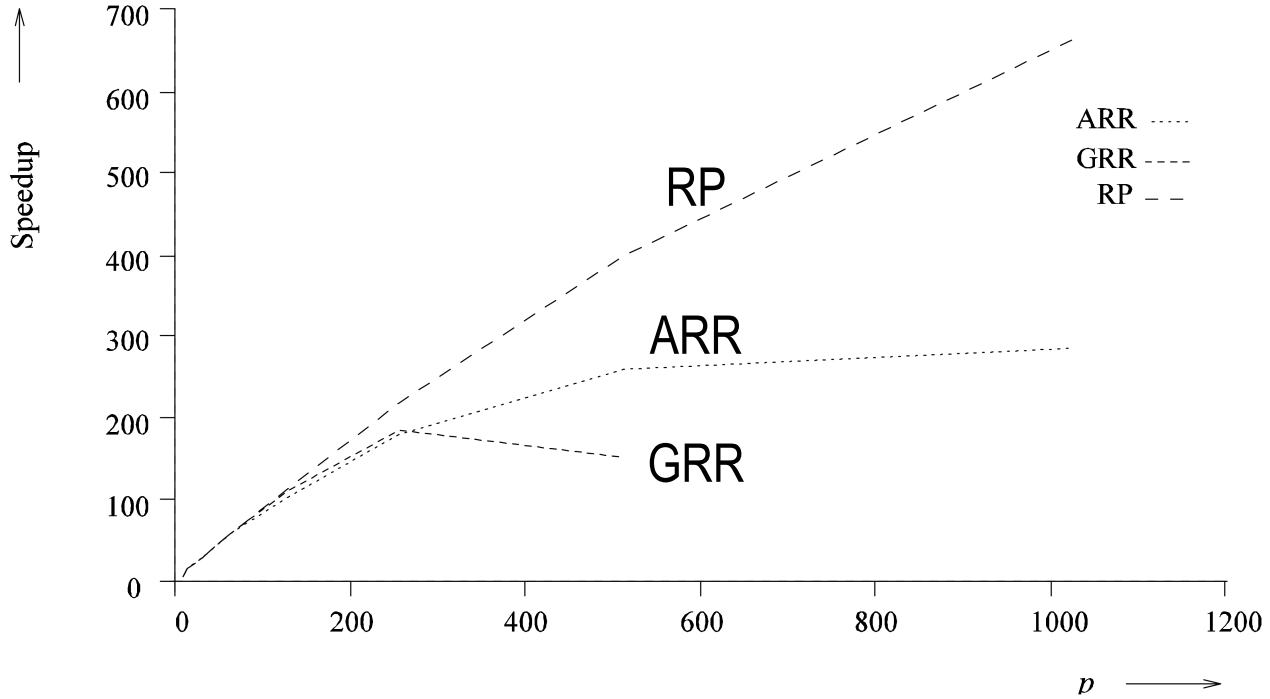
Experimental Validation: Satisfiability Problem



Number of work requests generated for RP and GRR and their expected values (*O(p* log *p* log *W)* and *O(p log W)* respectively). (Assuming *W=p* in the estimation)

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

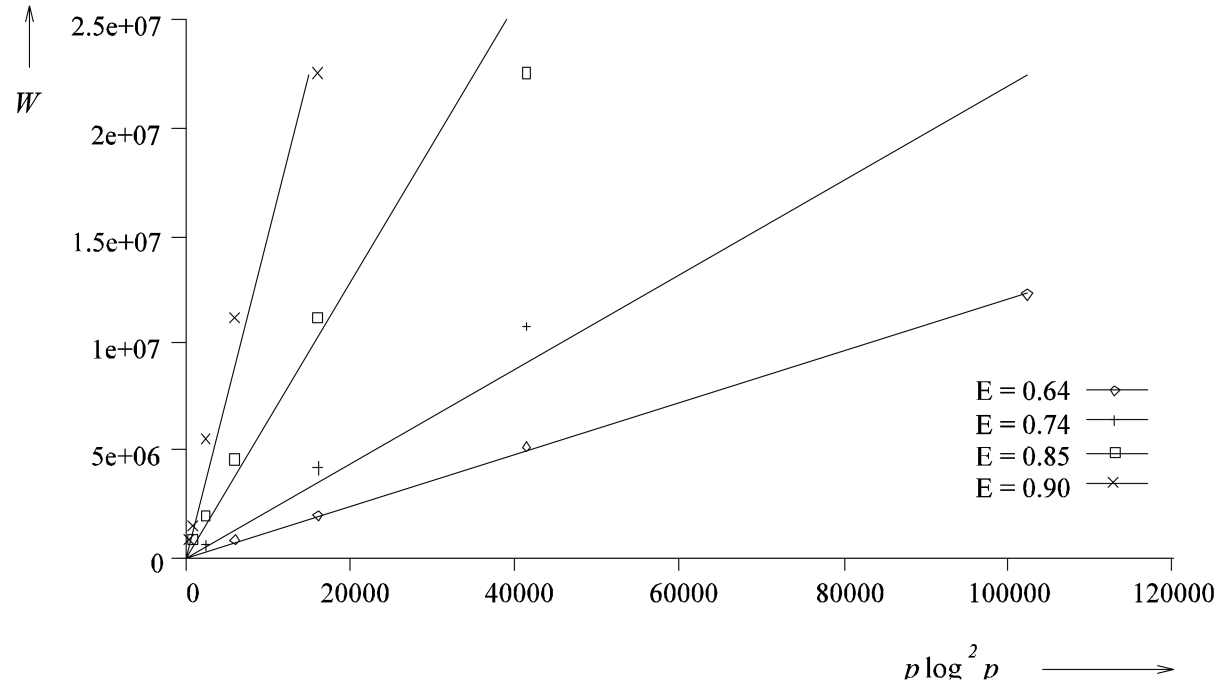
Experimental Validation: Satisfiability Problem



Speedups of parallel DFS using ARR, GRR and RP load-balancing schemes.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

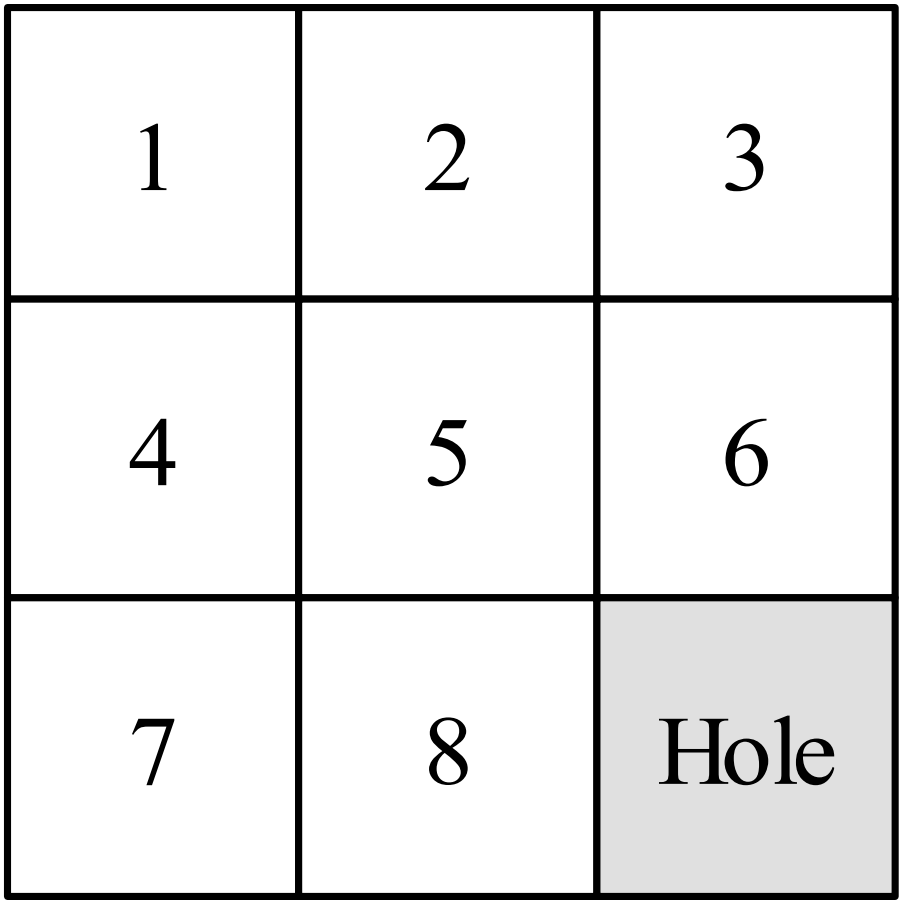
Experimental Validation: Satisfiability Problem



Experimental isoefficiency curves for RP for different efficiencies.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

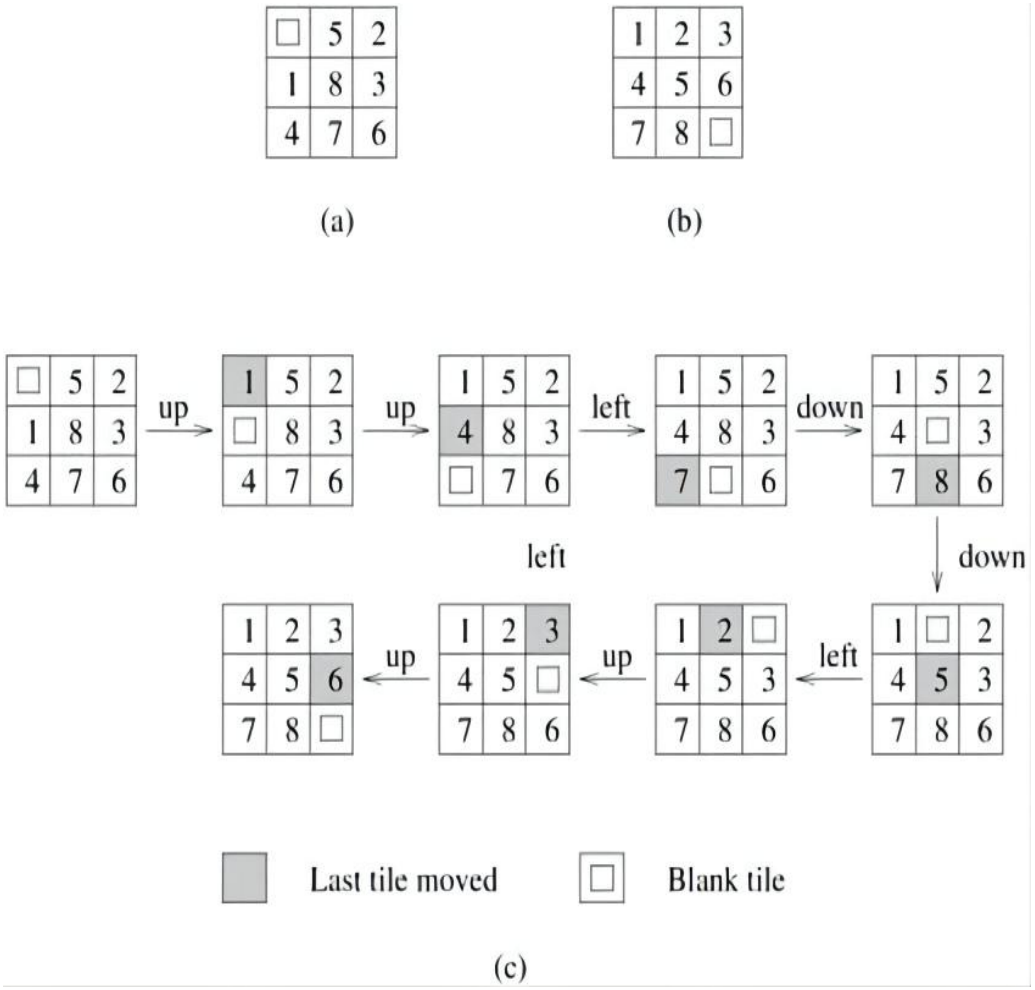
Best-First Search: 8-puzzle



This is the solution state. Tiles slide up, down, or sideways into hole.

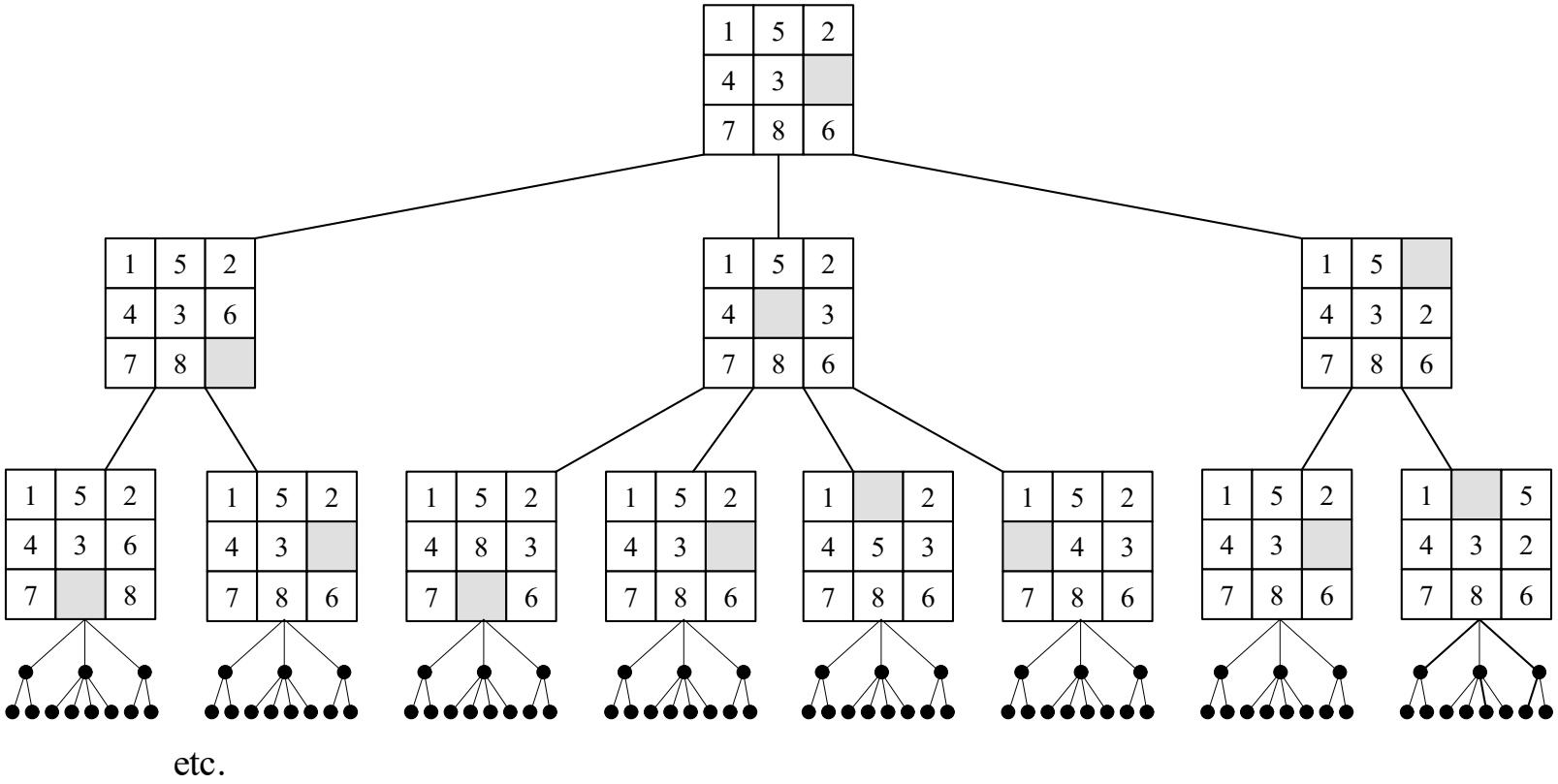
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Example: Solve 8-Puzzle Problem



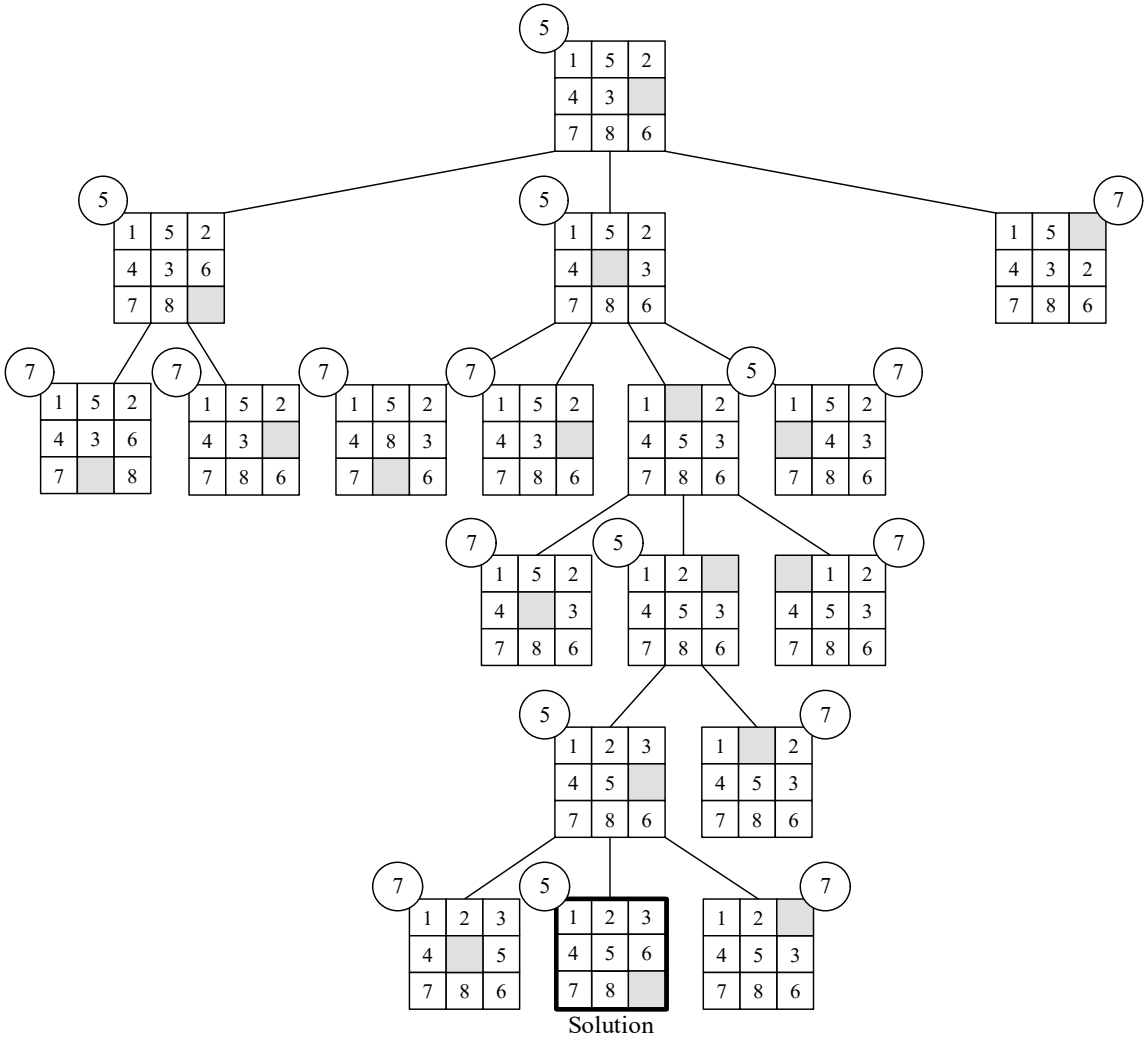
A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

State Space Tree Represents Possible Moves



Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Best-First Search of 8-puzzle



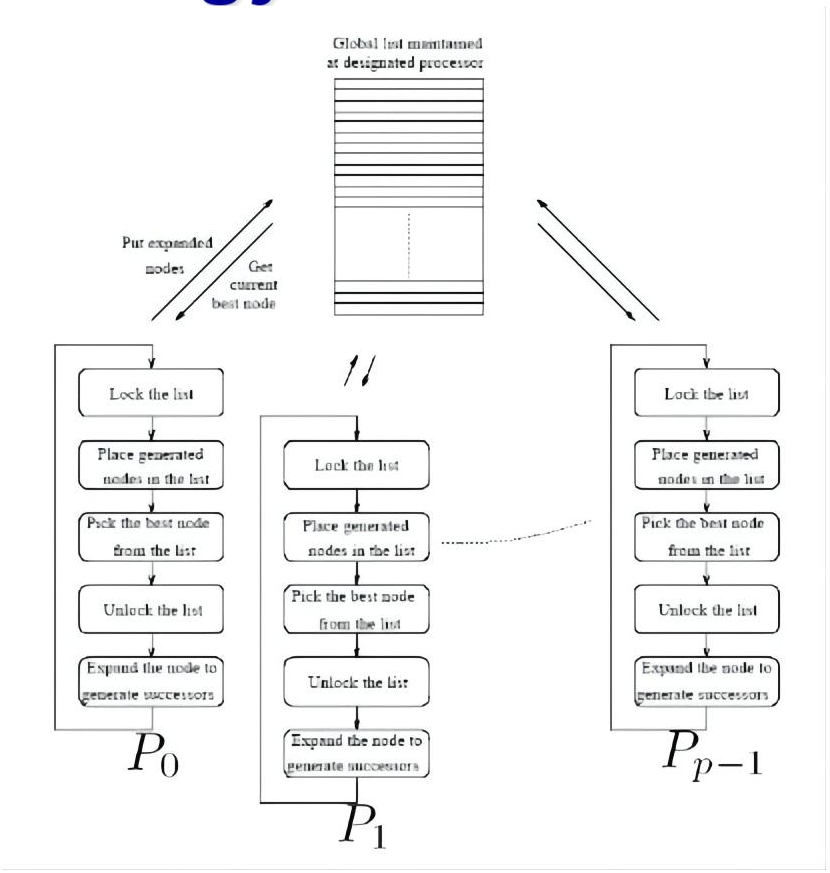
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

A Lower Bound Function

- A lower bound on number of moves needed to solve puzzle is sum of Manhattan distance of each tile's current position from its correct position
- Depth of node in state space tree indicates number of moves made so far
- Adding two values gives lower bound on number of moves needed for any solution, given moves made so far
- We always search from node having smallest value of this function (best-first search)

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

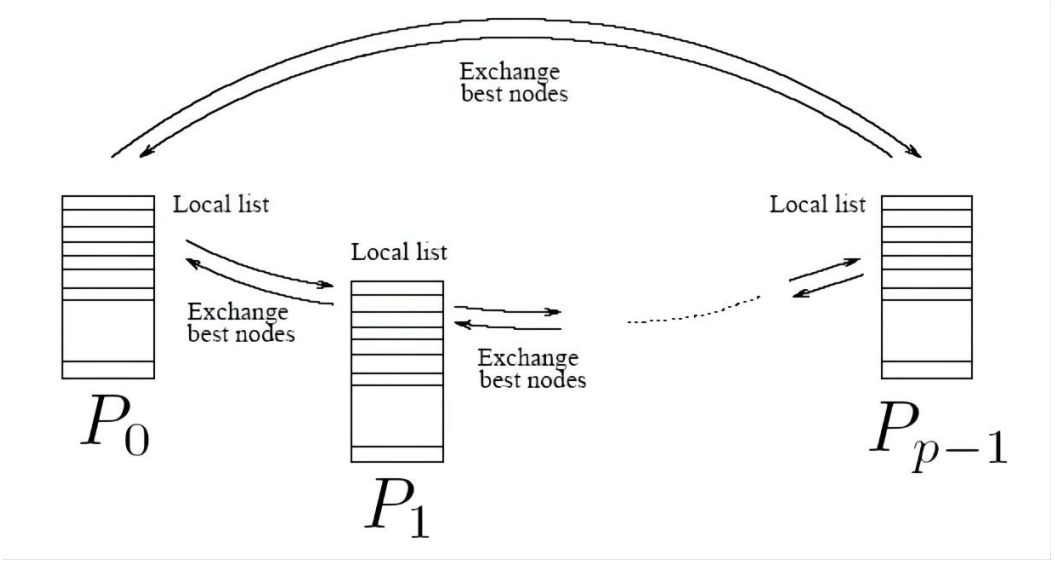
Parallel Best-First Search: Centralized Strategy



A general schematic for parallel best-first search using a centralized strategy. The locking operation is used here to serialize queue access by various processors.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

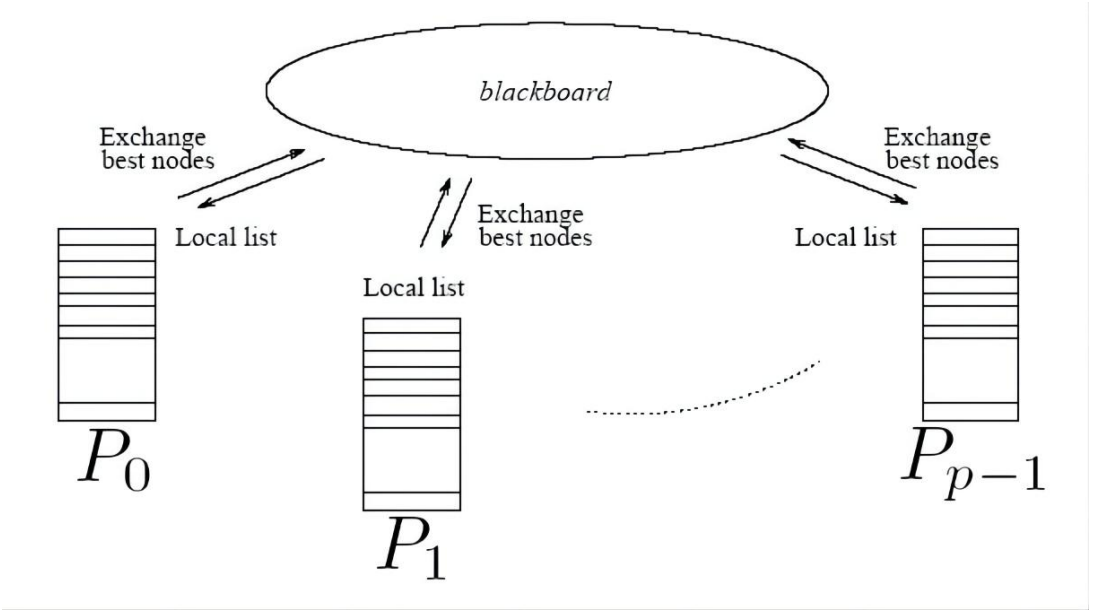
Parallel Best-First Search: Ring Communication Strategy



A message-passing implementation of parallel best-first search using the ring communication strategy.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Parallel Best-First Search: Blackboard Communication Strategy



An implementation of parallel best-first search using the blackboard communication strategy.

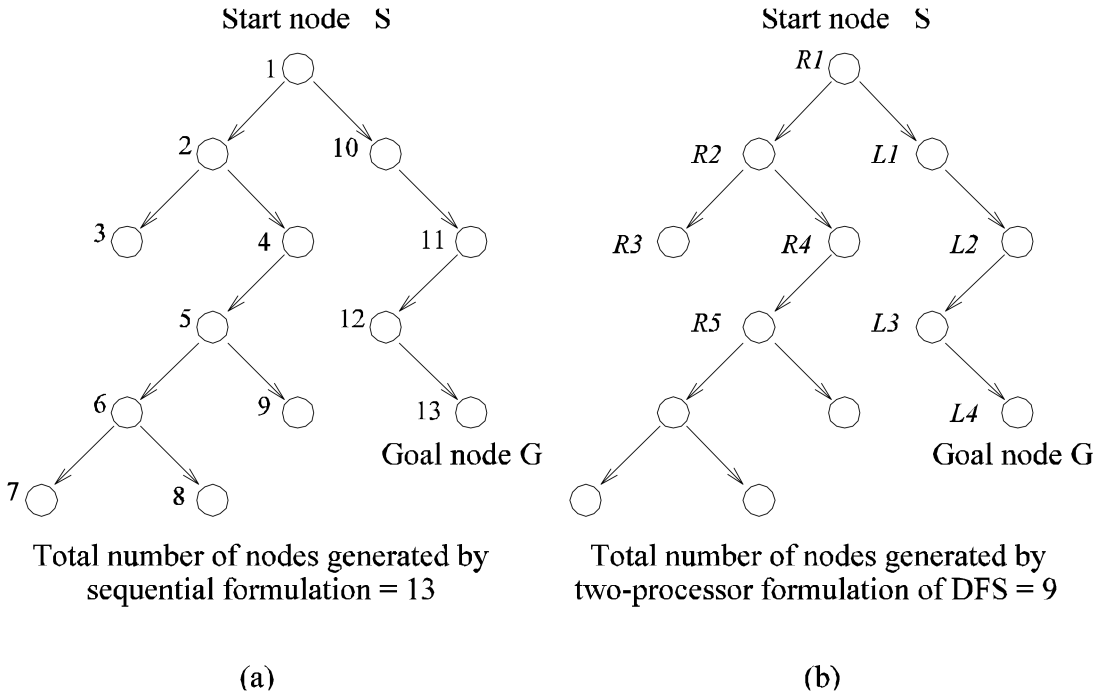
A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Speedup Anomalies in Parallel Search

- **Since the search space explored by processors is determined dynamically at runtime, the actual work might vary significantly.**
- **Executions yielding speedups greater than p by using p processors are referred to as *acceleration anomalies*. Speedups of less than p using p processors are called *deceleration anomalies*.**
- **Speedup anomalies also manifest themselves in best-first search algorithms.**
- **If the heuristic function is good, the work done in parallel best-first search is typically more than that in its serial counterpart.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

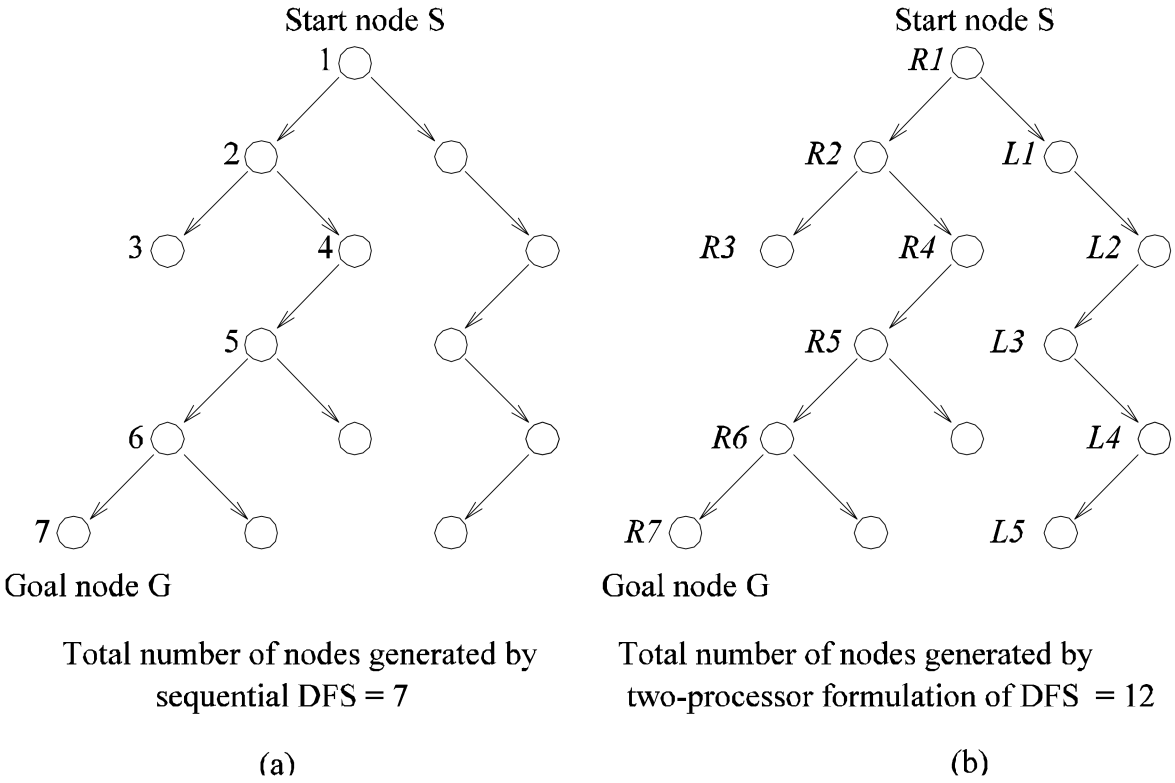
Speedup Anomalies in Parallel Search



The difference in number of nodes searched by sequential and parallel formulations of DFS. For this example, parallel DFS reaches a goal node after searching fewer nodes than sequential DFS.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Speedup Anomalies in Parallel Search



A parallel DFS formulation that searches more nodes than its sequential counterpart

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Summary

- **Parallel depth-first search**
  - Load balancing schemes: ARR, GRR, RP
  - Scalability analysis
- **Parallel best-first search**
  - Centralized strategy
  - Communication strategies: random, ring, blackboard
- **Speedup anomalies**

Backup Slides: Distributed Termination Detection

Distributed Termination Detection

- **Suppose we only want to print one solution**
- **We want all processes to halt as soon as one process finds a solution**
- **This means processes must periodically check for messages**
  - Every process calls MPI_Iprobe every time search reaches a particular level (such as the cutoff depth)
  - A process sends a message after it has found a solution

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

MPI_Iprobe

- **Nonblocking test for a message**

```
int MPI_Iprobe(
    int source,
    int tag,
    MPI_Comm comm,
    int *flag,
    MPI_Status *status);
```

- **flag: true if a message with the specified source, tag, and communicator is available**
- **It is not necessary to receive a message immediately after it has been probed for, and the same message may be probed for several times before it is received.**

Simple (Incorrect) Algorithm

- **A process halts after one of the following events has happened:**
  - It has found a solution and sent a message to all of the other processes
  - It has received a message from another process
  - It has completely searched its portion of the state space tree

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Why Algorithm Fails

- **If a process calls** MPI_Finalize **before another active process attempts to send it a message, we get a run-time error**
- **How this could happen?**
  - A process finds a solution after another process has finished searching its share of the subtrees

  OR
  - A process finds a solution after another process has found a solution

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Distributed Termination Problem

- **Distributed termination problem: Ensuring that**
  - all processes are inactive AND
  - no messages are en route
- **A number of algorithms have been proposed.**
  - Token termination detection
    - [Dijkstra, Seijen, Gasteren, 1983]
  - Tree-based termination detection
    - [Rokusawa, Ichiyoshi, Chikayama, Nakashima, 1988]
  - …

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

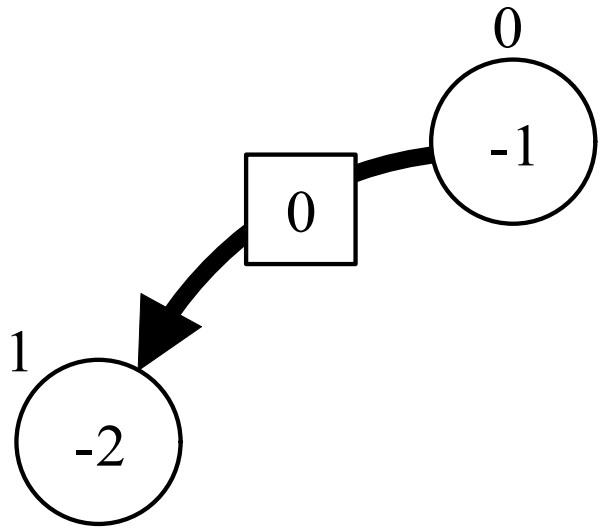
Dijkstra et al.'s Algorithm

- **Each process has a color and a message count**
  - Initial color is white
  - Initial message count is *0*
- **A process that sends a message turns black and increments its message count**
- **A process that receives a message turns black and decrements its message count**
- **If all processes are white and sum of all their message counts are *0*, there are no pending messages and we can terminate the processes**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Dijkstra et al.'s Algorithm (cont.)

- **Organize processes into a logical ring**
- **Process *0* passes a token around the ring**
- **Token also has a color (initially white) and count (initially *0*)**



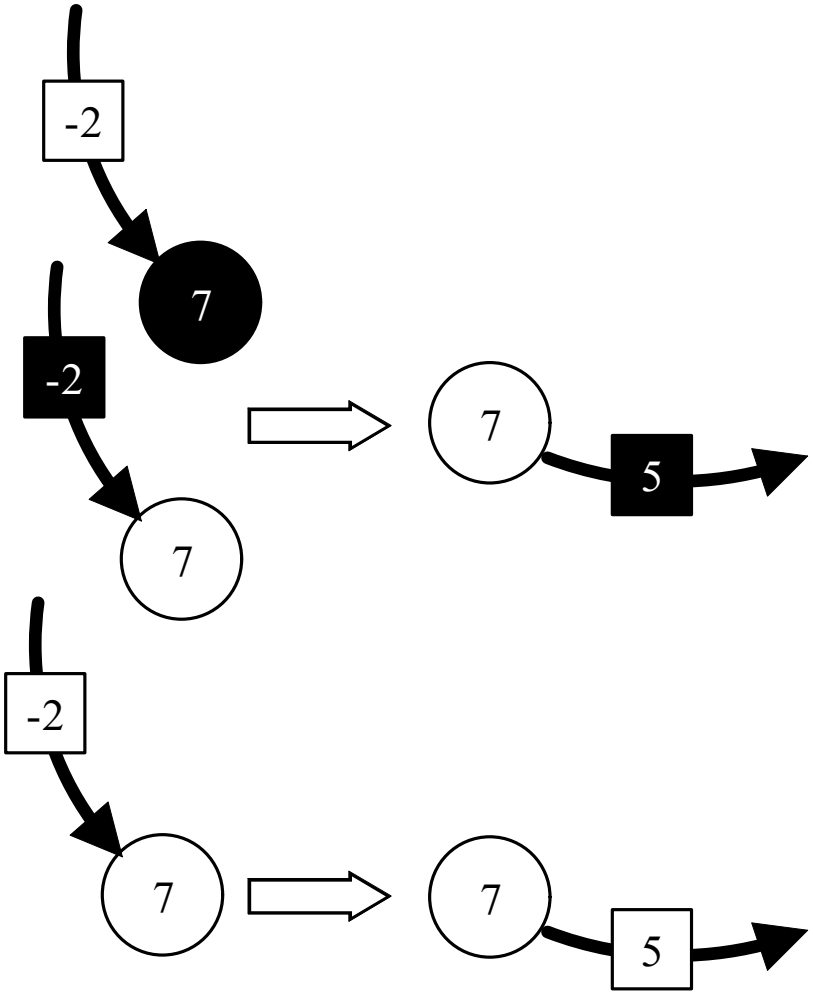
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Dijkstra et al.'s Algorithm (cont.)

- **A process receives the token**
  - If process is black
    - Process changes token color to black
    - Process changes its color to white
  - Process adds its message count to token's message count
- **A process sends the token to its successor in the logical ring**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Dijkstra et al.'s Algorithm (cont.)



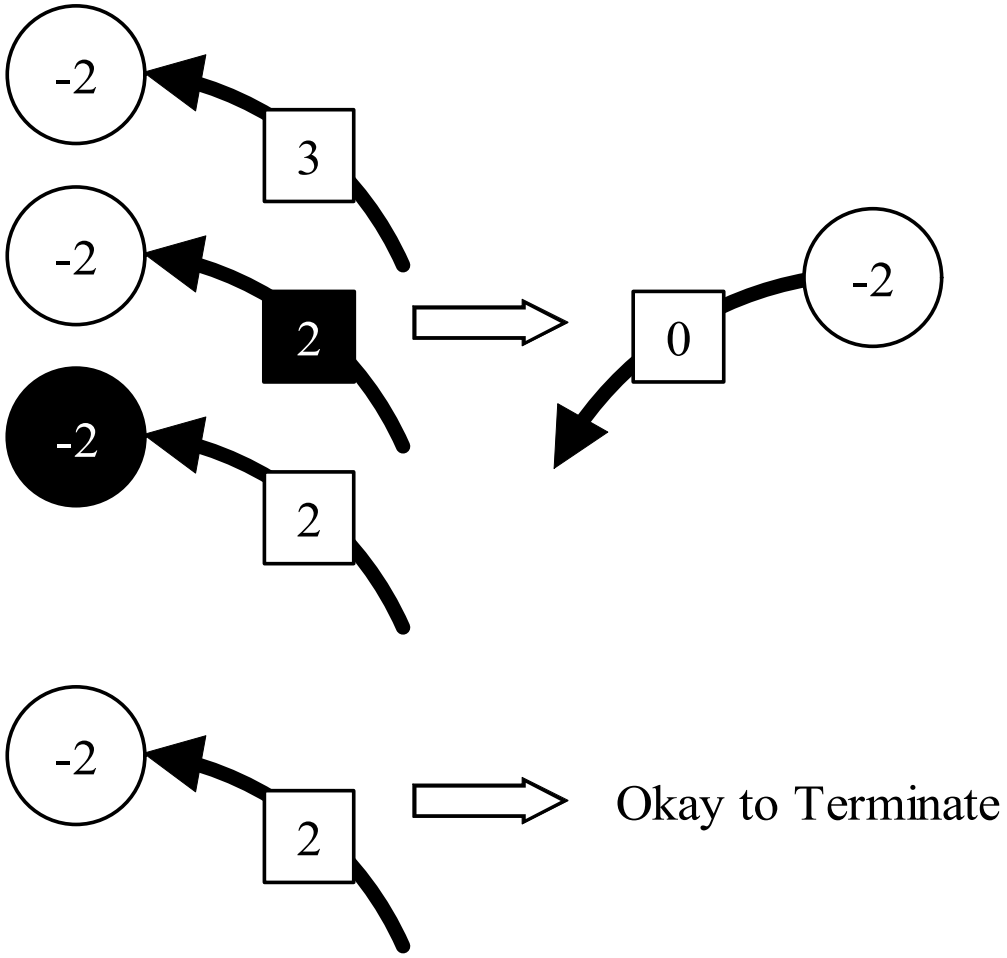
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Dijkstra et al.'s Algorithm (cont.)

- **Process *0* receives the token**
  - Safe to terminate processes if
    - Token is white
    - Process *0* is white
    - Token count + process *0* message count = *0*
  - Otherwise, process *0* must probe ring of processes again

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Dijkstra et al.'s Algorithm (cont.)



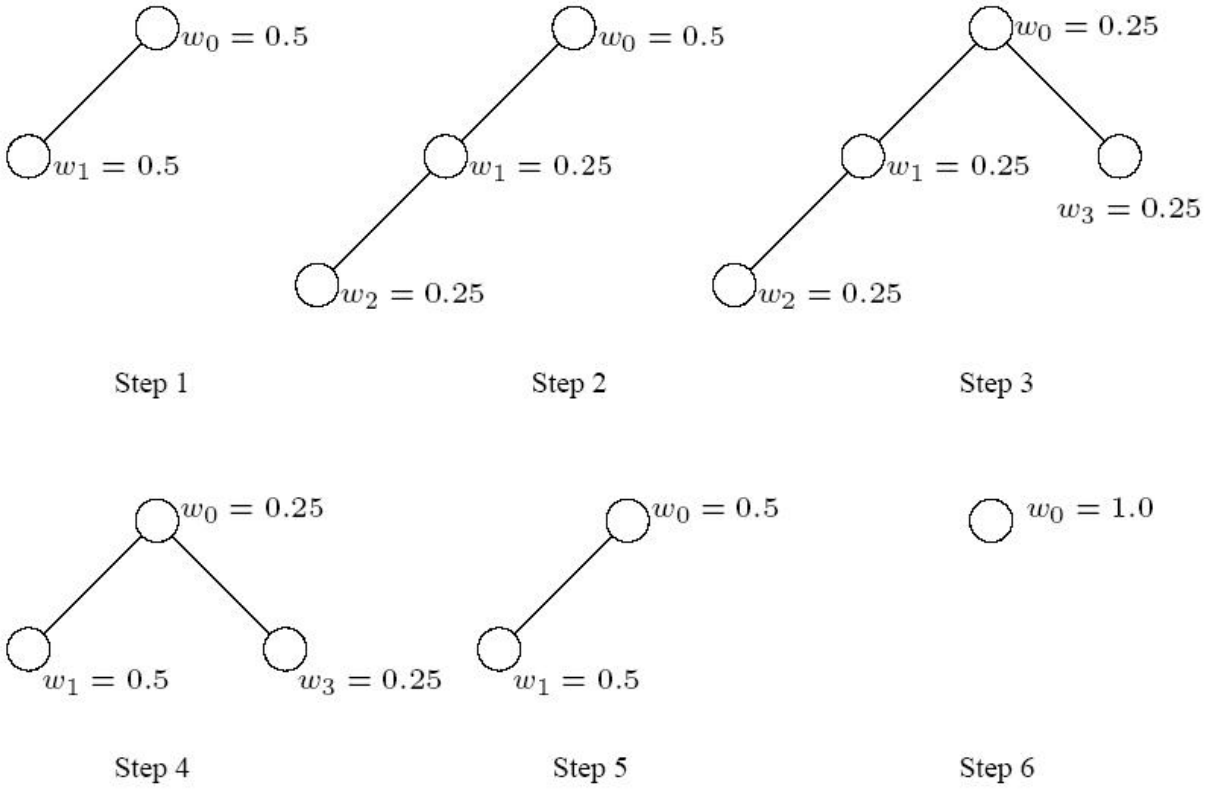
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Tree-Based Termination Detection

- **Algorithm**
  - Associate weights with individual workpieces. Initially, processor P_0 has all the work and a weight of one.
  - Whenever work is partitioned, the weight is split into half and sent with the work.
  - When a processor gets done with its work, it sends its parent the weight back.
  - Termination is signaled when the weight at processor P_0 becomes one again.
- **Note that underflow and finite precision are important factors associated with this scheme.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Tree-Based Termination Detection



Steps 1-6 illustrate the weights at various processors after each work transfer

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003