CIS 631
Parallel Processing

Lecture 6: Parallel Programming

Allen D. Malony
malony@cs.uoregon.edu

Department of Computer and Information Science
University of Oregon

Acknowledgements

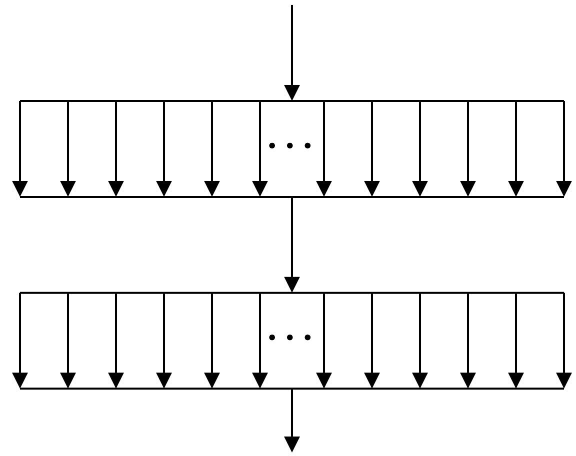
- Portions of the lectures slides were adopted from:
  - I. Foster, "Designing and Building Parallel Programs," 1995.
  - Vijay Pai, COMP 422, "Parallel Programming," Rice University, 2002.
  - A. Grama, A. Gupta, G. Karypis, and V. Kumar, "Introduction to Parallel Computing," 2003.

Outline

- Dependency and Synchronization
- Methodological design of parallel programs
- Types of parallel programs
  - Data parallel vs. task parallel
  - Pipelining
  - Task graphs
  - Master-slave
  - Producer-consumer
  - Divide-and-conquer
  - SPMD
  - Loop scheduling

Fork-Join Parallelism

```
x = g(a);
for( i=0; i<100; i++ ) a[i] = f(i);
y = h(a);
for( i=0; i<100; i++ ) b[i] = x + h( a[i]);
```



- First loop is a DOALL loop
- Middle statement is sequential
- Second loop is a DOALL loop
- Execution moves between sequential and parallel phases
- Call this *fork-join* parallelism

Fork-Join and Barrier Synchronization

- fork() causes a number of processes to be created and to be run in parallel
- join() causes all these processes to wait until all of them have executed a join() (*barrier* synchronization)

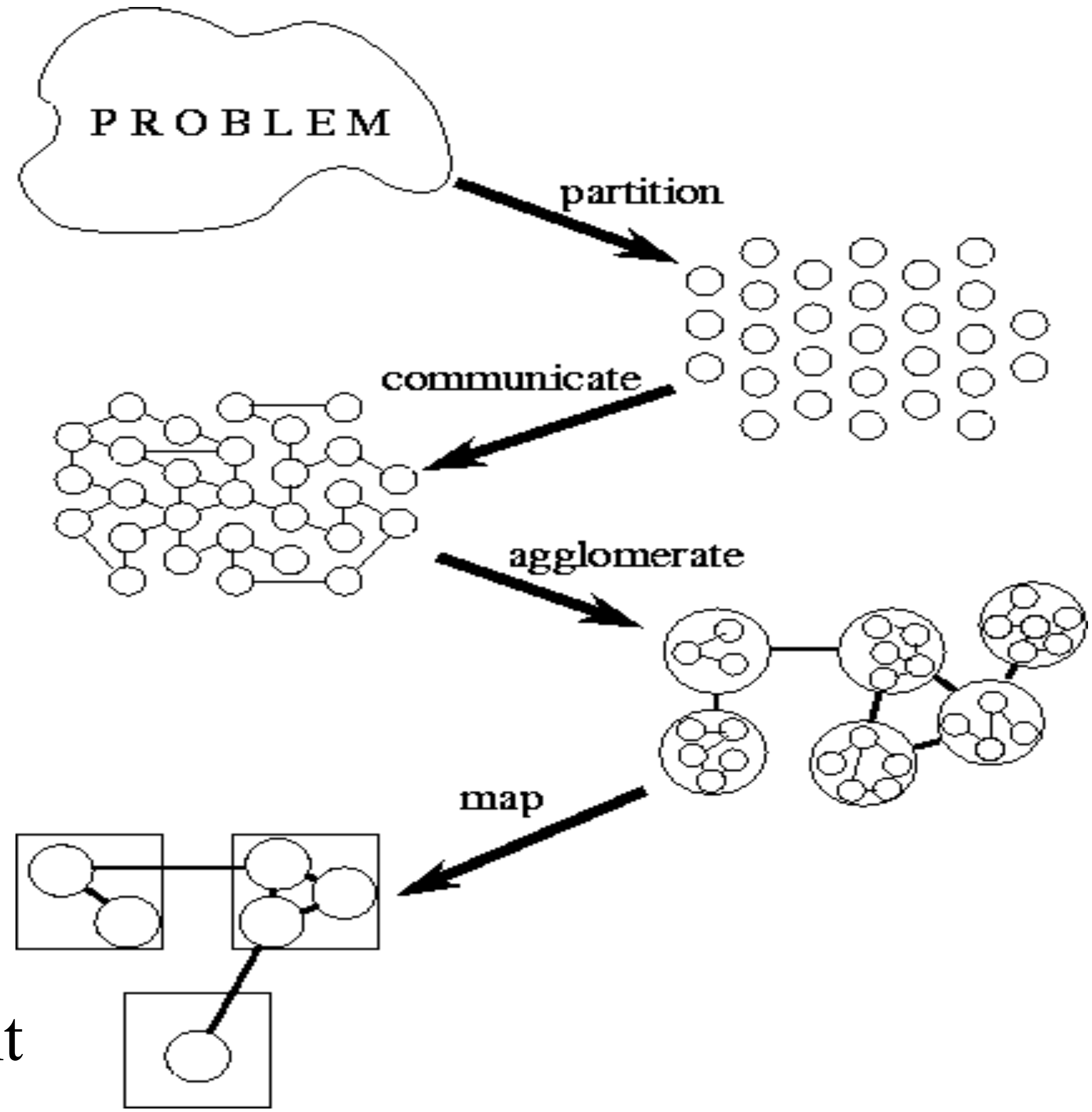
```
fork();
for( i=0; i<100; i++ ) a[i] = f(i);
join();
y = h(a);
fork();
for( i=0; i<100; i++ ) b[i] = x + h( a[i]);
join();
```

Synchronization Issues

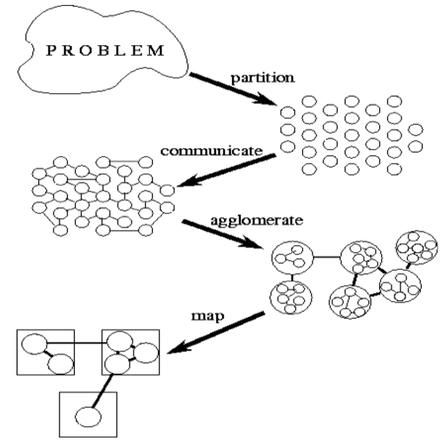
- Synchronization is necessary to make some programs execute correctly in parallel
- Dependences have to be "covered" by appropriate synchronization operations
- Different sychronization constructs exist in different parallel programming models
- However, synchronization is expensive
- To reduce synchronization
  - May need to limit parallelization
  - Look for opportunities to increase parallelism granularity

Methodological Design

- Partition:
  - Task/data decomposition
- Communication
  - Task execution coordination
- Agglomeration
  - Evaluation of the structure
- Mapping
  - Resource assignment



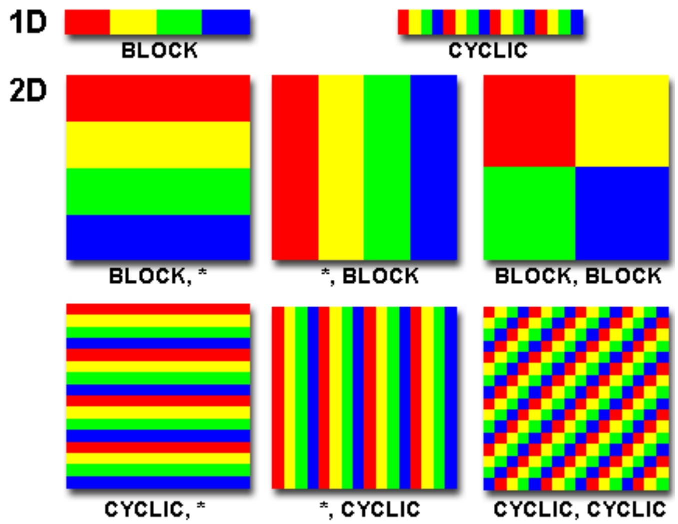
Partitioning

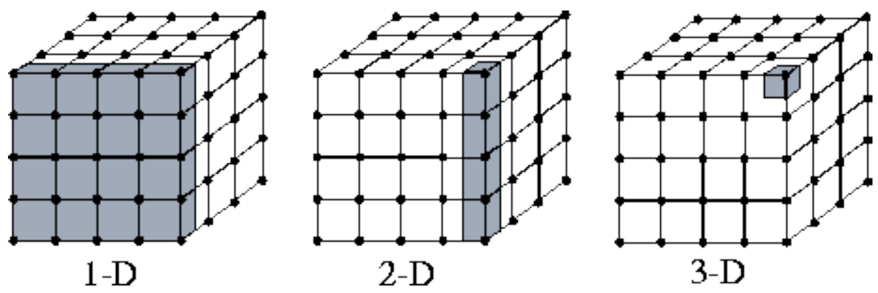


- Partitioning stage is intended to expose opportunities for parallel execution
- Focus on defining large number of small task to yield a fine-grained decomposition of the problem
- A good partition divides into small pieces both the *computation* associated with a problem and the *data* on which this computation operates
- *Domain decomposition* focuses on computation data
- *Functional decomposition* focuses on computation tasks
- Mixing domain/functional decomposition is possible

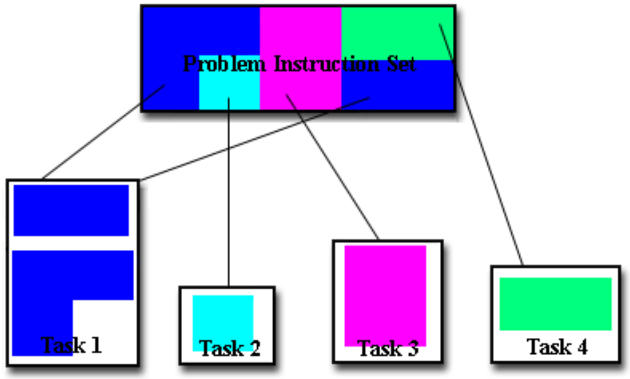
Domain and Functional Decomposition

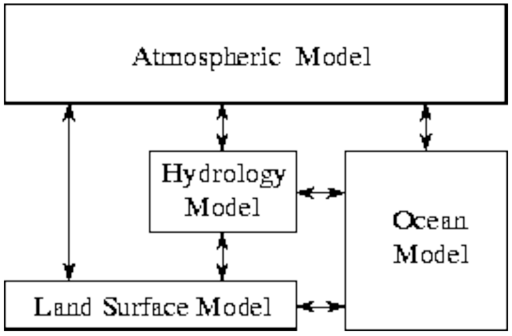
- Domain decomposition of two / three-dimensional grid





- Functional decomposition of a climate model

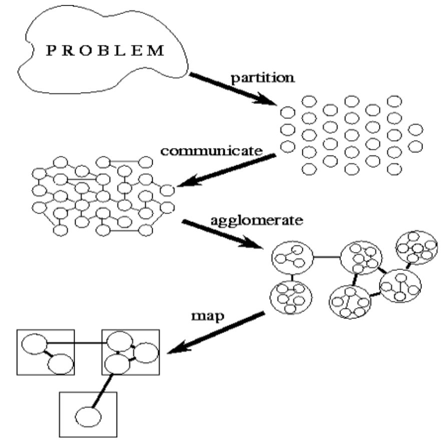




Partitioning Checklist

- Does your partition define at least an order of magnitude more tasks than there are processors in your target computer? If not, may loose design flexibility.
- Does your partition avoid redundant computation and storage requirements? If not, may not be scalable.
- Are tasks of comparable size? If not, it may be hard to allocate each processor equal amounts of work.
- Does the number of tasks scale with problem size? If not may not be able to solve larger problems with more processors
- Have you identified several alternative partitions?

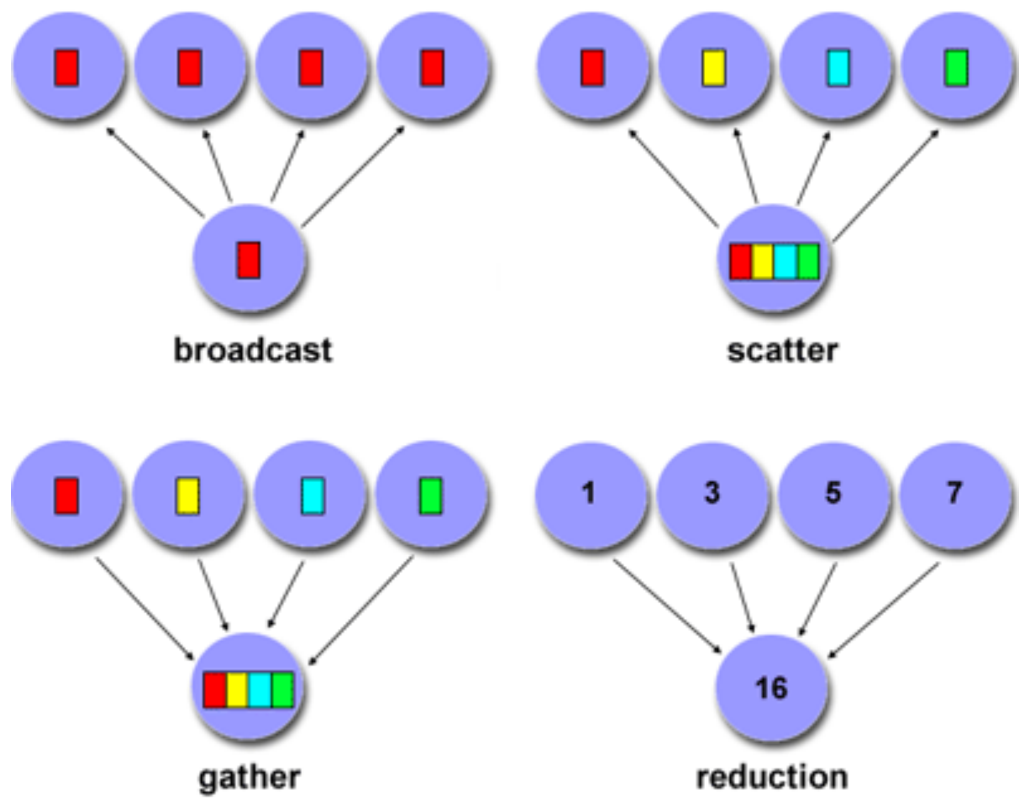
Communication



- Tasks generated by a partition must interact to allow the computation to proceed
  - Information flow: data and control
- Types of communication
  - *Local* vs. *Global*: locality of communication
  - *Structured* vs. *Unstructured*: communication patterns
  - *Static* vs. *Dynamic*: determined by runtime conditions
  - *Synchronous* vs. *Asynchronous*: coordination degree
- Granularity and frequency of communication
  - Size of data exchange
- Communication as control

Types of Communication

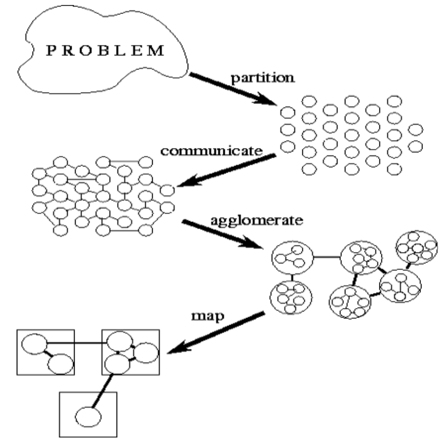
- Point-to-point
- Group-based
- Hierachical
- Collective



Communication Design Checklist

- Is the distribution of communications equal?
  - Unbalanced communication may limit scalability
- What is the communication locality?
  - Wider communication locales are more expensive
- What is the degree of communication concurrency?
  - Communication operations may be parallelized
- Is computation associated with different tasks able to proceed concurrently? Can communication be overlapped with computation?
  - Try to reorder computation and communication to expose opportunities for parallelism

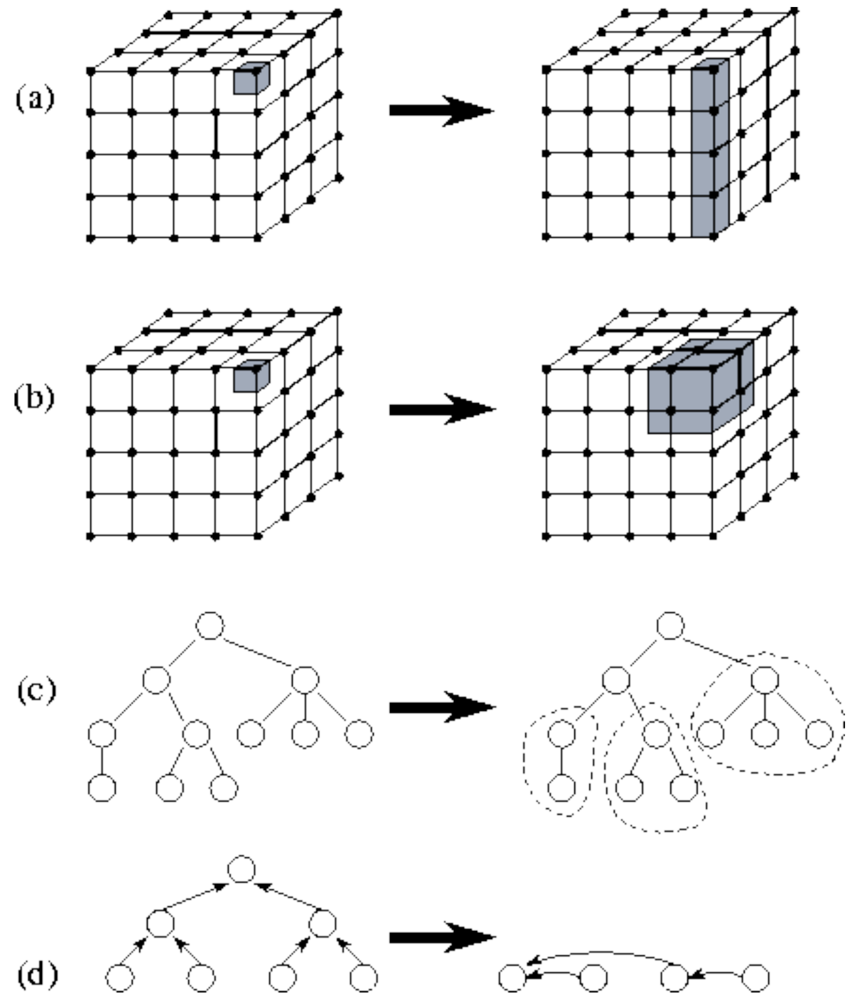
Agglomeration



- Move from parallel abstractions to real implementation
- Revisit partitioning and communication
  - View to efficient algorithm execution
- Is it useful to *agglomerate* (combine) tasks?
- Is it useful to *replicate* data and/or computation?
- Changes important algorithm and performance ratios
  - *Surface-to-volume*: reduction in communication at the expense of decreasing parallelism
  - *Communication/computation*: which cost dominates
- Replication may allow reduction in communication
- Maintain flexibility to allow overlap

Types of Agglomeration

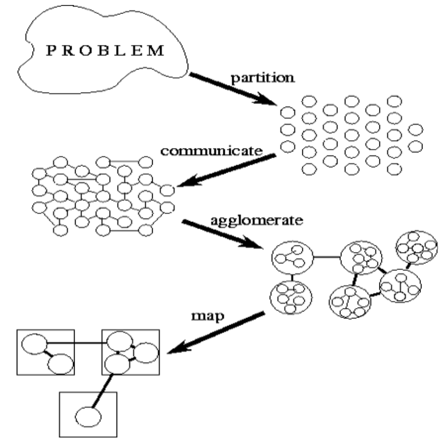
- Element to column
- Element to block
  - Better surface to volume
- Task merging
- Task reduction
  - Reduces communication



Agglomeration Design Checklist

- Has increased locality reduced communication costs?
- Is replicated computation worth it?
- Does data replication compromise scalability?
- Is the computation still balanced?
- Is scalability in problem size still possible?
- Is there still sufficient concurrency?
- Is there room for more agglomeration?
- Fine-grained vs. coarse-grained?

Mapping



- Specify where each task is to execute
  - Less concern on shared-memory computers
- Attempt to minimize execution time
  - Place concurrent tasks on different processors to enhance physical concurrency
  - Place communicating tasks on same processor, or on processors close to each other, to increase locality
  - Strategies can conflict!
- Mapping problem is *NP-complete*
  - Use problem classifications and heuristics
- Static and dynamic load balancing

Mapping Algorithms

- Load balancing (partitioning) algorithms
- Data-based algorithms
  - Think of computational load with respect to amount of data being operated on
  - Assign data (i.e., work) in some known manner to balance
  - Take into account data interactions
- Task-based (task scheduling) algorithms
  - Used when functional decomposition yields many tasks with weak locality requirements
  - Use task assignment to keep processors busy computing
  - Consider centralized and decentralize schemes

Mapping Design Checklist

- Is static mapping too restrictive and non-responsive?
- Is dynamic mapping too costly in overhead?
- Does centralized scheduling lead to bottlenecks?
- Do dynamic load-balancing schemes require too much coordination to re-balance the load?
- What is the tradeoff of dynamic scheduling complexity versus performance improvement?
- Are there enough tasks to achieve high levels of concurrency? If not, processors may idle.

Types of Parallel Programs

- Flavors of parallelism
  - Data parallelism
    - All processors do same thing on different data
  - Task parallelism
    - Processors are assigned tasks that do different things
- Parallel execution models
  - Data parallel
  - Pipelining (Producer-Consumer)
  - Task graph
  - Work pool
  - Master-Worker

Data Parallel

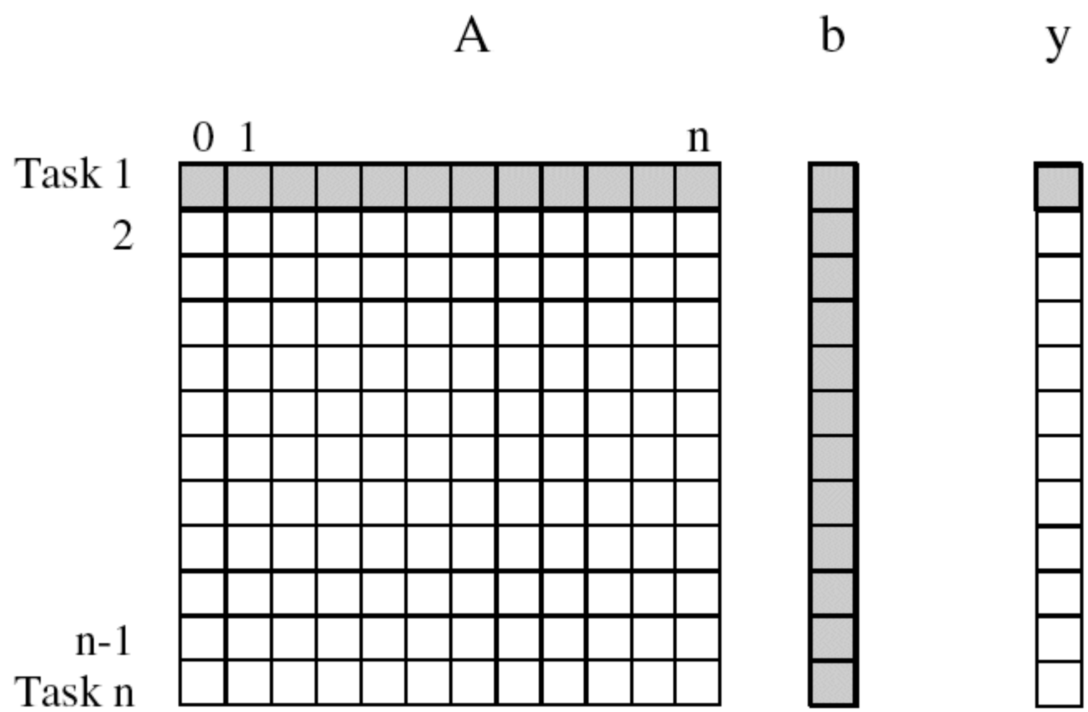
- Data is decomposed (mapped) onto processors
- Processors performance similar (identical) tasks on data
- Tasks are applied concurrently
- Load balance is obtained through data partitioning
  - Equal amounts of work assigned
- Certainly may have interactions between processors
- Data parallelism scalability
  - Degree of parallelism tends to increase with problem size
  - Makes data parallel algorithms more efficient
- *Single Program Multiple Data* (*SPMD*)
  - Convenient way to implement data parallel computation

Matrix - Vector Multiplication

- A x b = y
- Allocate tasks to rows of A

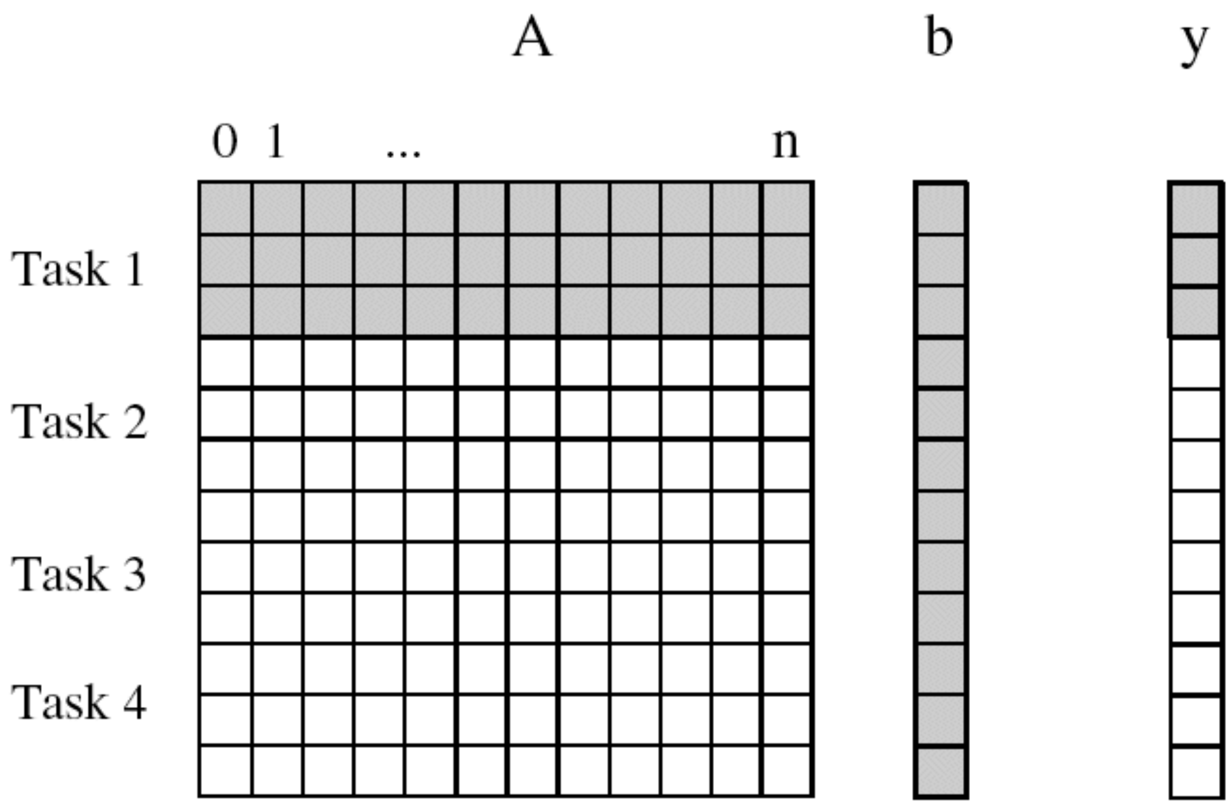
  $y[i] = \sum_{j} A[i,j]*b[j]$

- Dependencies?
- Speedup?
- Computing each element of y can be done independently



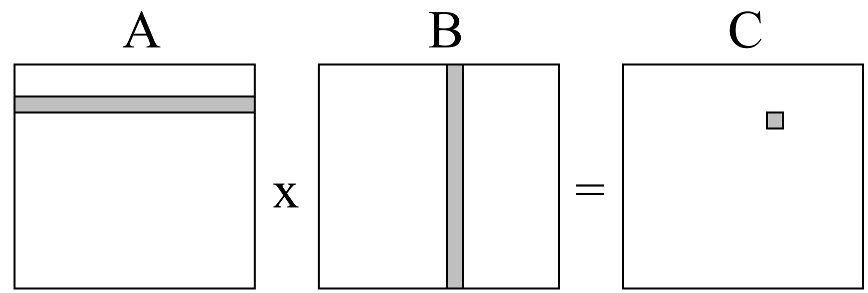
Matrix-Vector Multiplication with Limited Tasks

- Suppose we only have 4 tasks
- Dependencies?
- Speedup?

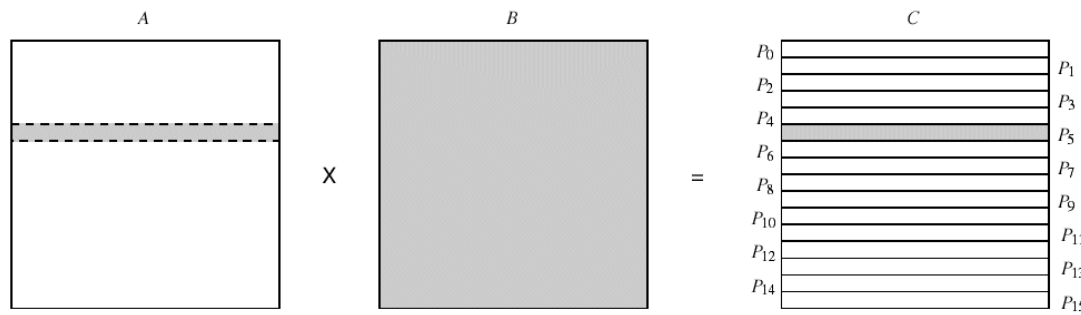


Matrix Multiplication

- A x B = C
- A[i,:] • B[:,j] = C[i,j]

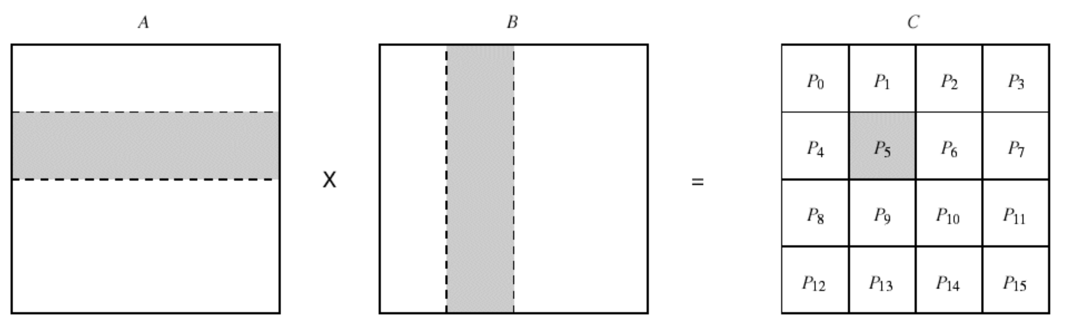


- Row partitioning
  - *N* tasks



(a)

- Block partitioning
  - *N*N/B* tasks



(b)

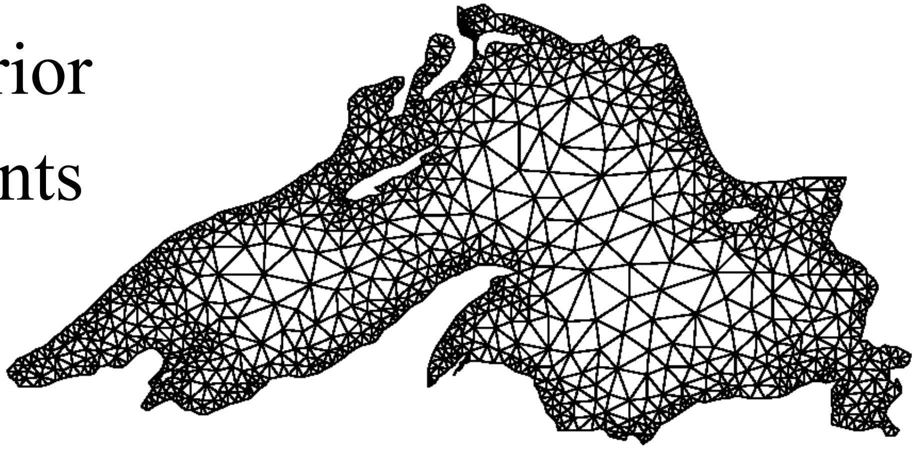
- Shading shows data sharing in B matrix

Granularity of Task and Data Decompositions

- Granularity can be with respect to tasks and data
- Task granularity
  - Equivalent to choosing the number of tasks
  - Fine-grained decomposition results in large # tasks
  - Large-grained decomposition has smaller # tasks
  - Translates to data granularity after # tasks chosen
    - consider matrix multiplication
- Data granularity
  - Think of in terms of amount of data needed in operation
  - Relative to data as a whole
  - Decomposition decisions based on input, output, input-output, or intermediate data

Mesh Allocation to Processors

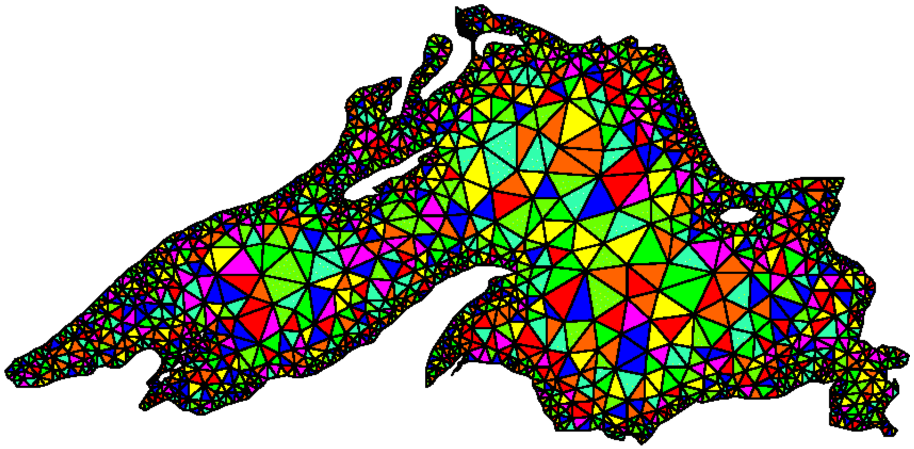
- Mesh model of Lake Superior
- How to assign mesh elements to processors

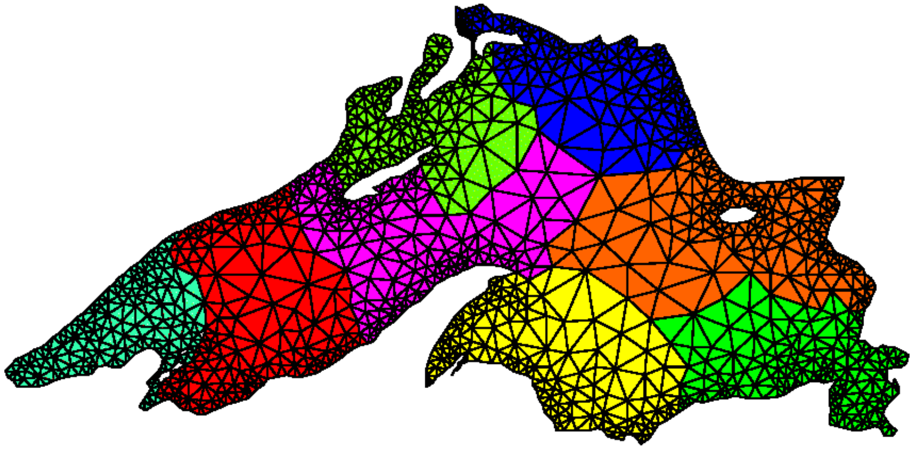


- Distribute onto 8 processors

randomly

graph partitioning for minimum edge cut



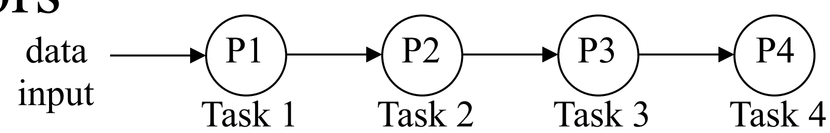


Pipeline Model

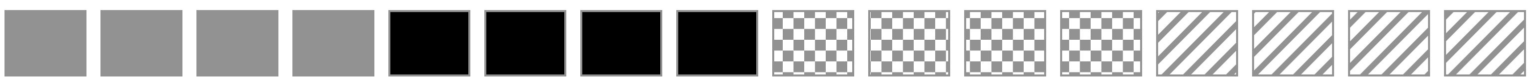
- Stream of data operated on by succession of tasks

  Task 1 Task 2 Task 3 Task 4

  - Tasks are assigned to processors

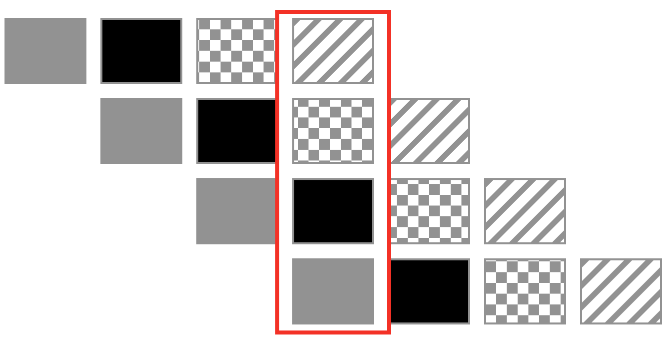


- Consider *N* data units
- Sequential



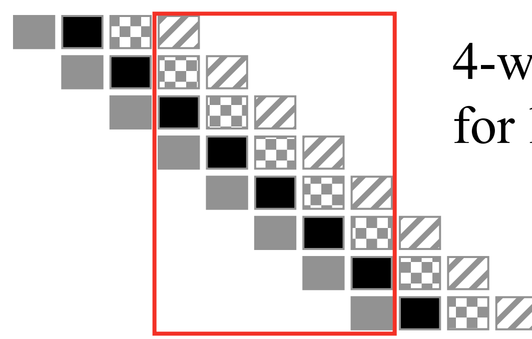
- Parallel (each task assigned to a processor)

4 data units



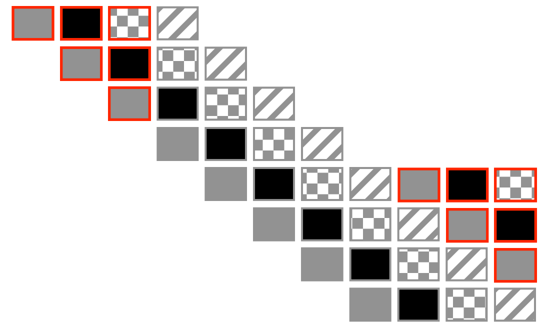
4-way parallel

8 data units



4-way parallel, but for longer time

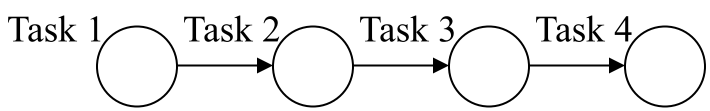
Pipeline Performance



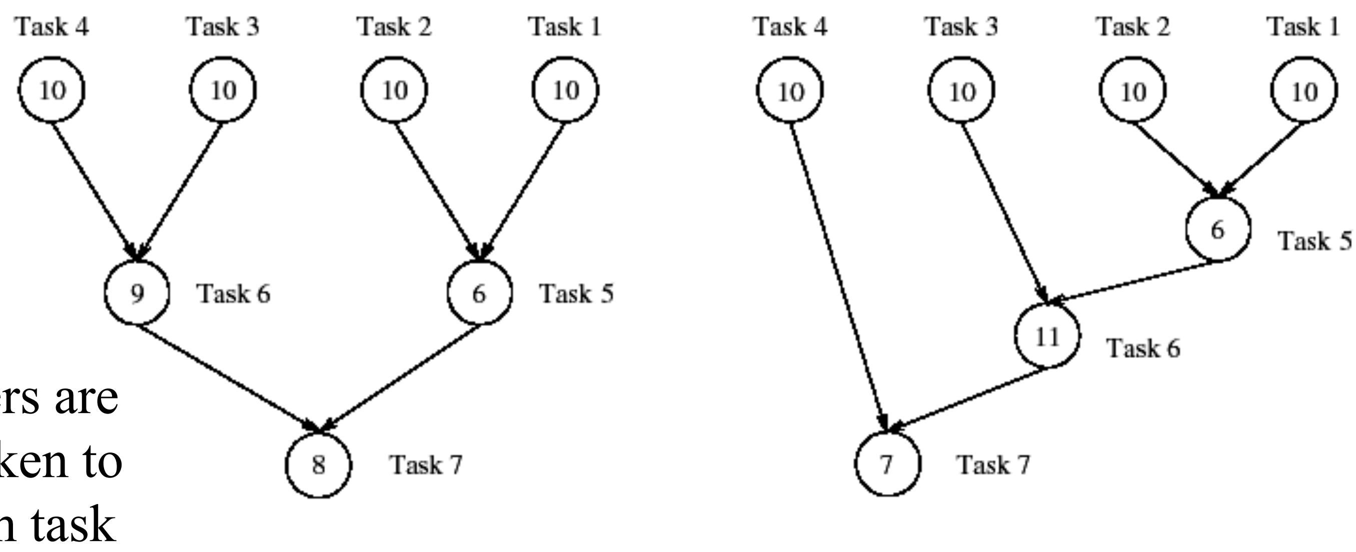
- N data and T tasks
- Each task takes unit time t
- Sequential time = $N*T*t$
- Parallel pipeline time = $start + finish + (N-2T)/T * t$

  = O(N/T) (for N>>T)
- Try to find a lot of data to pipeline
- Try to divide computation in a lot of pipeline tasks
  - More tasks to do (longer pipelines)
  - Shorter tasks to do
- Pipeline computation special form of *producer-consumer*
  - Producer tasks output data input by consumer tasks

Tasks Graphs

- Computations in any parallel algorithms can be viewed as a task dependency graph
- Task dependency graphs may be simple or non-trivial
  - Pipeline


  - Arbitrary (represents the algorithm dependencies)

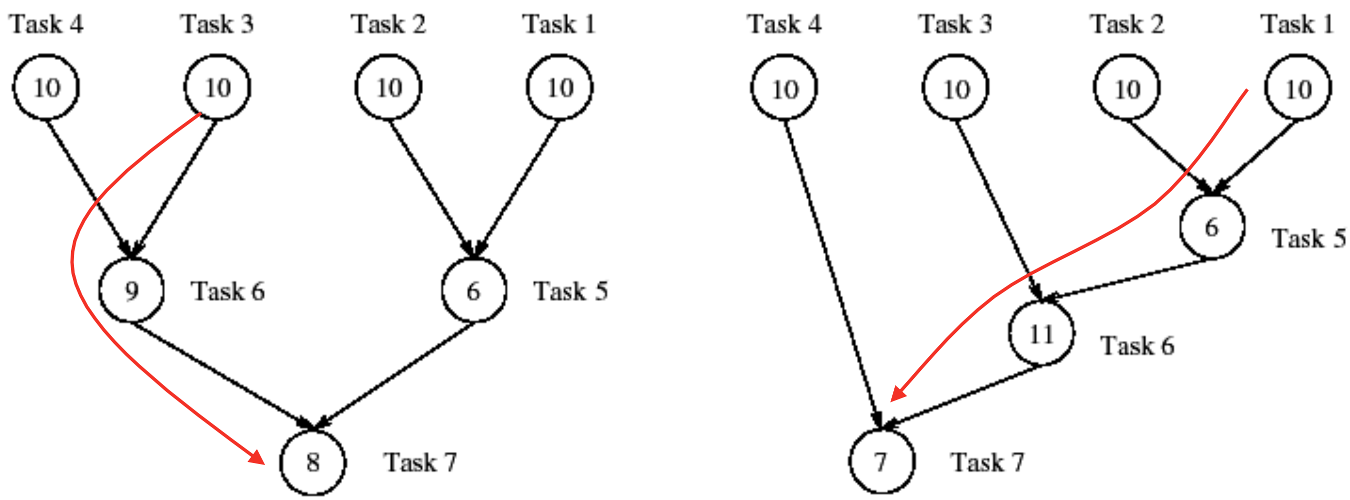


Numbers are time taken to perform task

(a) (b)

Task Graph Performance

- Determined by the *critical path*
  - Sequence of dependent tasks that takes the longest time



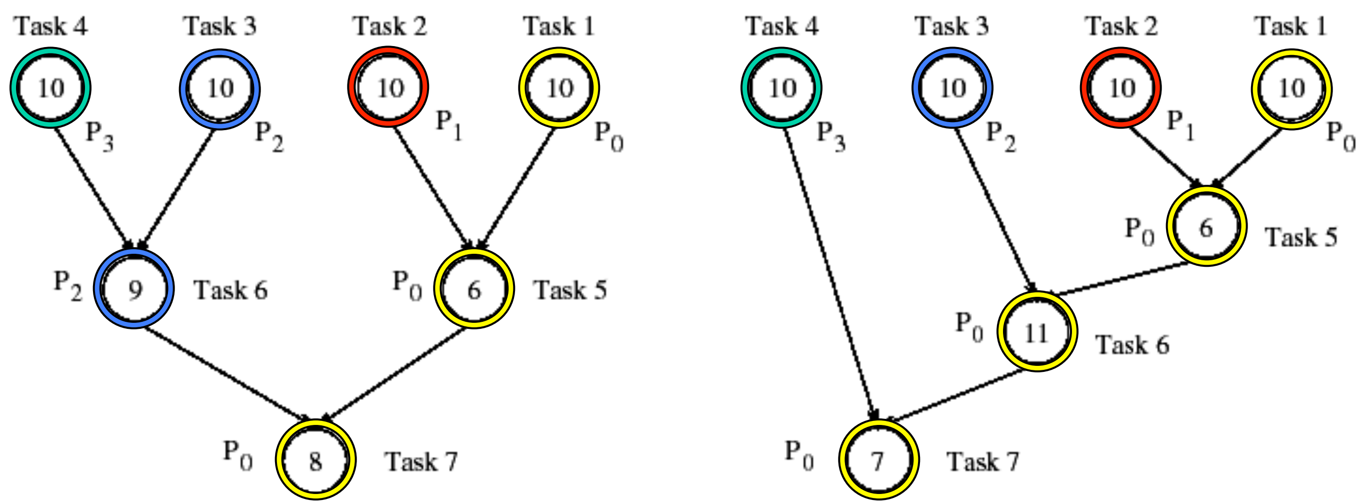
(a) (b)

Min time = 27 Min time = 34

  - *Critical path length* bounds parallel execution time

Task Assignment (Mapping) to Processors

- Given a set of tasks and number of processors
- How to assign tasks to processors?
- Should take dependencies into account
- Task mapping will determine execution time



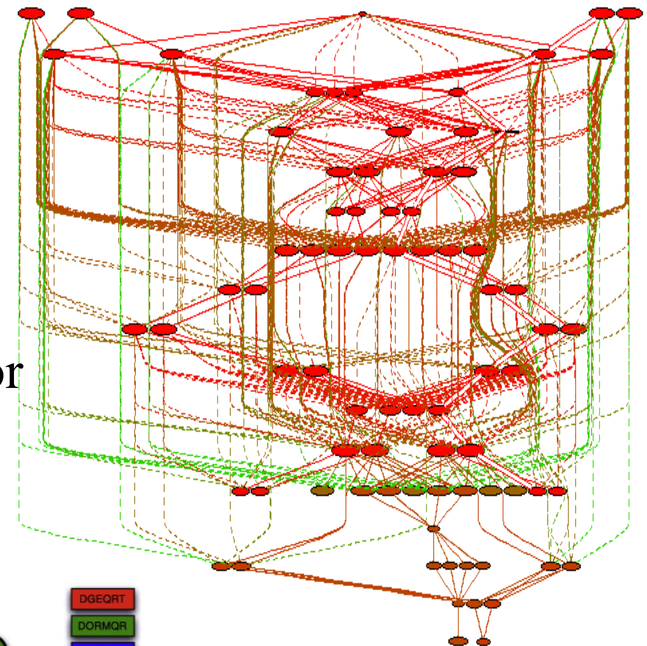
(a) Total time = ?

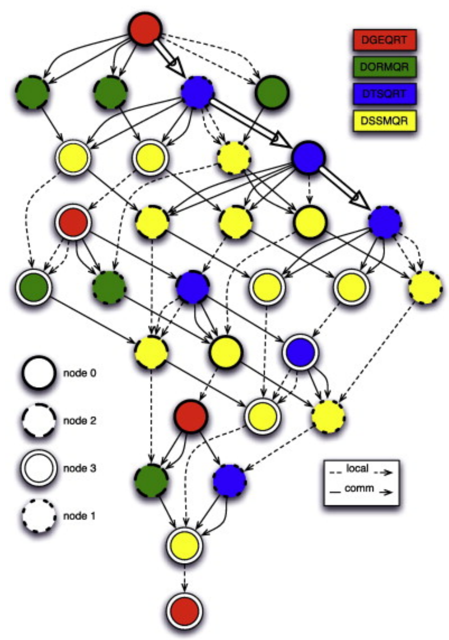
(b) Total time = ?

Task Graphs in Action

- Uintah task graph scheduler
  - C-SAFE: Center for Simulation of Accidental Fires and Explosions, University of Utah
  - Large granularity tasks
- PLASMA
  - DAG-based parallel linear algebra
  - DAGuE: A generic distributed DAG engine for HPC

Task graph for PDE solver

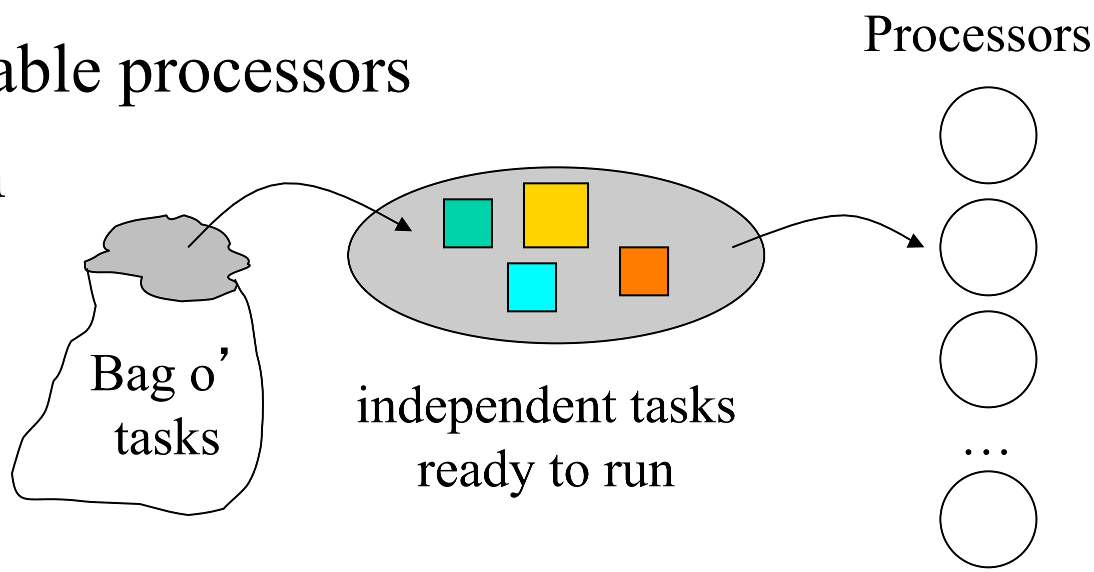




DAG of QR for a 4×4 tiles matrix on a 2×2 grid of processors.

Bag o' Tasks Model and Worker Pool

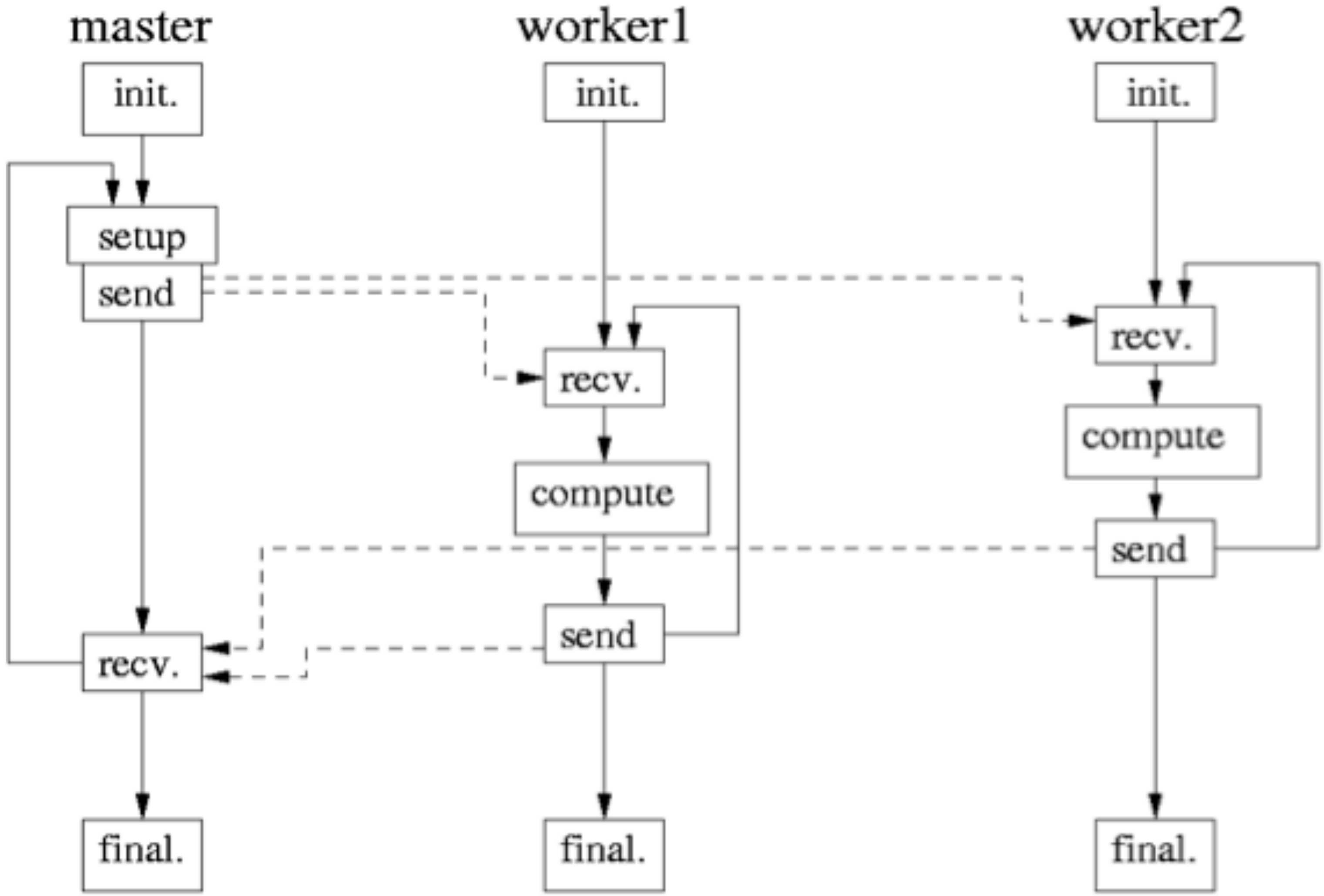
- Set of tasks to be performed
- How do we schedule them?
  - Find independent tasks
  - Assign tasks to available processors
- Bag o' Tasks approach
  - Tasks are stored in a bag waiting to run
  - If all dependencies are satisified, it is moved to a ready to run queue
  - Scheduler assigns a task to a free processor selected from a pool of (worker) processors



Master-Worker Parallelism

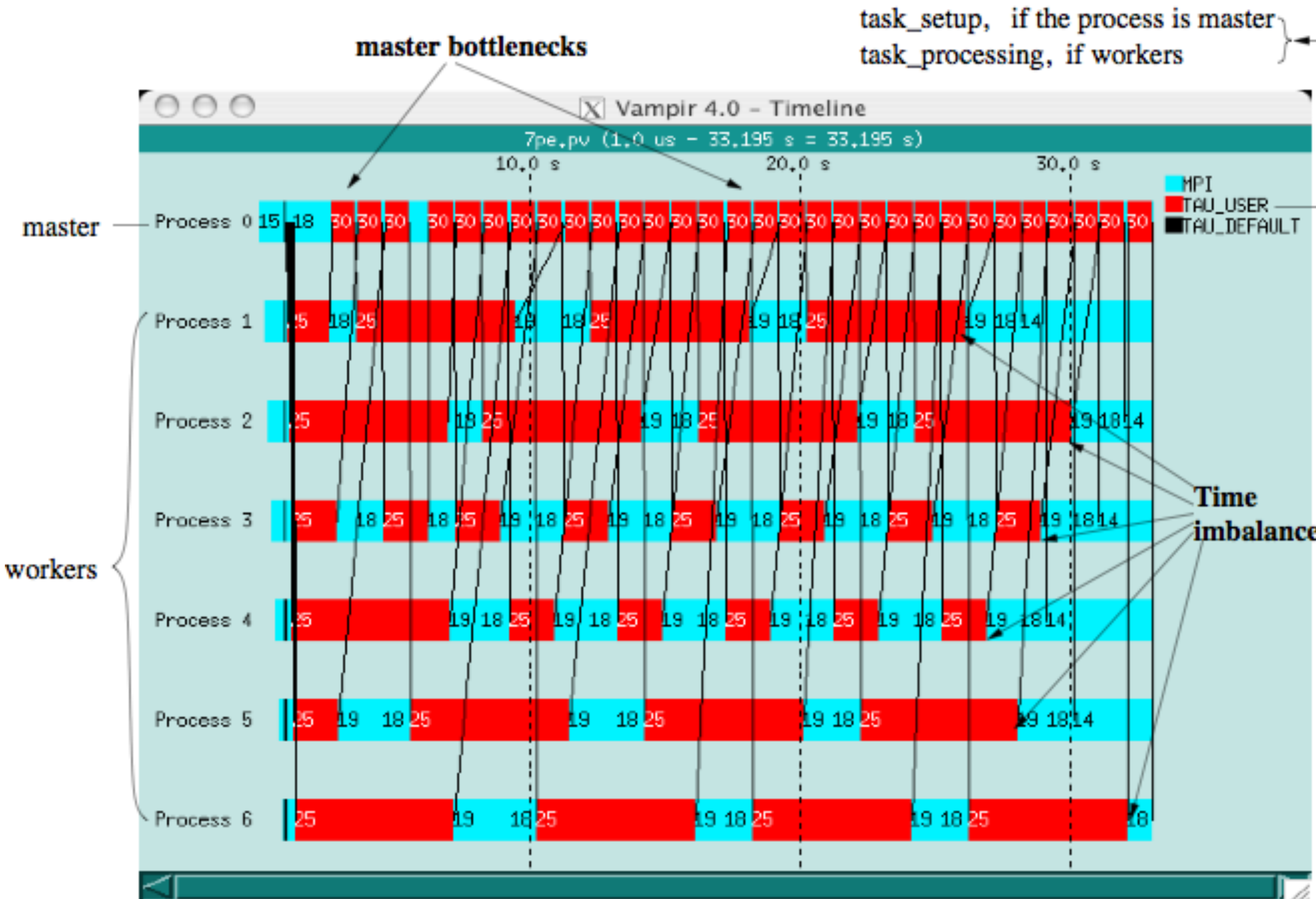
- One or more master processes generate work
- Masters allocate work to worker processes
- Workers idle if have nothing to do
- Workers are mostly stupid and must be told what to do
  - Execute independently
  - May need to synchronize, but most be told to do so
- Master may become the bottleneck if not careful
- What are the performance factors and expected performance behavior
  - Consider task granularity and asynchrony
  - How do they interact?

Master-Worker Execution Model (Li Li)



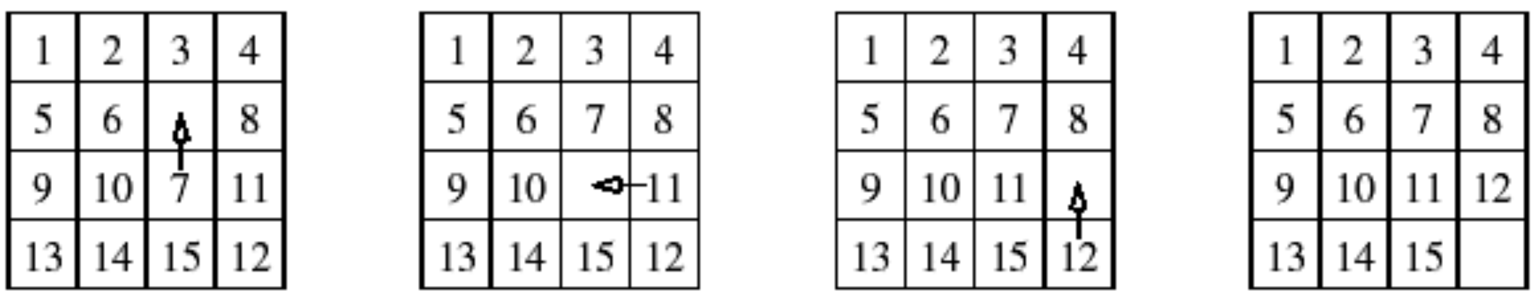
Li Li, "Model-based Automatics Performance Diagnosis of Parallel Computations," Ph.D. thesis, 2007.

M-W Execution Trace (Li Li)



Search-Based (Exploratory) Decomposition

- 15-puzzle problem
- 15 tiles numbered 1 through 15 placed in 4x4 grid
  - Blank tile located somewhere in grid
  - Initial configuration is out of order
  - Find shortest sequence of moves to put in order

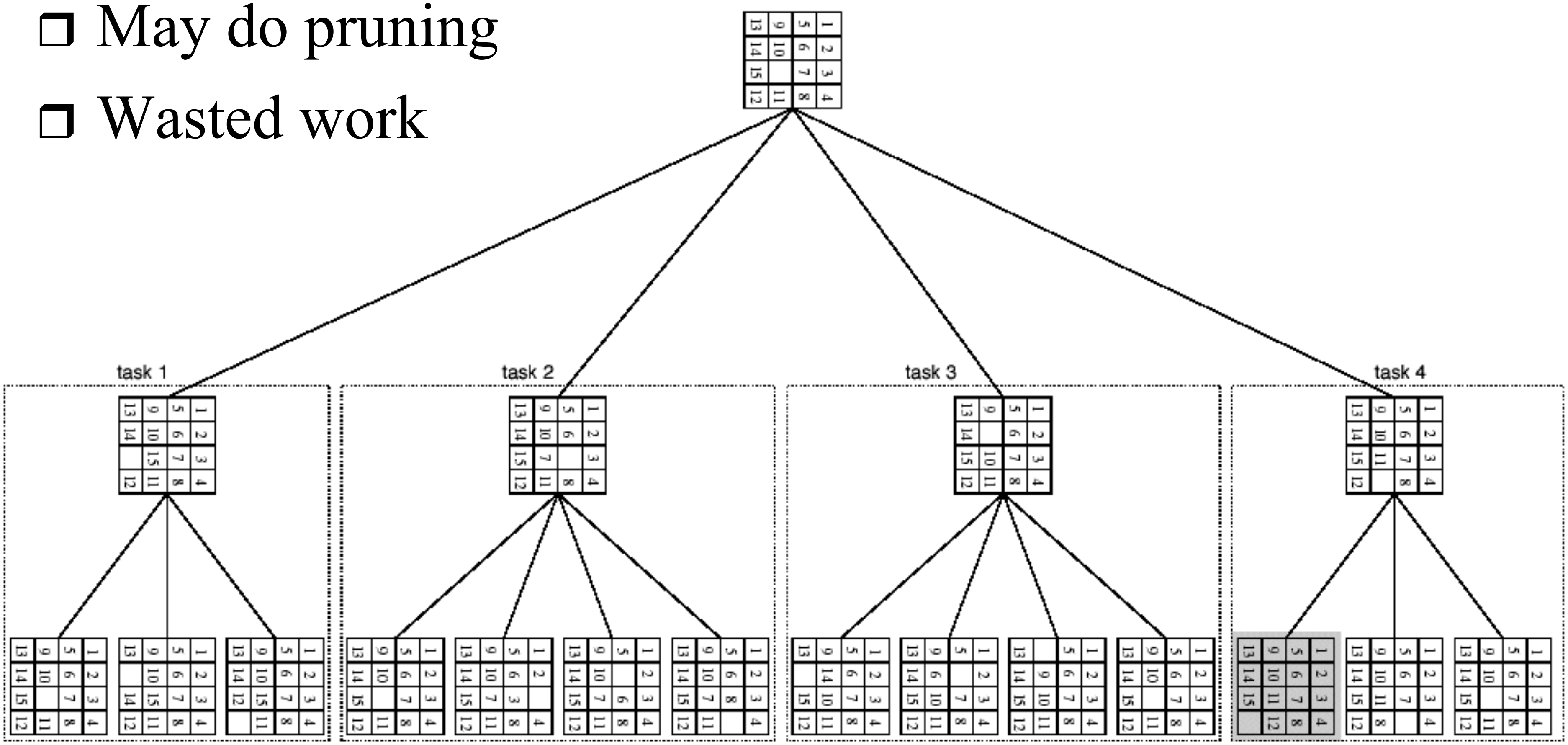


(a) (b) (c) (d)

- Sequential search across space of solutions
  - May involve some heuristics

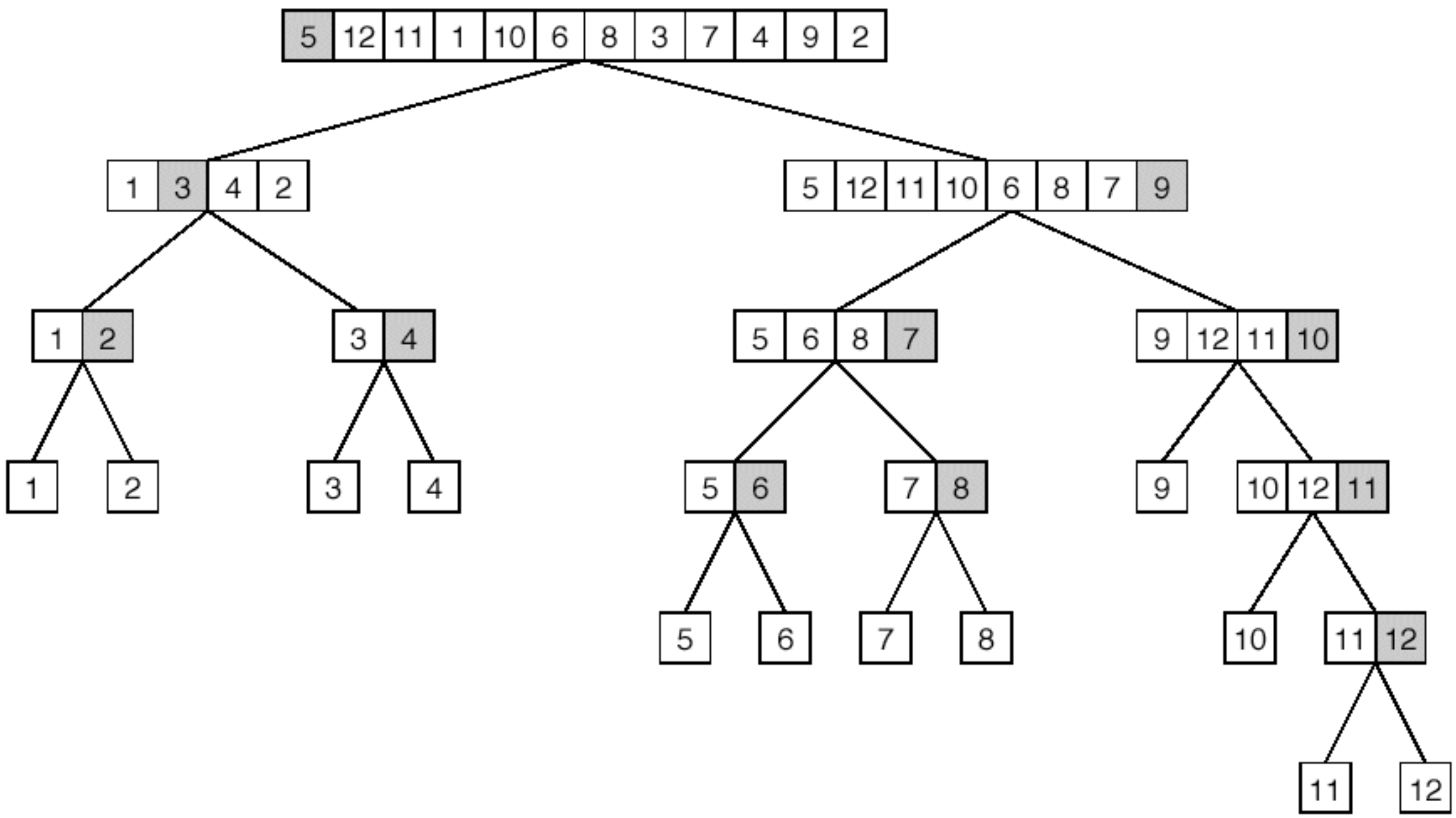
Parallelizing the 15-Puzzle Problem

- Enumerate move choices at each stage
- Assign to processors
- May do pruning
- Wasted work



Divide-and-Conquer Parallelism

- Break problem up in orderly manner into smaller, more manageable chunks and solve
- Quicksort example



Next Class

- Programming models
- Standard parallel programming techniques
  - shared memory (Pthreads)
  - message passing (MPI)
  - data parallelism (Fortran 90, CUDA)
  - shared memory + data parallelism (OpenMP)
  - object-oriented parallelism (?)