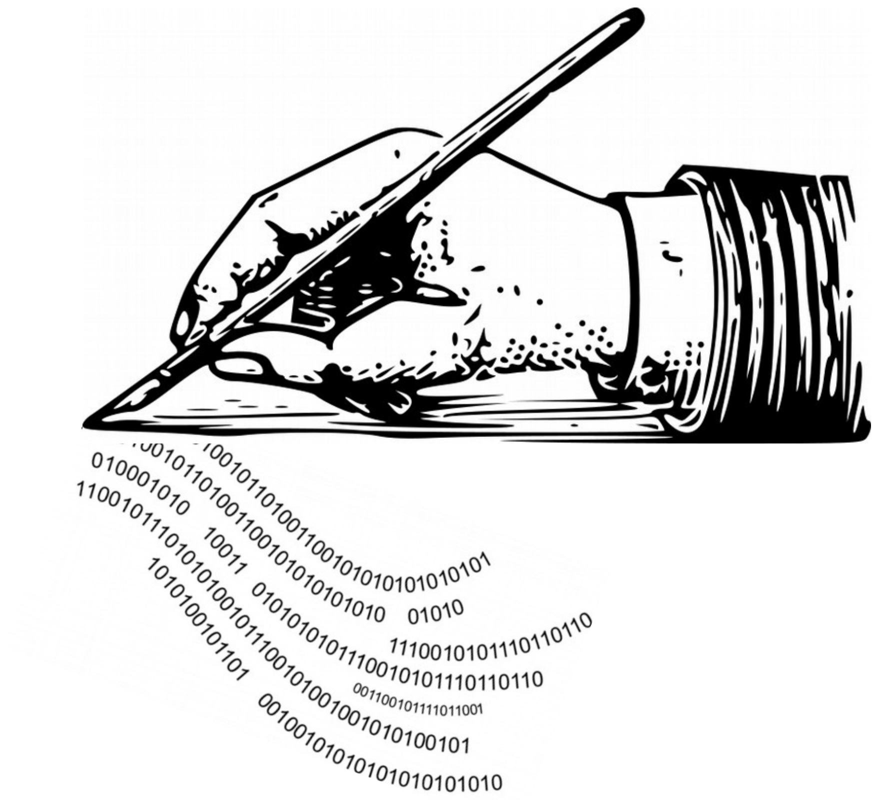
CS 470
Spring 2025

Mike Lam, Professor



Parallel Algorithms

Graphics and content taken from IPP section 2.7 and the following:
http://www.mcs.anl.gov/~itf/dbpp/text/book.html
http://compsci.hunter.cuny.edu/~sweiss/course_materials/csci493.65/lecture_notes/chapter03.pdf *(no longer accessible)*
https://fenix.tecnico.ulisboa.pt/downloadFile/3779577334688/cpd-11.pdf *(no longer accessible)*

Parallel program development

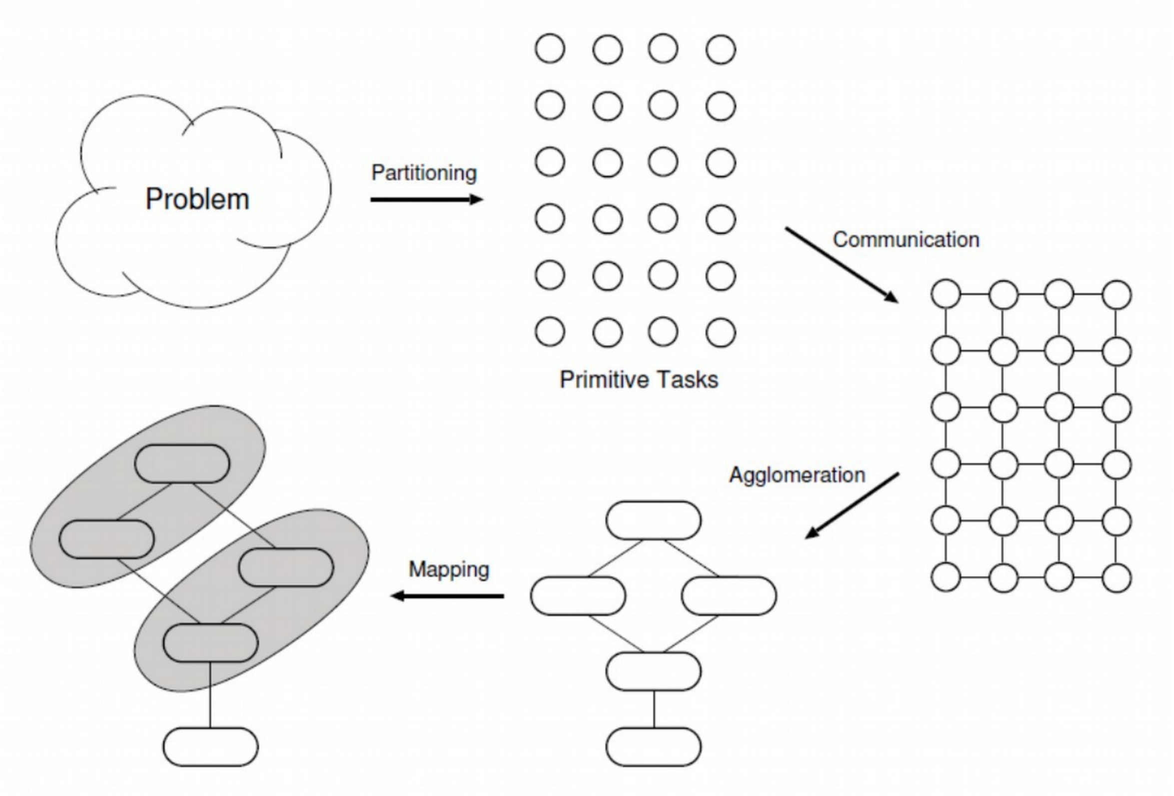
- **Writing efficient parallel code is hard**
- We have covered two generic paradigms ...
  - Shared-memory multithreading
  - Distributed message-passing
- … and four specific technologies
  - Pthreads
  - OpenMP
  - CUDA
  - MPI
- Given a problem, how do we *more generally* approach the development of a parallel program that solves it?

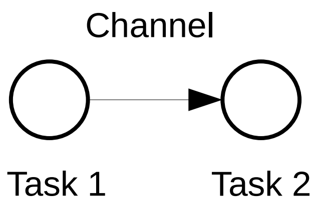
Method vs. methodology

- Method: a systematic process or way of doing a task
- Methodology: analysis of methods relevant to a discipline
  - Literally: "the study of methods"
  - Goal: guidelines or best practices for a class of methods
- Parallel algorithms
  - There is no single **method** for creating efficient parallel algorithms
  - However, there are some good **methodologies** that can guide us
  - Requires more abstract thinking about problems
  - We will study one: Foster's methodology

Foster's methodology

- Task: executable unit along with local memory and I/O ports
- Channel: message queue connecting tasks' input and output ports
- Drawn as a graph, tasks are vertices and channels are edges
- Steps:
  1) Partitioning
  2) Communication
  3) Agglomeration
  4) Mapping





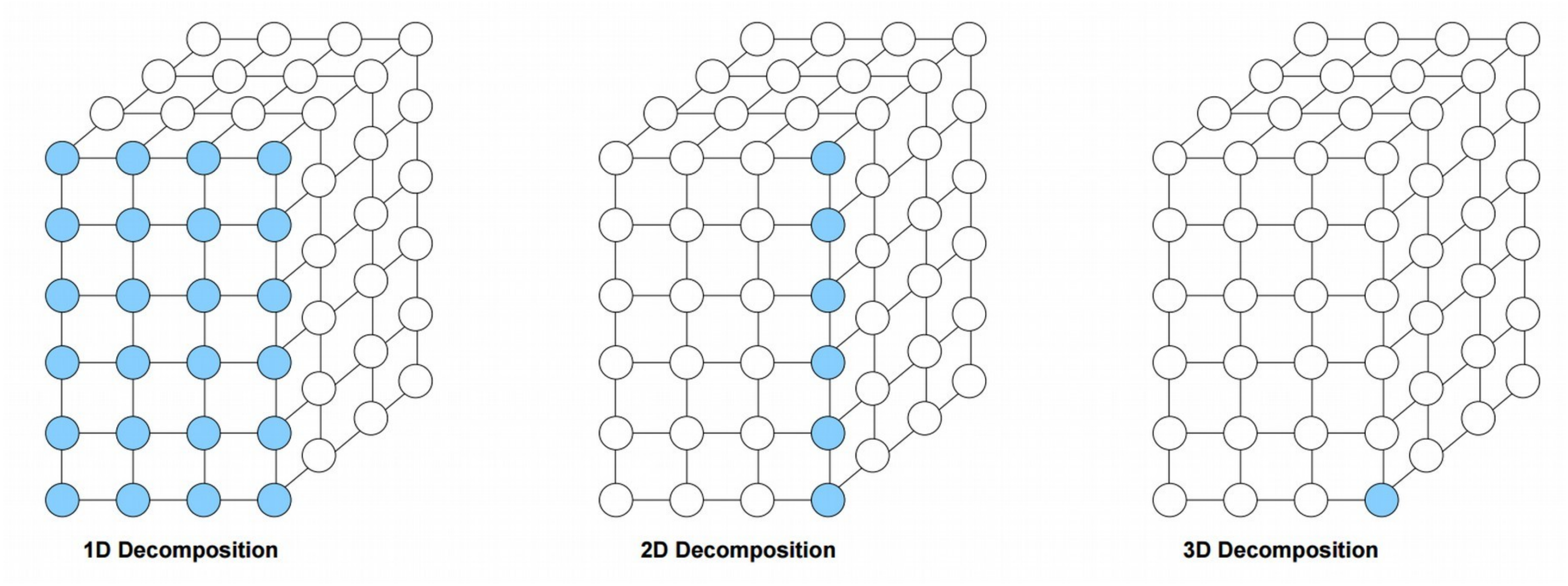
Foster's textbook is online:
http://www.mcs.anl.gov/~itf/dbpp/text/book.html

Partitioning

- Goal: discover as much parallelism as possible
- Divide computation into as many primitive tasks as possible
  - Avoid redundant computation
  - Primitive tasks should be roughly the same size
  - Number of tasks should increase as the problem size increases
    - This helps ensure good scaling behavior
  - Data vs. task decomposition

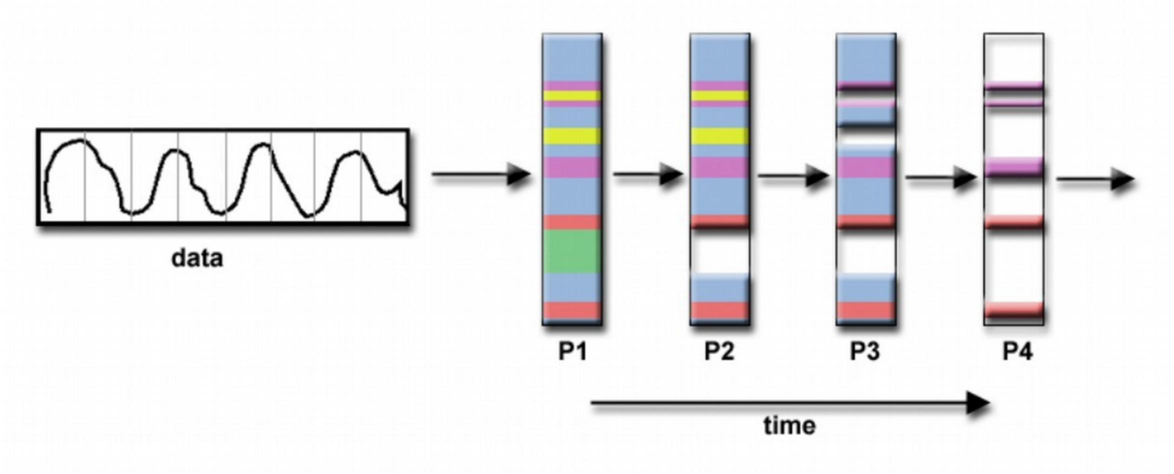
Partitioning

- Domain ("data") decomposition
  - Break tasks into segments of various granularities by data

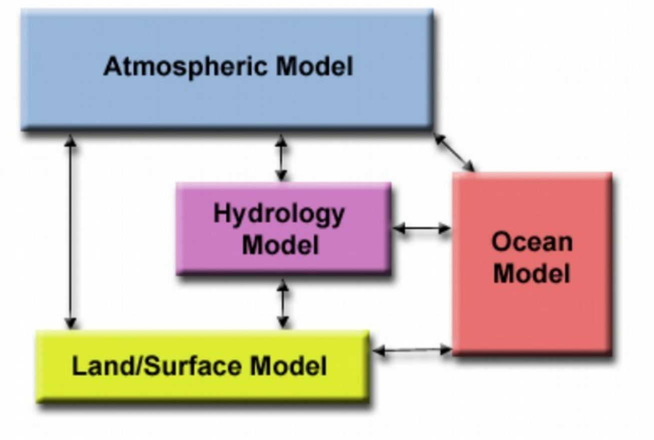


Partitioning

- Functional ("task") decomposition
  - Separation by task type
  - Domain/data decomposition can often be used inside of individual tasks



Pipelined



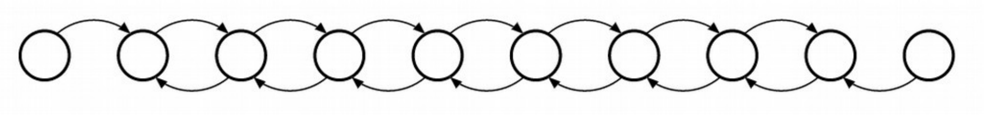
Non-pipelined

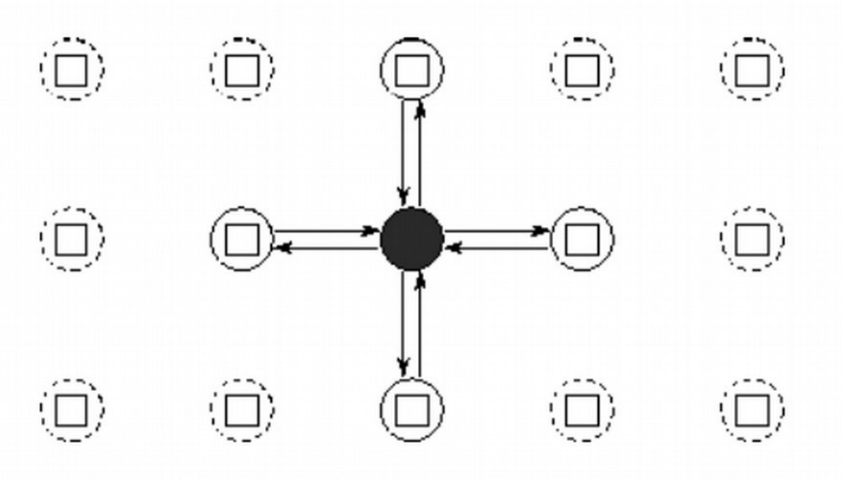
Communication

- Goal: minimize overhead
- Identify which tasks must communicate and how
  - Local (few tasks) vs. global (many tasks)
  - Structured (regular) vs. unstructured (irregular)
  - Prefer local, structured communication
  - Tasks should perform similar amounts of communication
    - This helps with load balancing
  - Communication should be concurrent wherever possible

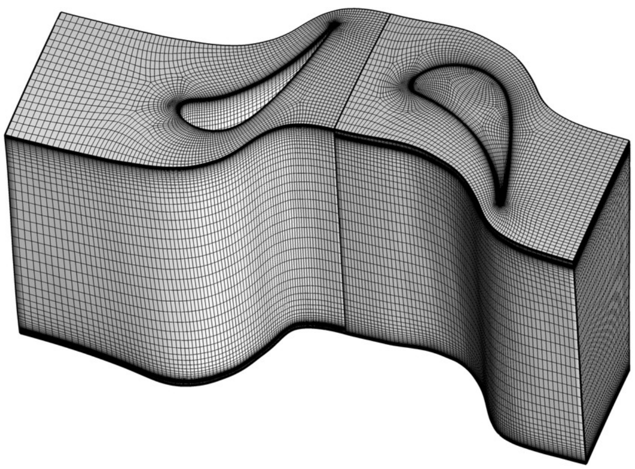
Communication

- Examples of local communication:





Structured

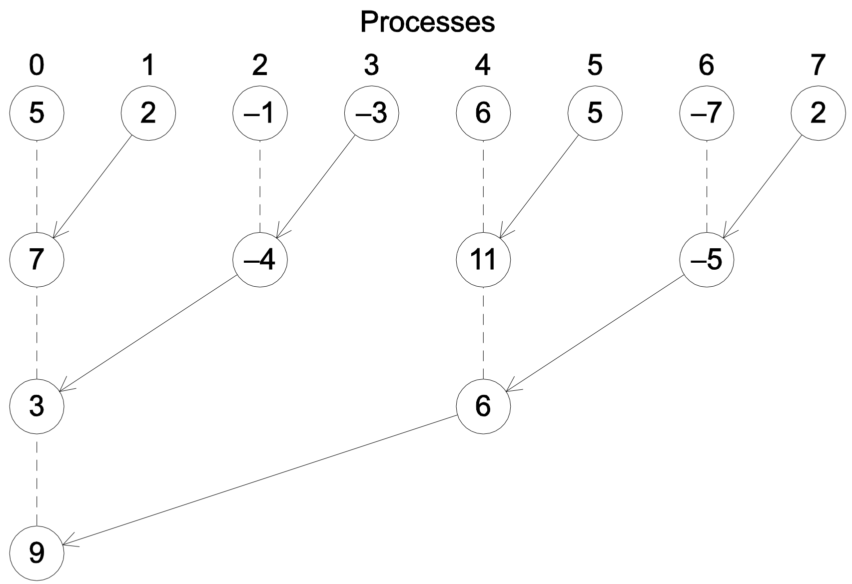


Unstructured

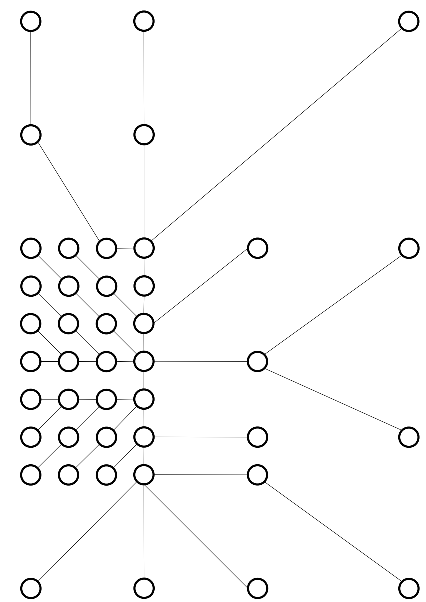
http://dx.doi.org/10.1051/epjconf/201611402115

Communication

- Examples of global communication:



Structured



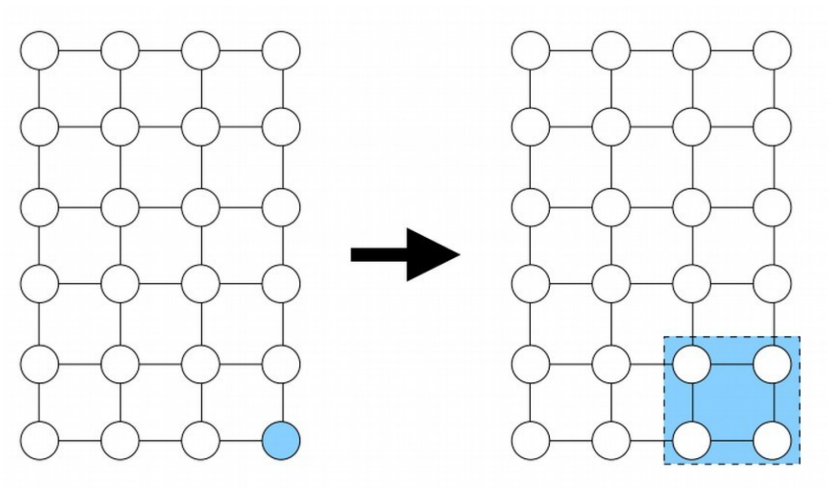
Unstructured

Agglomeration

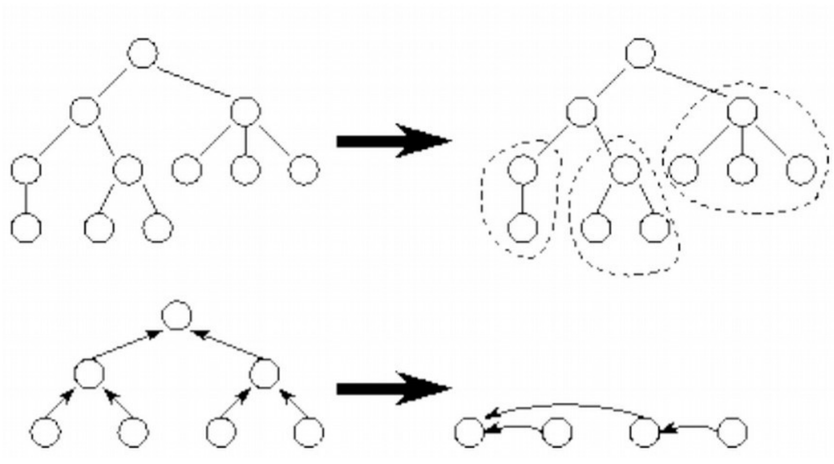
- Goal: Reduce messages and simplify programming
- Combine tasks into groups, increasing locality
  - Groups should have similar computation and communication costs
  - Task counts should still scale with processor count and /or problem size
  - Minimize software engineering costs
    - Agglomeration can prevent code reuse

Agglomeration

- Examples:



Agglomeration of four local tasks



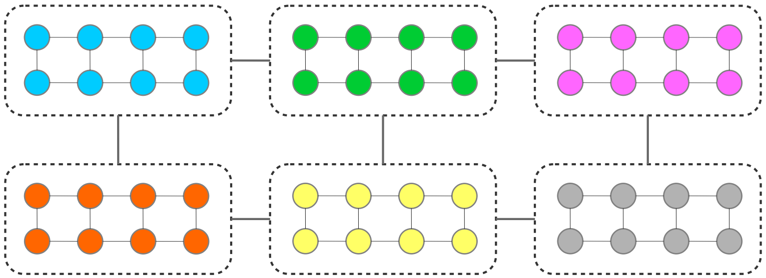
Agglomeration of tree-based tasks

Mapping

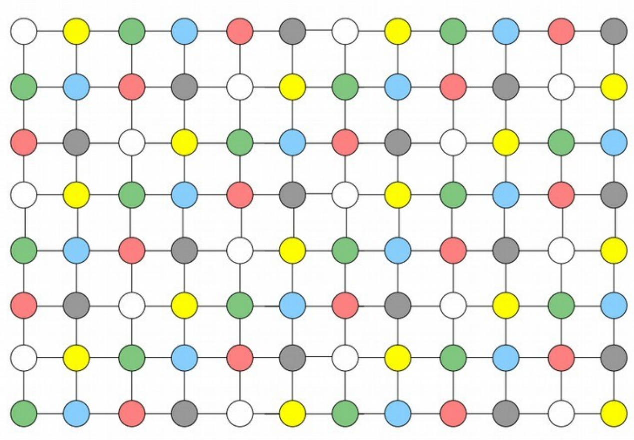
- Assign tasks (or task groups) to processors/nodes
  - Block vs. cyclic
  - Static vs. dynamic
- Goal: minimize execution time
  - Alternately: maximize processor utilization
  - On a distributed system: minimize communication
- Strategies:
  - 1) Place concurrent tasks on different nodes
  - 2) Place frequently-communicating tasks on the same node
- Problem: these strategies are **often** in conflict!
  - The general problem of optimal mapping is NP-complete

Mapping

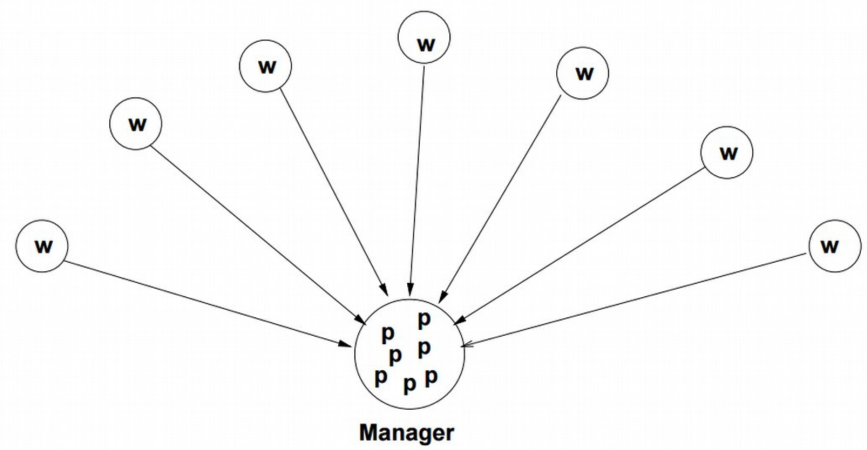
- Examples:



Block mapping



Cyclic mapping



Dynamic mapping

Boundary Value Problem

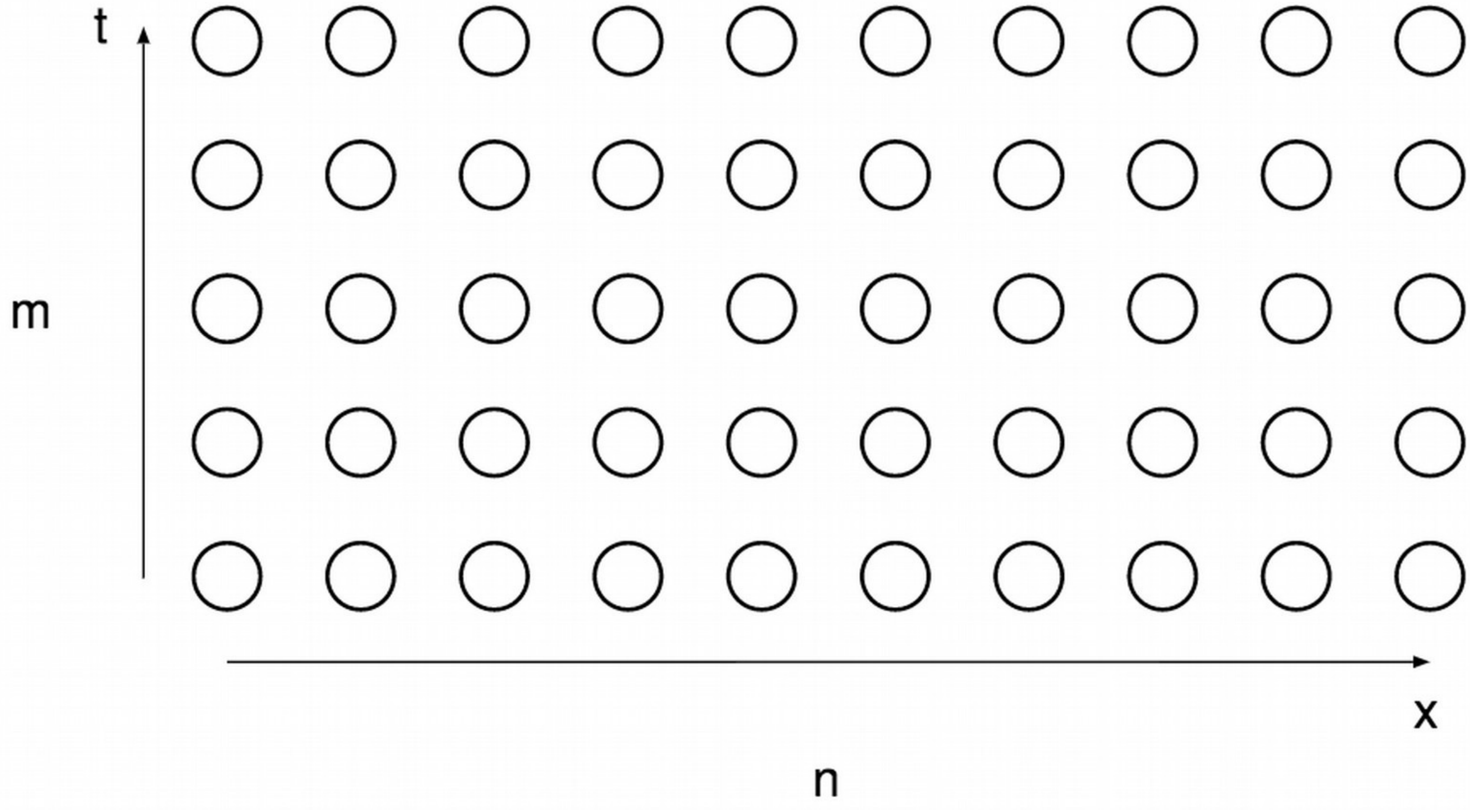
- Problem
  - General statement: *Determine the temperature changes in a thin cylinder of uniform material with constant-temperature boundary caps over a given time period, given the size of the cylinder and its initial temperature*
  - General solution: solve partial differential equation(s)
    - Often too difficult or expensive to solve analytically
  - Approximate solution: finite difference method
    - Discretize space (1d grid) and time (ms)
    - Approximate solution using Taylor polynomials

- Goal: Parallelize this solution, using Foster's methodology as a guide

Boundary Value Problem

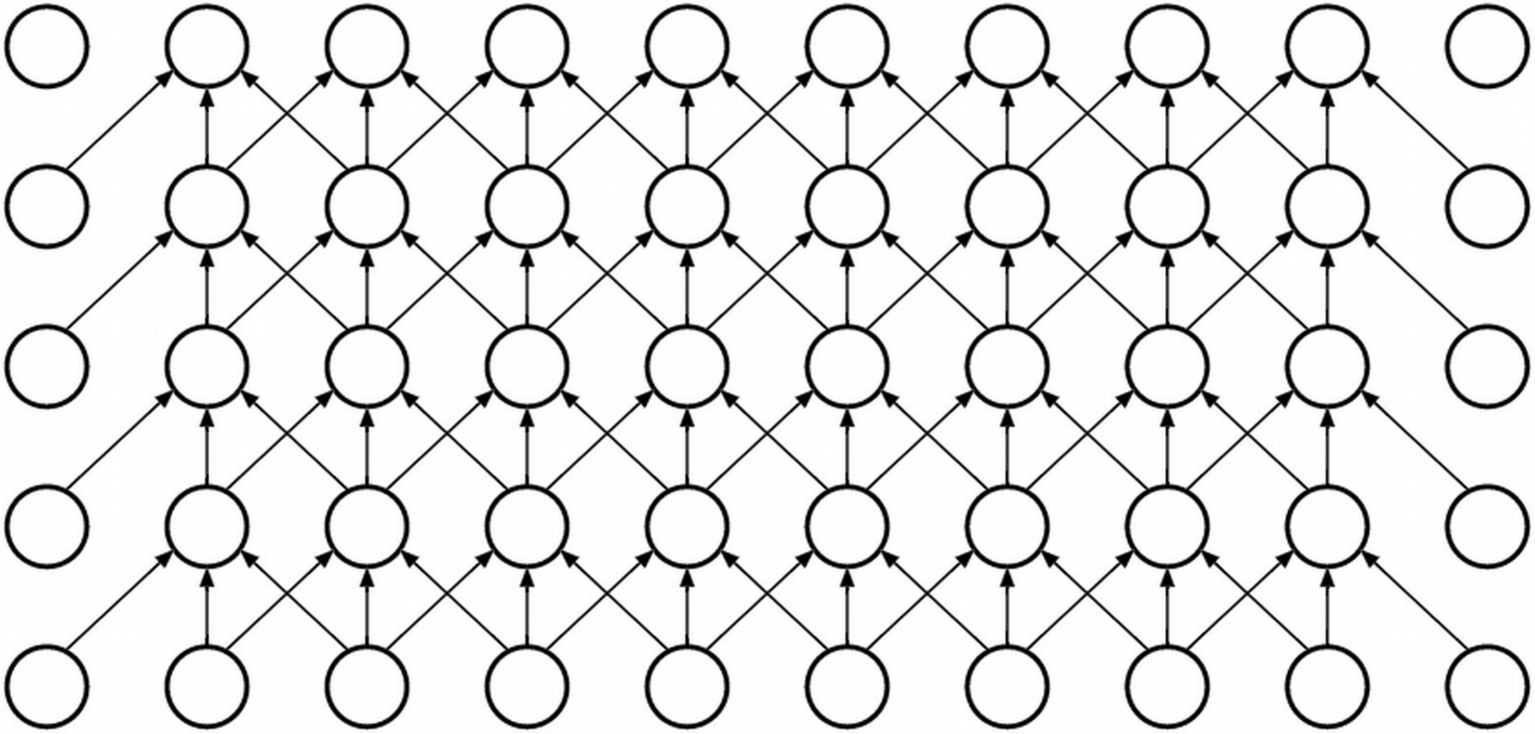
Partitioning:

Make each $T(x, t)$ computation a primitive task.
$\Rightarrow$ 2-dimensional domain decomposition



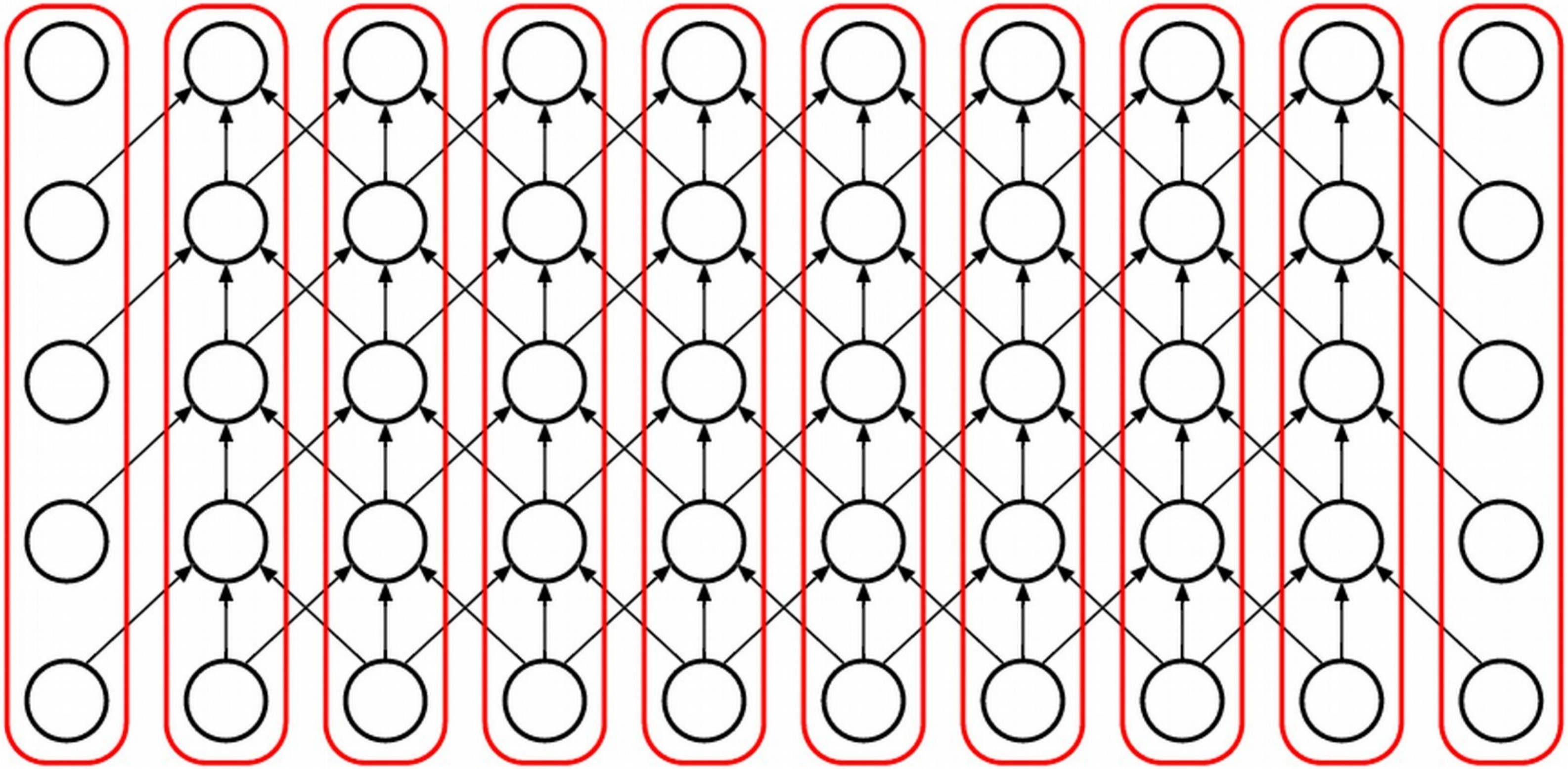
Boundary Value Problem

Communication:



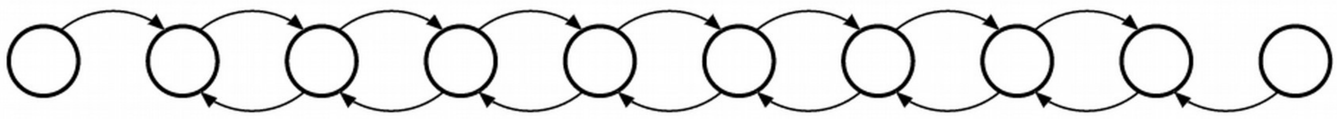
Boundary Value Problem

Agglomeration:

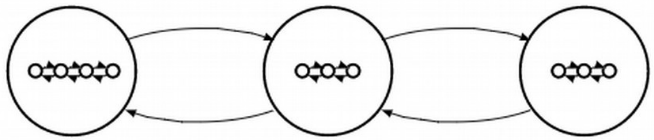


Boundary Value Problem

Agglomeration:



Mapping:



Pseudocode:

for each group:

for each timestep:

for each slice:

compute new temperatures

exchange temperatures w/ nearest neighbors

Boundary Value Problem

- Pseudocode:

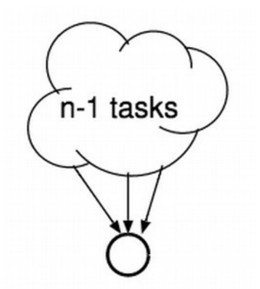
```
for each group:
    for each timestep:
        for each slice:
            compute new temperatures
        exchange temperatures w/ nearest neighbors
```

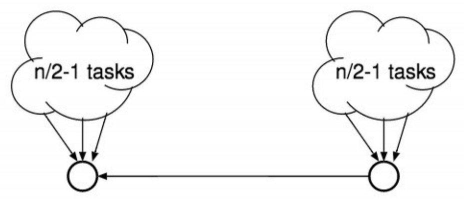
Finding a maximum

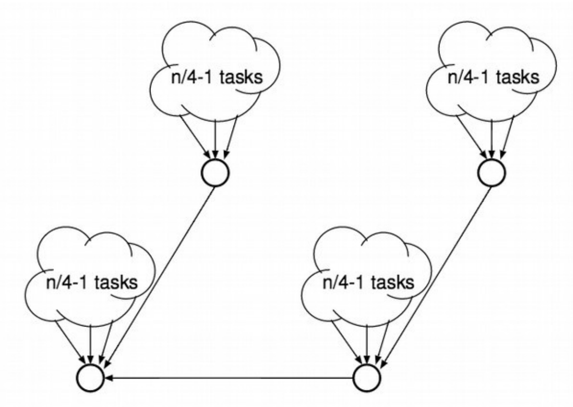
- Problem: Determine the maximum value among some large set of given values
  - Special case of a reduction
- Goal: Parallelize this solution, using Foster's methodology as a guide

Finding a maximum

- Partitioning: each pairwise comparison is a primitive task
  - (1d domain decomposition)
  - One task (root) will compute final solution
- Communication: divide-and-conquer
  - Root task needs to compute max after n-1 tasks
  - Keep splitting the input space in half



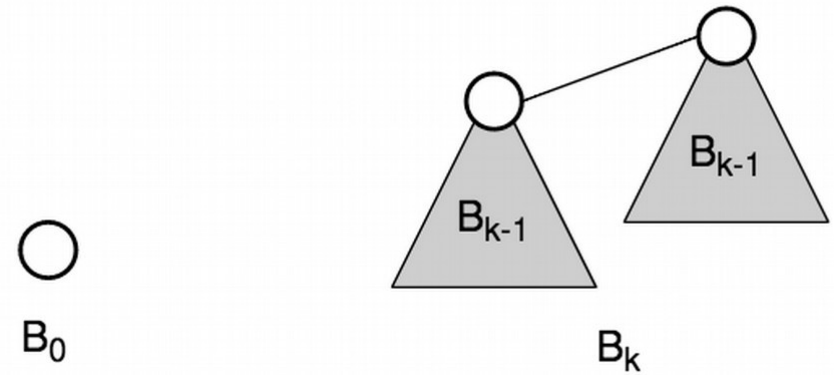




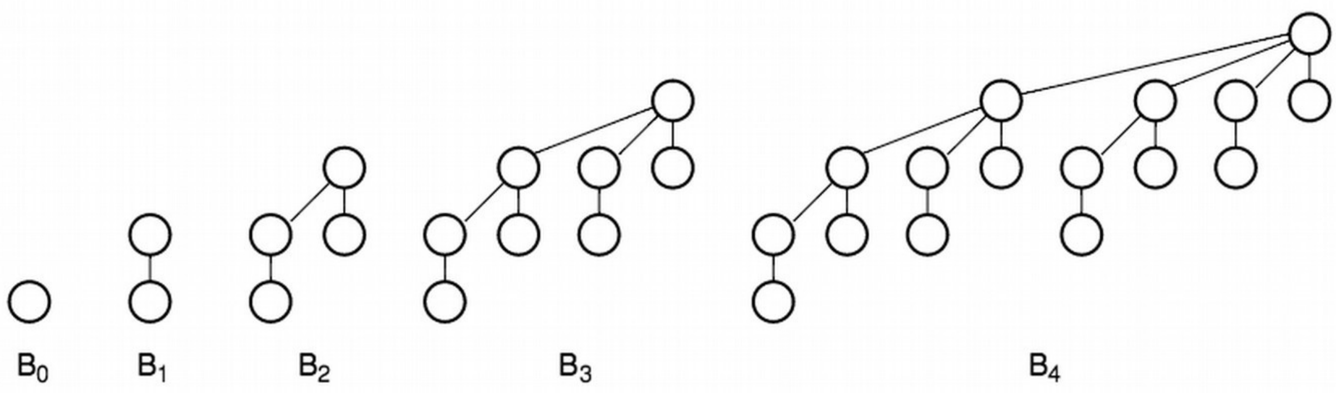
Finding a maximum

- Binomial tree with $n = 2^k$ nodes

Recursive definition:



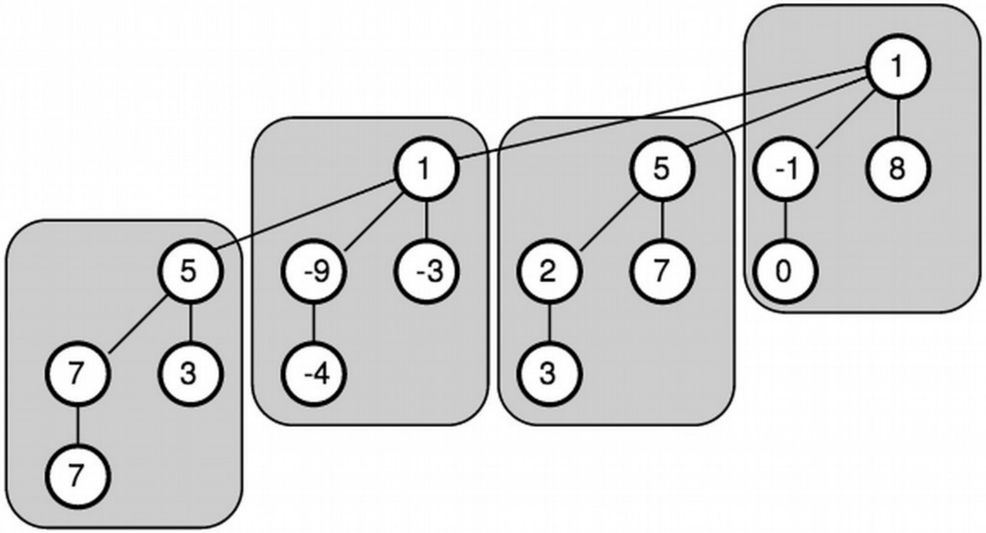
Examples:



Finding a maximum

Agglomeration:

Group n leafs of the tree:

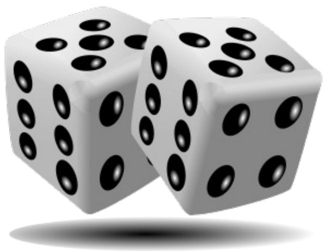


Mapping:

The same (actually, in the agglomeration phase, use n such that you end up with p tasks).

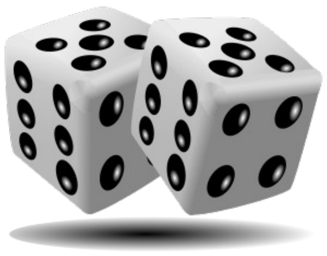
Random number generation

- Goal: Generate pseudo-random numbers in a distributed way
- Problem: We wish to retain some notion of reproducibility
  - Results should be **deterministic**, given the RNG seed
  - We can't depend on the ordering of distributed communications
- Problem: Avoiding duplicated series of generated numbers
  - This means we can't just use the same generator in all processes



Random number generation

- Naive solution:
  - Generate all numbers on one node and scatter them
    - Like in our MPI analysis lab
  - Too slow!
- Can we do better? (Foster's)
  - Generating each random number is a task
  - Channels between subsequent numbers from the same seed
  - Tweak communication & agglomeration
  - Minimize dependencies



Random number generation

Goal:
Uniform randomness and reproducibility

$$
\begin{aligned}
L_{k+1} &= a_L L_k \bmod m \\
R_{k+1} &= a_R R_k \bmod m
\end{aligned}
$$

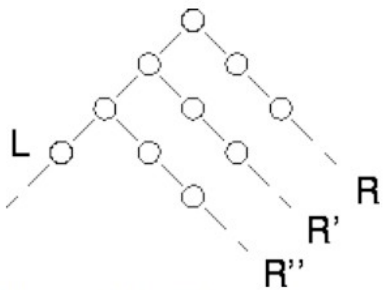


Figure 10.1: *The random tree method. Two generators are used to construct a tree of random numbers. The right generator is applied to elements of the sequence L generated by the left generator to generate new sequences R, R', R'', etc.*

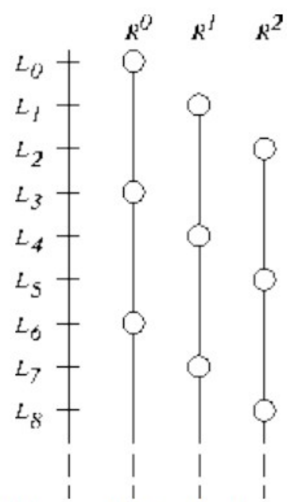


Figure 10.2: *The leapfrog method with n=3. Each of the three right generators selects a disjoint subsequence of the sequence constructed by the left generator's sequence.*

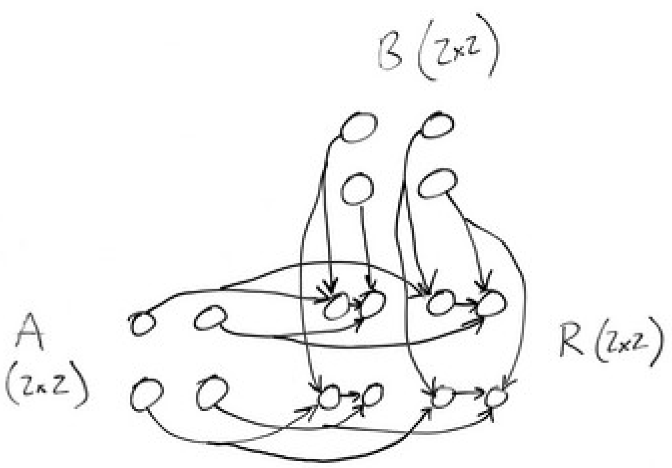
More info in Chapter 10 of
http://www.mcs.anl.gov/~itf/dbpp/text/book.html

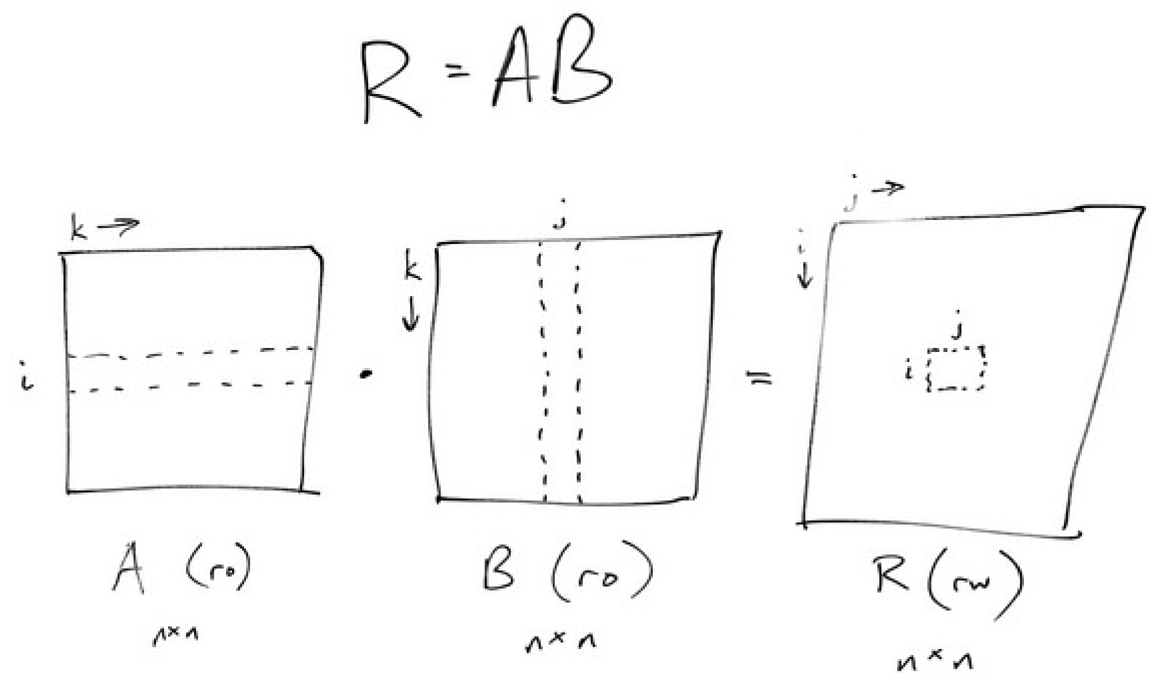
Tasks and data access patterns

- Often, data access patterns describe “communication” in Foster’s methodologies
- Examples:
  - Matrix multiplication
  - Gaussian elimination (P2)
  - Backward substitution (P2)

Data access patterns

```
void multiply_matrices(int *A, int *B, int *R, int n)
{
    int i, j, k;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            R[i*n+j] = 0;
            for (k = 0; k < n; k++) {
                R[i*n+j] += A[i*n+k] * B[k*n+j];
            }
        }
    }
}
```



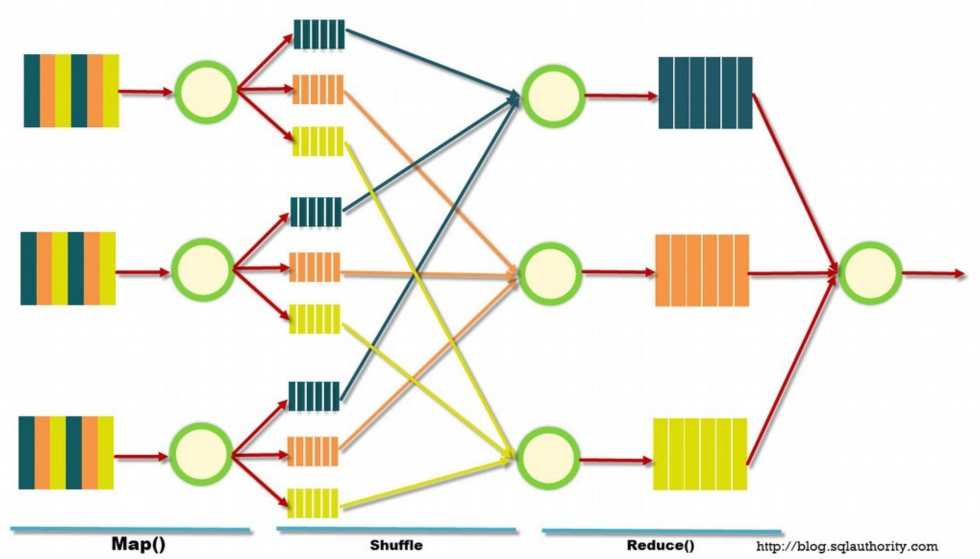


Common paradigms

- Grid/mesh-based nearest-neighbor simulation
  - Often includes math-heavy computations
    - Linear algebra and systems of equations
    - Dense vs. sparse matrices
  - Newer: adaptive mesh and multigrid simulations
- Worker pools / task queues
  - Newer: adaptive cloud computing
- Pipelined task phases
  - Newer: MapReduce
- Divide-and-conquer tree-based computation
  - Often combined with other paradigms (worker pools and pipelines)

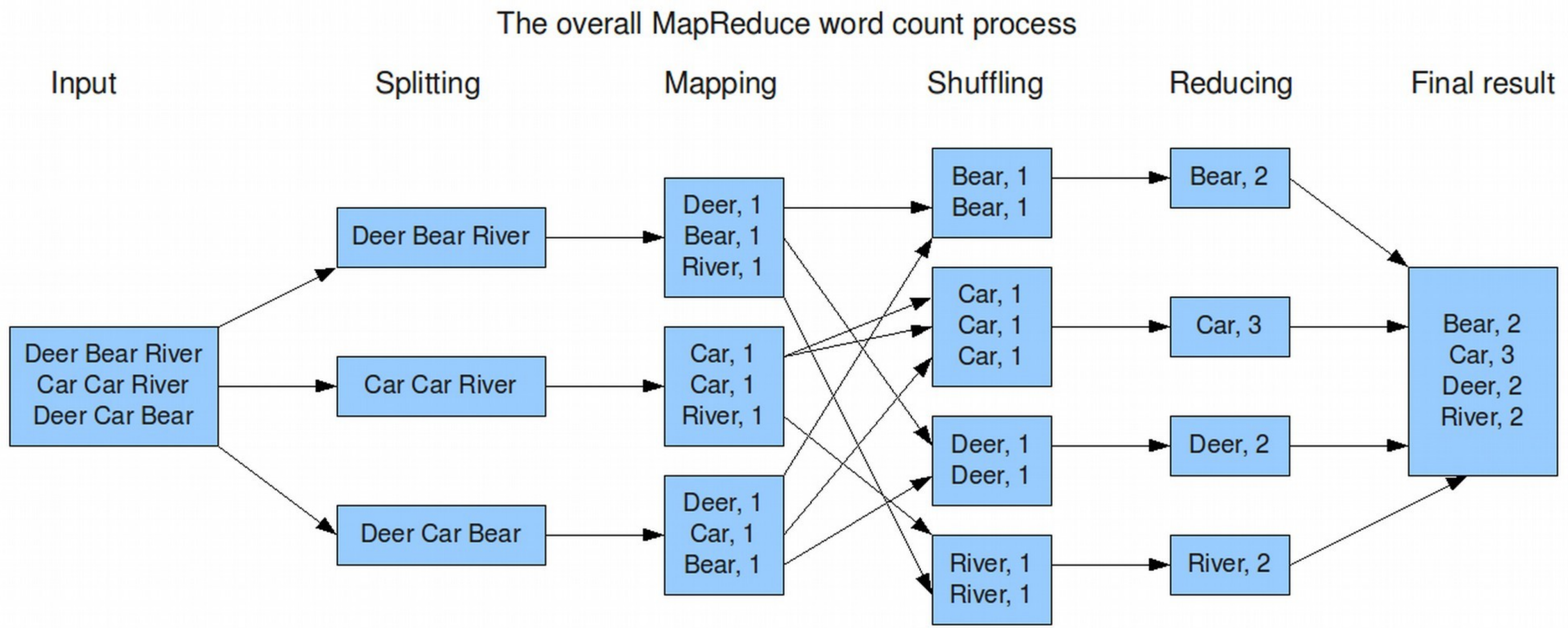
MapReduce

- Parallel/distributed system paradigm for "big data" processing
  - Uses a specialized file system and takes advantage of independent tasks
  - Originally developed at Google (along with GFS)
  - Currently popular: Apache Hadoop and HDFS
    - General languages: Java, Python, Ruby, etc.
    - Specialized languages: Pig (data flow language) or Hive (SQL-like)
    - Growing quickly: Apache Spark (more generic w/ in-memory processing)
    - For streaming data: Apache Storm, Google BigQuery, Azure Synapse
- Phases
  - Map (process local data)
  - Shuffle (distributed sort)
  - Reduce (combine results)



MapReduce

- Word count example



Apache Spark (Python)

WORD COUNT

```
text_file = sc.textFile("hdfs://docs/input.txt")
counts = text_file.flatMap(lambda line: line.split(" ")) \
                  .map(lambda word: (word, 1)) \
                  .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://results/counts.txt")
```

MONTE CARLO PI

```
def sample(p):
    x, y = random(), random()
    return 1 if x*x + y*y < 1 else 0

count = sc.parallelize(xrange(0, NUM_SAMPLES)) \
          .map(sample) \
          .reduce(lambda a, b: a + b)
print "Pi is roughly %f" % (4.0 * count / NUM_SAMPLES)
```

Big Data

- Big data is a broad term for analyzing or processing large data sets
  - Exact size depends on the organization and task
  - Ranges from gigabytes to petabytes or exabytes
  - Often requires handling streaming data
  - Informally understood to begin at “the point at which the current approach begins to fail”
  - Requires new tools or a revised approach

Data Science

- Data science is an interdisciplinary field that extracts knowledge and insight from data
  - The data are often large, unstructured, and/or noisy
  - Motivation often comes from social sciences
  - Process usually informed by statistics
  - Analysis usually requires application of CS
    - Databases and data processing
    - Machine learning and artificial intelligence
    - Data visualization and human-computer interaction
    - Parallel and distributed systems

RStudio®

A word of caution

- It is easy to over-engineer “big data” solutions
  - Most “big data” problems aren’t really that big
    - E.g., if your data set fits on a single hard drive, it’s probably not a big data problem
  - Traditional pipe-based or shared-memory solutions will be simpler and possibly even faster
    - Case study: “Command-line Tools can be 235x Faster than your Hadoop Cluster”
      - https://adamdrake.com/command-line-tools-can-be-235x-faster-than-your-hadoop-cluster.html
    - KISS principle: “Keep It Short and Simple”