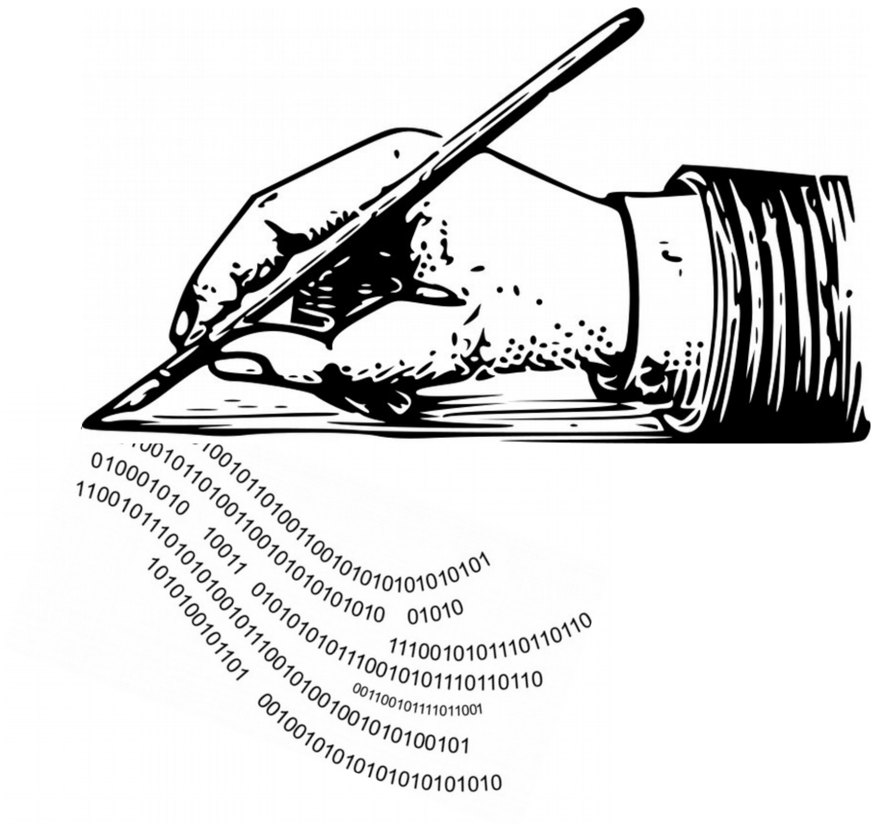
CS 470
Spring 2017

Mike Lam, Professor



Parallel Algorithm Development

(Foster's Methodology)

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
https://fenix.tecnico.ulisboa.pt/downloadFile/3779577334688/cpd-11.pdf

Parallel program development

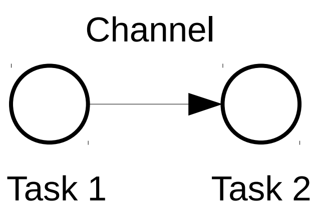
- **Writing efficient parallel code is hard**
- We've covered two generic paradigms ...
  - Shared-memory
  - Distributed message-passing
- … and three specific technologies
  - Pthreads
  - OpenMP
  - MPI
- Given a problem, how do we approach the development of a parallel program that solves it?

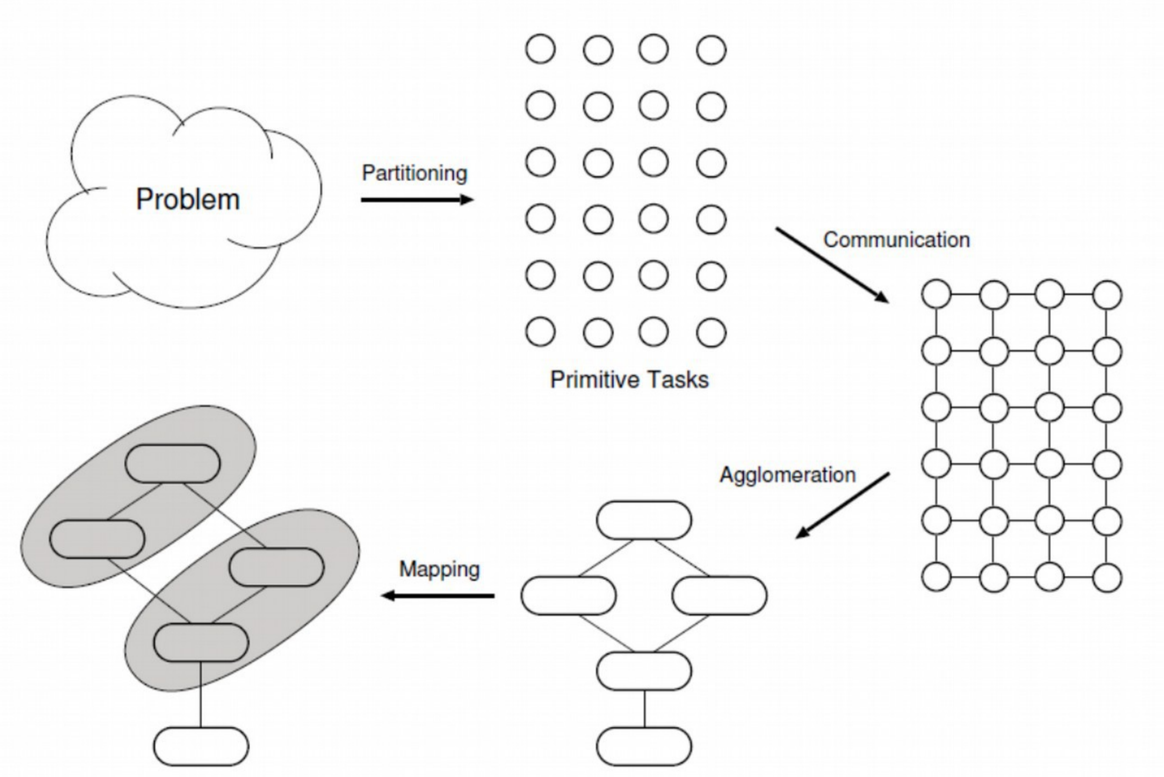
Method vs. methodology

- 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
  - 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





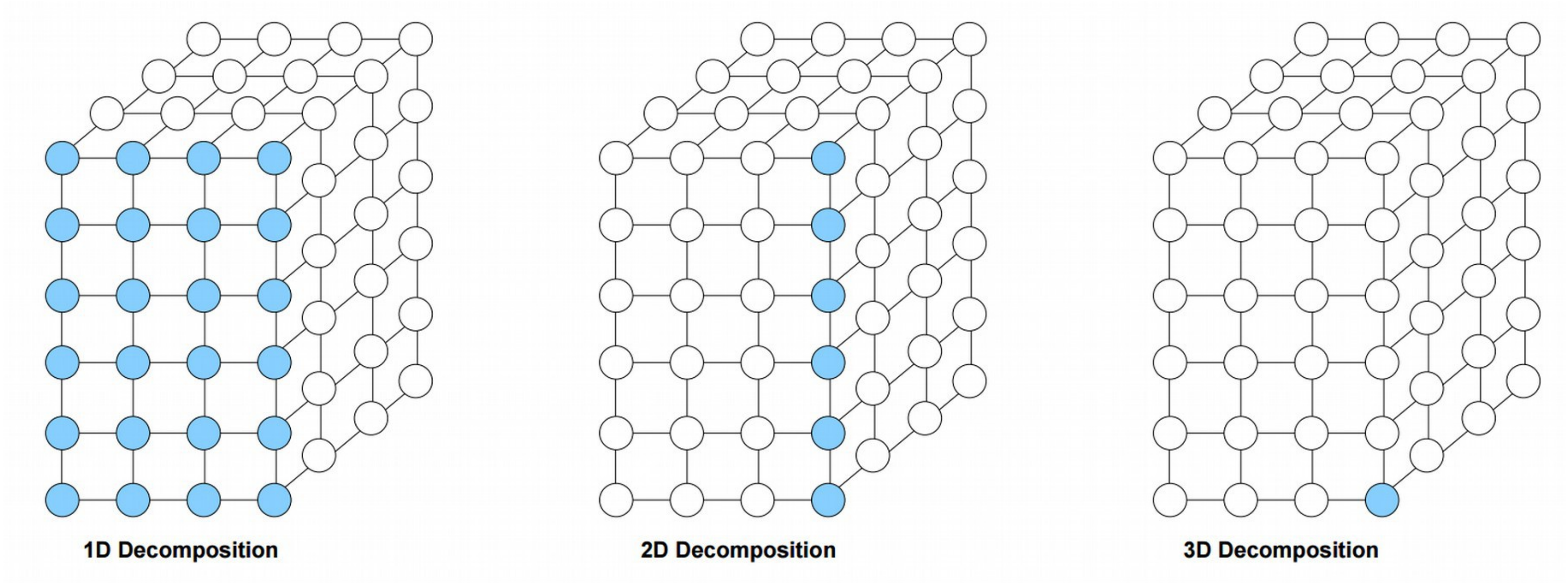
Foster's textbook is online:
`http://www.mcs.anl.gov/~itf/dbpp/text/book.html`

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

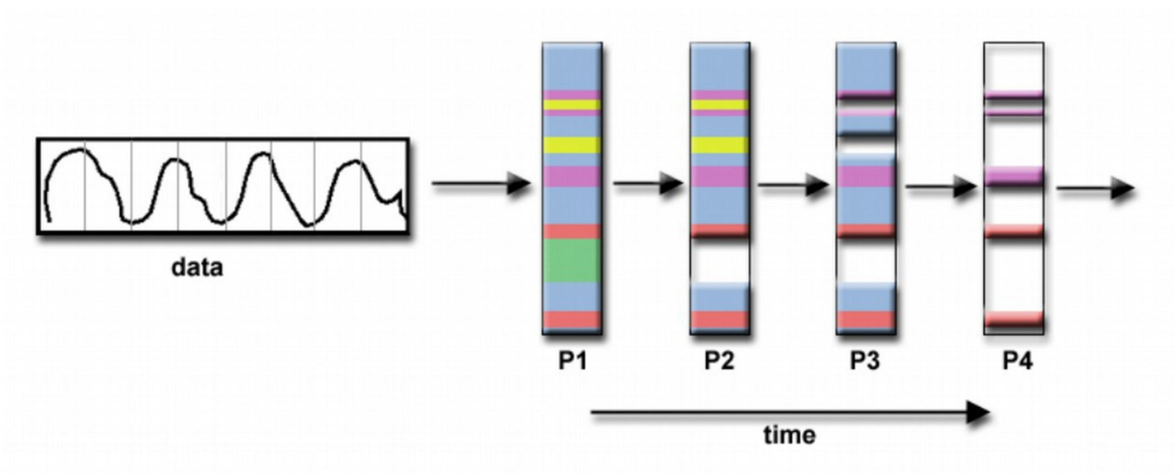
Partitioning

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

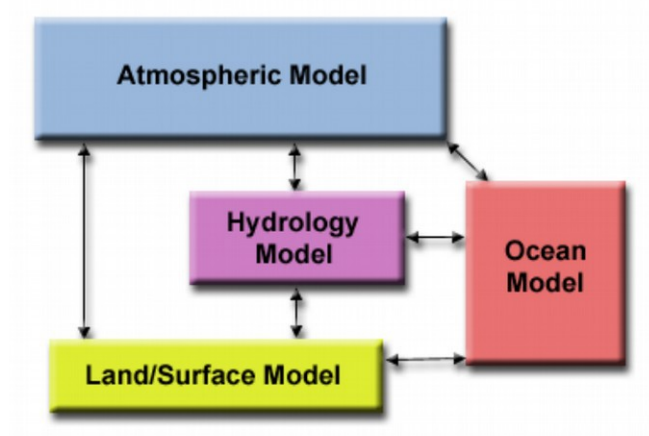


Partitioning

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



Pipelined



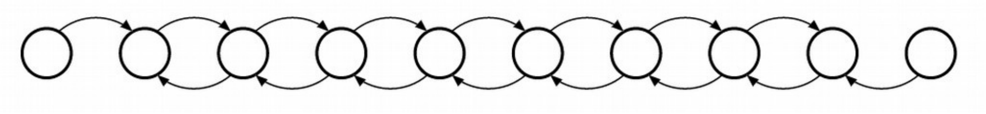
Non-pipelined

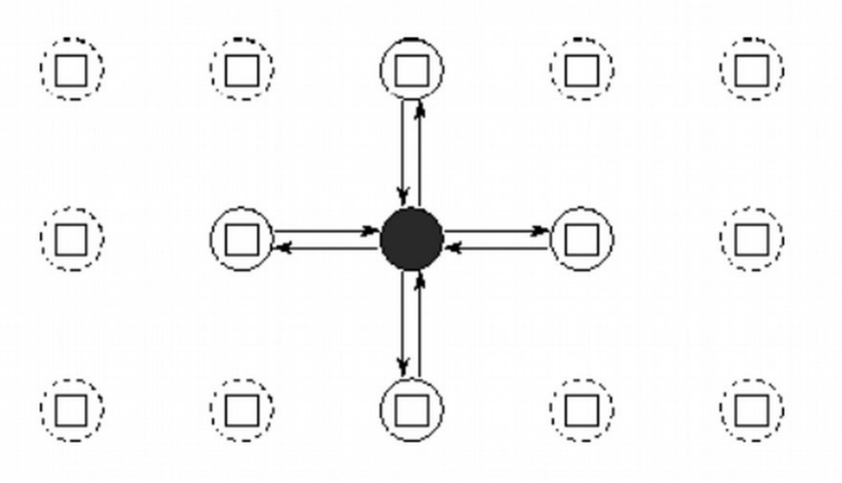
Communication

- 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

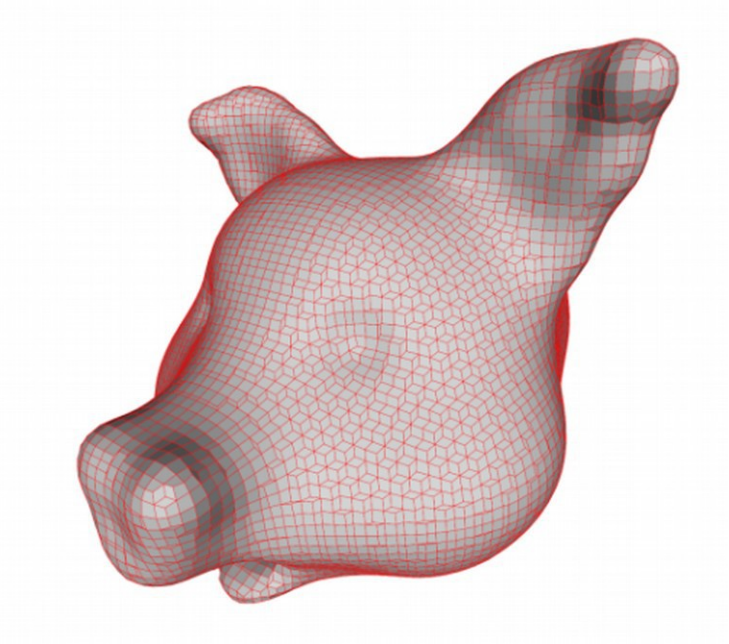
Communication

- Examples of local communication:





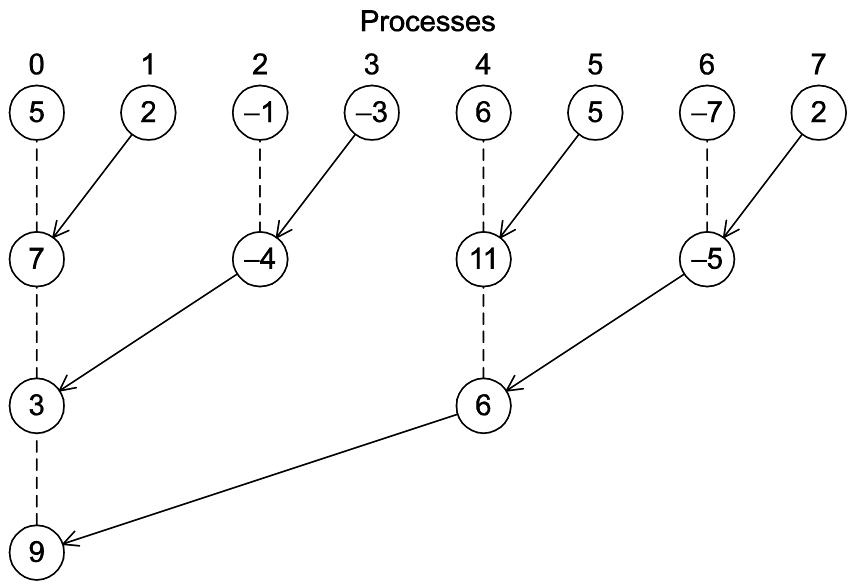
Structured



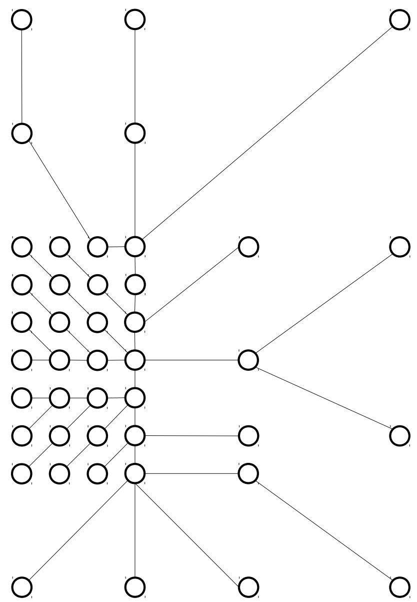
Unstructured

Communication

- Examples of global communication:



Structured



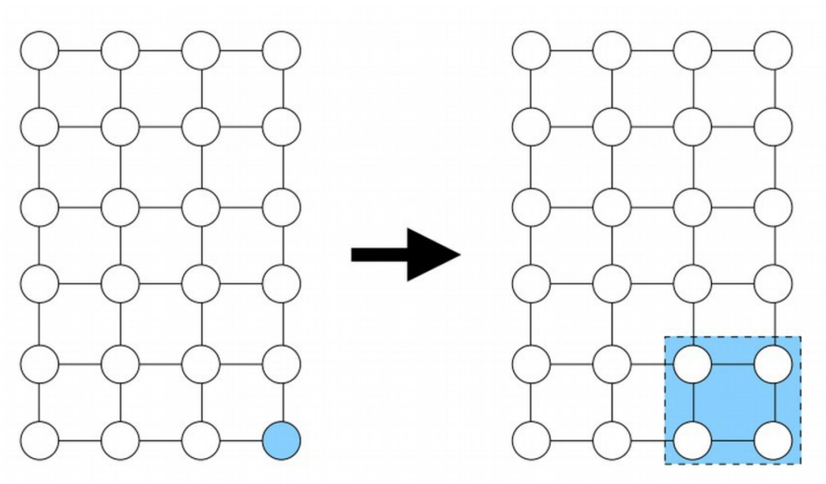
Unstructured

Agglomeration

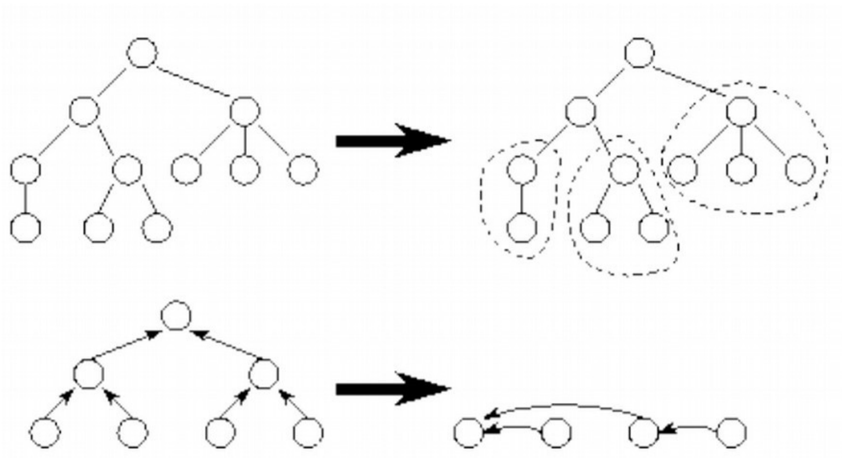
- Goal: Reduce messages and simplify programming
- Combine tasks into groups, increasing locality
  - Groups should have similar computation and communication costs
  - Task counts should still scale with processor count and /or problem size
  - Minimize software engineering costs
    - Agglomeration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

Agglomeration

- Examples:



Agglomeration of four local tasks



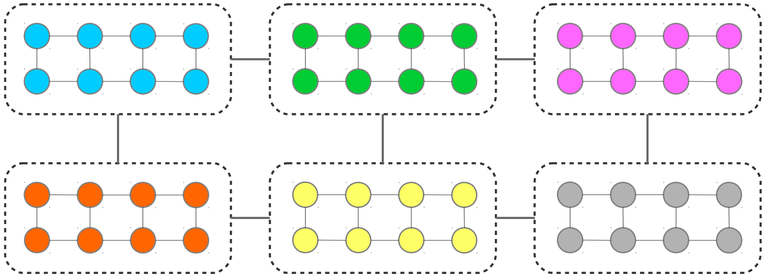
Agglomeration of tree-based tasks

Mapping

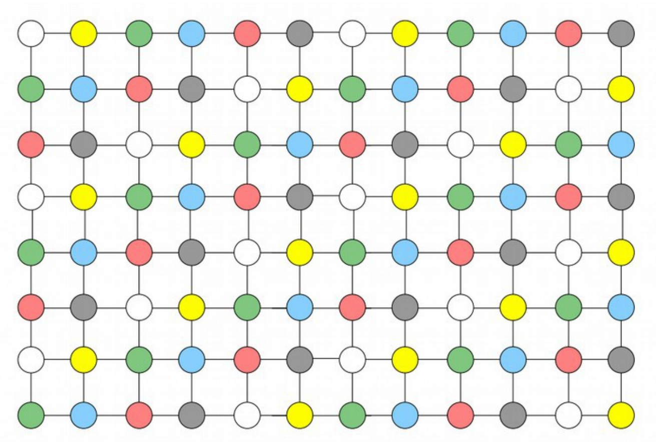
- Goal: minimize execution time
  - Alternately: maximize processor utilization
  - On a distributed system: minimize communication
- Assign tasks (or task groups) to processors/nodes
  - Block vs. cyclic
  - Static vs. dynamic
- Strategies:
  - 1) Place concurrent tasks on different nodes
  - 2) Place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

Mapping

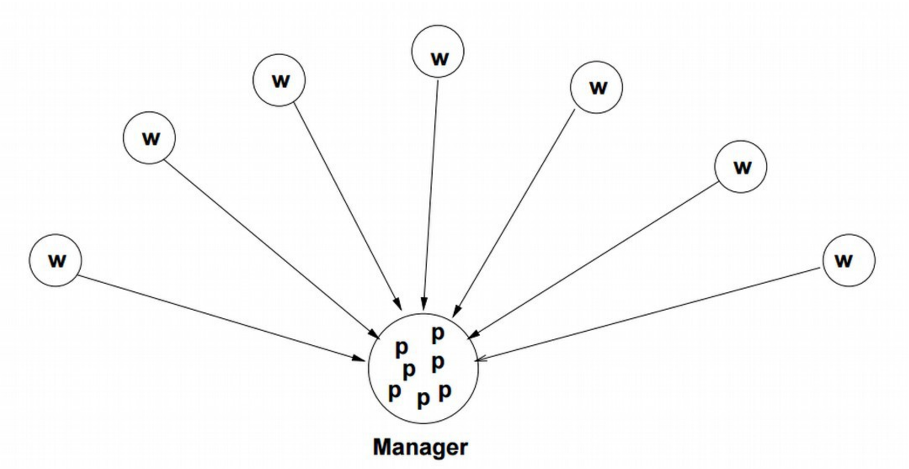
- Examples:



Block mapping



Cyclic mapping



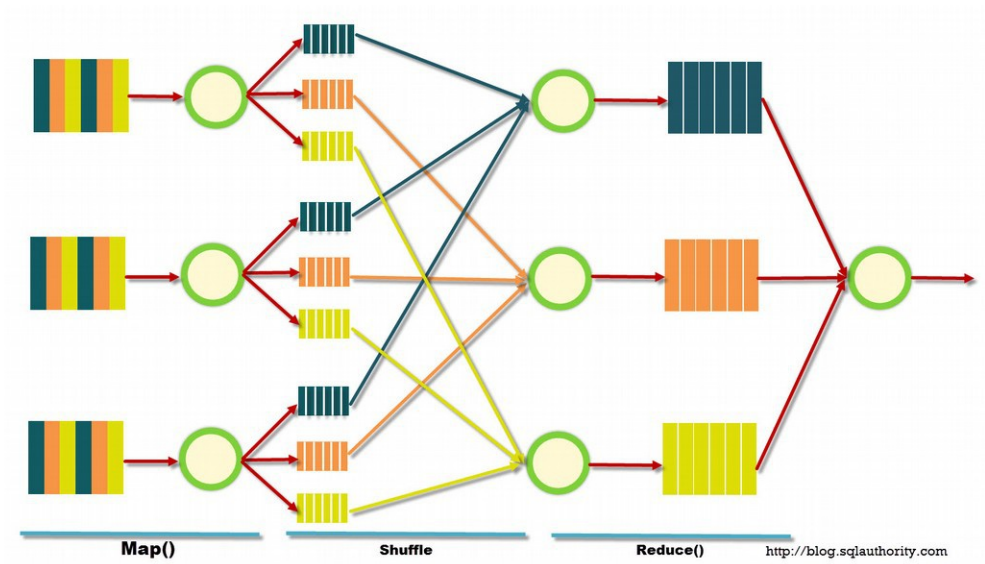
Dynamic mapping

Common paradigms

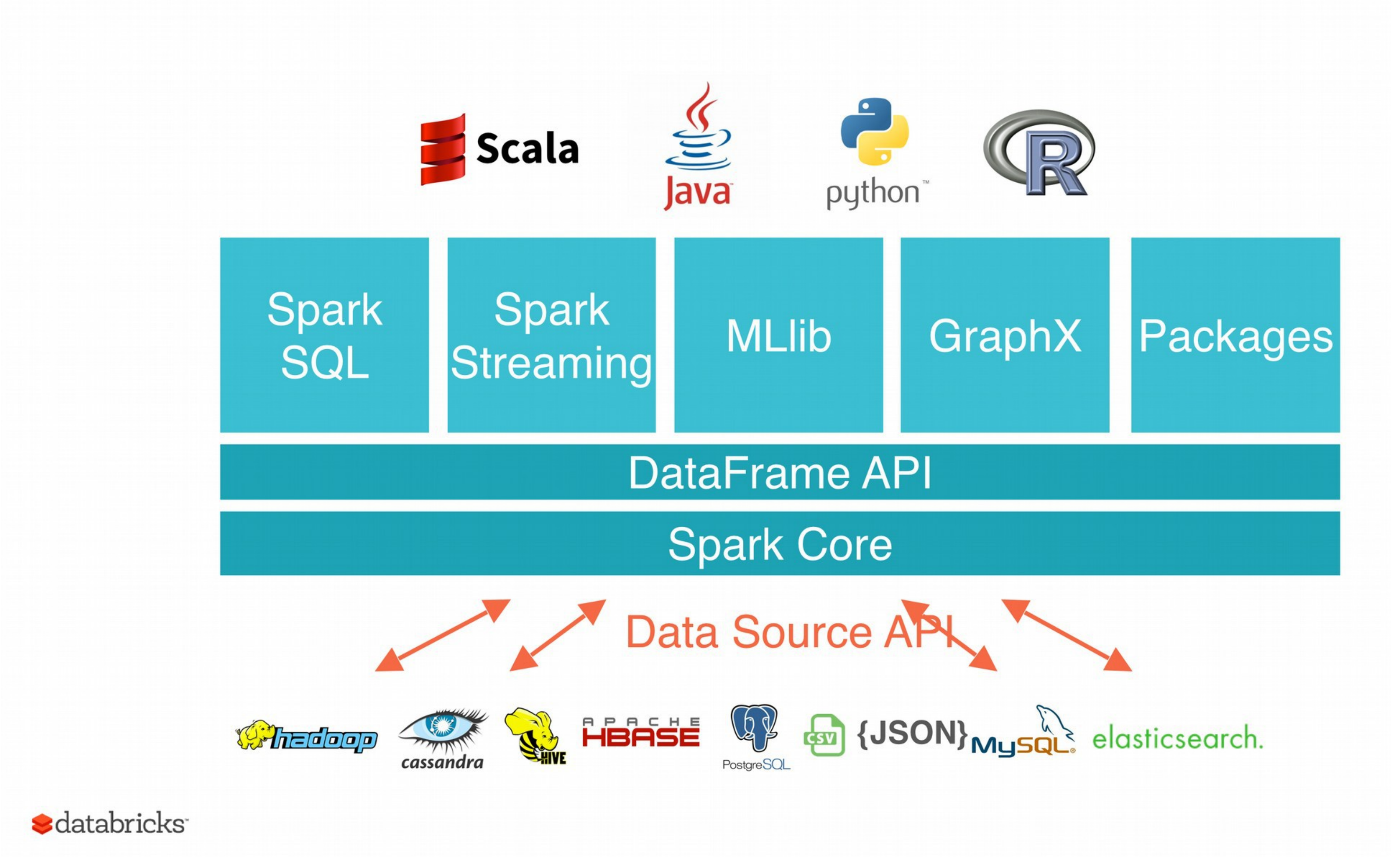
- Grid/mesh-based nearest-neighbor simulation
  - Often includes math-heavy computations
    - Linear algebra and systems of equations
    - Dense vs. sparse matrices
  - Newer: adaptive mesh and multigrid simulations
- Worker pools / task queues
  - Newer: adaptive cloud computing
- Pipelined task phases
  - 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
  - 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)
- Phases
  - Map (process local data)
  - Shuffle (distributed sort)
  - Reduce (combine results)

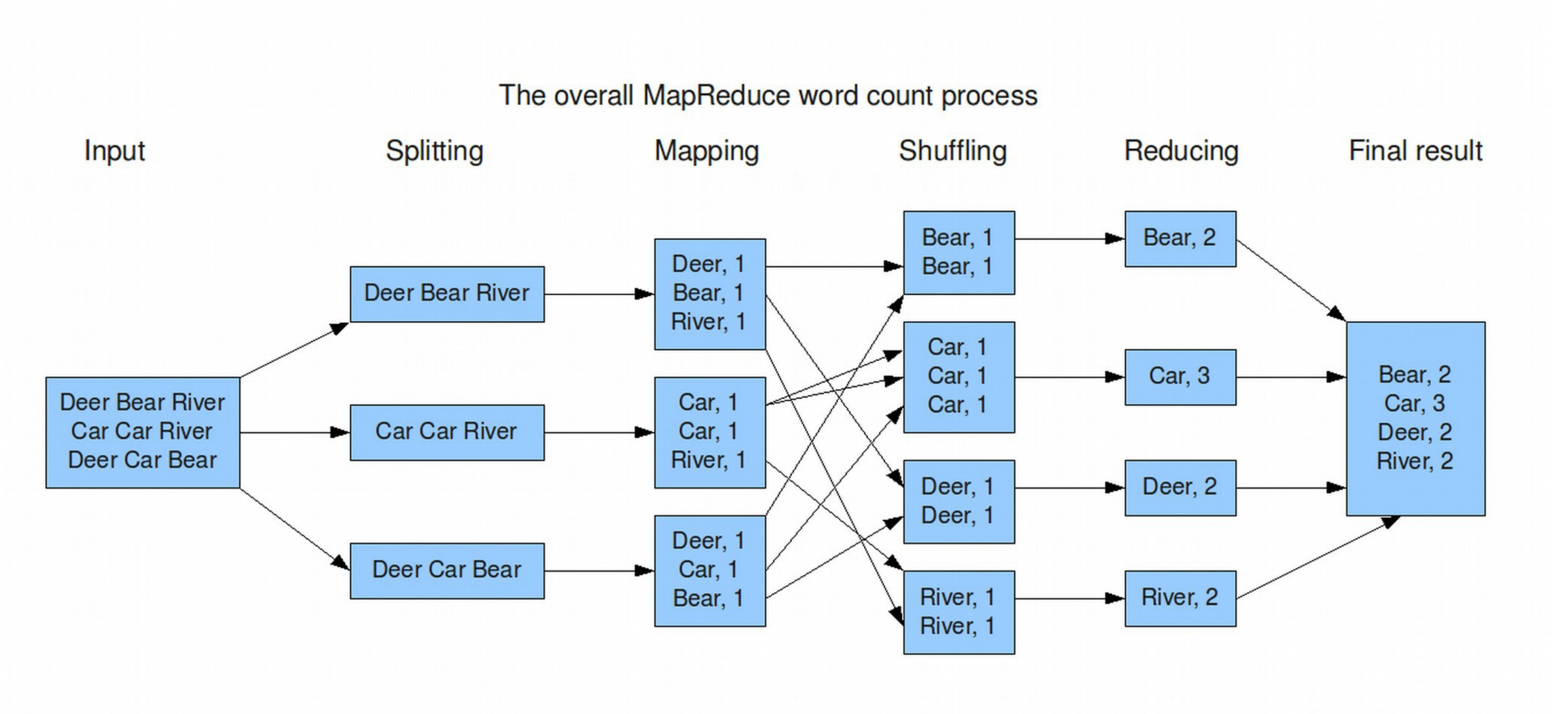


Apache Spark



MapReduce

- Word count example



Apache Hadoop (Java)

```
public class WordCount {

     public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {
       private final static IntWritable one = new IntWritable(1);
       private Text word = new Text();

       public void map(LongWritable key, Text value, OutputCollector<Text,
                       IntWritable> output, Reporter reporter) throws IOException {
         String line = value.toString();
         StringTokenizer tokenizer = new StringTokenizer(line);
         while (tokenizer.hasMoreTokens()) {
           word.set(tokenizer.nextToken());
           output.collect(word, one);
         }
       }
     }

     public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {
       public void reduce(Text key, Iterator<IntWritable> values,
                          OutputCollector<Text, IntWritable> output, Reporter orter) throws IOException {
         int sum = 0;
         while (values.hasNext()) {
           sum += values.next().get();
         }
         output.collect(key, new IntWritable(sum));
       }
     }
}
```

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)
```