Big Data Mining and Analytics Lecture

Parallel and Distributed Computing

Dr. Latifur Khan
Department of Computer Science
University of Texas at Dallas

Big Data processing boils down to…

- Divide-and-conquer
- Throwing more hardware at the problem

Simple to understand… a lifetime to master…

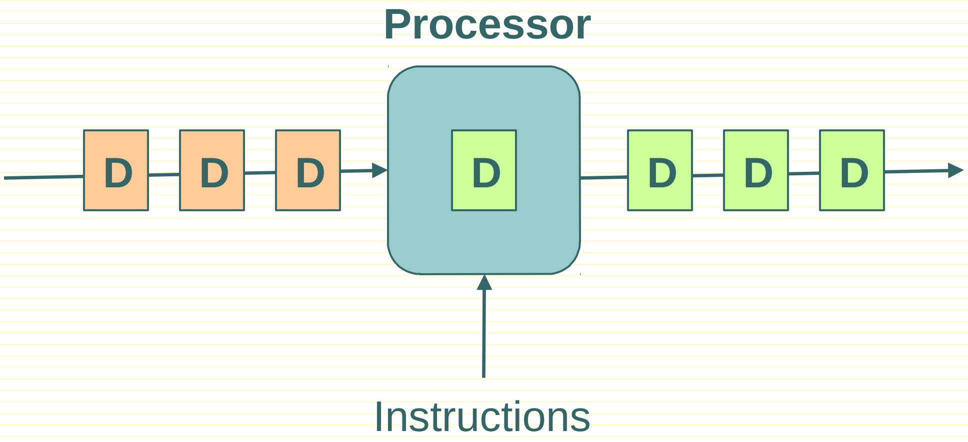
Parallel vs. Distributed

- Parallel computing generally means:
  - Vector processing of data
  - Multiple CPUs in a single computer
- Distributed computing generally means:
  - Multiple CPUs across many computers

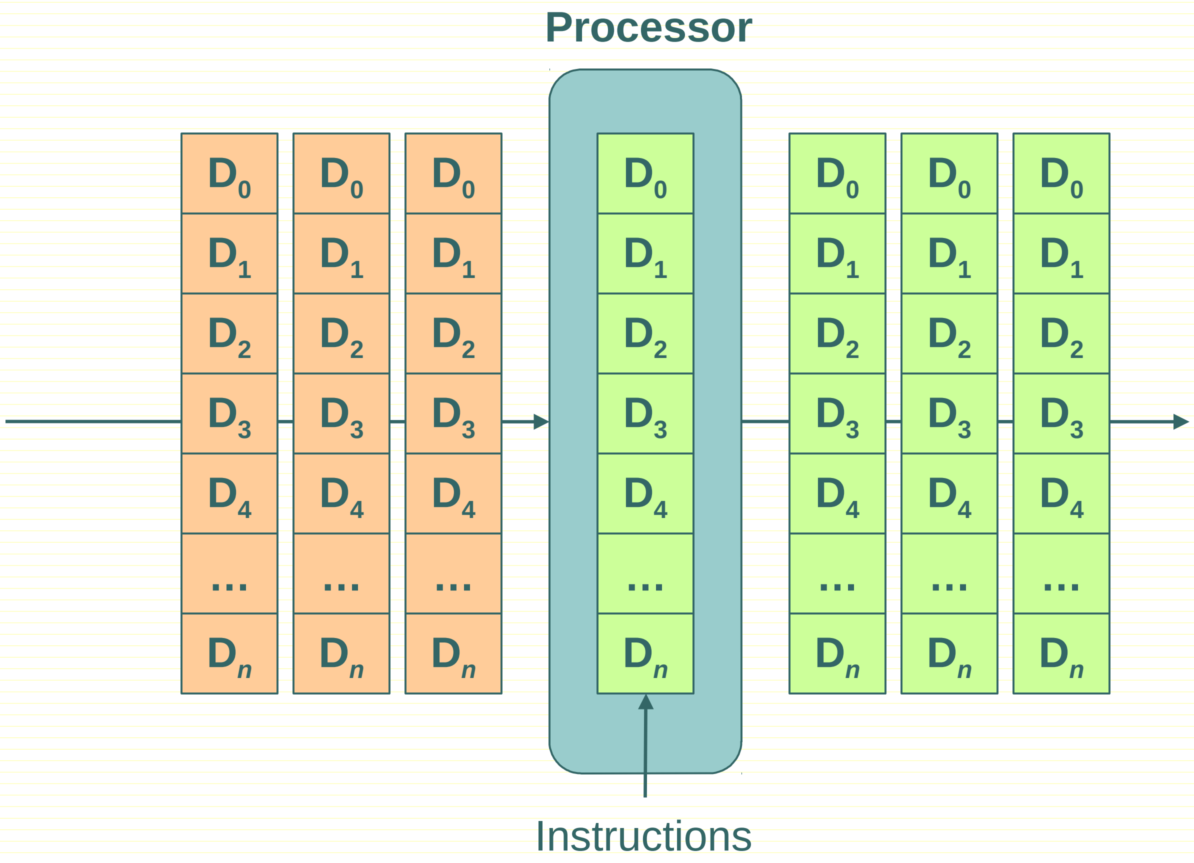
Flynn's Taxonomy

Data		Instructions: Single (SI)	Instructions: Multiple (MI)
Data	Single (SD)	**SISD** Single-threaded process	**MISD** Pipeline architecture
Data	Multiple (MD)	**SIMD** Vector Processing	**MIMD** Multi-threaded Programming

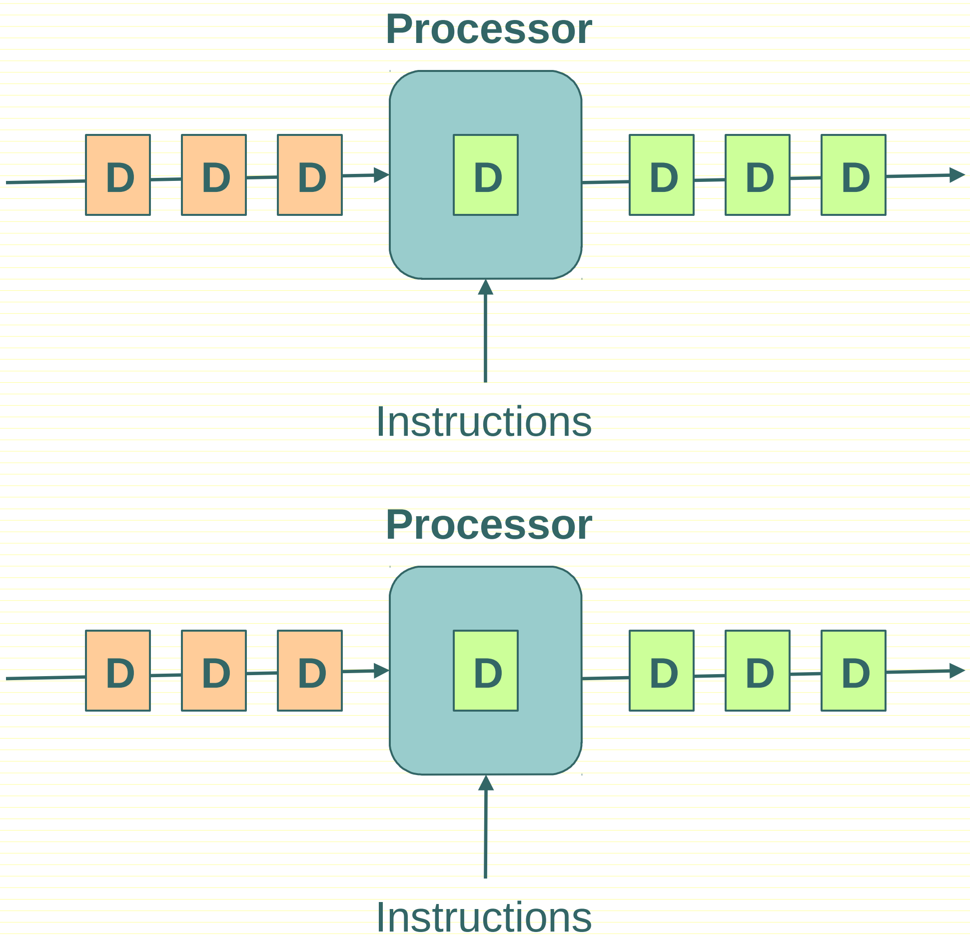
SISD



SIMD

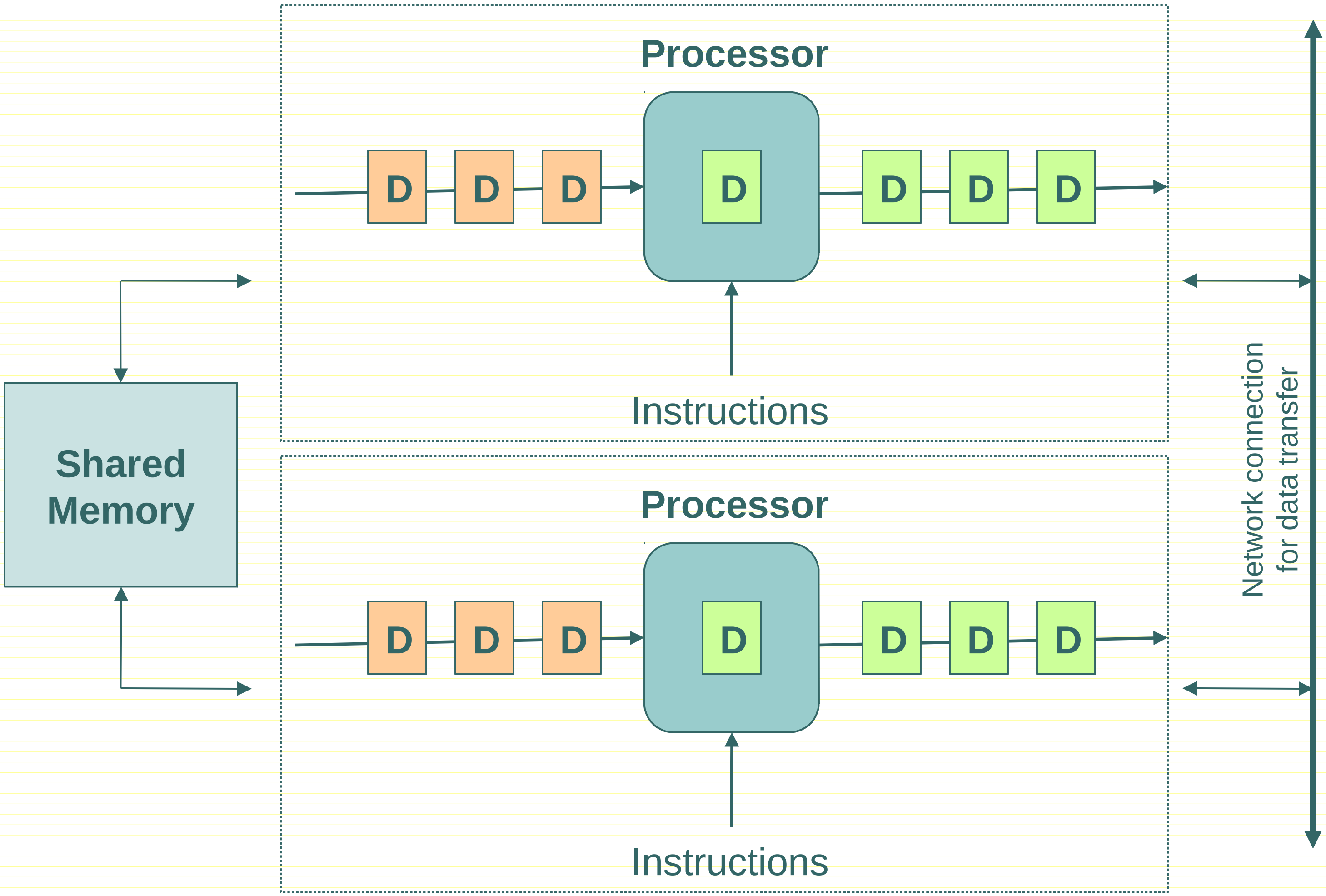


MIMD



Processor
D
D
D
D
D
D
D
Instructions
Processor
D
D
D
D
D
D
D
Instructions

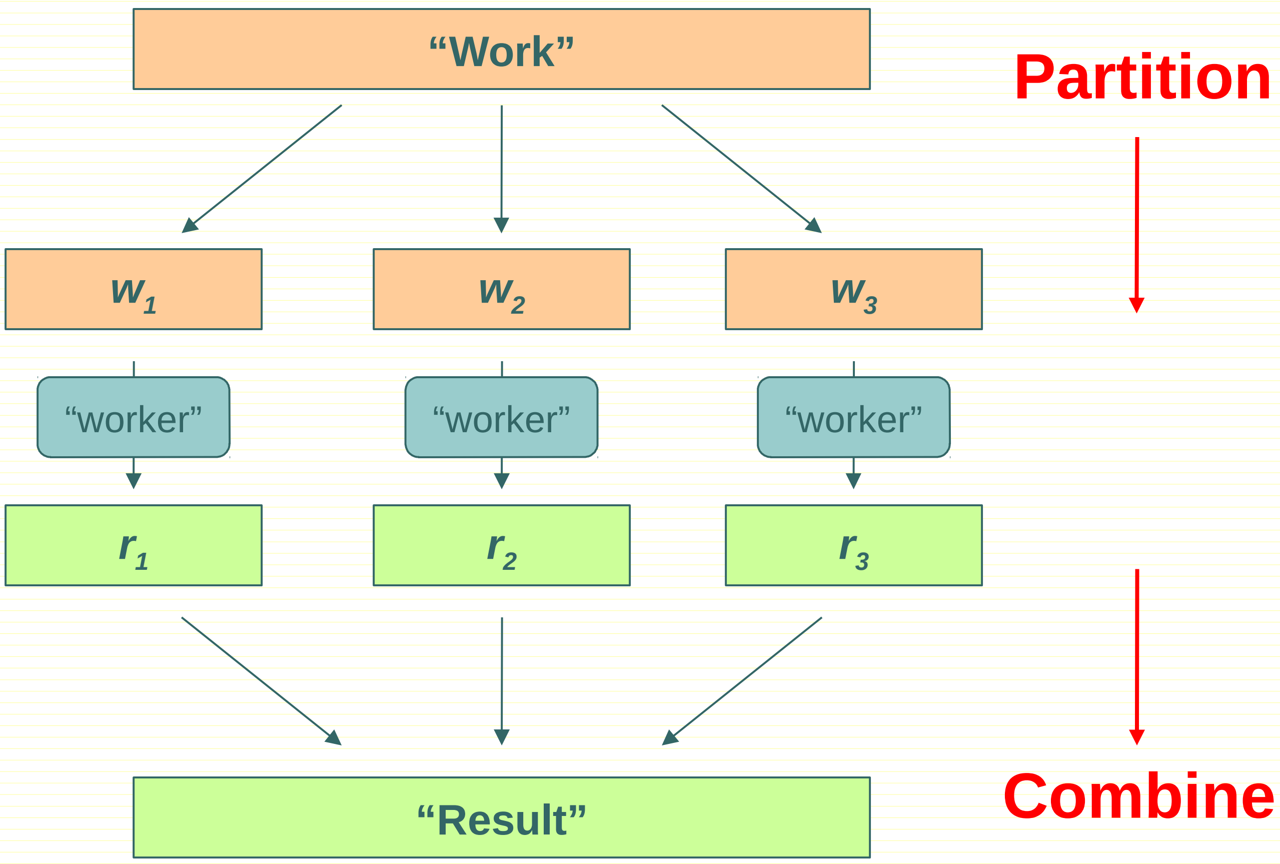
Parallel vs. Distributed



Parallel: Multiple CPUs within a shared memory machine

Distributed: Multiple machines with own memory connected over a network

Divide and Conquer



Parallelization Problems

- How do we assign work units to workers?
- What if we have more work units than workers?
- What if workers need to share partial results?
- How do we aggregate partial results?
- How do we know all the workers have finished?
- What if workers die?

What is the common theme of all of these problems?

General Theme?

- Parallelization problems arise from:
  - Communication between workers
  - Access to shared resources (e.g., data)
- Thus, we need a synchronization system!
- This is tricky:
  - Finding bugs is hard
  - Solving bugs is even harder

Cloud Computing Lecture #2

From Lisp to MapReduce and GFS

Today's Topics

- Functional Programming
- MapReduce
- Google File System (GFS)

Functional Programming

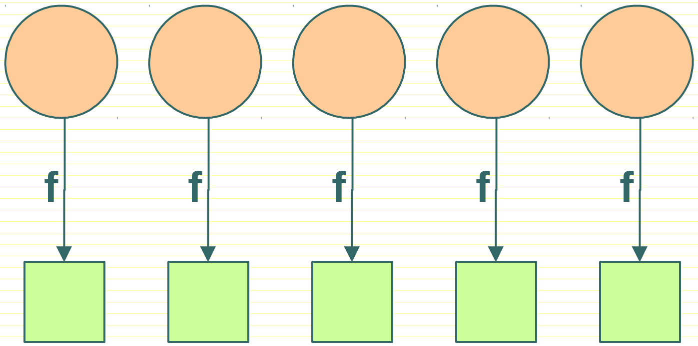
- MapReduce = functional programming meets distributed processing on steroids
  - Not a new idea… dates back to the 50's (or even 30's)
- What is functional programming?
  - Computation as application of functions
  - Theoretical foundation provided by lambda calculus
- How is it different?
  - Traditional notions of "data" and "instructions" are not applicable
  - Data flows are implicit in program
  - Different orders of execution are possible
- Exemplified by LISP and ML

Lisp → MapReduce?

- What does this have to do with MapReduce?
- After all, Lisp is about processing *lists*
- Two important concepts in functional programming
  - Map: do something to everything in a list
  - Fold: combine results of a list in some way

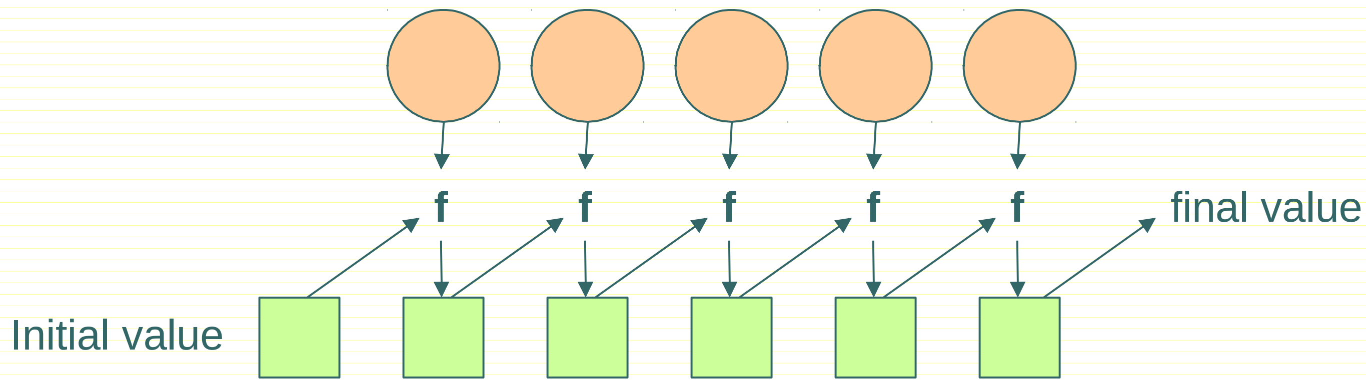
Map

- Map is a higher-order function
- How map works:
  - Function is applied to every element in a list
  - Result is a new list



Fold

- Fold is also a higher-order function
- How fold works:
  - Accumulator set to initial value
  - Function applied to list element and the accumulator
  - Result stored in the accumulator
  - Repeated for every item in the list
  - Result is the final value in the accumulator



Lisp
MapReduce
GFS

Map/Fold in Action

- Simple map example:

```
(map (lambda (x) (* x x))
     '(1 2 3 4 5))
 → '(1 4 9 16 25))
```

- Fold examples:

```
(fold + 0 '(1 2 3 4 5)) → 15
(fold * 1 '(1 2 3 4 5)) → 120
```

- Sum of squares:

```
(define (sum-of-squares v)
  (fold + 0 (map (lambda (x) (* x x)) v)))
(sum-of-squares '(1 2 3 4 5)) → 55
```

Lisp → MapReduce

- Let's assume a long list of records: imagine if...
  - We can distribute the execution of map operations to multiple nodes
  - We have a mechanism for bringing map results back together in the fold operation
- That's MapReduce! (and Hadoop)
- Implicit parallelism:
  - We can parallelize execution of map operations since they are isolated
  - We can reorder folding if the fold function is commutative and associative

Typical Problem

- Iterate over a large number of records
- Map: extract something of interest from each
- Shuffle and sort intermediate results
- Reduce: aggregate intermediate results
- Generate final output

Key idea: provide an abstraction at the point of these two operations

MapReduce

- Programmers specify two functions:

  map (k, v) → <k', v'>*

  reduce (k', v') → <k', v'>*

  - All v' with the same k' are reduced together

- Usually, programmers also specify:

  partition (k', number of partitions) → partition for k'

  - Often a simple hash of the key, e.g. hash(k') mod n
  - Allows reduce operations for different keys in parallel