Parallelism

CS242

Lecture 16

Concurrency vs. Parallelism

- Concurrency
  - Multiple threads of control
  - May interleave at any granularity
    - Above atomic operations
  - Makes sense even on a single processor
  - A structuring device to deal with unpredictable latencies

- Parallelism
  - Multiple threads of control that execute at the same time
  - On multiple hardware devices
  - Inherently about performance

Amdahl's Law

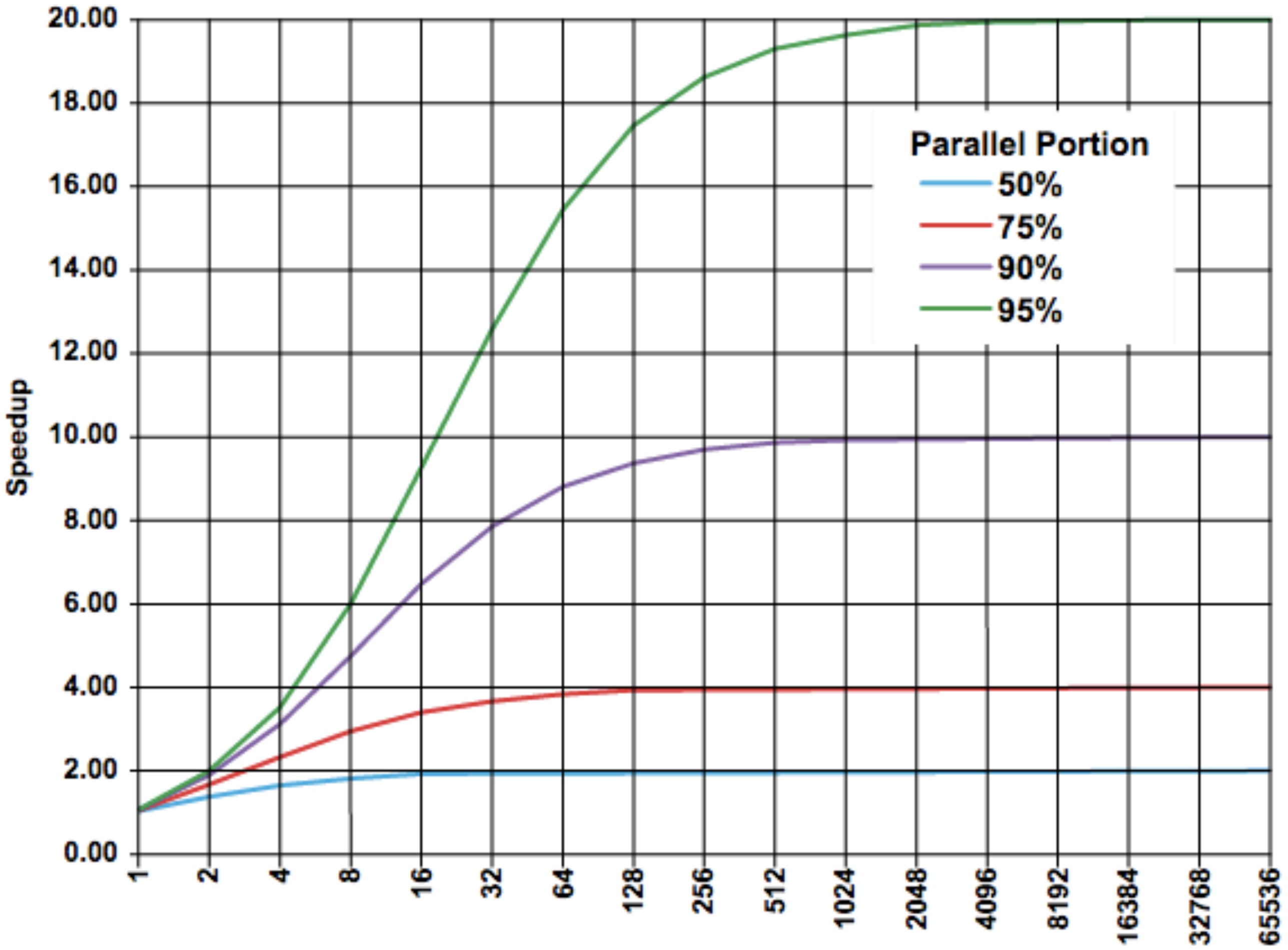
$$\text{Speedup} = \frac{1}{(1 - p) \; + \; (p / s)}$$

where

p = portion of the program sped up

s = factor improvement of that portion

Parallelism: Speed vs. Processors



Discussion

- Amdahl's law is simple and general
  - Not about a specific machine or program

- And unforgiving
  - To speed up by 1000x, must parallelize 99.9%
  - To reach 10,000x, must parallelize 99.99%
  - And these are not very aggressive targets!

Three Topics

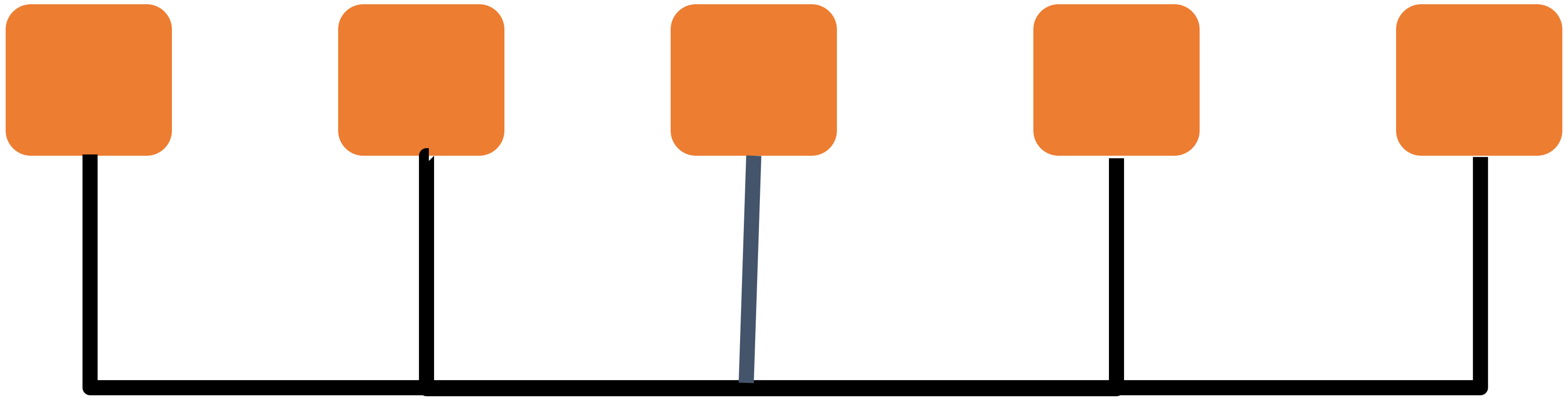
- Bulk synchronous model
- SPMD
- MapReduce

Bulk Synchronous Model

- A model
  - An idealized machine

- Originally proposed for analyzing parallel algorithms
  - Leslie Valiant
  - “A Bridging Model for Parallel Computation”, 1990

The Machine



What are some properties of this machine model?

Computations

- A sequence of *supersteps:*

- Repeat:
- *All processors do local computation*
- *Barrier – all threads must reach the barrier before any proceed*
- *All processors communicate*
- *Barrier*

What are properties of this computational model?

Basic Properties

- Uniform
  - compute nodes
  - communication costs
- Separate communication and computation
- Synchronization is global

The Idea

- Programs are
  - written for v virtual processors
  - run on p physical processors

- If $v \geq p \log p$ then
  - Managing memory, communication and synchronization can be done automatically within a constant factor of optimal

How Does It Work?

- Roughly
  - Memory addresses are hashed to a random location in the machine
  - Guarantees that on average, memory accesses have the same cost
  - The extra log p factor of threads are multiplexed onto the p processors to hide the latency of memory requests
  - The processors are kept busy and do no more compute than necessary

SPMD

Terminology

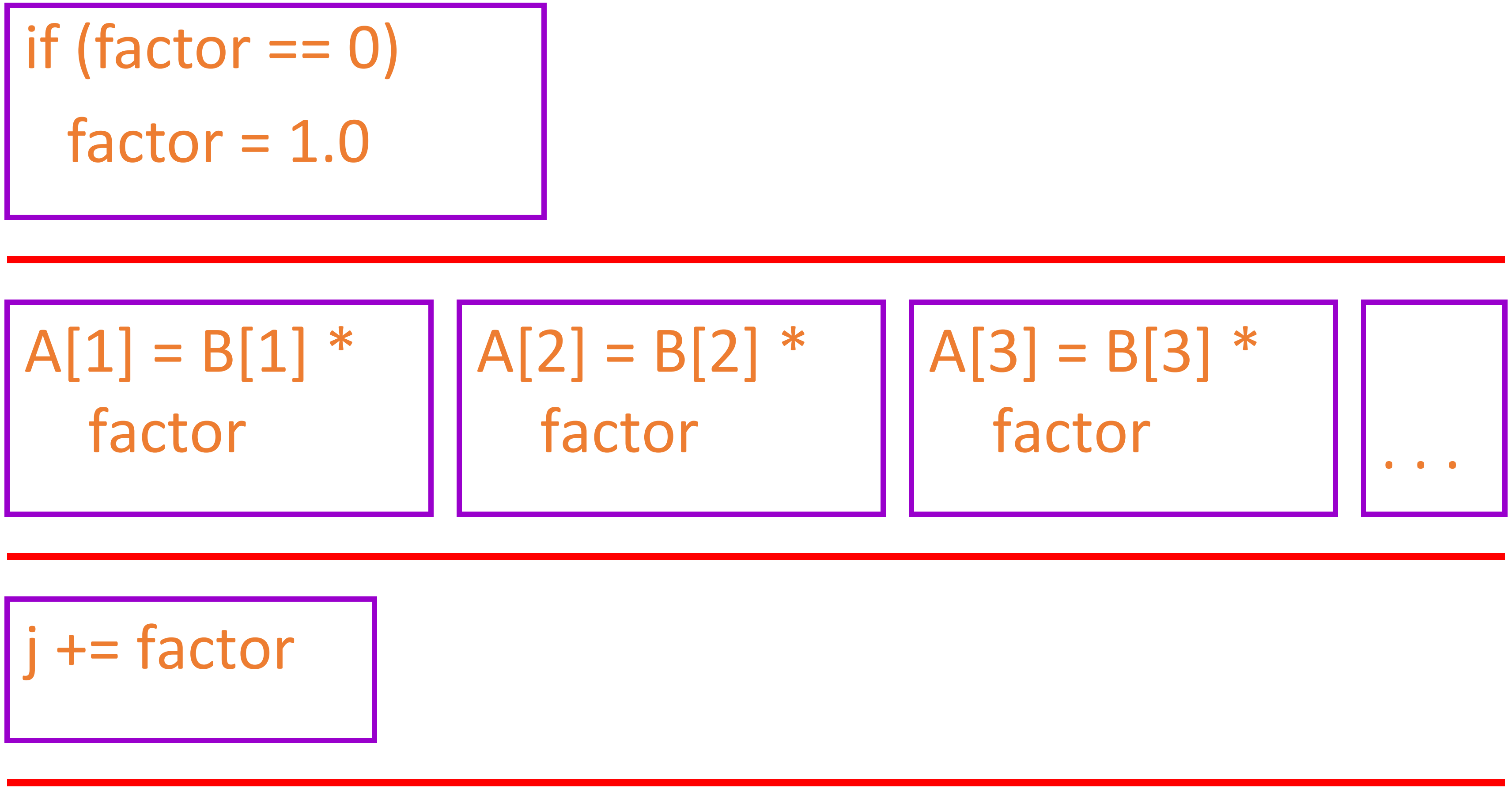
- SIMD
  - Single Instruction, Multiple Data

- SPMD
  - Single Program, Multiple Data

SIMD = Vector Processing

```
if (factor == 0)
  factor = 1.0
A[1..N] = B[1..N] * factor;
j += factor;
```

Picture


if (factor == 0)
factor = 1.0
A[1] = B[1] * factor
A[2] = B[2] * factor
A[3] = B[3] * factor
. . .
j += factor

Comments

- Single thread of control
  - Global synchronization at each program instruction
- Can exploit fine-grain parallelism
  - Assumption of hardware support

SPMD = Single Program, Multiple Data

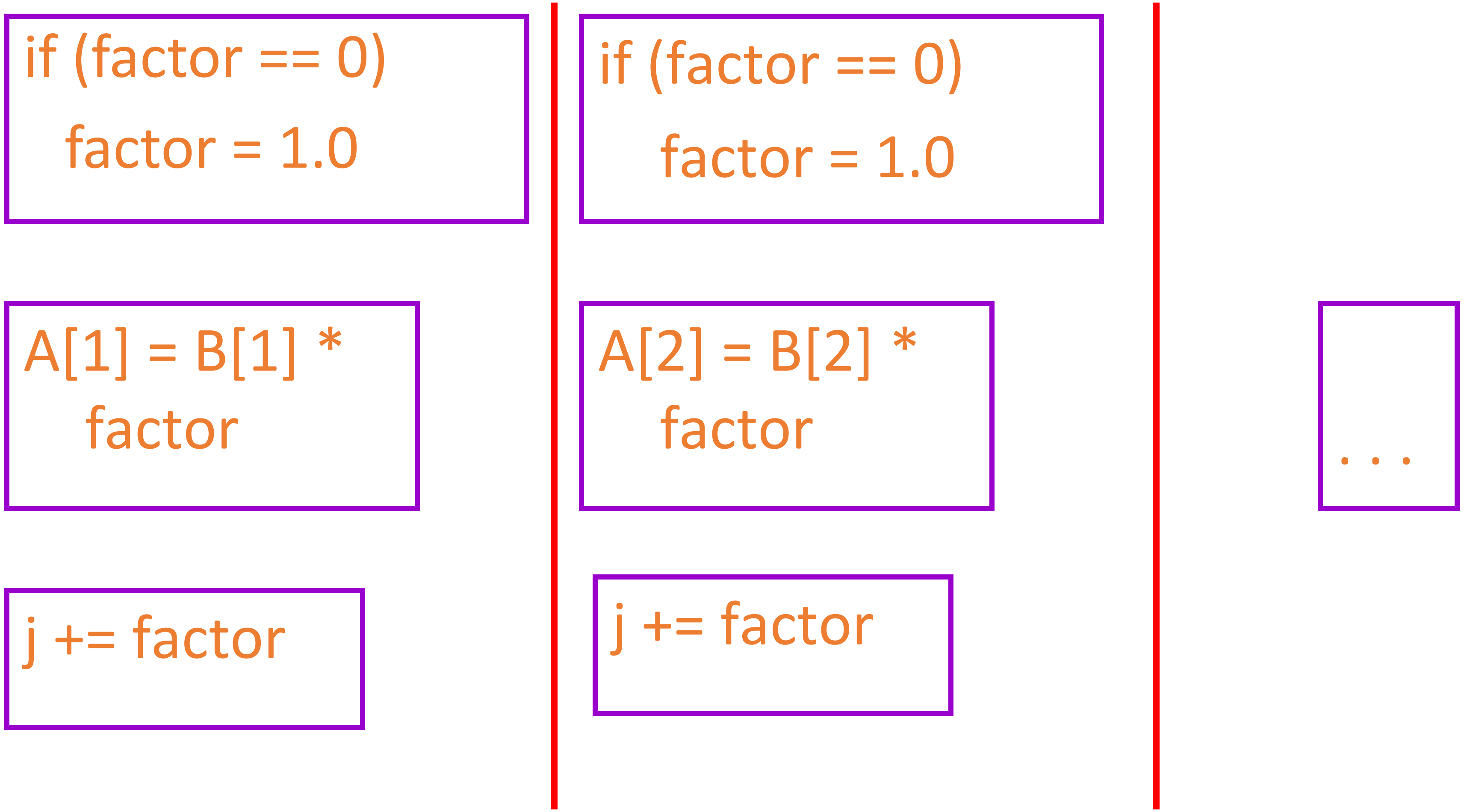
SIMD

```
if (factor == 0)
  factor = 1.0
A[1..N] = B[1..N] * factor;
j += factor;
…
```

SPMD

```
if (factor == 0)
  factor = 1.0
A[myid] = B[myid] * factor;
j += factor;
…
```

Picture



if (factor == 0)
factor = 1.0
A[1] = B[1] * factor
j += factor
if (factor == 0)
factor = 1.0
A[2] = B[2] * factor
j += factor
. . .

Comments

- Multiple threads of control
  - One (or more) per processor
- Asynchronous
  - All synchronization is programmer-specified
- Threads are distinguished by myid
- Choice: Are variables local or global?

Comparison

- SIMD
  - Designed for tightly-coupled, synchronous hardware
  - i.e., vector units

- SPMD
  - Designed for clusters
  - Too expensive to synchronize every statement
  - Need a model that allows asynchrony

MPI

- Message Passing Interface
  - A widely used standard
  - Runs on everything

- Some similarities to the Pi Calculus
  - For one-to-one communication
  - Also has one-to-all and all-to-all communication primitives

- Most popular way to write SPMD programs

MPI Programs

- Standard sequential programs
  - All variables are local to a thread
- Augmented with calls to the MPI interface
  - SPMD model
  - Every thread has a unique identifier
  - Threads can send/receive messages
  - Synchronization primitives

MPI Point-to-Point Routines

- MPI_Send(buffer, count, type, dest, …)
- MPI_Recv(buffer, count, type, source, …)

Example

```
 for (....)  {
// p = number of chunks of 1D grid, id = process id, h[] = local chunk of the grid
// boundary elements of h[] are copies of neighbors boundary elements

  .... Local computation ...

// exchange with neighbors on a 1-D grid
 if ( 0 < id )
      MPI_Send ( &h[1], 1, MPI_DOUBLE, id-1, 1, MPI_COMM_WORLD );
if ( id < p-1 )
      MPI_Recv ( &h[n+1], 1,  MPI_DOUBLE, id+1, 1, MPI_COMM_WORLD, &status );
if ( id < p-1 )
      MPI_Send ( &h[n], 1, MPI_DOUBLE, id+1, 2, MPI_COMM_WORLD );
if ( 0 < id )
      MPI_Recv ( &h[0], 1, MPI_DOUBLE, id-1, 2, MPI_COMM_WORLD, &status );

… More local computation ...

}
```

MPI Collective Communication Routines

- MPI_Barrier(…)
- MPI_Bcast(…)
- MPI_Scatter(…)
- MPI_Gather(…)
- MPI_Reduce(…)

Typical Structure

```
communicate_get_work_to_do();
barrier;                // not always needed
do_local_work();
barrier;
communicate_write_results();
```

What does this remind you of?

Bulk Synchronous/SPMD Model

- Easy to understand
- Phase structure guarantees no data races
  - Barrier synchronization also easy to understand
- Fits many problems well

PGAS Model

- PGAS = *Partitioned Global Address Space*

- There is one global address space

- But each thread owns a partition of the address space that is more efficient to access
  - i.e., the local memory of a processor

- Equivalent in functionality to MPI
  - But typically presented as a programming language
  - Examples: Split-C, UPC, Titanium

PGAS Languages

- No library calls for communication

- Instead, variables can name memory locations on other machines

```
// Assume y points to a remote location
// The following is equivalent to a send/receive
x = *y
```

PGAS Languages

- Also provide collective communication
- Barrier
- Broadcast/Reduce
  - 1-many
- Exchange
  - All-to-all

PGAS vs. MPI

- Programming model very similar
    - Both provide SPMD

- From a pragmatic point of view, MPI rules
    - Easy to add MPI to an existing sequential language

- For productivity, PGAS is better
    - Programs filled with low-level details of MPI calls
    - PGAS programs easier to modify
    - PGAS compilers can know more/do a better job

Summary

- SPMD is well-matched to cluster programming
  - Also works well on shared memory machines

- One thread per core
  - No need for compiler to discover parallelism
  - No danger of overwhelming # of threads

- Model exposes memory architecture
  - Local vs. Global variables
  - Local computation vs. sends/receives

MapReduce

The Motivation

- Organizations today produce enormous amounts of data
  - Customer orders
  - Patient records
  - Social media interactions
  - Log files from all the computers
  - ...

- The only way to process this data is in parallel
  - Would take far too long on a single machine
  - Memory is the main the issue, not processing power

The Programming Model

- Programs have a type $List(String) \rightarrow a$

- A string here is (substantial) block of data
  - Simply interpreted as a sequence of bytes; i.e., a string
  - Example: a file consisting of tens of megabytes to terabytes

- A program has the form

$$(reduce\ g) \diamond (map\ f)$$

where $f: String \rightarrow a$ and $g: (a,a) \rightarrow a$

Semantics

(reduce g) ◊ (map f)

What does ◊ do?

- Group the pairs returned by the map on the first components
    - the *keys*
- Apply the reduction function to the set of second components for each key
    - The *values*
- Return the set of key-value pairs after the reduction

Example: Word Count

Count the number of occurrences of each word in a corpus.

$$(\text{reduce } +) \diamond \text{ map } ((\text{map } \lambda w.(w,1)) \text{ o split } `\ `)$$

where split: Char → String → List(String)

Discussion

- A combinator-based language!
  - Recall SKI
- The ◊ combinator is perhaps unexpected
  - More than function composition
- Makes it easy to write parallel, distributed computations
  - Without knowing anything about parallelism, different kinds of parallel hardware, synchronization, etc.
  - Just need to understand basic functional programming

Discussion

- Why is MapReduce functional?
- Using state makes the most sense when there is a hardware-provided global state
  - i.e., a single address space
- But in distributed machines, there is no single address space
  - Or rather, it is expensive to provide one in software
- Functional programming makes sense regardless of the scale of the underlying machine

Evolution

- MapReduce became very popular
  - Gave programmers an easy way to write simple computations that they otherwise couldn't write themselves
- Downside: Many algorithms can't be expressed in MapReduce
  - At least efficiently
- Spark is a well-known successor to MapReduce

Summary

- Two very different parallel programming models

- Bulk synchronous/SPMD
  - Message-passing based
  - Run many copies of the same program in parallel
  - Threads are distinguished by a *myid* unique to each thread
  - Threads communicate via message passing
  - Used mostly for scientific programming

- MapReduce
  - Based on functional combinators with obvious parallel implementations
  - More expressive than simple composition because of the flexibility of the group-by operation that MapReduce adds between the map and reduce stages
  - Used for data analytics