EECE5698
Parallel Processing for Data Analytics

Lecture 3: Map and Reduce Operations in Spark

Outline

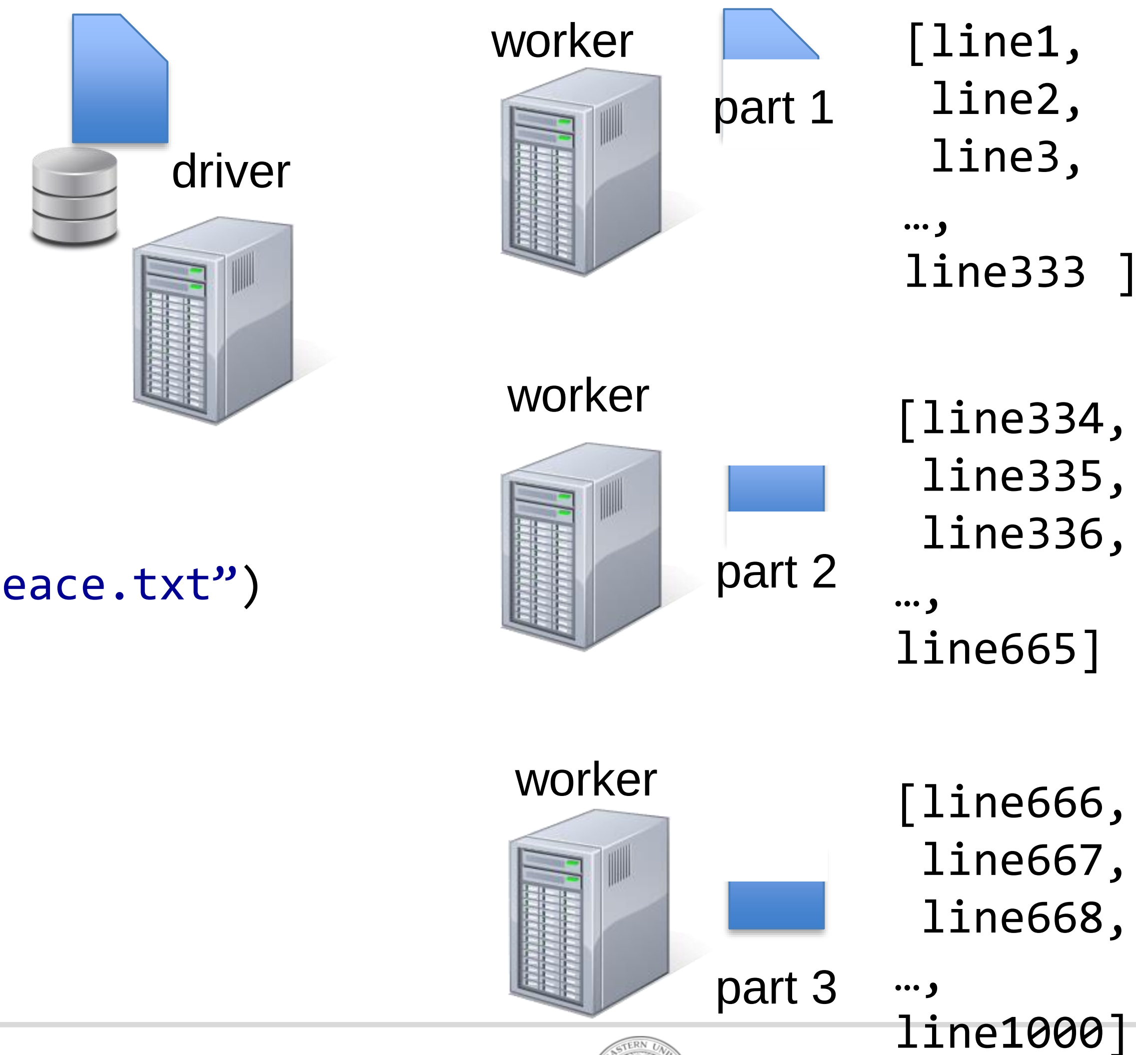
- Map and other Transforms
- Reduce and other Actions

Outline

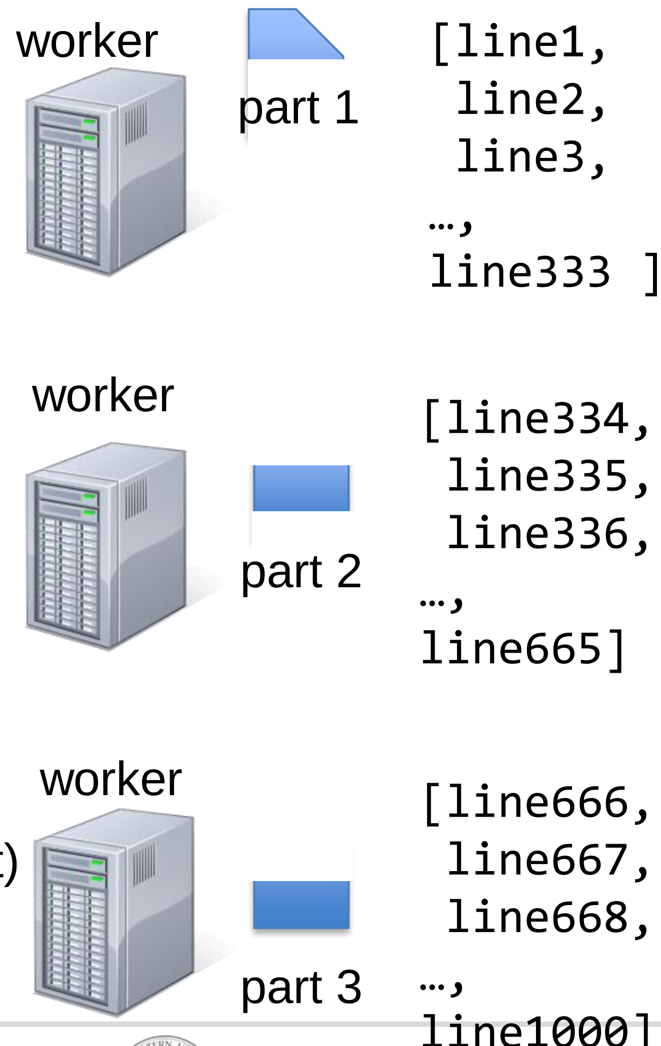
- ❑Map and other Transforms
- ❑Reduce, and other Actions

Resilient Distributed Datasets (RDDs)

```
myrdd=sc.textFile("WarAndPeace.txt")
```



Resilient Distributed Datasets (RDDs)



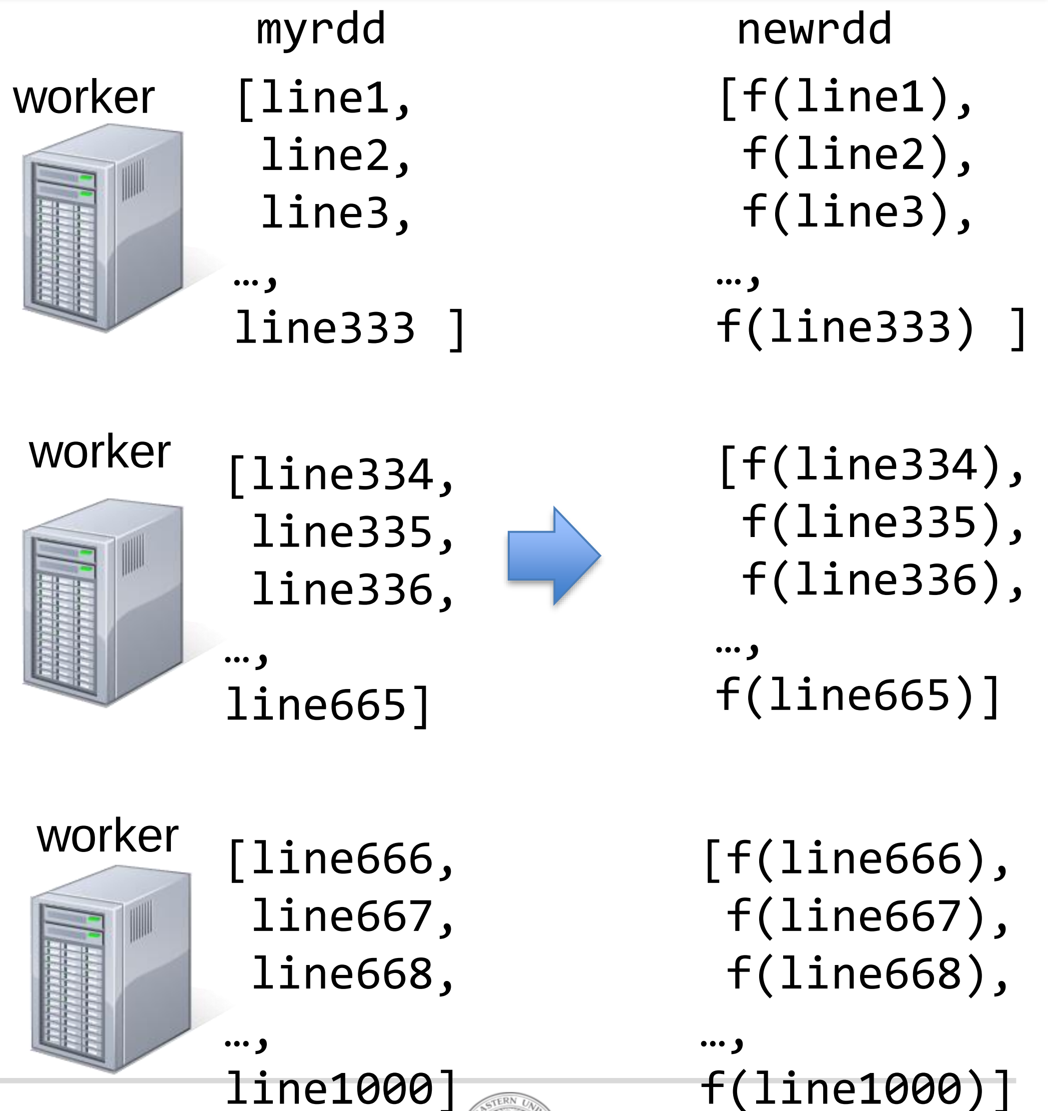

```
myrdd=sc.textFile("WarAndPeace.txt")
```

- Think: `myrdd` is a distributed list, **but..**
  - You **cannot change** it,
  - Cannot **see individual elements** (e.g., 5th element)
  - Can only interact with it through **specific ops**

The Map Transformation

```
newrdd=myrdd.map(f)
```

- Return a new RDD by applying a function to each element of this RDD.



Examples

User-defined function, defined through `def` statement

```
def myf(x):
    return x+2

newrdd=myrdd.map(myf)
```

myrdd

```
[1,
 2,
 3,
…,
10]
```

newrdd

```
[3,
 4,
 5,
…,
12]
```

Examples

User-defined function, defined through `lambda` operator

```
newrdd=myrdd.map(lambda x:x+2)
```

```
myrdd          newrdd

[1,            [3,
 2,             4,
 3,             5,
…,             …,
10]            12]
```

Examples

User-defined function, defined through `lambda` operator

```
newrdd=myrdd.map(lambda x:x+'_')
```

```
myrdd          newrdd

['a',          ['a_',
 'b',           'b_',
 'c',    ⇒      'c_',
…,             …,
'z']           'z_']
```


Northeastern

Examples

Python built-in function

```
newrdd=myrdd.map(eval)
```

```
myrdd

['1.4',
 '2.3',
 '(1,3)',
  '{"apples":2,"oranges":3}'
'array([1,3,5])'
]
```

```
newrdd

[1.4,
 2.3,
(1,3),
{"apples":2,"oranges":3},
array([1.0,3.0,5.0])
]
```

Other Important Transforms

`flatMap(f)`: Map each element to multiple elements

```
newrdd=myrdd.flatMap(lambda x:x.split())
```

myrdd		newrdd
['Words are like leaves', 'and where most abound', 'much fruit of sense beneath', 'is most rarely found']	⇨	['Words', 'are', 'like', 'leaves', 'and', 'where', …, 'rarely', 'found']

NOTE: `f` must return a **collection** (e.g., list)

Other Important Transforms

`distinct()`: Remove duplicates

```
newrdd= myrdd.distinct()
```

```
myrdd

 [1,
  1,
  2,
  2,
  3
 ]
```

```
newrdd

 [1,
  2,
  3
 ]
```

Other Important Transforms

`sample(withReplacement,fraction)`: Return a random sampled subset of this RDD

```
newrdd= myrdd.sample(False,0.5)
```

```
myrdd                      newrdd

 [1,                        [1,
  2,                         3,
  3,           →             5
  4,                        ]
  5
 ]
```

(result is non-deterministic)

Other Important Transforms

`sortBy(keyfunc, ascending=True)`: Sort RDD contents

```
newrdd= myrdd.sortBy(lambda x:-x)
```

```
myrdd                newrdd

 [1,                  [5,
  4,                   4,
  5,                   3,
  2,                   2,
  3                    1
 ]                    ]
```

Warning: future transforms may change ordering

Northeastern

How does map work?

```
def myf(x):
    return x+2
```

1

010
101

Serialize to binary representation

driver

worker

```
[line1,
 line2,
 line3,
…,
line333 ]
```

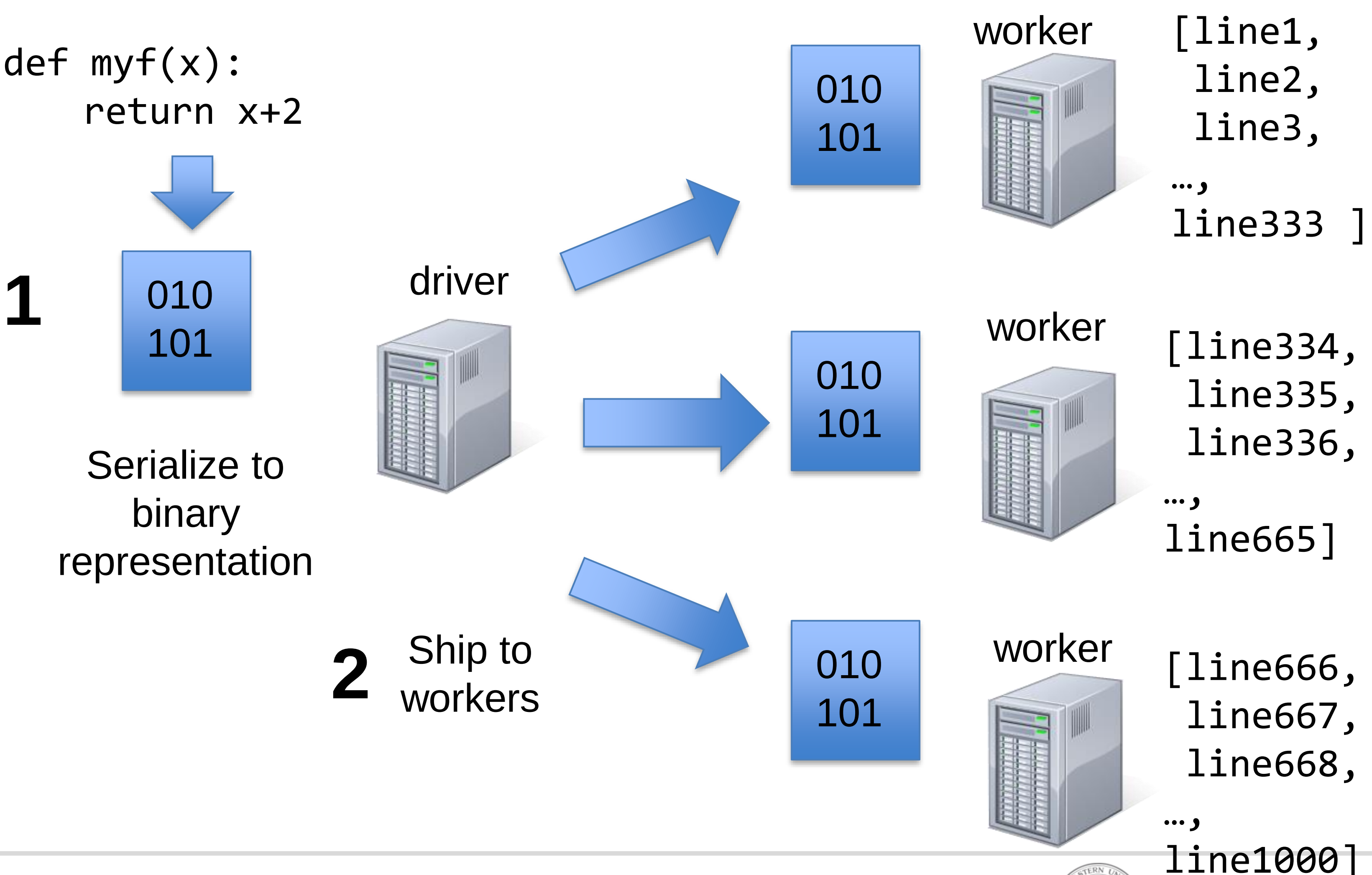
worker

```
[line334,
 line335,
 line336,
…,
line665]
```

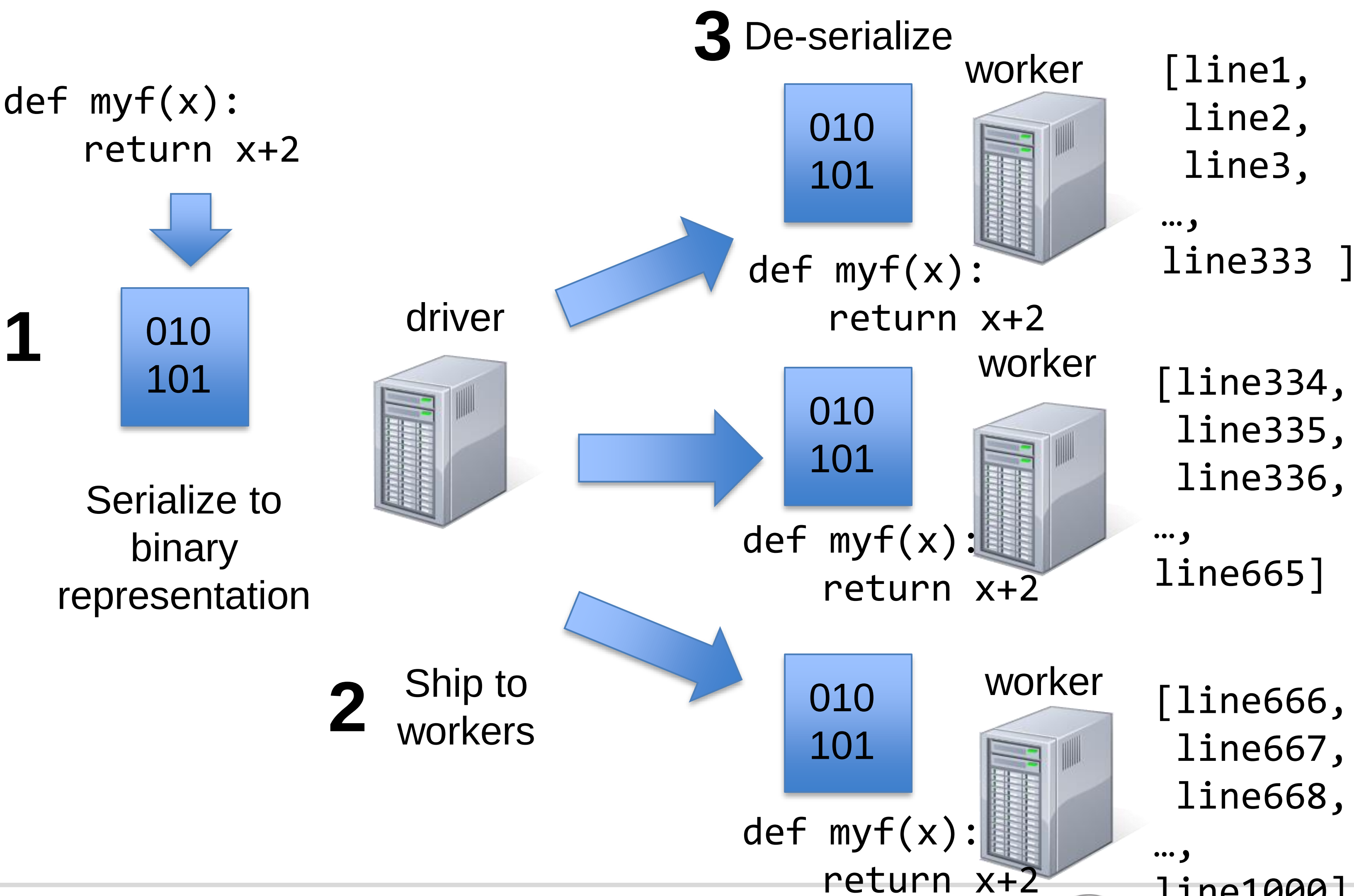
worker

```
[line666,
 line667,
 line668,
…,
line1000]
```

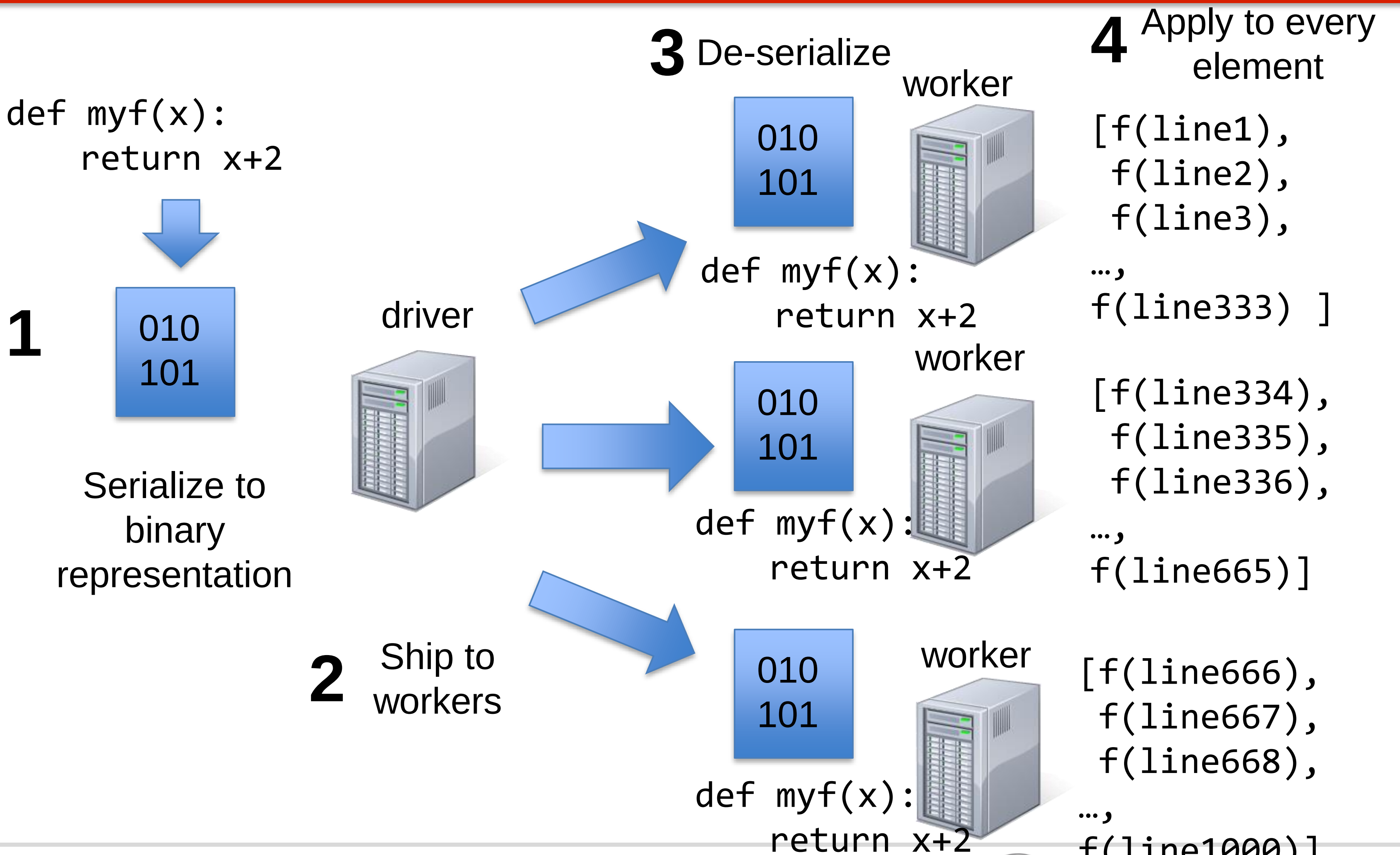
How does map work?



How does map work?



How does map work?



`maps` Involve Communication

- **Function** f is **shipped** to data
- Data **do not move** (stay at workers)
- Serialize-deserialize: `pickle` (default option)

Northeastern

Shipping Local Variables

- **Variables defined in the driver program** will automatically be shipped to the cluster along with function definition:

```
query = raw_input("Enter a query:")
pages.filter(lambda x: x.startswith(query))
```

- Some caveats:
  - Each task gets a new copy (**updates aren't sent back**)
  - Variable **must be pickle-able**
  - Variable **cannot be** an rdd
  - Don't use **fields of an object** (ships all of it!)
  - Beware of **shipping large variables**!

Example

```
class myClass(object):
  def __init__(self):
    self.mylist = range(10000) # 10000 element list
    self.query = “ERROR”

myOb=myClass()

pages.filter(lambda x: x.startswith(myOb.query))
```

Ships ENTIRE myOB , including `mylist`
(no error reported)

Example

```
class myClass(object):
   def __init__(self):
      self.myfile = file('temp.pkl', 'w') # file handle
      self.query = "ERROR"

myOb=myClass()

pages.filter(lambda x: x.startswith(myOb.query))
```

This will produce an error (attempts to ship myObj, which contains non-picklable `myfile`)

Northeastern

Example

```
class myClass(object):
  def __init__(self):
    self.myrdd = sc.textFile('WarAndPeace.txt')  # rdd
    self.query = "ERROR"

myOb=myClass()

pages.filter(lambda x: x.startswith(myOb.query))
```

This will produce an error (attempts to ship myObj, and rdds are not allowed in functions)

Outline

- ❑Map and other Transforms
- ❑Reduce and other Actions

The Reduce Action

```
val=myrdd.reduce(f)
```

myrdd

worker

```
[x1,
 x2,
 x3,
…,
x333 ]
```

```
def add(x,y):
    return x+y
```

- Reduces the elements of this RDD using the specified **commutative** and **associative binary** operator f.

worker

```
[x334,
 x335,
 x336,
…,
x665]
```

f=add

```
val=x1 +
 x2 +
 x3 +
…
 x1000
```

- Result return to **driver** program

worker

```
[x666,
 x667,
 x668,
…,
x1000]
```

The Reduce Action

```
def add(x,y):
    return x+y
val=myrdd.reduce(add)
```

- Reduces the elements of this RDD using the specified **commutative** and **associative binary** operator f.

only this is checked!!

- **f** is **binary:** takes 2 arguments
- **f** is **commutative:**

```
f(x,y) == f(y,x)
```

- **f** is **associative:**

```
f(f(x,y),z) == f(x,f(y,z))
```

Examples

```
def add(x,y):                      # numbers (addition)
    return x+y                     # strings,lists (concatenation, if order
                                                    does not matter)
                                   # arrays, matrices (element-wise addition)
def mult(x,y):                     # numbers
    return x*y

def max(x,y):                      # numbers, strings
    if x>y:
        return x
    else:
        return y


def intersection(x,y):                       # collections (lists, sets)
    return Set(x).intersection(y)
```

Other examples: minimum, logical **and**, **or**, **xor**,, bit-wise **and**,**or**,**xor**

Non-Examples

```
def addSquares(x,y):
    return x**2+y**2                    # commutative but not associative

def scaledAdd(x,y):
    return 2*x + 2*y                    # commutative but not associative

    ?                                   # associative but not commutative

def subtract(x,y):
    return x - y                        # neither associative nor commutative

def ImbalancedAdd(x,y):
    return 2*x + y                      # neither associative nor commutative
```

Non-Examples

```
def addSquares(x,y):
    return x**2+y**2            # commutative but not associative

def scaledAdd(x,y):
    return 2*x + 2*y            # commutative but not associative

def matrixMult(A,B):
    return numpy.dot(A,B)       # associative but not commutative

def subtract(x,y):
    return x - y                # neither associative nor commutative

def ImbalancedAdd(x,y):
    return 2*x + y              # neither associative nor commutative
```

How does reduce work?

```
def add(x,y):
    return x+y
```

worker

```
[x1,
 x2,
 x3,
…,
x333 ]
```

```
partial_sum1=x1+x2+…+x333
```

```
def add(x,y):
    return x+y
```

worker

```
[x334,
 x335,
 x336,
…,
x665]
```

```
partial_sum2=x334+x335+…+x665
```

```
def add(x,y):
    return x+y
```

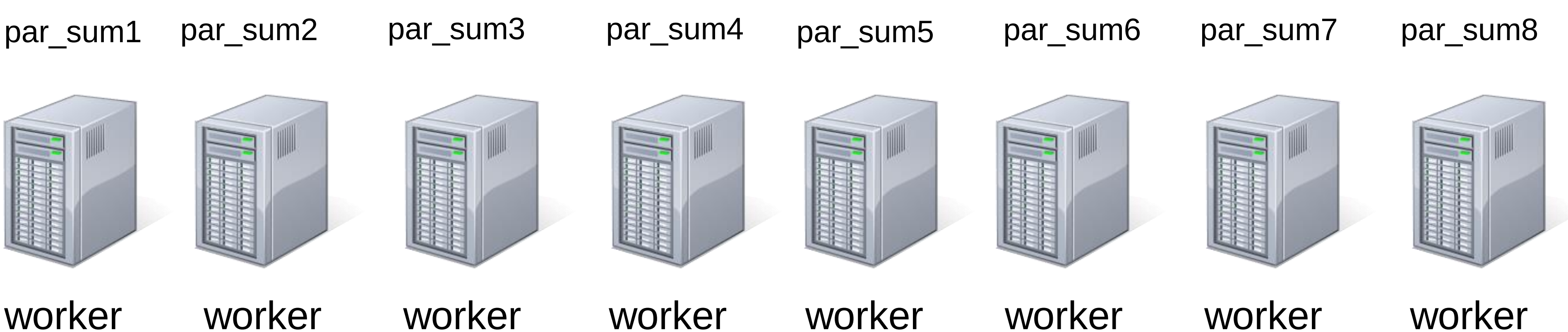
worker

```
[x666,
 x667,
 x668,
…,
x1000]
```

```
partial_sum3=x666+x335+…+x1000
```

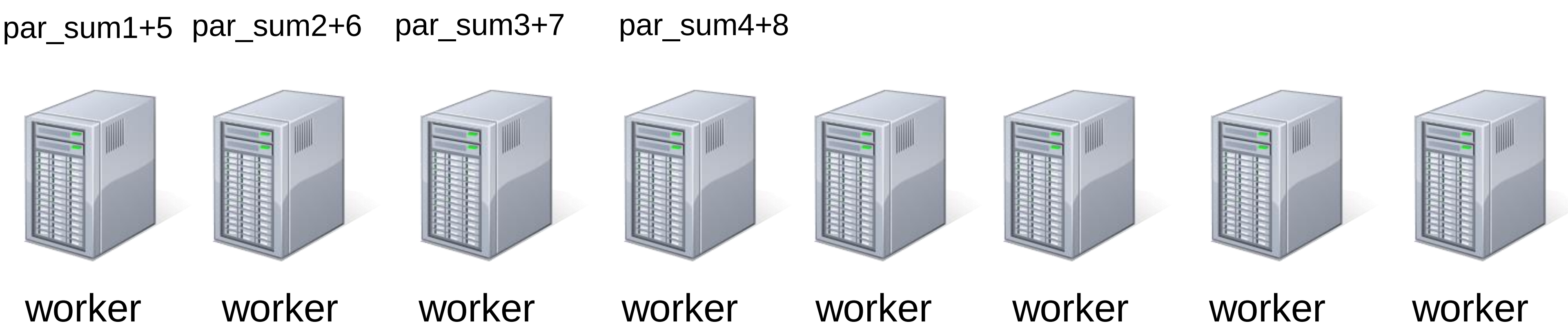
How does `reduce` work?

Round 1: Move results to n/2 processors

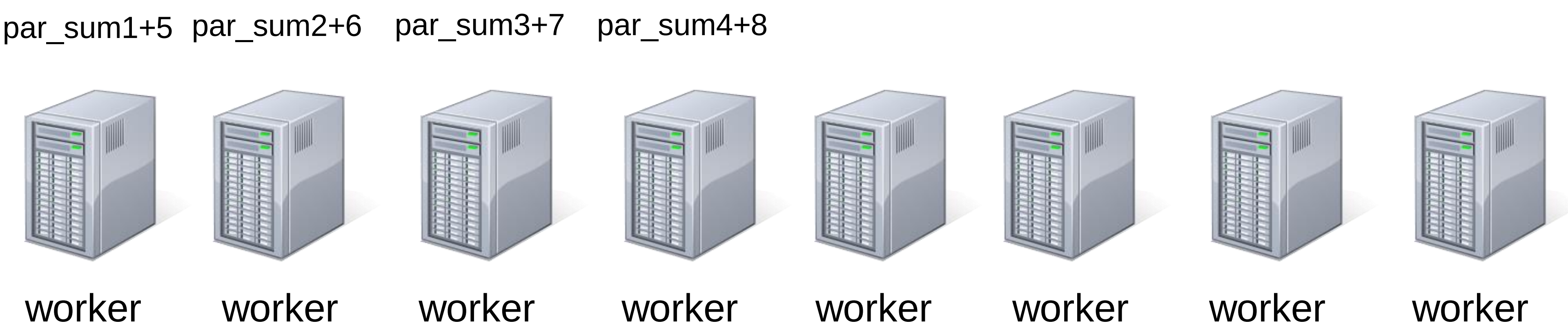


How does `reduce` work?

Round 1: Combine results

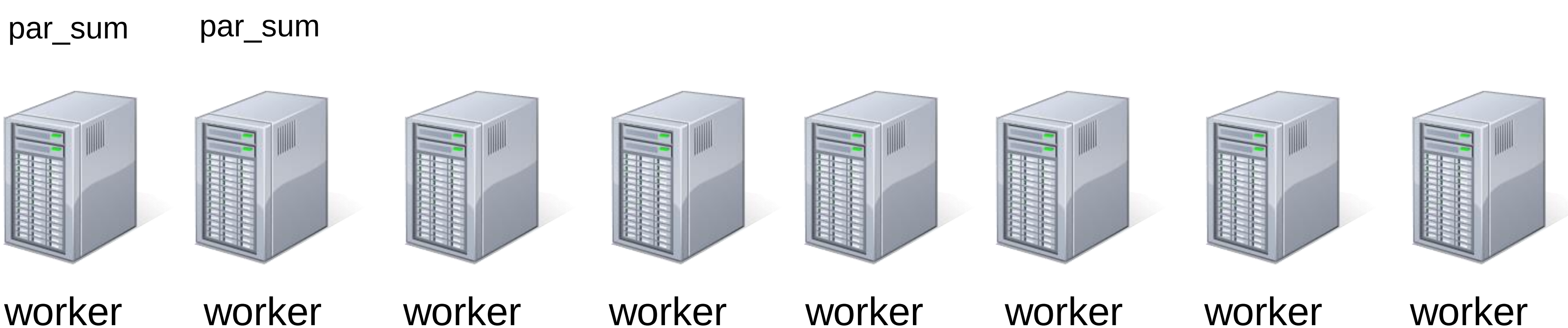


Round 2: Repeat



How does `reduce` work?

Round 3: Repeat



Northeastern

Serial Reduce Execution Time

Suppose we want to apply binary op (e.g. +) on n elements in `li`

```
serial_reduce(li,op)
```

Cost of one op (in sec): c

Number of op executions: $n-1$

Serial execution time: $T_S \approx (n-1)\cdot c = \Theta(n)$

Northeastern

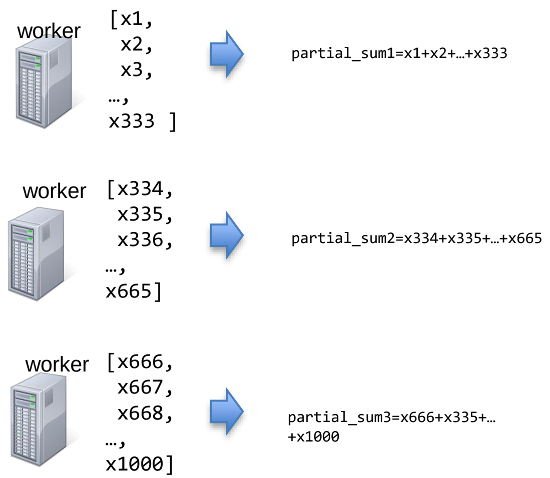
Parallel Reduce Execution Time

`rdd.reduce(op)`

n elements in `rdd`

p processors

Stage 1:



Assume `rdd` is equally partitioned

Each processor has n/p elements

Cost of one op (in sec): c

Number of op executions: $\frac{n}{p} - 1$

Decreases as we throw more machines to the problem!

Parallel execution time:

$$T_P^1 \approx (\frac{n}{p} - 1) \cdot c \approx \frac{nc}{p} = \Theta\left(\frac{n}{p}\right)$$

Northeastern

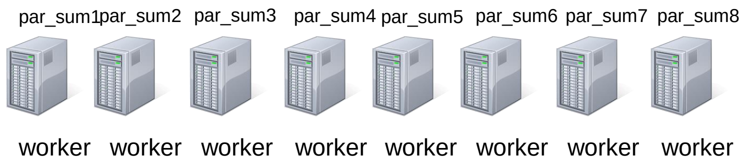
Parallel Reduce Execution Time

`rdd.reduce(op)`

n elements in `rdd`

p processors

Stage 2:



Total #of rounds for $p = 2^k$ processors: $k = \log_2 p$

Cost of one op (in sec): c

Cost to transfer/receive 1 element (in sec): c'

Each round terminates in: $c + c'$

Increases as we throw more machines to the problem!

Parallel execution time: $T_P^2 \approx \log_2 p \cdot (c + c') = \Theta\left(\log_2(p)\right)$

Northeastern

Speedup

Serial execution time: $T_S \approx (n-1) \cdot c$

Parallel execution time:
$$\begin{aligned} T_P &= T_P^1 + T_P^2 \\ &\approx (\frac{n}{p} - 1) \cdot c + \log_2 p \cdot (c + c') \end{aligned}$$

Speedup:

$$\frac{\text{Best } T_S}{\text{Best } T_P} \approx \frac{(n-1) \cdot c}{(\frac{n}{p} - 1)c + \log_2 p(c + c')} = \Theta\left(p \cdot \frac{1}{1 + \frac{p \log_2 p}{n}}\right)$$

Communication Costs

Total #of rounds for $p = 2^k$ processors: $k = \log_2 p$

Total #of messages:

Round 1: $p/2$
Round 2: $p/4$
. . .
Round log p: 1

$$\frac{p}{2} + \frac{p}{4} + \ldots + 2 + 1 = p \sum_{i=1}^{\log_2 p} \frac{1}{2^i} =$$

$$= p \cdot \frac{1}{2} \cdot \frac{1 - \frac{1}{2^{\log_2 p}}}{1 - \frac{1}{2}} = p \cdot (1 - \frac{1}{p}) = p - 1$$

total_sum

worker worker worker worker worker worker worker worker

Communication Costs

- Reduce **ships aggregated data around**
- **$\Theta(n)$ messages** for n processors
- Message size **may increase** with each aggregation!!!
  - Compare, e.g.,

```
def add(x,y):
    return x+y
```

applied to **numbers** vs.
applied to **strings** or **lists**

A Map-Reduce Example: Computing the Average

```
rdd=sc.parallelize(range(1000))

total,count =rdd.map(lambda x: (x,1))\
                .reduce(lambda x,y:
                        (x[0]+y[0],x[1]+y[1]) )

print 1.*total/count
```


Northeastern

Other Useful Actions

```
nums = sc.parallelize([1, 2, 3])

# Retrieve RDD contents as a local collection
nums.collect() # => [1, 2, 3]

# Return K elements
nums.take(2)   # => [1, 2]

# Count number of elements
nums.count()   # => 3
```

Exercise: how would I implement these with a reduce?

Other Useful Actions

```
nums = sc.parallelize([1, 5, 6, 3, 1, 2 ])

# Get the N elements from a RDD ordered in ascending order or as
# specified by the optional key function.
nums.takeOrdered(3)   # => [1, 1, 2]

nums.takeOrdered(3, key=lambda(x):-x)   # => [6,5,3]

# Write elements to a text file
nums.saveAsTextFile("file.txt")
```

`reduce` vs. `aggregate`

- `reduce`: **input** and **output** of aggregation are of the same type.

- `aggregate` allows them to be different

  aggregate(*zeroValue*, *seqOp*, *combOp*)

reduce vs. aggregate

- `reduce`: **input** and **output** of aggregation are of the same type.

- `aggregate` allows them to be different

aggregate(*zeroValue, seqOp, combOp*)

Combine pair (OT,IT)

IT: Input Type
OT: Output Type

`reduce` vs. `aggregate`

- `reduce`: **input** and **output** of aggregation are of the same type.

- `aggregate` allows them to be different

aggregate(*zeroValue*, *seqOp*, *combOp*)

Combine pair (OT,OT)

IT: Input Type
OT: Output Type

`reduce` vs. `aggregate`

- `reduce`: **input** and **output** of aggregation are of the same type.

- `aggregate` allows them to be different

aggregate(*zeroValue*, *seqOp*, *combOp*)

Starting OT value
("zero" element)

IT: Input Type
OT: Output Type

Computing an Average through aggregate

```
rdd=sc.parallelize(range(1000))

total,count =rdd.aggregate(
                  (0,0),                          #zero element
                  lambda pair,data:               #seqOp
                              (pair[0]+data,pair[1]+1),
                  lambda pair1,pair2:             #combOp
                     (pair1[0]+pair2[0],pair1[1]+pair2[1])
                  )

print 1.*total/count
```

Actions Available for Numerical RDDs only

`mean()`	Average of the elements
`sum()`	Total
`max()`	Maximum value
`min()`	Minimum value
`variance()`	Variance of the elements
`sampleVariance()`	Variance of the elements, computed for a sample
`stdev()`	Standard deviation
`sampleStdev()`	Sample standard deviation

Next Lectures

- ❑Working with key,value pairs
- ❑Controlling Parallelism
- ❑Lazy Evaluation & Persistence