CS 696 Intro to Big Data: Tools and Methods
Fall Semester, 2016
Doc 9 Parallel Computing
Sep 22, 2016

Parallel Computing

Concurrent computing

Running multiple processes or threads on same processor

Processes or threads are time-sliced

Parallel computing

Running multiple processes on different processors

Processes in same program run at the same time on different processors

Scaling up

- Adding more resources to a machine to allow it to handle larger tasks
  - Memory
  - Disk space
  - Faster processor

Scaling out

- Adding more machines/processors to handle larger tasks
- Requires parallel programming

Julia Parallel Processing

Low level constructs

High level constructs

Runs on
- Multicore processors
- Clusters

Cluster management

Experimental Julia to C/C++ compilers from Intel Labs
- Run Julia code 20 to 100 times faster than Spark
- Spark claims to be 10 to 100 times faster than Hadoop

Julia Parallel Processing - Low Level

@spawn, @spawnat

Run code on separate/remote processor

@everywhere

Run a command on all processors

fetch

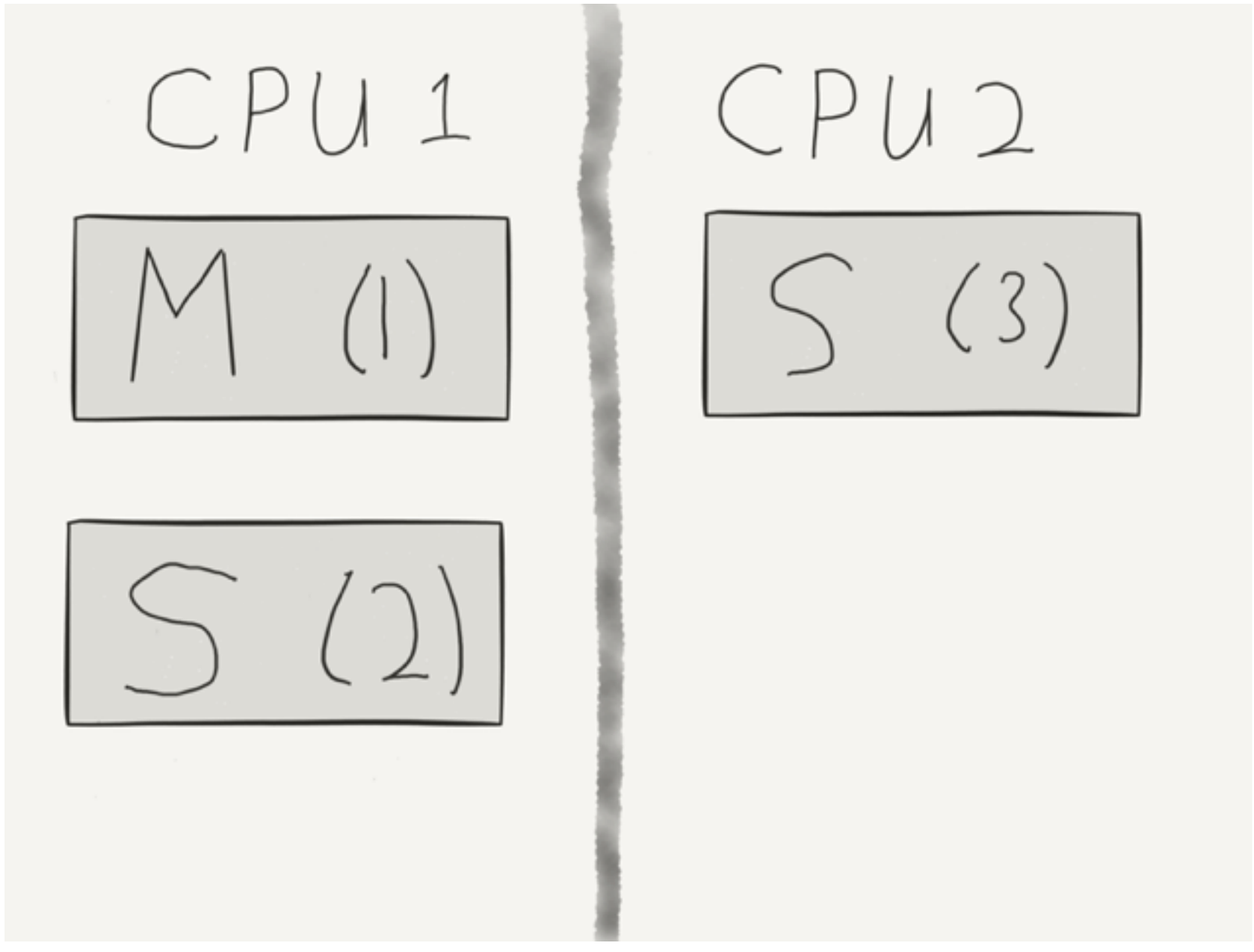
Obtain results from separate processor

put!

store a value on a separate processor

```
addprocs(2)
workers()
procs()
```

```
# only have two cores
# [ 2, 3]
# [ 1, 2, 3]
```

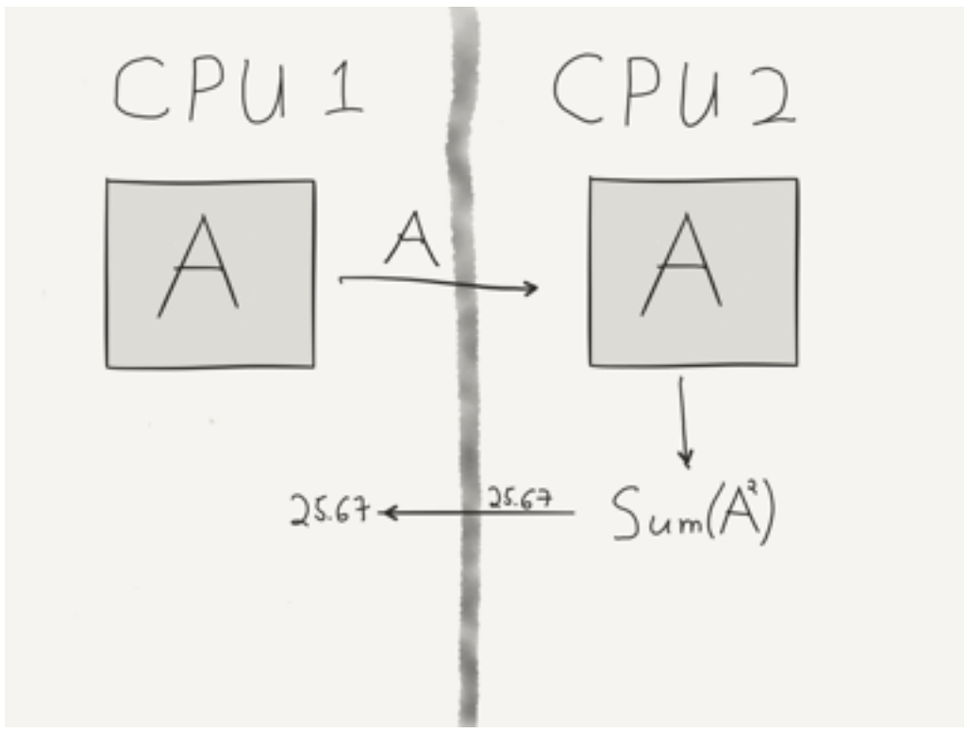


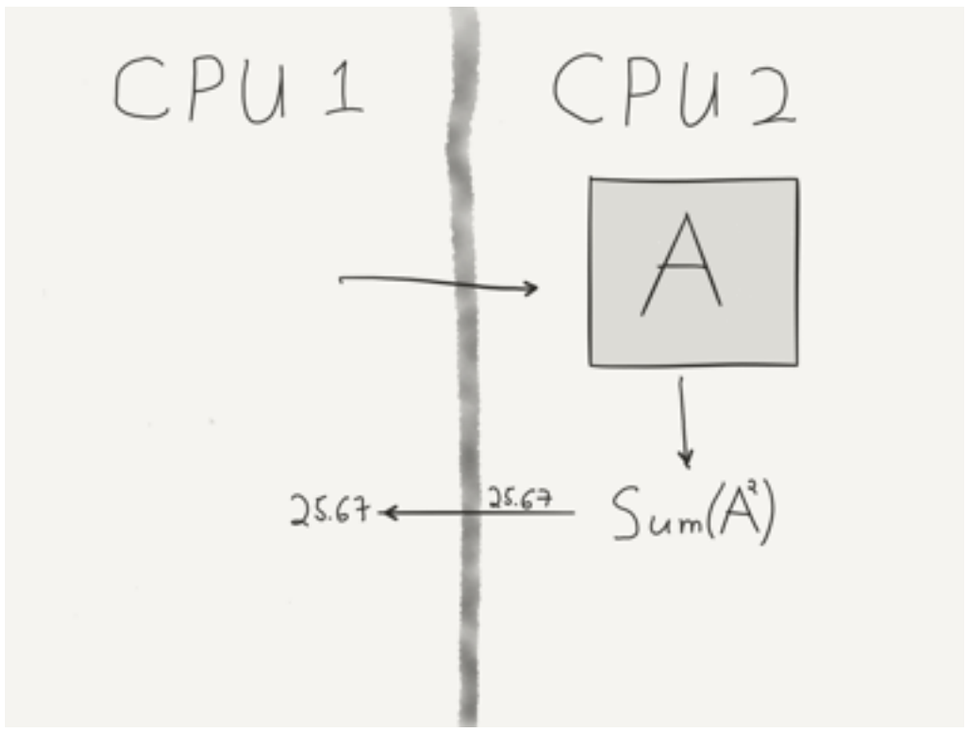
Adding 1 elementwise In parallel

```
addprocs(2)                                  # only have two cores
workers()                                    # [ 2, 3]
procs()                                      # [ 1, 2, 3]


remote = @spawn rand(2,2)                    # RemoteRef{Channel{Any}}(2,1,3)

fetch(remote)                                #= [0.477549  0.193374;
                                                     0.250799  0.0512077]


result = @spawn 1 .+ fetch(remote)

fetch(result)                                # [1.47755  1.19337;
                                                   1.2508   1.05121]
```

```
A = rand(n,n)
Aref = @spawn sum(A^2)
fetch(Aref)
```

```
Bref = @spawn sum(rand(n,n)^2)
fetch(Bref)
```





You need to be aware what you are doing on each processor

Do you need to send A from P1 to P2?

```
function count_heads(n)
    c::Int = 0
    for i=1:n
        c += rand(Bool)
    end
    c
end

a = @spawn count_heads(100000000)
b = @spawn count_heads(100000000)
fetch(a)+fetch(b)
```

```
On worker 5:
function count_heads not defined on process 3
```

count_heads.jl

```
function count_heads(n)
    c::Int = 0
    for i=1:n
        c += rand(Bool)
    end
    c
end
```

Put count_heads.jl in Julia path

In Julia

```
require("count_heads")

a = @spawn count_heads(100000000)
b = @spawn count_heads(100000000)
fetch(a)+fetch(b)
```

High-level Parallel/Performance Constructs

@parallel

@parallel

```
@parallel reducer for var = range
    body
end
```

Divide the loop among worker processes

Each process accumulates results and used reducer to combine the results

Result is send back to master and reduce is used combine all results

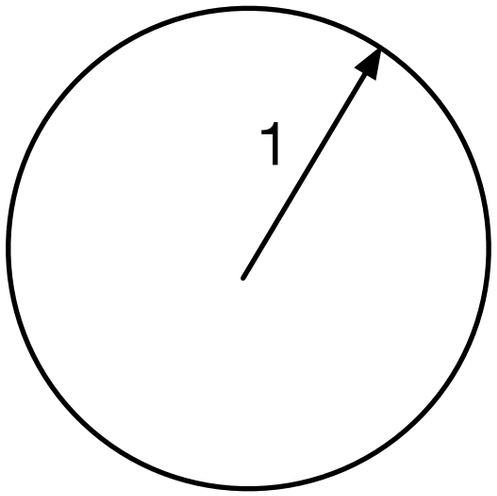
```
addprocs(10)
@parallel (+) for k = 1:100_000
      rand(1)
end
```

Each worker will sum 10_000 random numbers

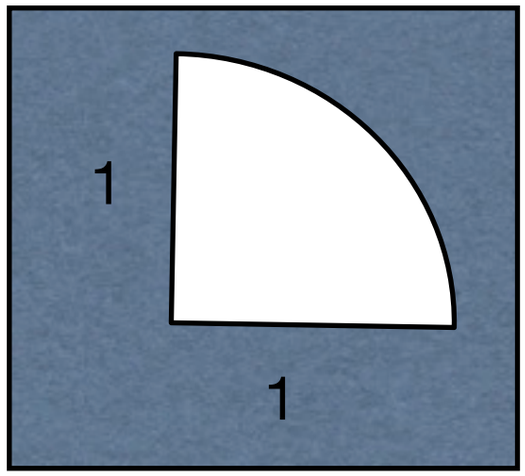
Master will sum up the 10 results

Assuming you have 10 processors

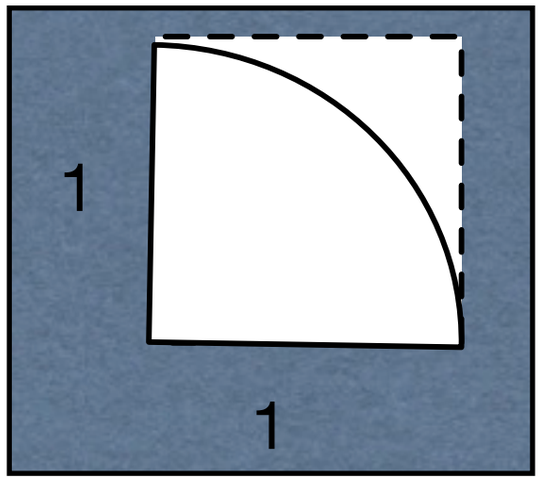
Computing Pi



Area = pi*r*r = pi



Area = pi/4



Area of Square = 1

Select random point in unit square

Probability that point is in the quarter Circle is (pi/4)/1 = pi/4

Select N random points in unit square

Let K = number of points in quarter circle

K should be about N*pi/4

4K/N should be about pi

Computing Pi

```
function findpi(n)
  inside = 0
  for i = 1:n
    x, y = rand(2)
    if (x^2 + y^2 <= 1)
      inside +=1
    end
  end
  4 * inside / n
end
```

rand(2)

returns two random numbers between 0 and 1

N	findpi(N)
1_000	3.148
100_000	3.15028
100_000_000	3.14169832
1_000_000_000	3.141595912

π = 3.1415926535897...

Code from http://www.exegetic.biz/blog/2015/09/monthofjulia-day-12-parallel-processing/. Of course we are measuring how random the random number generator is

Parallel Version

```
function parallel_findpi(n)
  inside =  @parallel (+) for i = 1:n
    x, y = rand(2)
    x^2 + y^2 <= 1 ? 1 : 0
  end
  4 * inside / n
end
```

findpi verses parallel_findpi

On Edora - Has 4 Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz processors

addprocs(4)

Time to Run in Seconds

N	findpi(N)	parallel_findpi(N)	Speedup
1_000	0.000079	0.003326	0.02
100_000	0.012105	0.06380	0.19
100_000_000	7.234	2.19617	3.29
1_000_000_000	72.36	18.660	3.88

Speedup

T(1) = time for sequnetial program to run

T(N) = time for parallel program to run on N processors

S(N) = speedup using N processors

S(N) = T(1)/T(N)

Timings on JuliaBox - 16 CPU

Time to Run in Seconds

N	findpi(N)	parallel_findpi(N) 16 processor	parallel_findpi(N) 8 processor
100_000	0.009	0.009	0.010
1_000_000	0.083	0.030	0.069
10_000_000	0.813	0.234	0.216
100_000_000	8.143	1.656	2.133
1_000_000_000	82.219	14.663	20.762
10_000_000_000		125.585	246.524

Speedup

100_000	0.009	1	0.9
1_000_000	0.083	2.8	1.2
10_000_000	0.813	3.5	3.8
100_000_000	8.143	4.9	3.8
1_000_000_000	82.219	5.6	4.0
10_000_000_000		6.5	3.3

Amdahl's Law

Ts = time of task that is inherently sequential

Tp = time of task that can be parallelized

T(1) = time for sequnetial program to run
T(N) = time for parallel program to run on N processors
S(N) = speedup using N processors

T(1) = Ts + Tp

T(N) = Ts + Tp/N Assuming we can parallelize perfectly

S(N) = T(1)/T(N)
= (Ts + Tp)/(Ts + Tp/N)

Amdahl's Law

$S(N) = (Ts + Tp)/(Ts + Tp/N)$

if Ts = 0 and we can perfectly parallelize the task we get

$S(N) = Tp/(Tp/N) = N$

Ts is never zero
Perfect parallization is not possible

So

$S(N) < N$

Amdahl's Law

Theory

S(N) = (Ts + Tp)/(Ts + Tp/N)

S(N) < N

Practice

It is possible for S(N) > N

How

Single processor may not be able to fit data in physical memory

Paging will significally slow sequenial program down

N processors can have more total memory that single processor

So parallel version may not have paging issues

More Realistic Amdahl's Law

Ts = time of task that is inherently sequential

Tp = time of task that can be parallelized

Tis = Average additional serial time doing interprocessor communication

Assume each processor takes same amount of time

Total time is N*Tis

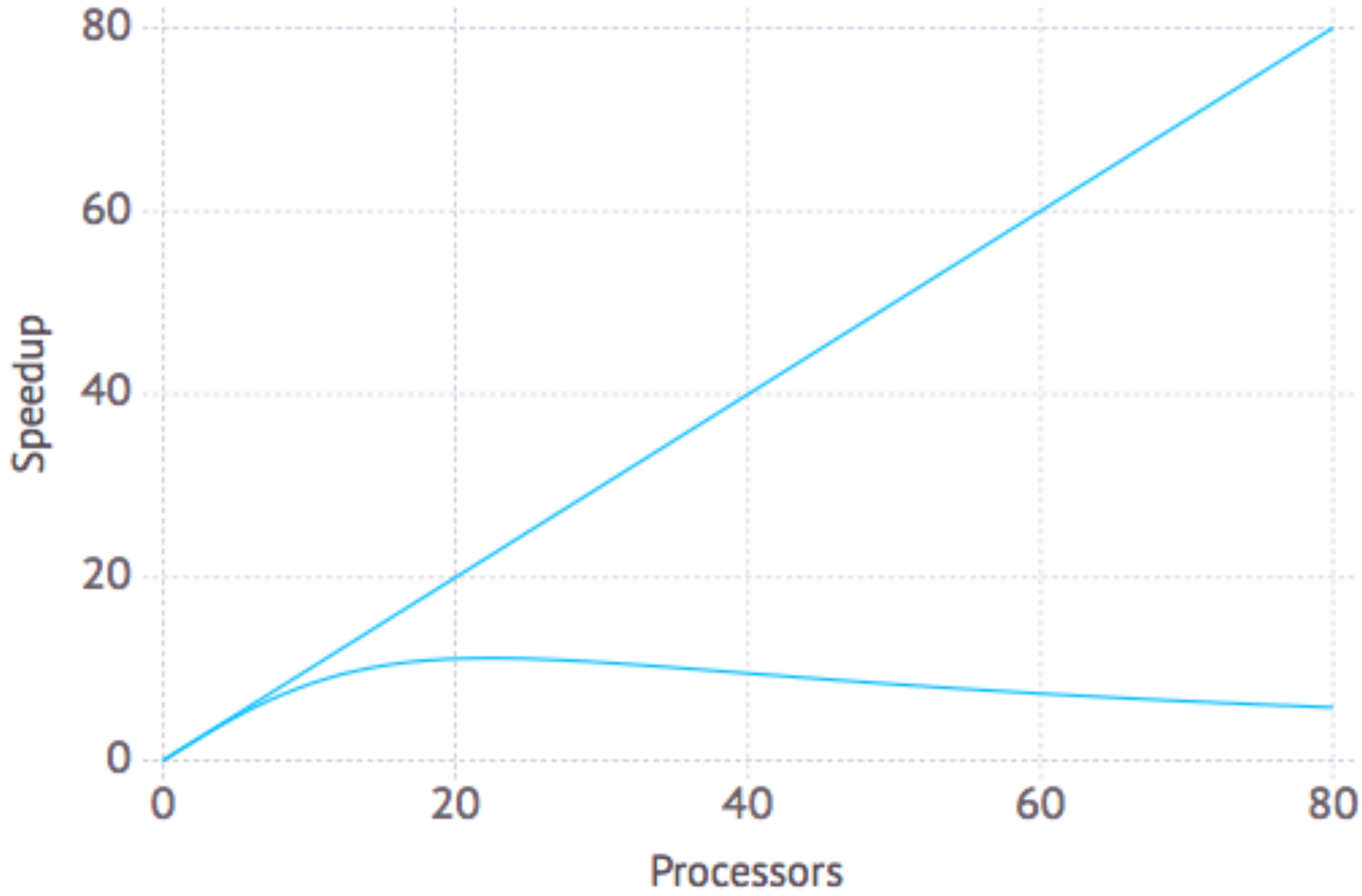
Tip - Average additional time by each processor doing set up, idle time, etc.

$$S(N) = (Ts + Tp)/(Ts + N*Tis + Tip + Tp/N)$$

Source http://www.phy.duke.edu/~rgb/brahma/Resources/als/als/node3.html

$$S(N) = (Ts + Tp)/(Ts + N*Tis + Tip + Tp/N)$$

Ts = Tip = 0
Tp = 10_000
Tis = 20 = 0.2% * Tp



Monte Carlo Method

Uses repeated random sampling to obtain numerical results

Used mainly in:

- Optimization
- Numerical Integration
- Generating draws from probability distribution

Embarrassingly (Pleasingly) Parallel

Little or no effort needed to separate problem into parallel tasks

Little or no communication needed between parallel tasks

Searching a web page that contains key words

These are the types of problems that can be

Solved using Hadoop & Spark

Compilers can detect some forms and parallelize for you

Distributed Arrays - DistributedArrays.jl

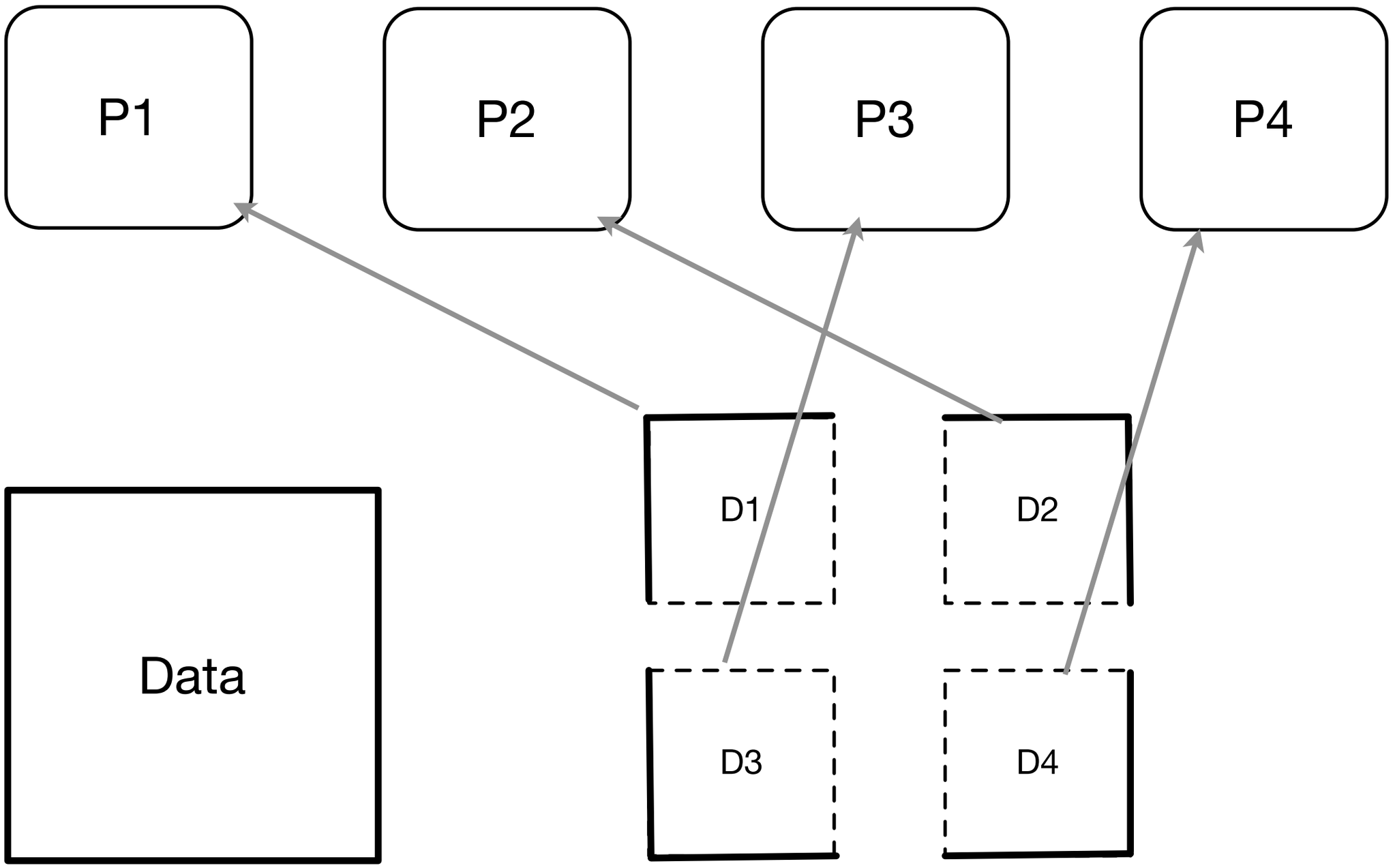
Distributes Arrays among processors

Can distribute arrays from master to slaves

Can create arrays on slaves

Master can work arrays on slaves

Distributing Data



Creating a Distributed Array

dzeros(100,100,10)
dones(100,100,10)
drand(100,100,10)
drandn(100,100,10)
dfill(x,100,100,10)

Using DistributedArrays.jl

```
onmaster = rand(100,100)
distributed = distribute(onmaster)          #distribute onmaster to the workers

sum(distributed)                            # compute sum locally on workers
                                                    # combine the result on master

heads = map(x -> x > 0.5,distributed)       # apply map on workers
                                                    $ return result on master
```

SharedArrays

Each worker has access to the array

```
addprocs(3)
            3-element Array{Int64,1}:
             2
             3
             4
```

```
S = SharedArray(Int, (3,4), init = S -> S[Base.localindexes(S)] = myid())
```

```
3×4 SharedArray{Int64,2}:
 2  2  3  4
 2  3  3  4
 2  3  4  4
```

ClusterManagers

Launches worker processes in a cluster environment
Managing events during the lifetime of each worker
Providing data transport

Julia Cluster

The initial Julia process, also called the master, is special and has an id of 1
Only the master process can add or remove worker processes
All processes can directly communicate with each other

Types of Cluster Managers

LocalManager,

used when addprocs() or addprocs(np::Integer) are called

SSHManager

used when addprocs(hostnames::Array) is called with a list of hostnames

Remote hosts need passwordless login enabled

ArrayFire.jl

GPU computing

```
using ArrayFire

a = rand(10, 10)
on_gpu = AFArray(a)

result_on_gpu = (on_gpu + 1)/5

result_on_cpu = Array(result_on_cpu)
```

HPAT.jl, ParallelAccelerator.jl

Intel Labs projects to provide high level efficient & fast parallel code

ParallelAccelerator.jl

- Converts Julia code to C/C++
- Imports C/C++ code into Julia

- Supports subset of Julia

- Uses implicit parallelism in
  - map, reduce, comprehension
  - .+, .- , .* , ./ converted into data-parallel map operations

HPAT.jl

- Using ParallelAccelerator converts Julia code to C/C++ & MPI calls for distributed computing

Sample Using ParallelAccelertor

```
using ParallelAccelerator
@acc function calc_pi(n)
        x = rand(n) .* 2.0 .- 1.0
        y = rand(n) .* 2.0 .- 1.0
        return 4.0 * sum(x.^2 .+ y.^2 .< 1.0)/n
end
```

```
function calc_pi_normal(n)
        x = rand(n) .* 2.0 .- 1.0
        y = rand(n) .* 2.0 .- 1.0
        return 4.0 * sum(x.^2 .+ y.^2 .< 1.0)/n
end
```

```
@time calc_pi(10_000_000)
```

0.284697 seconds
(28 allocations: 1.641 KB)

```
@time calc_pi_normal(10_000_000)
```

1.167740 seconds
(7.49 k allocations: 688.223 MB, 52.57% gc time)

Using for loop rather than .* etc

1.105030 seconds
(10.00 M allocations: 915.528 MB, 18.67% gc time)

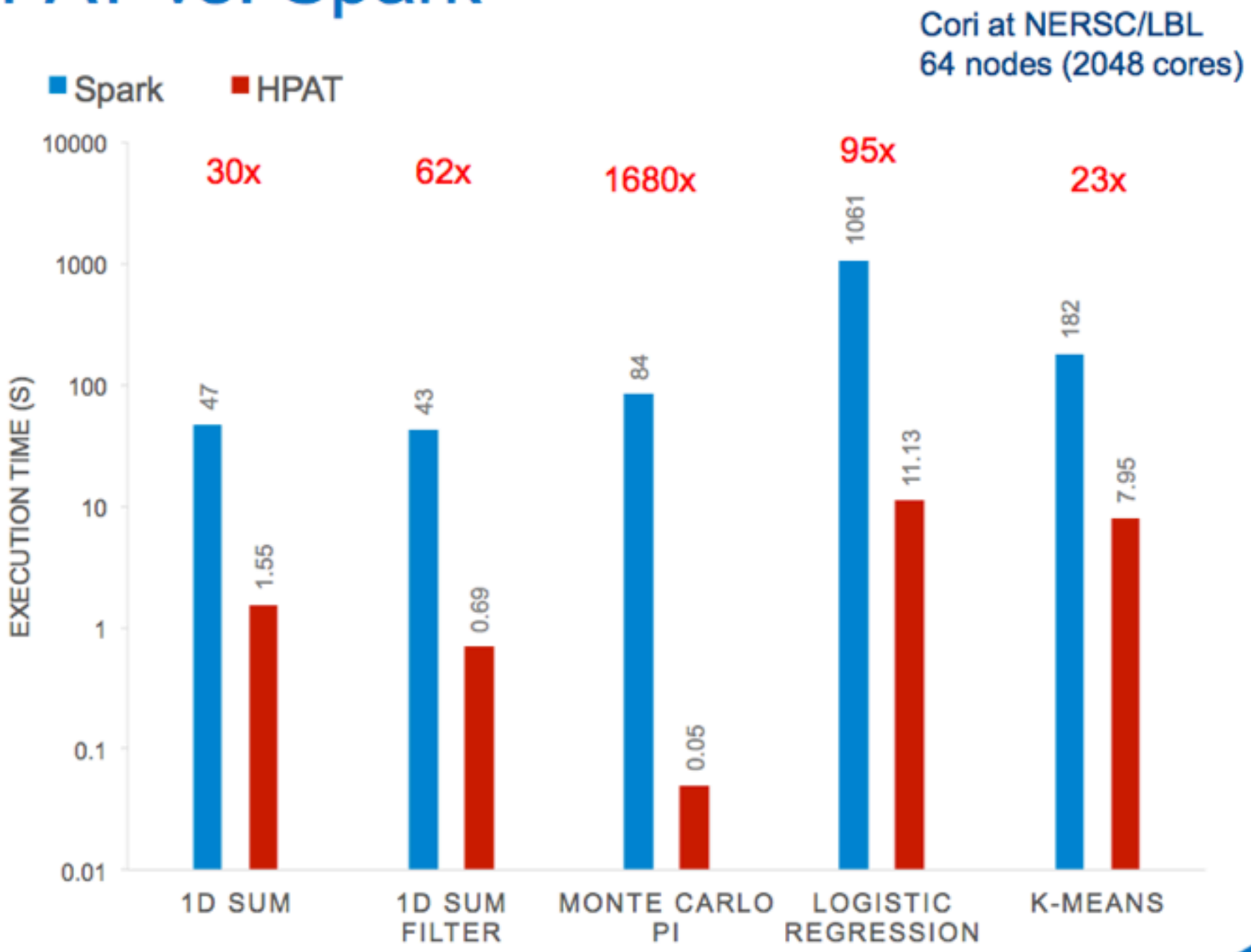
Pkg.add("ParallelAccelerator") adds a lot of stuff
The first time running calc_pi takes a long time

Sample Using HPAT

```
using HPAT
@acc hpat function calc_pi(n)
        x = rand(n) .* 2.0 .- 1.0
        y = rand(n) .* 2.0 .- 1.0
        return 4.0 * sum(x.^2 .+ y.^2 .< 1.0)/n
end
```

Now can be run on machines supporting mpi

HPAT vs. Spark



Wednesday, September 21, 16

Source: http://www.slideshare.net/EhsanTotoni/hpat-presentation-at-juliacon-2016/1