CIBR Mini-Workshop on Parallel Computing

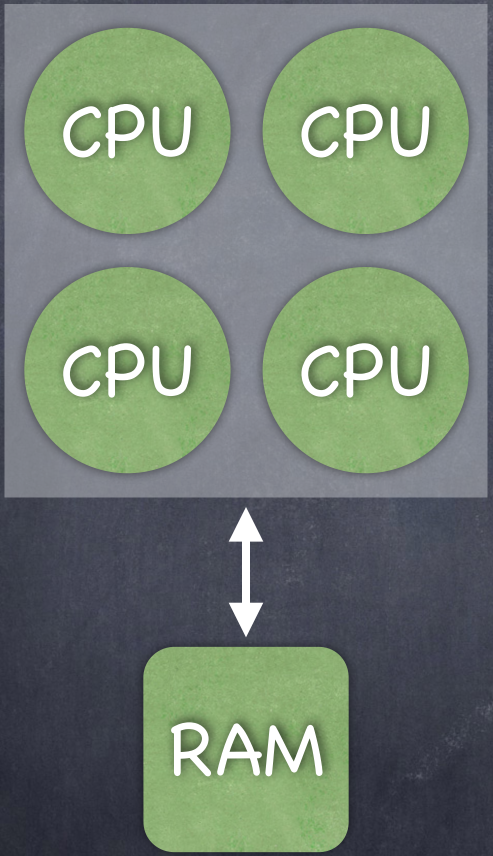
http://blake.bcm.edu/CIBRClusters

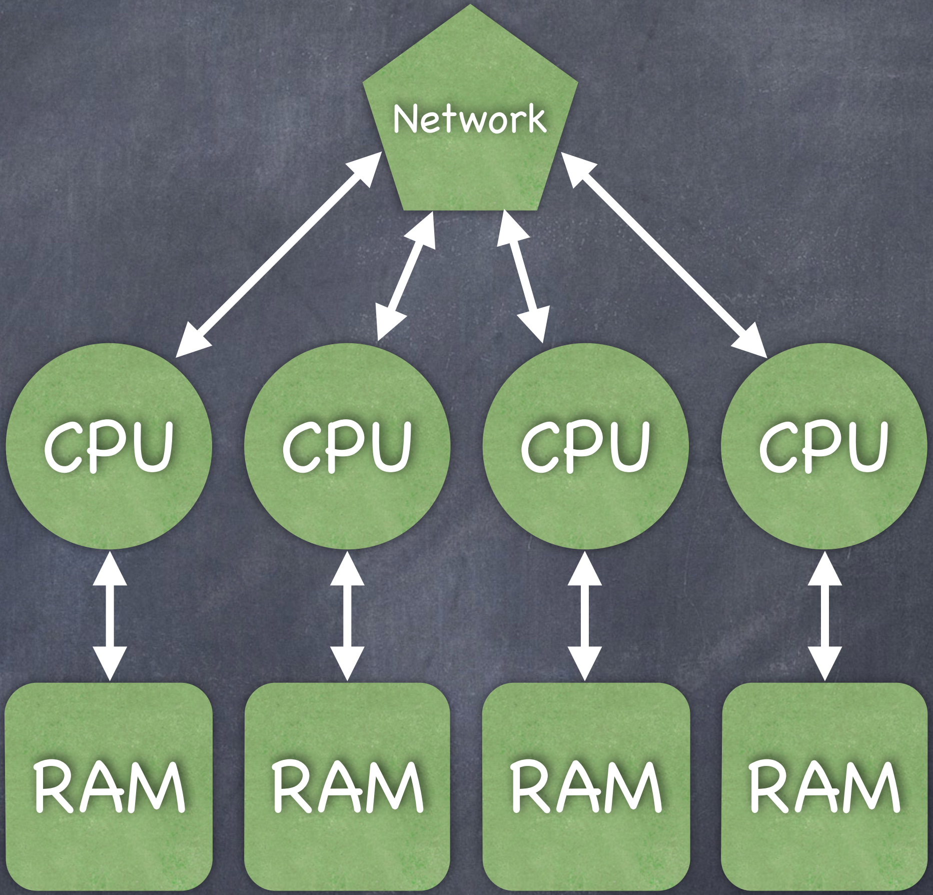
Part 1
Cluster Architecture

Shared Clusters at BCM

- Genome Center
- Cancer Center
- CIBR Co-op:
  - 5 clusters (CIBR + 6 PIs)
    - 960+704+640+256+180 = 2740 cores
    - ~24,000,000 CPU-hr/year
    - 350 TB reliable storage
    - 60,000 CPU-hr/qtr free for any CIBR PI

Shared or Distributed ?

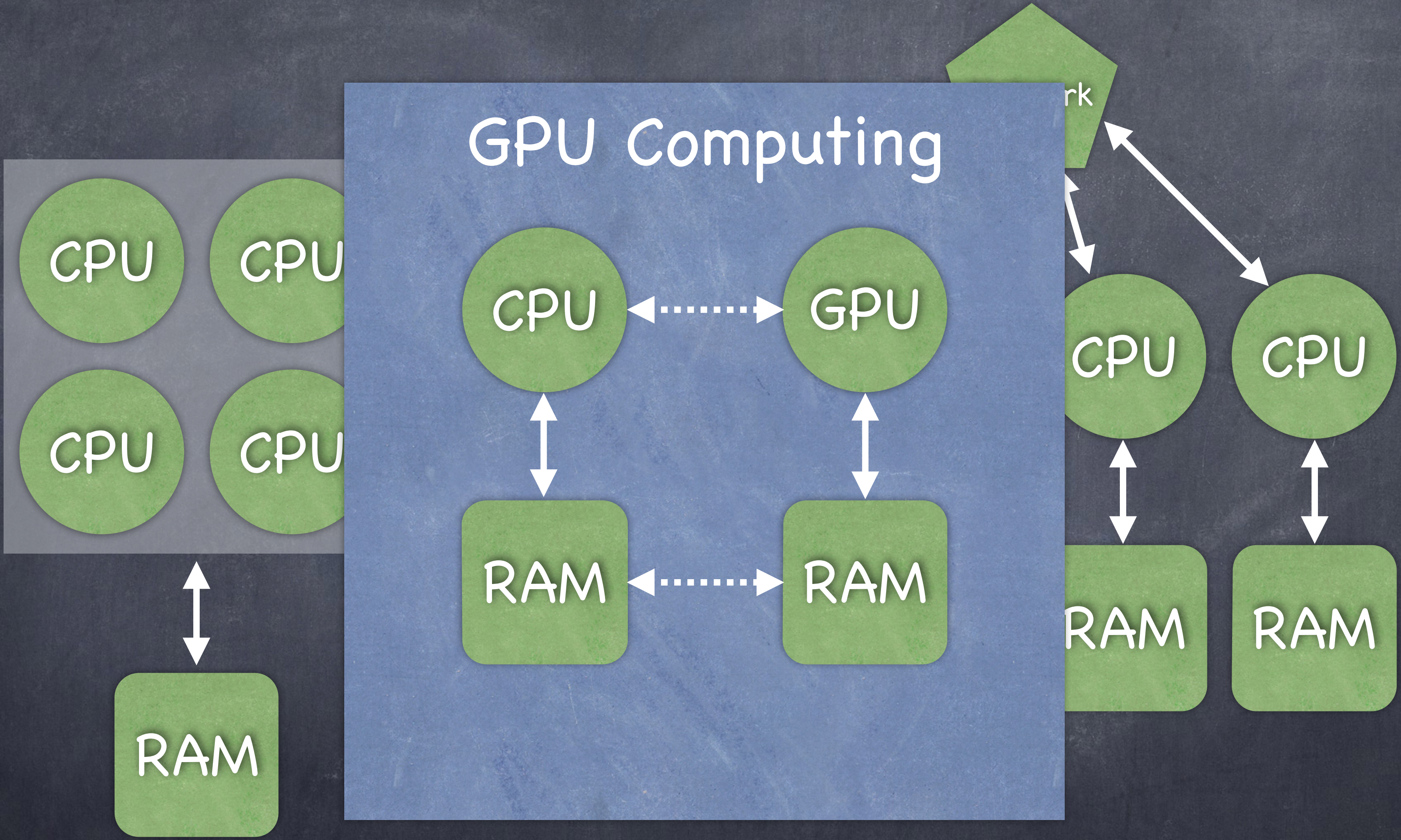




Easy Parallelism

Inexpensive

Shared or Distributed ?



Easy Parallelism

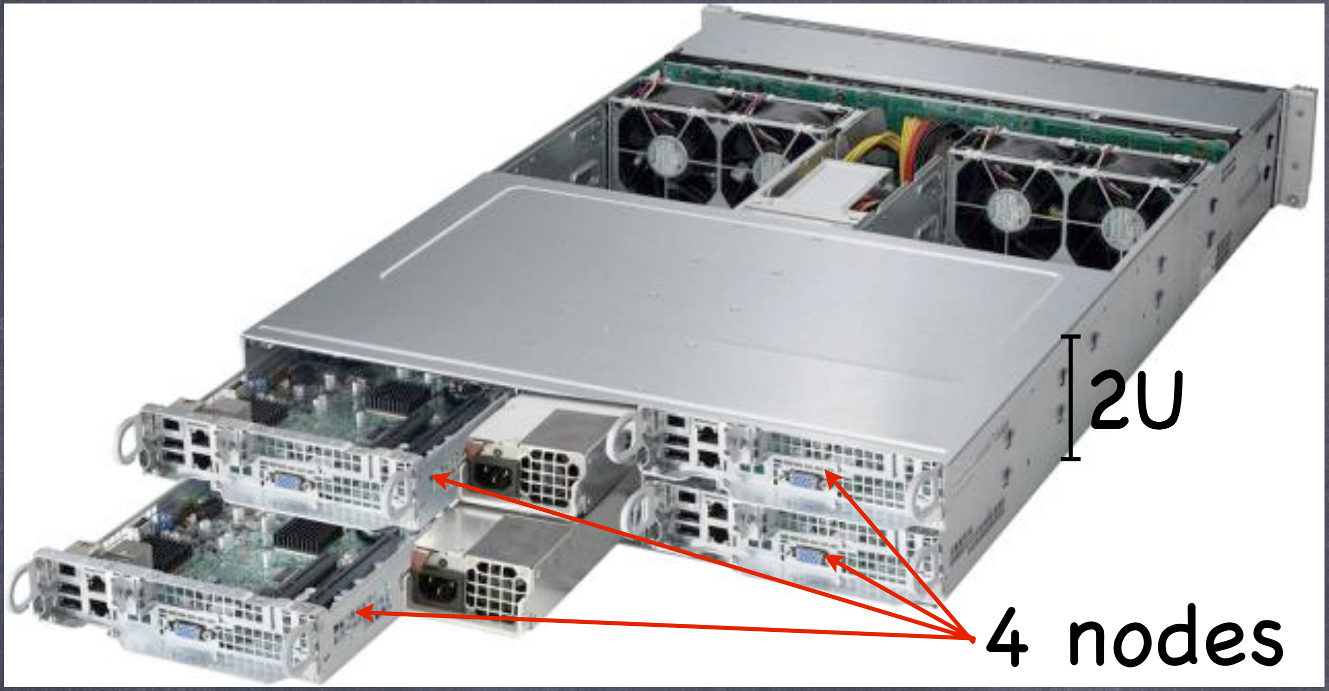
Inexpensive

SMP or Distributed

- S 1964, CDC 6600, $60m (2012 $), 500 kFlops, 1 CPU
- S 1977, CRAY 1, $33m, 80 MFlops, 1 CPU ← iPhone4S
- S 1984, CRAY XMP, $25m, 800 MFlops, 4 CPUs - vector
- D 1987, CM-2, $22m, 6 GFlops, 65,536 CPUs, 2048 MPU
- S 1996, Origin 2000, ~$3m, 10 GFlops, 32 CPUs SMP
- D 2005, Cluster, $0.4m, 900 GFlops, 106 nodes, 212 cores
- ? 2011, Cluster, $0.22m, 5.5 TFlops, 48 nodes, 576 cores
- ! 2015, Tesla K80 GPU, $0.005m, 5.6 TFlops, 1 PCIe board

Typical Rack 42U

Cluster Hardware



- 1 Chassis:
  - 4 nodes
    - 2 processors/node:
      - 12 cores/processor
    - 128 GB RAM/node
    - 2 TB Hard Drive/node
  - 10 Gb ethernet

- 96 cores
  - 2-4 TFLOPS
  - 68 GB/sec RAM
- 512 GB RAM (5GB/core)

- $26,000 ($270/core)

Cluster Hardware

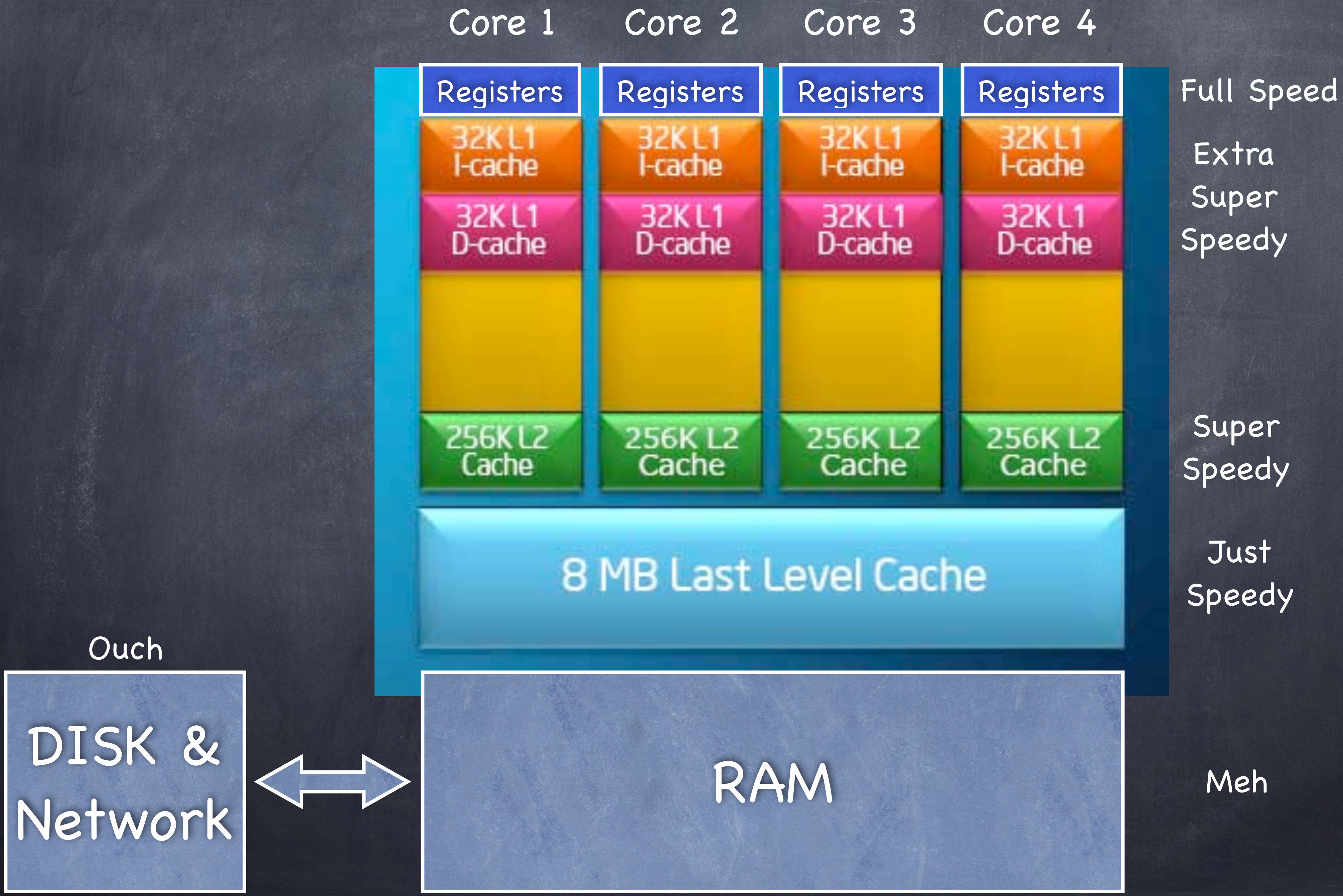
- 1 Rack:
  - 20 * 2U ->
    - $26,000 * 20 -> $520k + ~$30k (rack, etc.)
    - 20*96 cores -> 1920 cores
      - 40-80 TFLOPS Peak
      - ~30 KW
        - 30 KW * 8700 hr/yr = 260 MWH/yr
        - ~$30,000/yr electric bill
        - A/C bill !?

Comparison of Languages

Loop/Array/Math Benchmark

Language	Time
C++ (-O2)	1
C++ (no opt)	2
Javascript (JIT)	2
Java	5.1
Python	16.5
Perl	24.6
PHP	55.6

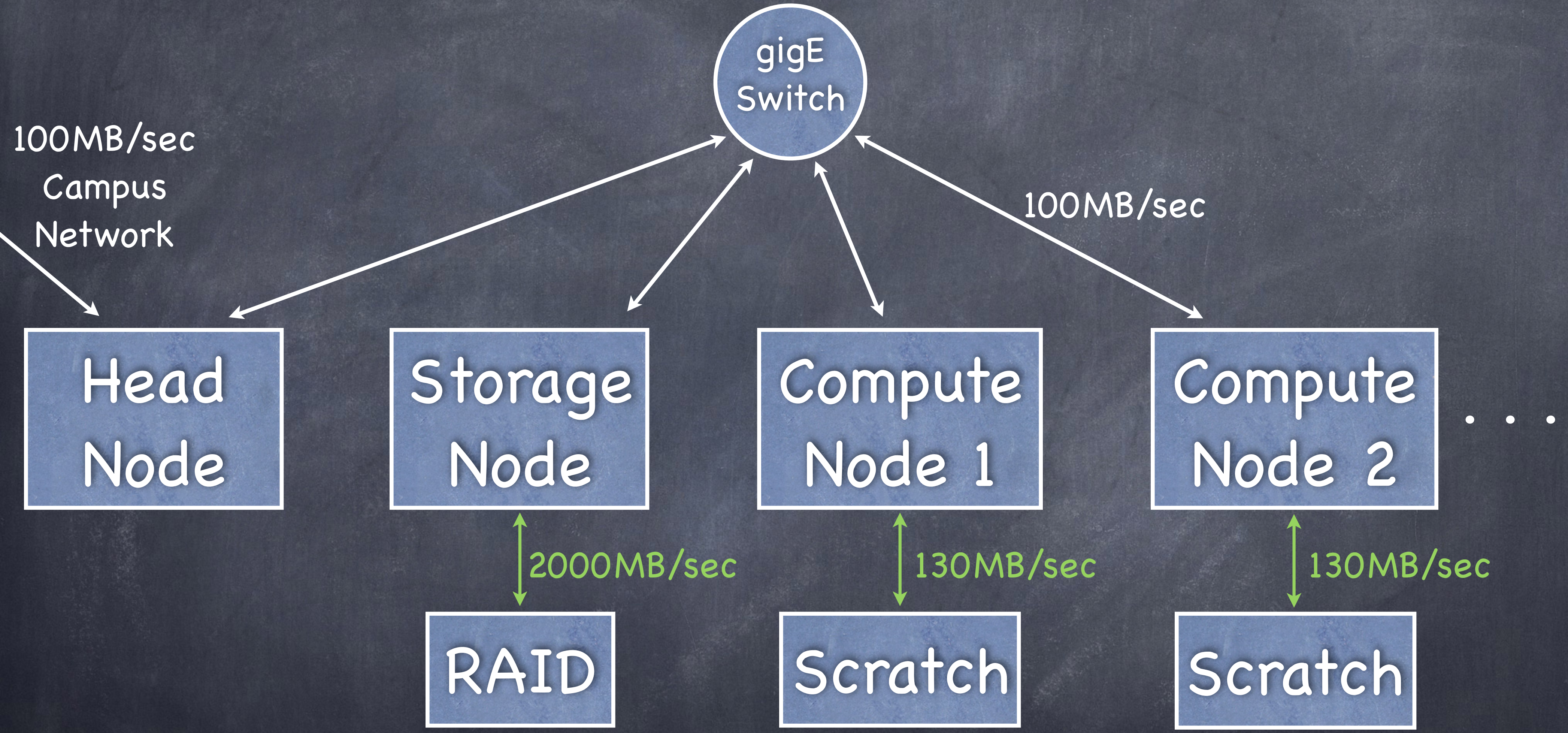
Quad Core Cache Structure



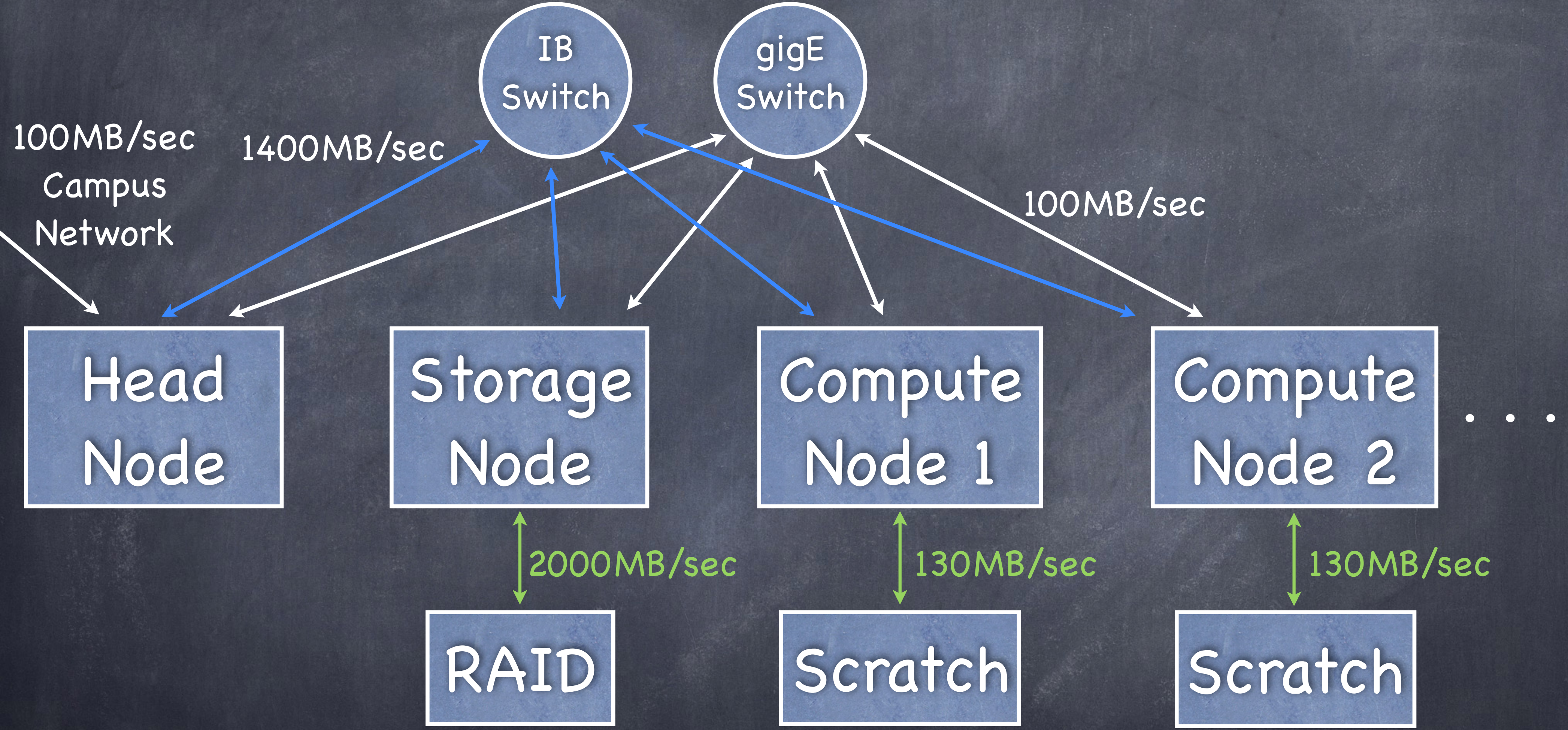
Speed

- 300,000 MIPS - (Million instructions per second) current peak capabilities of a single CPU (with multiple cores)
- 100,000 MB/sec - Level 1 cache memory bandwidth (32 kbytes/core)
- 50,000 MB/sec - Level 2 cache memory bandwidth (256 kbytes/core)
- 35,000 MB/sec - Level 3 cache memory bandwidth (8000 kbytes/CPU)
- 18,000 MB/sec - RAM (typical DDR3 dual channel)
- 8,000 MB/sec - PCIe x16 (2.0)
- 1,500 MB/sec - 12 drive RAID6 with PCIe controller
- 800 MB/sec - QDR Infiniband
- 150 MB/sec - Typical sequential disk read bandwidth for one drive
- 100 MB/sec - Gigabit network

Hypothetical Cluster



Hypothetical Cluster



What about the cloud?

- Amazon EC2:
  - c3.8xlarge, $1.68/hour
  - 16 physical core, 4GB RAM/core

What about the cloud?

- Amazon EC2:
  - c3.8xlarge, $1.68/hour
  - 16 physical core, 4GB RAM/core
- Cluster
  - Prism: $4800 /(3 years * 365 * 24) = $0.18/hour
  - ~140k CPU-hr/yr
  - 16 physical cores, 4GB RAM/core

XSEDE/TACC

- Multiple clusters available, eg:
  - Stampede: 6,400 nodes + Phi coprocessors
  - ~2 + 7 PF
  - FREE allocation grants for academic projects
  - 2-3M CPU-hr/year allocations possible

Part 2
Parallelism

Simple Task

- Take a 20 GB sequence and locate all of the TATA blocks within it.
  - Choice of language ?
  - Run on a cluster ?
  - Multiple cores ?
  - How long will it take to run ?
  - How to make it faster ?

Another Task

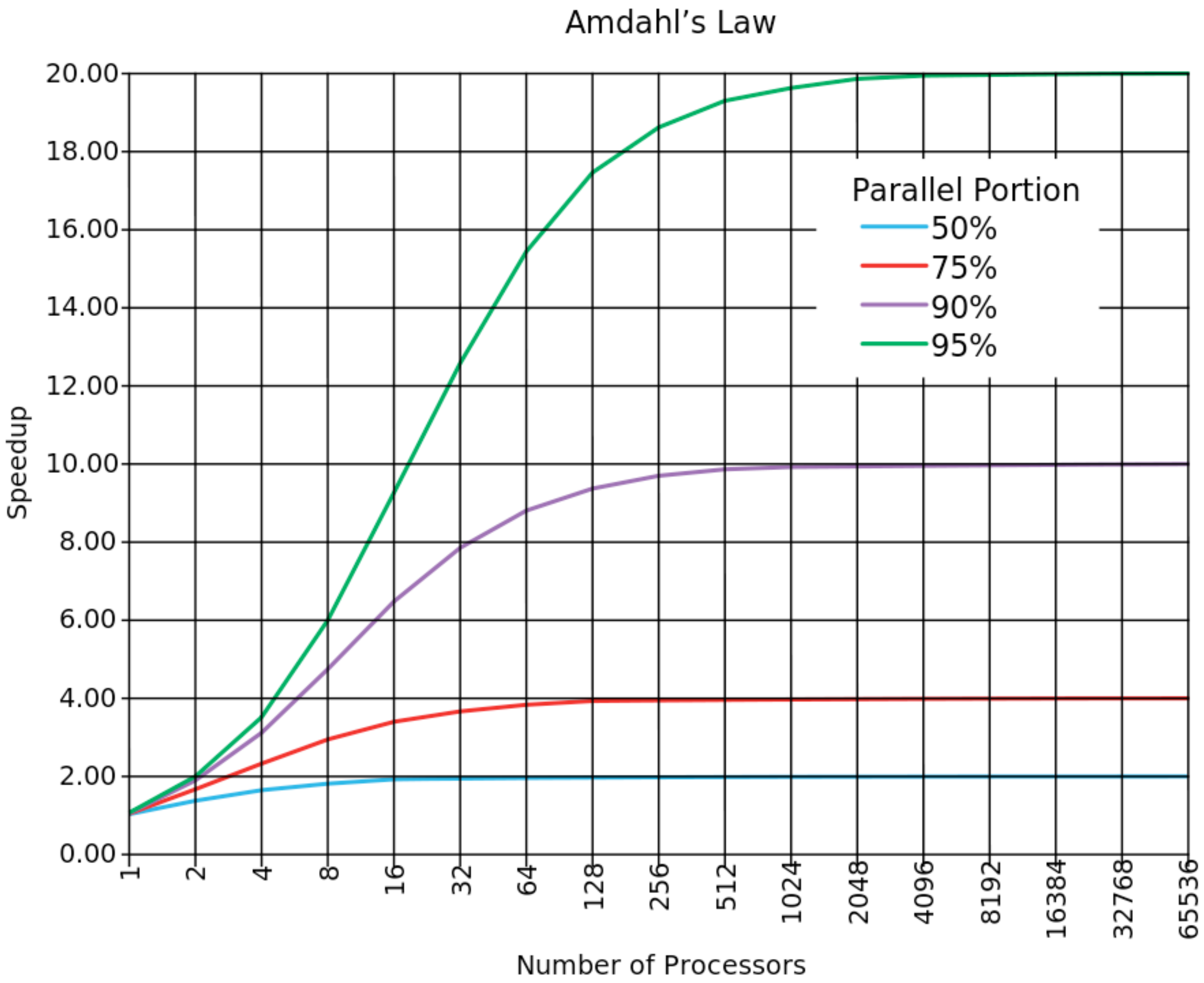
- You have 500, 4096x4096 pixel floating point images. You need to apply a (Fourier) low-pass filter to all of them
  - Read -> FFT -> multiply -> IFT -> Write
  - Image size: 64 MB
  - Total time for one image on desktop PC: ~3.5 sec
- Run on multiple cores ?
- Run on a cluster ?

Slightly Trickier

- Iterative Image Alignment - You have a set of 1000, 256x256 images:
  - average all images together
  - align each image to the average
  - repeat 10x
- How to handle communications ?

Amdahls Law

- Speedup achievable with many processors is limited by the non-parallel portions of the task:
- S=1/(B+(1-B)/n)
- B=fraction of the code which cannot run in parallel
- n=number of processors


Amdahl's Law
Speedup
20.00
18.00
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of Processors
Parallel Portion
50%
75%
90%
95%

Slightly Trickier

- Iterative Image Alignment - You have a set of 1000, 256x256 images:
  - average all images together
  - align each image to the average
  - repeat 10x
- How to handle communications ?

Slightly Trickier

- average all images together
  - All images on 1 node ? (serial !)
  - How else to handle ?
- align each image to the average
  - Each node needs:
    - the reference
    - 1 or more images to align

Coarse vs Fine

- Coarse-grained parallelism
  - Tasks are completely independent (may have shared input data)
  - Example: filter 1000 images
- Fine-grained parallelism
  - Tasks need to communicate between each other continuously
  - Example: Matrix inversion

Example

- You have 200 sequences and wish to run a multiple sequence alignment against a set of 20 shorter reference sequences. How to parallelize ?

Coarse Grained?

- Each of the 200 sequences to one processor, which computes all of the 20 alignments for that sequence
- Advantages:
  - Very coarse, easy to distribute
  - Potentially 'perfectly' parallel
- Disadvantages:
  - Only works if you have at most 200 cores
  - If the 200 sequences vary significantly in length, total time will be limited by the longest sequence

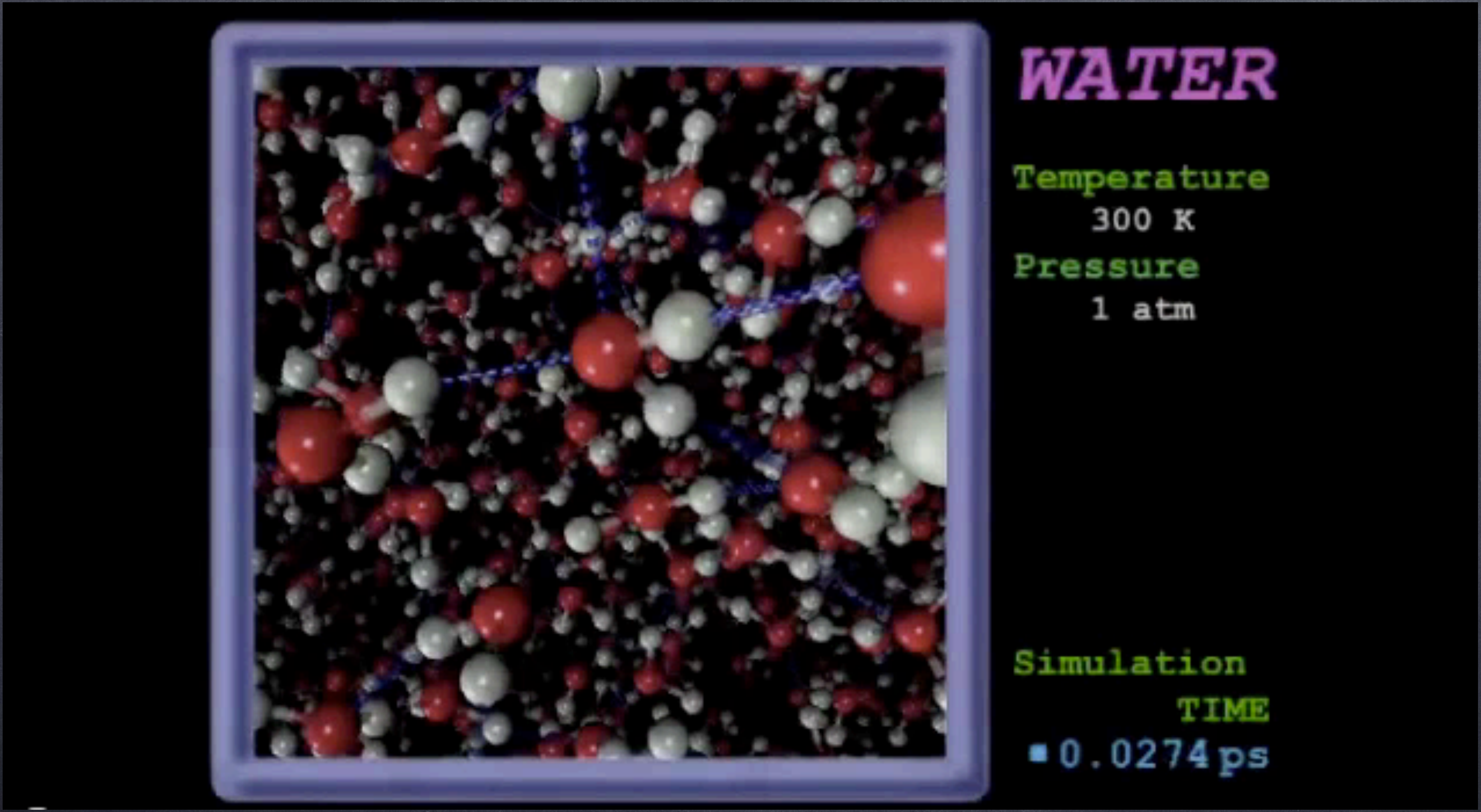
Fine Grained

- Tackle 1 sequence and 1 reference at a time. Each processor helps compute the local score
- Advantages:
  - Fine grained - more uniformly scalable
- Disadvantages
  - May be VERY inefficient due to communications bottlenecks

Intermediate Approach

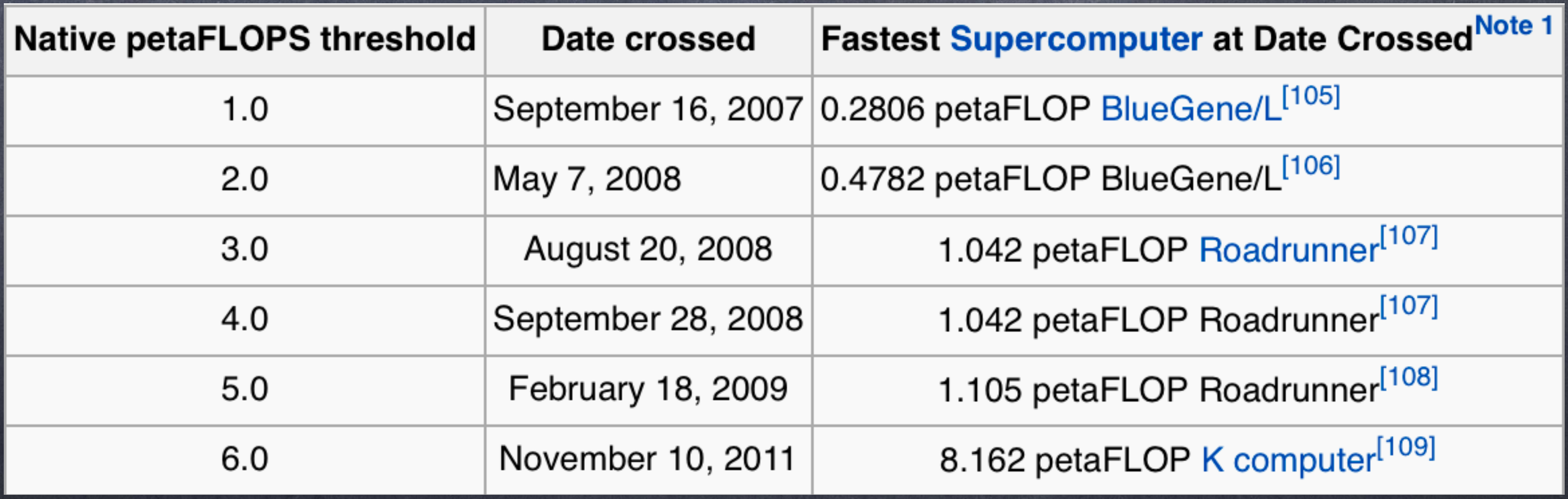
- Split the overall process into 200*20 = 4000 individual alignment tasks, and send one to each core as it becomes available
- Advantages
  - Each task independent, so still 'perfectly' parallel
  - Parallelizable up to 4000 cores
- Disadvantages
  - May still have some inefficiencies with differing sequence lengths, particularly for large number of processors

MD Simulations



MD Simulations

- VERY high CPU/Disk ratio
- Long-time single simulation
- Many short-time simulations (folding@home)

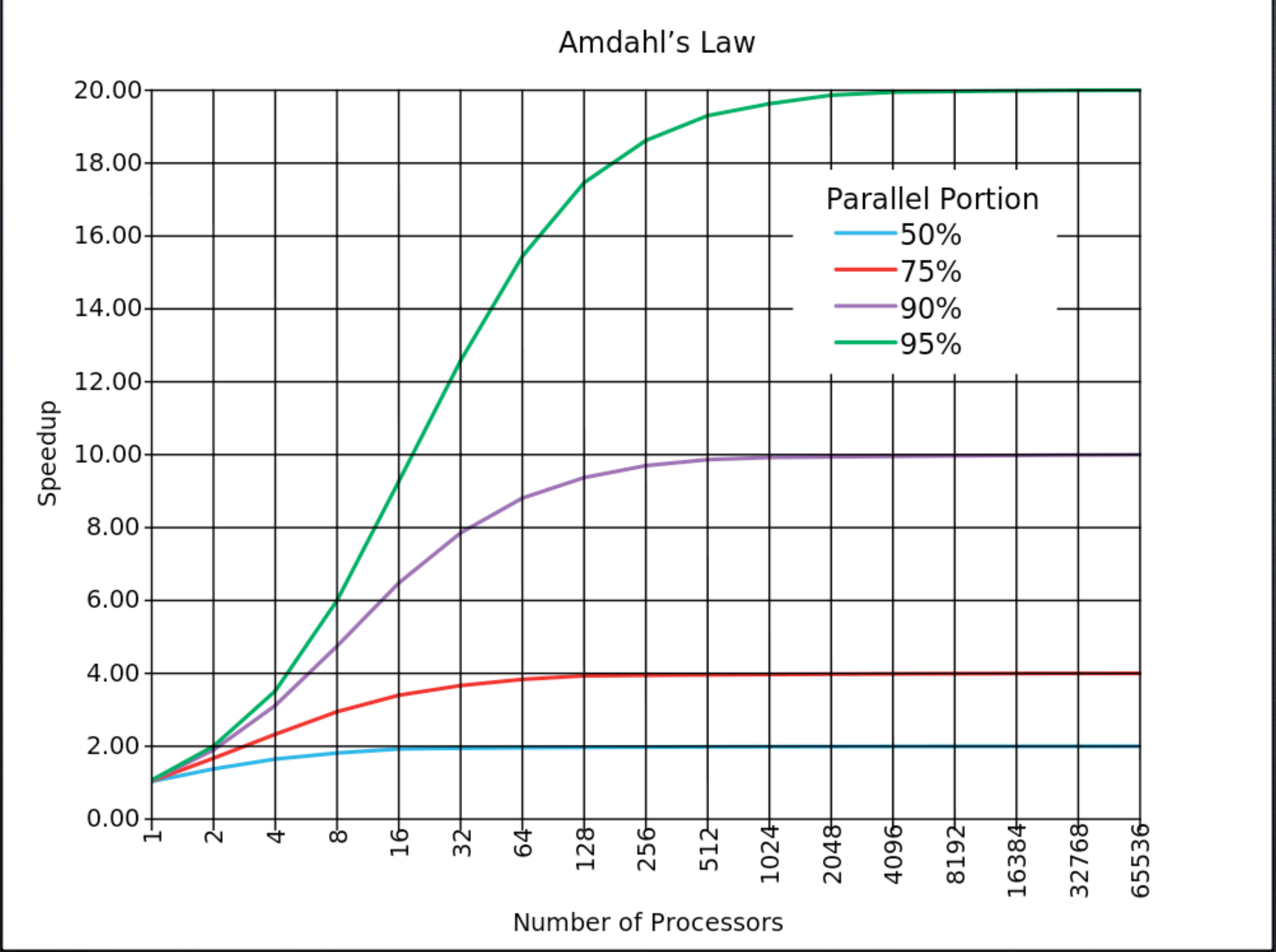


Native petaFLOPS threshold	Date crossed	Fastest Supercomputer at Date Crossed[Note 1]
1.0	September 16, 2007	0.2806 petaFLOP BlueGene/L[105]
2.0	May 7, 2008	0.4782 petaFLOP BlueGene/L[106]
3.0	August 20, 2008	1.042 petaFLOP Roadrunner[107]
4.0	September 28, 2008	1.042 petaFLOP Roadrunner[107]
5.0	February 18, 2009	1.105 petaFLOP Roadrunner[108]
6.0	November 10, 2011	8.162 petaFLOP K computer[109]

All of our clusters combined total ~0.1 Petaflops

Questions to Ask Yourself

- Total time required for I/O
- Possible to share data?
- Total time required for processing
- Memory usage
- Interprocess communication


Amdahl's Law
Speedup
20.00
18.00
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of Processors
Parallel Portion
50%
75%
90%
95%

Disc Tricks

(for data intensive jobs)

- Run on 'storage node'
- Purpose-specific node/workstation
- Clone data via broadcasting
- Copy data to scratch storage on appropriate nodes
- Lustre filesystem
- Process while copying

Part 3
Using Clusters

How to

Interrogate Your Cluster !

Cluster Resources

- RTFM (http://blake.bcm.edu/CIBRClusters)
- cat /etc/hosts
- df -h
- mount
- ifconfig
- /proc filesystem (cpuinfo, meminfo)
- qstat -q
- Filesystem speed ?
  - dd if=/dev/zero of=test bs=1M count=2000; rm test

Subsystems

- BQS (Batch Queuing System)
  - PBS (OpenPBS, Torque, etc.)
  - SGE (Sun Grid Engine)
  - HTCondor (UW)
- Parallelized programs
  - pthreads
  - OpenMP
  - MPI

BQS

- OpenPBS/Torque
  - Edit batch script
  - Submit job (qsub)
    - to a specific queue (qstat -q)
  - Job waits in queue (qstat -a)
  - Nodes allocated ($PBS_NODEFILE)
  - Script run on the first node ($PBS_O_WORKDIR)
  - Cleanup/logfiles
    - Kill a bad job (qdel)
  - Accounting updated (resources used)

Batch Script

```
#!/bin/bash
#
# This is an example PBS/Torque script
# modify the number of nodes, ppn (processors per node), and walltime
#

#PBS -l nodes=2:ppn=12
#PBS -l walltime=2:00:00

cd $PBS_O_WORKDIR

YOUR COMMANDS HERE
```

```
qsub -q <queuename> myscript.pbs
```

:q

- cput - Maximum amount of CPU time used by all processes in the job. Units: time.
- file - The largest size of any single file that may be created by the job. Units: size.
- nodes - Number of nodes to allocate
- pcput - Maximum amount of CPU time used by any single process in the job. Units: time.
- pmem - Maximum amount of physical memory (workingset) used by any single process of the job. Units: size.
- ppn - Number of processors to use per node
- pvmem - Maximum amount of virtual memory used by any single process in the job. Units: size.
- walltime - Maximum amount of real time during which the job can be in the running state. Units: time.

Part 4
Parallel Programming

Parallel programming

- pthreads
- OpenMP
- MPI
- Other niche systems...

pthreads

- Only one node at a time (SMP)
- SMP -> easy communications
- Somewhat painful to program
  - Synchronization issues
  - May be limits in some languages
- Available in multiple programming languages

pthreads

Python example:

```
from threading import Thread
import time,sys

def func(n):
        for i in range(10):
                time.sleep(1)
                print n,i
                sys.stdout.flush()

threads=[Thread(target=func,args=[i]) for i in xrange(4)]

for t in threads:
        t.start()
        time.sleep(0.1)
```

OpenMP

- Very good speedups with limited effort
- Same code can compile parallel and serial
- One node only (SMP)
- Needs to be part of the compiler (available in gcc)

- http://www.openmp.org

OpenMP Example

```
#include <stdio.h>
#include <math.h>
#include <omp.h>

int main() {

int i,j;
double sum=0;

for (i=0; i<100000000; i++) {
        sum+=pow(1.00001,i/1000);
}

printf("%lf\n",sum);
}
```

OpenMP Example

```
#include <stdio.h>
#include <math.h>
#include <omp.h>

int main() {

int i,j;
double sum=0;

#pragma omp parallel
{
#pragma omp for
for (i=0; i<100000000; i++) {
        sum+=pow(1.00001,i/1000);
}
}
printf("%lf\n",sum);
}
```

OpenMP Example

```
#include <stdio.h>
#include <math.h>
#include <omp.h>

int main() {

int i,j;
double sum=0;

#pragma omp parallel
{
#pragma omp for reduction(+:sum)
for (i=0; i<100000000; i++) {
        sum+=pow(1.00001,i/1000);
}
}
printf("%lf\n",sum);
}
```

MPI

- MPI: Message Passing Interface
- Written by computer scientists for computer scientists
- Operates on distributed processors
- Bindings for many languages
- Explicit interprocess communication via messages
- All nodes run the same program
- Communications problems are common
- Zero fault tolerance

MPI

- Many variants - OpenMPI, MPICH, Intel MPI,...
- mpicc - MPI aware C compiler
- which mpicc - identify which MPI installation
- mpirun - convenient program launching tool
  - Runs the exact same program on each processor!
- On most clusters, automatically talks to BQS

MPI

- Outline of one strategy for MPI program:
- MPI_Init() - Initialize MPI on all nodes
- MPI_Barrier() - Synchronize nodes
- MPI_Comm_rank() - Identify CPU (rank)
- rank 0:
  - coordinate processing, perhaps do some
- rank 1-n:
  - perform work assigned by rank 0
- MPI_finalize() - clean everything up

MPI

```
#include <mpi.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv)
{
  int rank;
  char hostname[256];

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  gethostname(hostname,255);

  printf("Hello world!  I am process number: %d on host %s\n", rank, hostname);

  MPI_Finalize();

  return 0;
}
```

https://hpcc.usc.edu/support/documentation/examples-of-mpi-programs/

MPI - python

```
#!/usr/bin/env python
from sys import argv,stdout
from mpi import *

mpi_init(0,[])
mpi_barrier(MPI_COMM_WORLD)
proc=mpi_comm_rank(MPI_COMM_WORLD)
nproc=mpi_comm_size(MPI_COMM_WORLD)
print "Running on %d/%d"%(proc,nproc)

if proc==0 :

   print "Stage 1, synchronous send/receive"
   print "Rank ",
   for i in range(1,nproc):
      mpi_send("TESTING",7,MPI_CHAR,i,1,MPI_COMM_WORLD)
      print i,
      stdout.flush()
   print "\nTransmit complete"

else :
   data=mpi_recv(7,MPI_CHAR, 0,1,MPI_COMM_WORLD)
   print proc," received ",data

mpi_barrier(MPI_COMM_WORLD)
mpi_finalize()
```

Other systems?

- Many other language dependent systems
- May not be broadly supported on 'big iron' clusters
- Sysops may be hostile to use of anything but MPI

Where to Learn More

- Passing interest
  - Youtube has many good videos
- Somewhat interested
  - TACC offers multi-day workshops on parallelism
- Really Committed
  - (if you are a GS) Rice offers Comp 422, a full semester course on parallel computing
  - iTunesU - full courses (eg - Stanford)