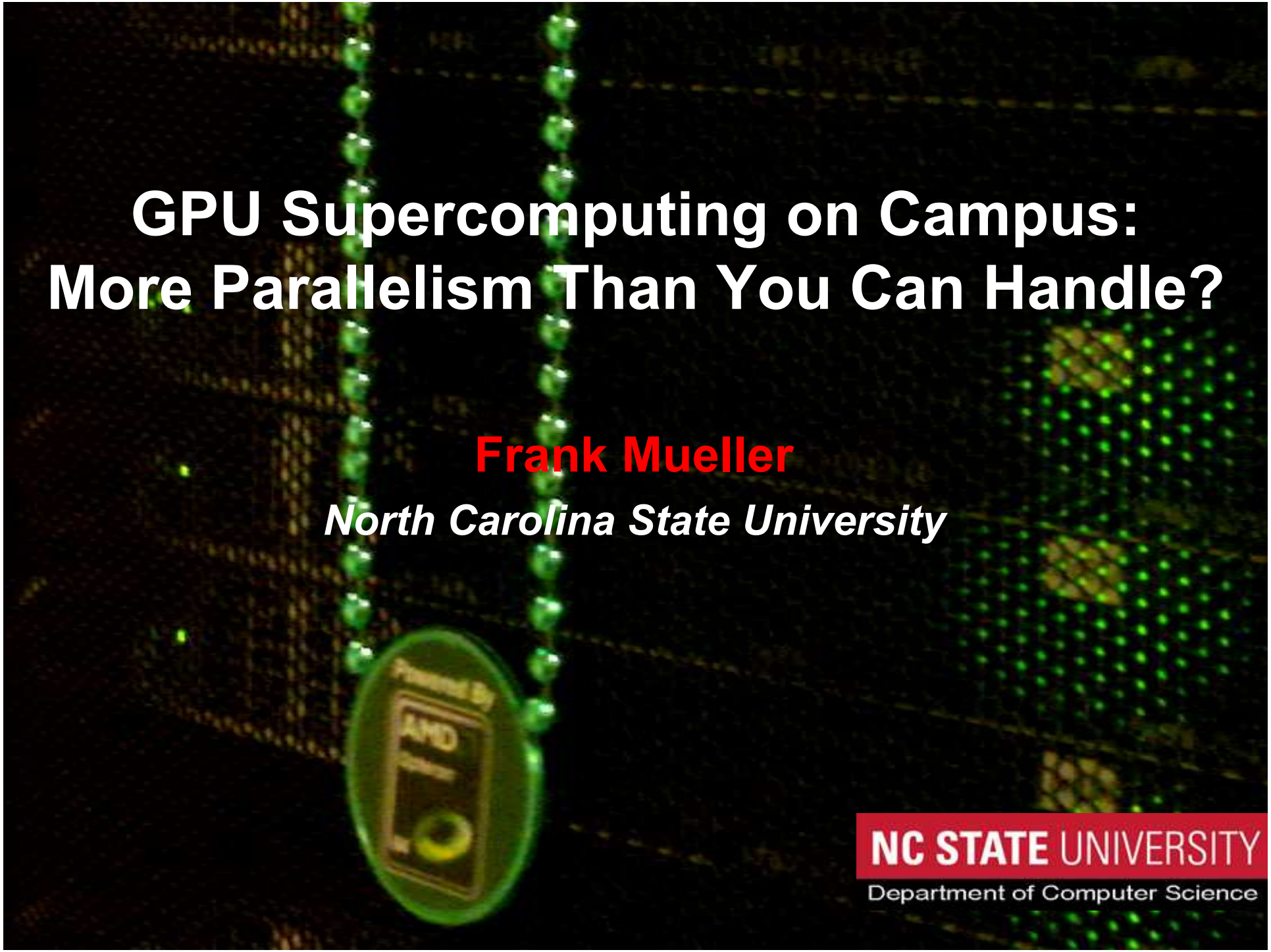


GPU Supercomputing on Campus: More Parallelism Than You Can Handle?

Frank Mueller

North Carolina State University

NC STATE UNIVERSITY

Department of Computer Science

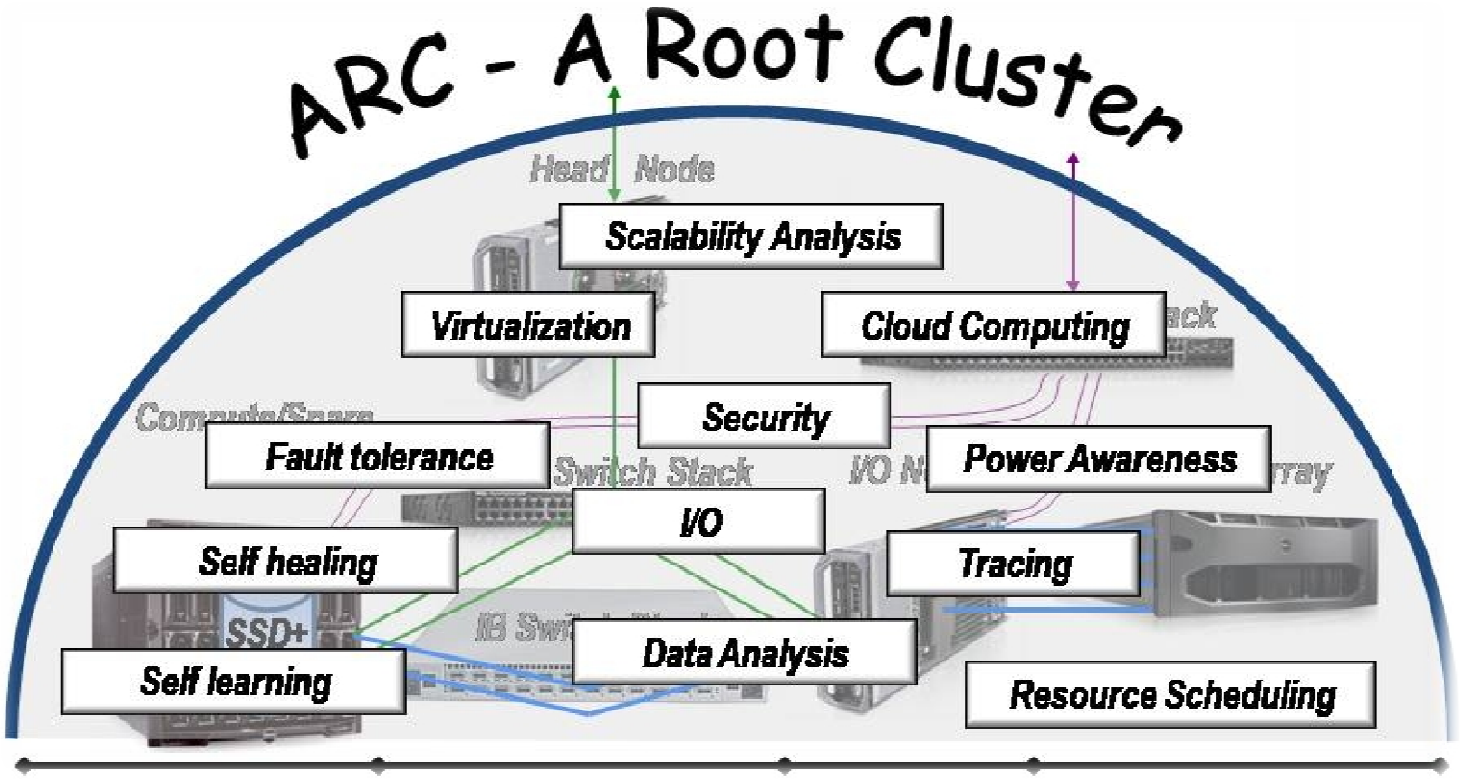
What's Supercomputing?

- Scientific computing on the biggest computers ever
  - Top500.org → Linpack benchmark: "Indie 500 of computers"
  - Up to 100 Petaflops today: 100×10^{15} floating point ops/second!
- How? → Lots of parallel computation
- Parallelism in Computers
  - Inside CPU: pipline, instruction parallelism, out-of-order exec., speculation
  - Multicores: up to 72 processing cores on a chip, or 3,500+ GPU cores
- Cluster: connect 10-200 such computers together
- Supercomputer: really large cluster w/ 1k-1M computers
  - Plus a custom network: 100X faster than a 1 Gbps connect.

➢ This talk: How to utilize a 100 node GPU cluster, called "ARC"

Who Needs It?

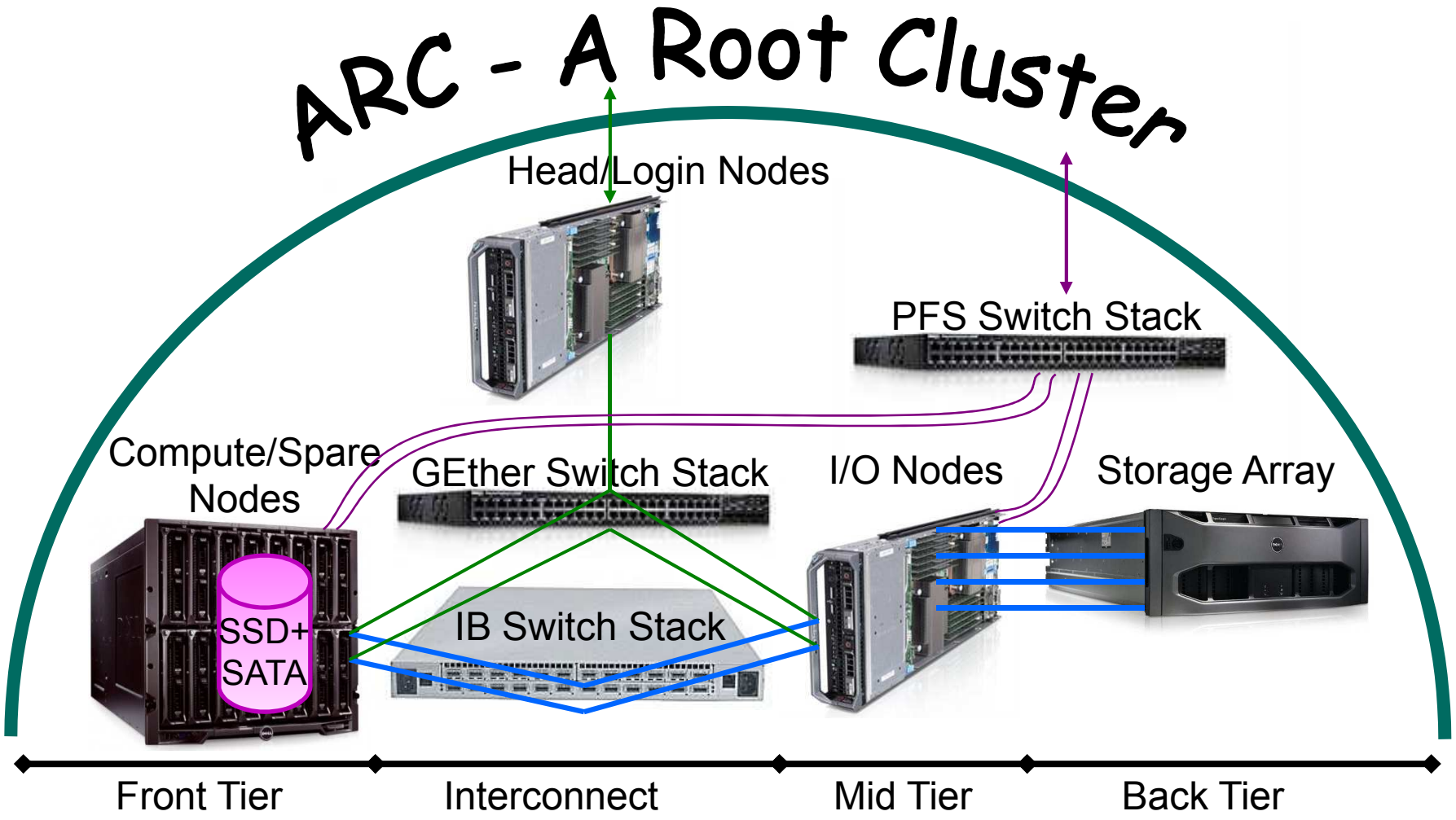
- Scientific computing
- Big data



ARC Cluster: In the News

- "NC State is Home to the Most Powerful Academic HPC in North Carolina" (CSC News, Feb 2011)
- "Crash-Test Dummy For High-Performance Computing" (NCSU, The Abstract, Apr 2011)
- "Supercomputer Stunt Double" (insideHPC, Apr 2011)

System Overview



Hardware

- 108 Compute Nodes
  - 2-way SMPs with AMD Opteron/ Intel Sandy/Ivy/Broadwell procs
  - 8 cores/socket, 16 cores/node
  - 16-64 GB DRAM per node
- **1728 compute cores available**

Example: Approximation of Pi

Integration to evaluate π

Computer approximations to π by using numerical integration
Know

$$tan(45^0) = 1;$$

same as

$$tan\frac{\pi}{4} = 1;$$

So that;

$$4 * tan^{-1}1 = \pi$$

From the integral tables we can find

$$tan^{-1}x = \int \frac{1}{1+x^2}dx$$

or

$$tan^{-1}1 = \int_0^1 \frac{1}{1+x^2}dx$$

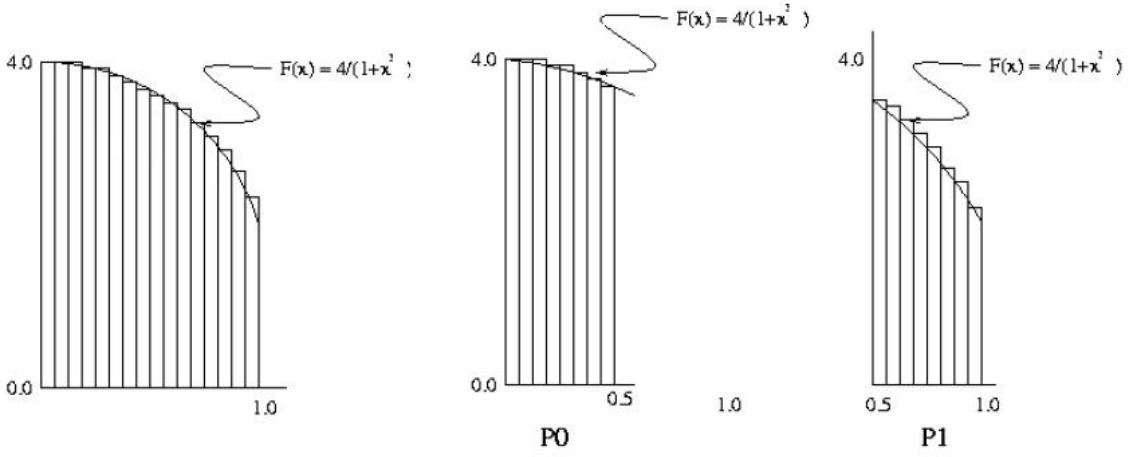
Using the mid-point rule with panels of
uniform length $h = 1/n$, for various values of n.
Evaluate the function at the midpoints of
each subinterval (x_{i-1}, x_i) $i * h - h/2$ is the midpoint.
Formula for the integral is

$$x = \sum_{i=1}^{n} f(h * (i - 1/2))$$

$$\pi = h * x$$

where

$$f(x) = \frac{4}{1+x^2}$$



- **Let's write some code!**
- http://moss.csc.ncsu.edu/~mueller/mpigpu/

PI in C (stylized)

```
#include <stdio.h>
#include <math.h>
#include "mytime.h"

double integrate(int n) {
  double sum, h, x;
  int i;
  h = 1.0 / (double) n;
  for (i = 1; i <= n; i++) {
    x = h*((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
  }
  return sum * h;
}
```

```
int main(int argc, char *argv[]) {
  int n;
  double PI25DT = 3.14159…;
  double pi;

  while (1) {
    printf("Enter # intervals: ");
    scanf("%d",&n);
    if (n == 0)
      break;
    pi = integrate(n);

   printf("pi=%.16f, error=%.16f\n",
           pi, fabs(pi - PI25DT));
}
```

```
make pi-c
pi-c
Enter the number of intervals: (0 quits) 1000
pi is approximately 3.1415926535897643, E
wall clock time = 0.038800
Enter the number of intervals: (0 quits) 0
```

- Let's parallelize it!

PI w/ OpenMP in C (stylized)

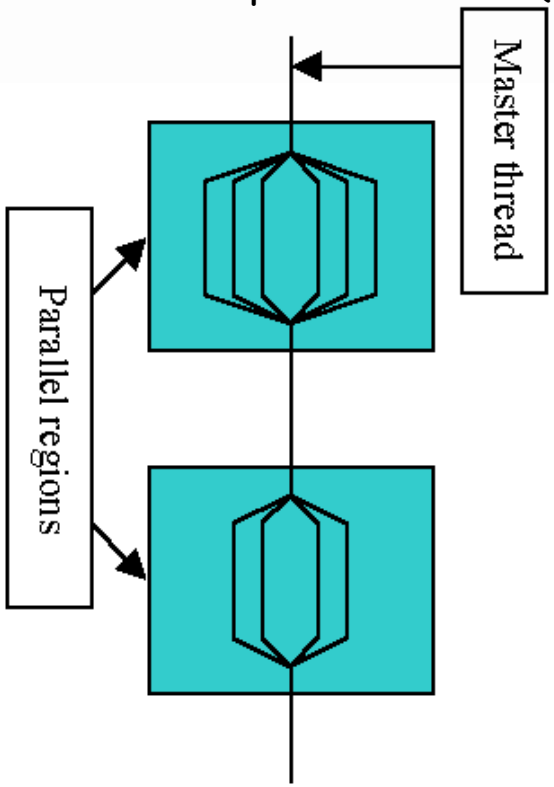
```
#include <stdio.h>
#include <math.h>
#include "mytime.h"

double integrate(int n) {
  double sum, h, x;
  int i;
  h = 1.0 / (double) n;
#pragma omp parallel for
   reduction(+:sum) private(x)
  for (i = 1; i <= n; i++) {
    x = h*((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
  }
  return sum * h;
}
```

```
int main(int argc, char *argv[]) {
  int n;
  double PI25DT = 3.14159…;
  double pi;

  while (1) {
    printf("Enter # intervals: ");
    scanf("%d",&n);
    if (n == 0)
      break;
    pi = integrate(n);

   printf("pi=%.16f, error=%.16f\n",
           pi, fabs(pi - PI25DT));
  }
}
```

PI w/ OpenMP in C (stylized)

```
#include <stdio.h>
#include <math.h>
#include "mytime.h"

double integrate(int n) {
  double sum, h, x;
  int i;
  h = 1.0 / (double) n;
#pragma omp parallel for
   reduction(+:sum) private(x)
  for (i = 1; i <= n; i++) {
    x = h*((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
  }
  return sum * h;
}
```

- What did we just do? OpenMP!
- pragma → create threads
  - Run loop in parallel over all cores
  - Split iterations (evenly) over cores



PI w/ OpenMP in C (stylized)

```
#include <stdio.h>
#include <math.h>
#include "mytime.h"

double integrate(int n) {
  double sum, h, x;
  int i;
  h = 1.0 / (double) n;
#pragma omp parallel for
   reduction(+:sum) private(x)
  for (i = 1; i <= n; i++) {
    x = h*((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
  }
  return sum * h;
}
```

- What did we just do? OpenMP!
- pragma → create threads
  - Run loop in parallel over all cores
  - Split iterations (evenly) over cores
  - But: NO dependency b/w statements
    - Ex.: a[i] = a[i-1]+b[i]; /* dependence!!! */
  - Who checks? Burden is on YOU!
    - Ex.: x = h*… /* every thread writes to x!!! */

PI w/ OpenMP in C (stylized)

```
#include <stdio.h>
#include <math.h>
#include "mytime.h"

double integrate(int n) {
  double sum, h, x;
  int i;
  h = 1.0 / (double) n;
#pragma omp parallel for
   reduction(+:sum) private(x)
  for (i = 1; i <= n; i++) {
    x = h*((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
  }
  return sum * h;
}
```

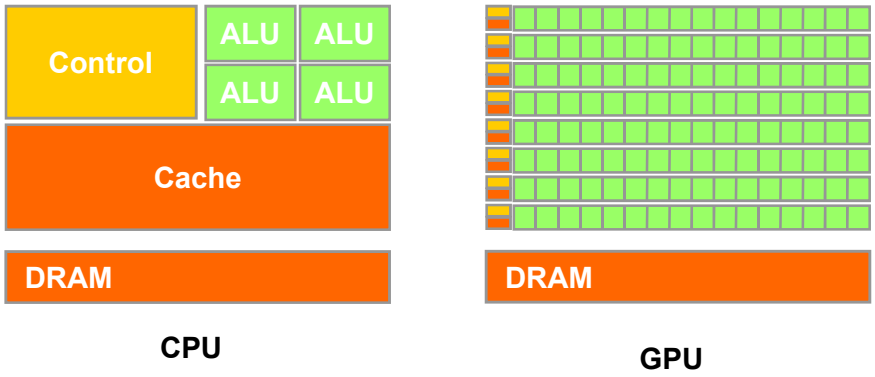
- What did we just do? OpenMP!
- pragma → create threads
  - Run loop in parallel over all cores
  - Split iterations (evenly) over cores
  - But: NO dependency b/w statements
    - Ex.: a[i] = a[i-1]+b[i]; /* dependence!!! */
  - Who checks? Burden is on YOU!
    - Ex.: x = h*… /* every thread writes to x!!! */
  - to avoid overwrites, each thread needs a copy of x → `private(x)`
    - Ex.: sum = sum + … /* dependence??? */
  - Same again, but need to compute aggregate → `reduction(+:sum)`
- More? openmp.org / take a class

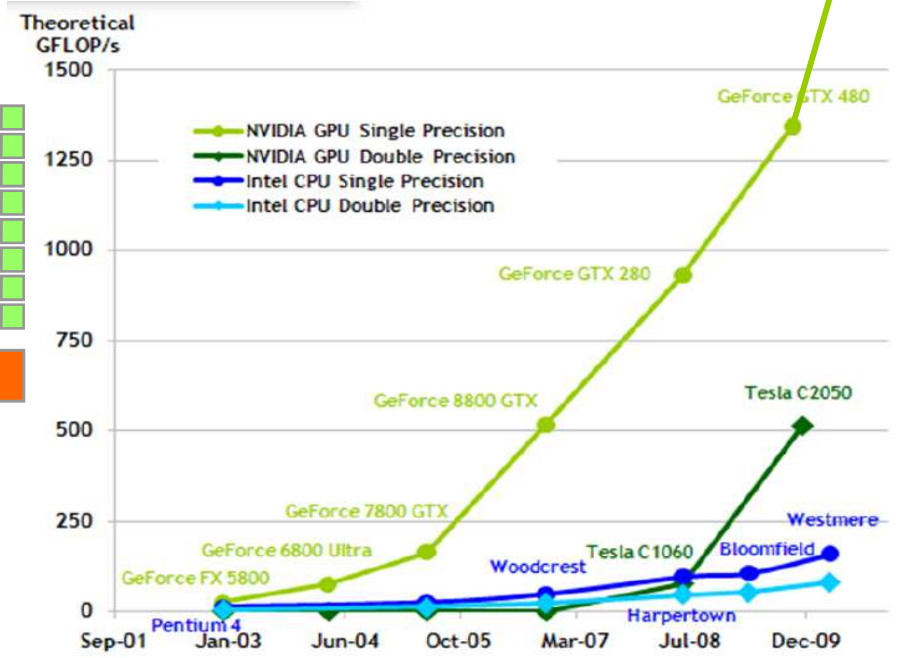
➢ 4-16 Threads? I want more!

NVIDIA GPUs in the Cluster

- 10s of thousands of threads!
- Programmed via
  - CUDA or OpenACC pragmas
- Good for lots of calculations
  Not good for braches: if-then-else

- C/M2070
- C2050
- GTX480
- K20c
- GTX780
- GTX680
- K40c
- GTX Titan X
- GTX 1080
- Titan X





PI for PGI w/ OpenACC in C (stylized)

```
#include <stdio.h>
#include <math.h>
#include "mytime.h"

double integrate(int n) {
  double sum, h, x;
  int i;
  h = 1.0 / (double) n;
#pragma acc parallel
   reduction(+,sum) private(x)
  for (i = 1; i <= n; i++) {
    x = h*((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
  }
  return sum * h;
}
```

```
int main(int argc, char *argv[]) {
  int n;
  double PI25DT = 3.14159…;
  double pi;

  while (1) {
    printf("Enter # intervals: ");
    scanf("%d",&n);
    if (n == 0)
      break;
    pi = integrate(n);

   printf("pi=%.16f, error=%.16f\n",
           pi, fabs(pi - PI25DT));
```

```
make pi-pgicu-c
pi-pgicu-c
Enter the number of intervals: (0 qu
pi is approximately 3.14159265358
wall clock time = 0.038800
Enter the number of intervals: (0 quits) 0
```

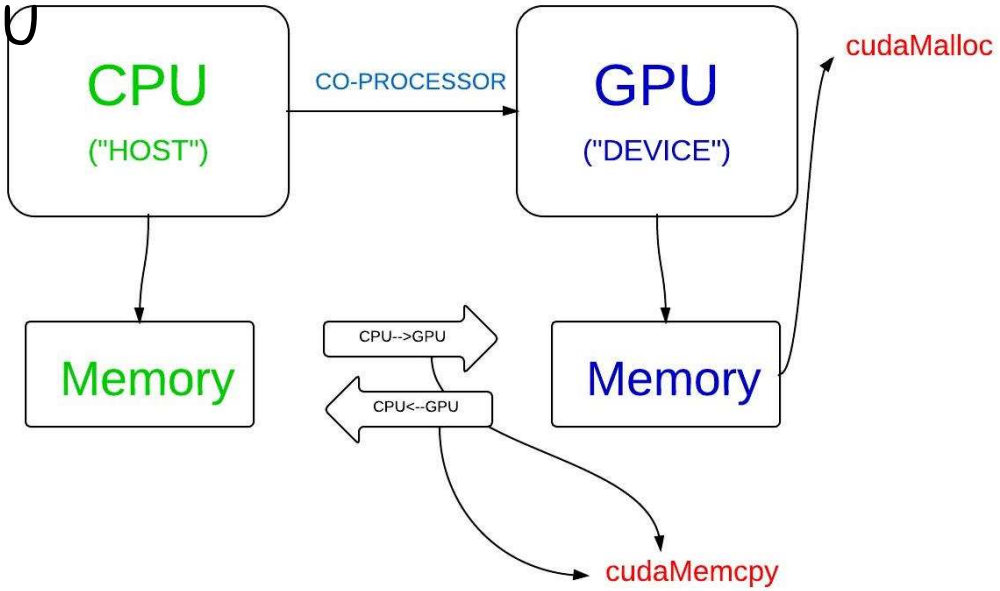
- ~4k Threads (depends on GPU).

Done? No!

NVIDIA CUDA

- 10k threads + full control → CUDA (Compute Unified Device Archit.)
- CPU controls execution
- GPU is fast co-processor

1. allocate memory: input/output
2. copy inputs: CPU → GPU
3. Invoke kernel → 4. Run compute kernel
5. Copy outputs GPU→ CPU
6. Print output



- But: High copy cost
  - Need large data to offset copy cost

PI for CUDA in C: The CPU sSide

```
int main(int argc, char *argv[]) {
  int n;
  int *n_d; // device copy of n8
  …
  double *pi_d; // device copy of pi
  struct timeval startwtime, …;
  // Allocate memory on GPU
  cudaMalloc( (void **) &n_d, sizeof(int) * 1 );
  cudaMalloc( (void **) &pi_d, sizeof(double) * 1 );

  while (1) {
  …
    if (n == 0)
      break;
    // copy from CPU to GPU
    cudaMemcpy( n_d, &n, sizeof(int) * 1, cudaMemcpyHostToDevice);
    integrate<<< 1, 1 >>>(n_d, pi_d);
    // copy back from GPU to CPU
    cudaMemcpy( &pi, pi_d, sizeof(double) * 1, cudaMemcpyDeviceToHost);
    …
  }
  // free GPU memory
  cudaFree(n_d);
  cudaFree(pi_d);
}
```

- Need to double up on data
  - CPU array + GPU shadow array
- copy inputs: CPU → GPU
- Invoke kernel
- Copy outputs GPU→ CPU
- Print output (not shown)

PI for CUDA in C: The GPU Side

```
#include <stdio.h>
#include <math.h>
#include "mytime.h"

// GPU kernel
__global__ void integrate(int
   *n, double *sum) {
  double h, x;
  int i;

  *sum = 0.0;
  h = 1.0 / (double) *n;
  for (i = 1; i <= *n; i++) {
    x = h * ((double)i - 0.5);
    *sum += 4.0 / (1.0 + x*x);
  }
  *sum *= h;
}
```

GPU Kernel:

- Specifies what each thread does

- Are we there yet? No!
  → only 1 Thread

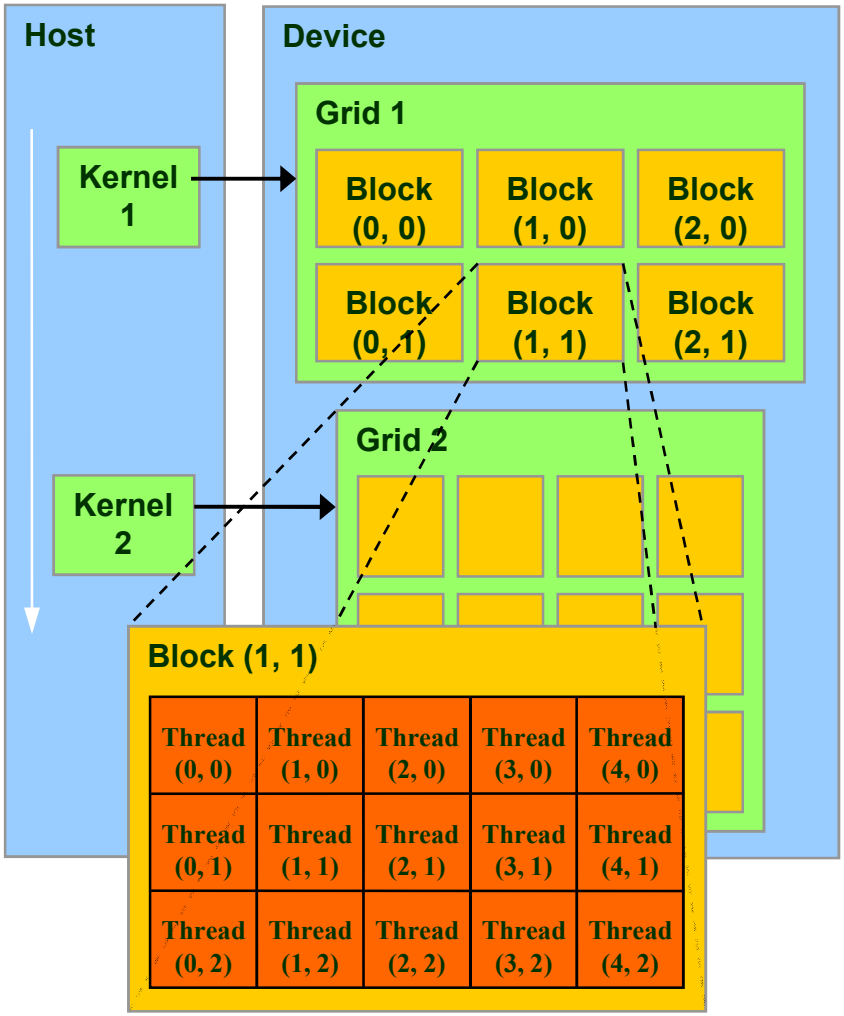
```
cp pi-c.c pi-cu.cu
vim pi-cu.cu
make pi-cu
pi-cu
Enter the number of intervals: (0 quits) 1000000
…
```

Thread Batching: Grids and Blocks

- GPU Kernel: grid of blocks
- block is a batch of threads
- threads and blocks have IDs
  - thread.Idx.x: thread ID in a block
  - blockDim.x: # threads in a block
  - block.Idx: # blocks in a grid
- Challenge: each thread has different data to work on

```
sum[threadIdx.x] += 4.0/…
```

1. CPU: pass array of sum elements → GPU
2. GPU: each thread updates its sum[…]
3. CPU: get updated sum array from← GPU
4. CPU: Add up partial sums



Courtesy: NDVIA

PI for CUDA with 512 Threads (1 Block)

1. CPU: pass array of sum elements → GPU
2. GPU: each thread updates its sum[…]
3. CPU: get updated sum array from← GPU
4. CPU: add up partial sums

```
int main(int argc, char *argv[]) {
  double mypi[THREADS];
  double *mypi_d; // device copy
   of pi
  …
  cudaMalloc( (void **) &mypi_d,
   sizeof(double) * THREADS );
    …
    integrate<<<1, THREADS>>>
      (n_d, mypi_d);
    …
    cudaMemcpy(&mypi, mypi_d,
     sizeof(double) * THREADS,
     cudaMemcpyDeviceToHost);
    pi = 0.0;
    for (i = 0; i < THREADS; i++)
      pi += mypi[i];
    …
  }
  …
  cudaFree(mypi_d);
}
```

PI for CUDA with 512 Threads (1 Block)

```
…
#include "mytime.h"
#define THREADS 512
// GPU kernel
…
  sum[threadIdx.x] = 0.0;
  for (i=threadIdx.x+1; i<=*n;
       i+=THREADS) {
    x = h * ((double)i - 0.5);
    sum[threadIDx.x]+=4.0/(1.0+x*x);
  }
  sum[threadIDx.x] *= h;
}
```

- threads and blocks have IDs
  - thread.Idx.x: thread ID in a block
  - blockDim.x: # threads in a block
  - block.Idx: # blocks in a grid
- Challenge: each thread has different data to work on

  `sum[threadIdx.x] += 4.0/…`

PI for CUDA with 512 Threads (1 Block)

```
…
#include "mytime.h"
#define THREADS 512
// GPU kernel
…
  sum[threadIdx.x] = 0.0;
  for (i=threadIdx.x+1; i<=*n;
       i+=THREADS) {
    x = h * ((double)i - 0.5);
    sum[threadIDx.x]+=4.0/(1.0+x*x);
  }
  sum[threadIDx.x] *= h;
}
```

```
int main(int argc, char *argv[]) {
  double mypi[THREADS];
  double *mypi_d; // device copy
   of pi
  …
  cudaMalloc( (void **) &mypi_d,
   sizeof(double) * THREADS );
    …
    integrate<<<1, THREADS>>>
      (n_d, mypi_d);
  …
  cudaMemcpy(&mypi, mypi_d,
   sizeof(double) * THREADS,
   cudaMemcpyDeviceToHost);
  pi = 0.0;
  for (i = 0; i < THREADS; i++)
    pi += mypi[i];
}
```

```
cp pi-cu.cu pi-cu-block.cu
vim pi-cu-block.cu
make pi-cu-block
pi-cu-block
Enter the number of intervals…
```

- Are we there yet? No!
  → only 512 Threads… but 1st

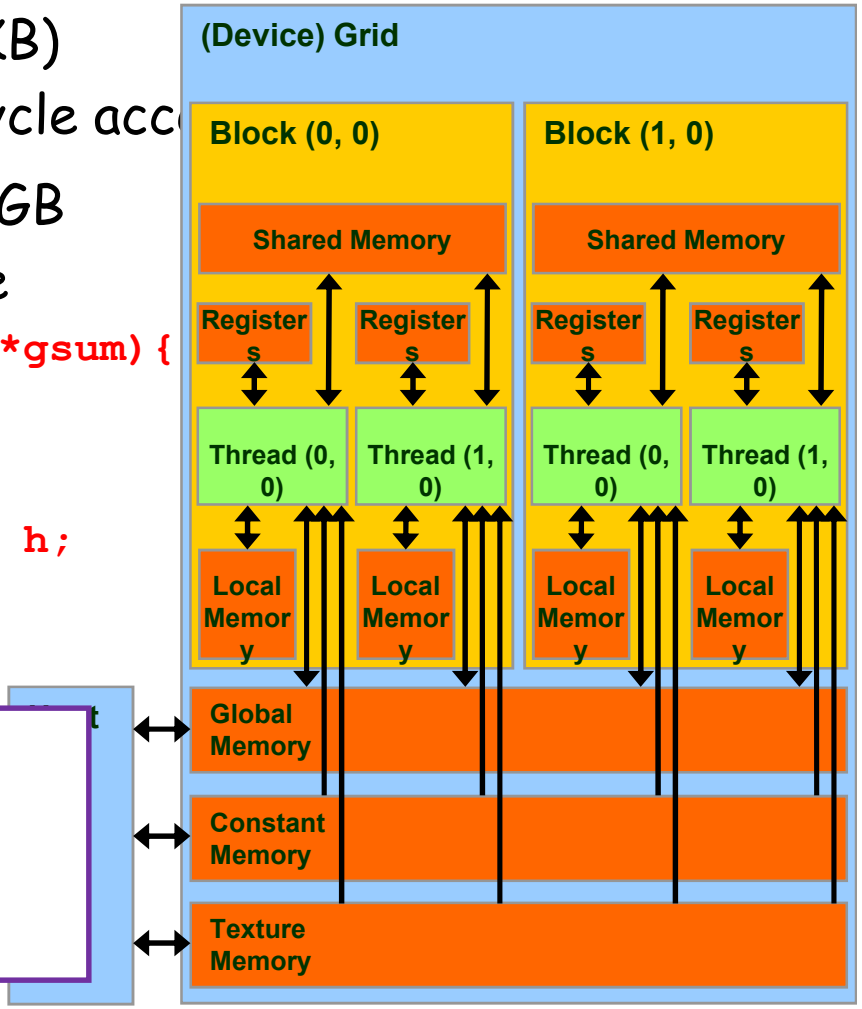
PI for CUDA with Shared (Fast) Memory

- Shared memory
  - Small (a few KB)
  - Fast: single cycle acc
- Global memory: ~1GB
  - Slow but large

```
…
#include "mytime.h"
#define THREADS 512
// GPU kernel
__global__ void integrate(int *n,double *gsum){
  __shared__ double sum[THREADS];
  sum[threadIdx.x] = 0.0;
…
  gsum[threadIdx.x] = sum[threadIdx.x] * h;
}
```

```
cp pi-cu-block.cu pi-cu-shared.cu
vim pi-
make p
pi-cu-s
Enter th
```

- And next: 32k Threads!



PI for CUDA with Grid (32k Threads)

```
#define THREADS 512
#define MAX_BLOCKS 64
// GPU kernel, we know: THREADS == blockDim.x
__global__ void integrate(int *n, int *blocks,
   double *gsum) {
  …
  for (i = blockIdx.x*blockDim.x + threadIdx.x +
   1; i <= *n; i += blockDim.x * *blocks) {
    …
  gsum[blockIdx.x*blockDim.x + threadIdx.x] =
   sum[threadIdx.x] * h;
}
```

- threads and blocks have IDs
  - thread.Idx.x: thread ID in a block
  - blockDim.x: # threads in a block
  - block.Idx: # blocks in a grid

- 64 blocks x 512 threads
- Each block has own shared mem!!!
  - Block 0: sum[0..511]
  - Block 1: sum[0..511]
  - etc.

PI for CUDA with Grid (32k Threads)

```
int main(int argc, char *argv[]) {
  int n, i, blocks;
  int *n_d, *blocks_d; // device copy
  …
  cudaMalloc( (void **) &blocks_d, sizeof(int) * 1 );
  cudaMalloc( (void **) &mypi_d, sizeof(double) * THREADS
                                              * MAX_BLOCKS );
  …
    cudaMemcpy( blocks_d, &blocks, sizeof(int) * 1,
                cudaMemcpyHostToDevice );

    integrate<<< blocks, THREADS >>>(n_d, blocks_d, mypi_d);

    // copy back from GPU to CPU
    cudaMemcpy( &mypi, mypi_d,
      sizeof(double) * THREADS * blocks,
      cudaMemcpyDeviceToHost );
    pi = 0.0;
    for (i = 0; i < THREADS * blocks
      pi += mypi[i];
  …
  cudaFree(blocks_d);
```

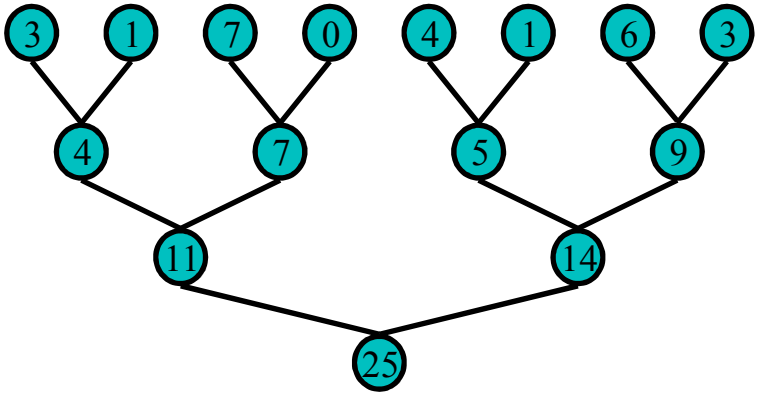
```
cp pi-cu-shared.cu pi-cu-grid.cu
vim pi-cu-grid.cu
make pi-cu-grid
pi-cu-grid
```

- We can do better yet…

PI for CUDA with Reduction

```
int main(int argc, char *argv[]) {
…
    integrate<<< blocks, THREADS >>>(n_d, blocks_d, mypi_d);
    if (blocks > 1)
       global_reduce<<< 1, blocks >>>(n_d, blocks_d, mypi_d);
     // copy back from GPU to CPU
     cudaMemcpy( &pi, mypi_d, sizeof(double) * 1,
   cudaMemcpyDeviceToHost );
```

- 2 kernels <<<…>>> needed !!!




```
make pi-cu-grid-reduce
pi-cu-grid-reduce
Enter the number of intervals…
```

PI for CUDA with Reduction

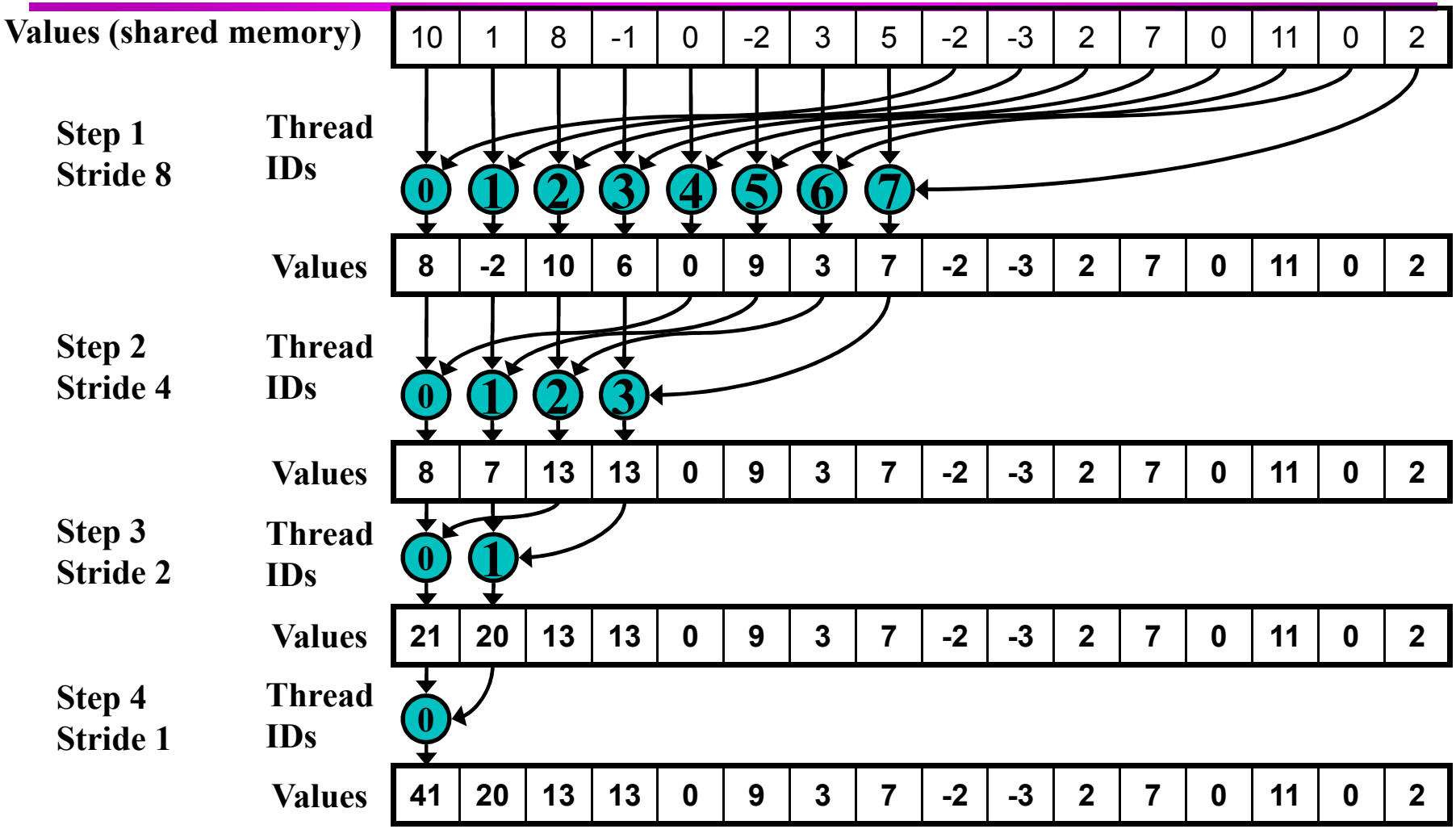
```
__global__ void integrate(int *n, int *blocks, double *gsum) {
  const unsigned int bid = blockDim.x * blockIdx.x + threadIdx.x;
  const unsigned int tid = threadIdx.x;
  …
  __shared__ double ssum[THREADS];
  double sum;
  sum = 0.0;
  h   = 1.0 / (double) *n;
  for (i = bid + 1; i <= *n; i += blockDim.x * *blocks) {
    x = h * ((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
  }
  ssum[tid] = sum * h;
  // block reduction
  __syncthreads();
  for (i = blockDim.x / 2; i > 0; i >>= 1) { /* per block */
    if (tid < i)
      ssum[tid] += ssum[tid + i];
    __syncthreads();
  }
  if (tid == 0)
    gsum[blockIdx.x] = ssum[tid];
}
```

Synchronize calculations b/w threads in a block → needed for dependence

- `ssum[tid] += ssum[tid+i];`
- `gsum[…] = ssum[tid];`

- OK, just believe me, it works!

Actual Reduction Indexing



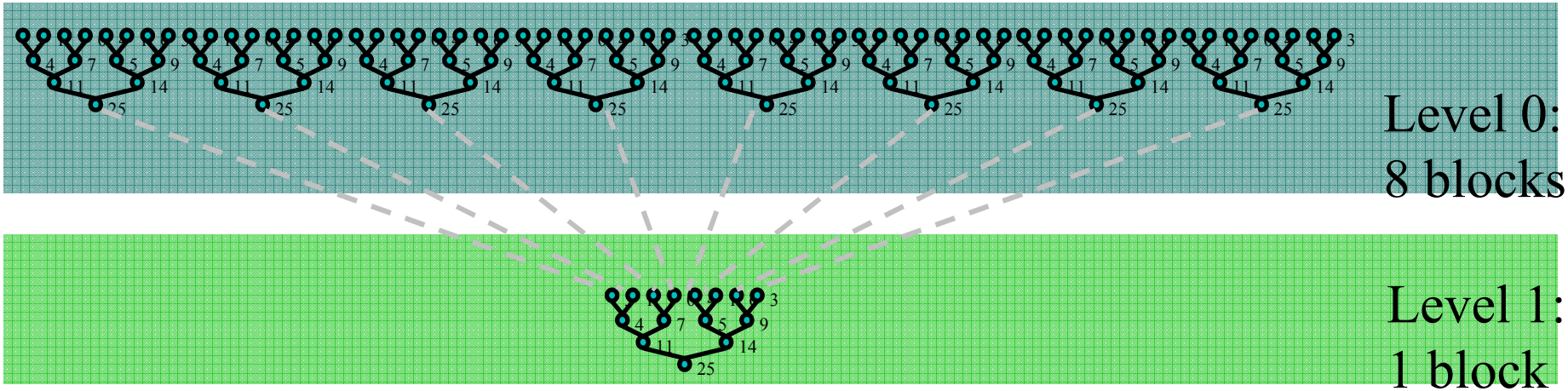
Sequential addressing is conflict free

PI for CUDA with Reduction

```
// number of threads must be a power of 2
__global__ static void global_reduce(int *n, int *blocks, double
   *gsum) {
     __shared__ double ssum[THREADS];
     const unsigned int tid = threadIdx.x;
     unsigned int i;

     ssum[tid] = gsum[tid];
     __syncthreads();
     for (i = blockDim.x / 2; i > 0; i >>= 1) { /* per block */
         if (tid < i)
            ssum[tid] += ssum[tid + i];
         __syncthreads();
     }
     if (tid == 0)
       gsum[tid] = ssum[tid];
}
```

Reduction across Block

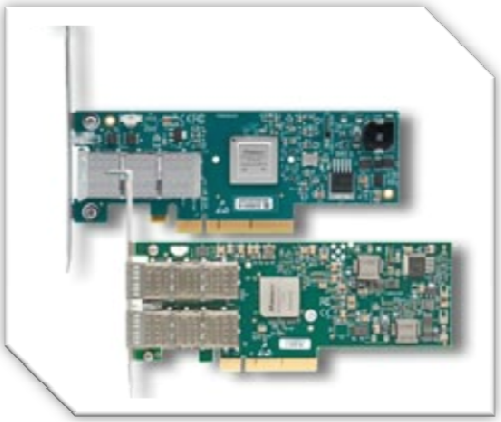
- Problem: no synchronization support across blocks
- Solution: decompose computation into multiple kernel invocations



- Could even use the same code for both invocations
  - Just different number of grid/block sizes

Interconnects

- Gigabit Ethernet
  - interactive jobs, ssh, service
  - Home directories
- 40Gbit/s Infiniband (OFEDstack)
  - MPI Communication
  - Open MPI, MVAPICH
  - IP over IB

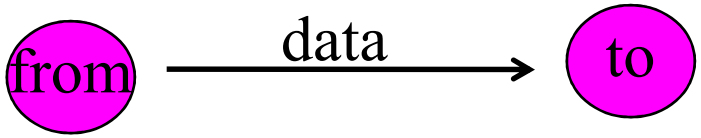


Message Passing Interface (MPI)

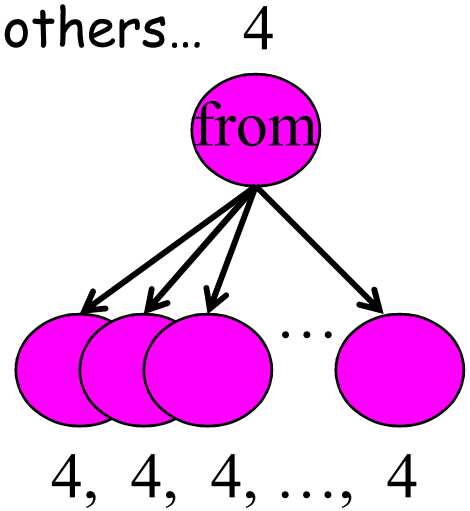
- De-facto standard to run program on 2, 10, 10k… computers
  - Same code
  - But each compute node operates on different data
- Same as OpenMP? → No!
  - OpenMP: shared memory
  - MPI: shared nothing, we need to communicate to xchange data
- MPI API + library → portable message passing
  - MPI_Send(data, to, …) → MPI_Recv(data, from, …)
  - Also collectives: all nodes participate [optionally some subset]
  - MPI_Bcast(data, from): send data to everyone else
  - MPI_Reduce(data, op): aggregate data via op, e.g., sum: "+"
  - others…

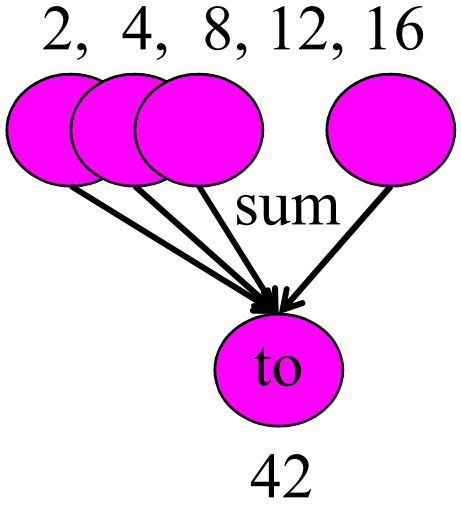
Message Passing Interface (MPI)

- MPI API + library → portable message passing
  - MPI_Send(data, to, …) → MPI_Recv(data, from, …)



  - Also collectives: all nodes participate [optionally some subset]
  - MPI_Bcast(data, from): send data to everyone else
  - MPI_Reduce(data, op): aggregate data via op, e.g., sum: "+"
  - others…





PI for MPI in C

```
#include <mpi.h>
…
double integrate(int n, int myid, int numprocs) {
  …
  for (i = myid + 1; i <= n; i += numprocs) {
  …
}

int main(int argc, char *argv[]) {
  int n, myid, numprocs;
  double PI25DT = 3.14159265358979323
  double mypi, pi;
  double startwtime, endwtime;
  int  namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  MPI_Get_processor_name(processor_name,&namelen);
  printf("MPI Task %2d on %20s\n", myid, processor_name);
  sleep(1); // wait for everyone to print
```

cp pi-c.c pi-mpi-c.c
vim pi-mpi-c.c
make pi-mpi-c
mpirin –np 4 pi-mpi-c
Enter the number of intervals…

- Initialize MPI system
- How many nodes in total?
- What's my node ID? and my name?

PI for MPI in C

```
while (1) {
   if (myid == 0) {
     printf("Enter the number of intervals: (0 quits)
  ");fflush(stdout);
      scanf("%d",&n);
   }
   MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
   if (n == 0)
     break;
   mypi = integrate(n, myid, numprocs);

   MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);

   if (myid == 0) {
     printf…
   }
  }
  MPI_Finalize();

  return 0;
}
```

- I am the leader: "rank 0"
- Tell everyone: # nodes total
- Everyone does their work indexed by myid & numoprocs
- Reduce result over everyone
- And the leader prints output

```
cp pi-c.c pi-mpi-c.c
vim pi-mpi-c.c
```

➢ Putting it all together…

PI for MPI+OpenMP in C


```
double integrate(int n, int myid, int numprocs) {
  double h, x, sum;
  int i;

  sum = 0.0;
  h   = 1.0 / (double) n;
#pragma omp parallel for reduction(+:sum) private(x)
  for (i = myid + 1; i <= n; i += numprocs) {
    x = h * ((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
  }
  return sum * h;
}
```

- Up to 100 nodes, 16 cores/node = 160X parallel

cp pi-mpi-c.c pi-mpi-omp-c.c
vim pi-mpi-omp-c.c
make pi-mpi-omp-c
pi-mpi-omp-c
Enter the number of intervals…

PI for PGI w/ MPI+OpenACC in C

OpenACC

```
double integrate(int n, int myid, int numprocs) {
  double h, x, sum;
  int i;

  sum = 0.0;
  h   = 1.0 / (double) n;
#pragma acc parallel reduction(+:sum) private(x)
  for (i = myid + 1; i <= n; i += numprocs) {
    x = h * ((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
  }
  return sum * h;
}
```

- Up to 100 nodes, 4k GPU threads/node = 400,000X parallel

cp pi-mpi-c.c pi-mpi-pgicu-c.c
vim pi-mpi-pgicu-c.c
make pi-mpi-pgicu-c
pi-mpi-pgicu-c
Enter the number of intervals…

PI for MPI+CUDA in C

```
__global__ void integrate(int *n, int *blocks, int *myid,
                          int*numprocs, double *gsum) {
  const unsigned int bid = blockDim.x * blockIdx.x + threadIdx.x;
  const unsigned int gid = *blocks * blockDim.x * *myid + bid;
  …
  for (i = gid + 1; i <= *n; i += blockDim.x * *blocks *
   *numprocs) {
    …
```

make pi-cu-mpi
mpirun –np 4 pi-cu-mpi
Enter the number of intervals…

PI for MPI+CUDA in C

```
int main(int argc, char *argv[]) {
  int n, blocks, myid, numprocs;
  int *n_d, *blocks_d, *myid_d, *numprocs_d; // device copy
  double PI25DT = 3.14159265358979323846264643;
  double mypi, pi;
  double *mypi_d; // device copy of pi
  int  namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  MPI_Get_processor_name(processor_name,&namelen);

  // Allocate memory on GPU
  cudaMalloc( (void **) &n_d, sizeof(int) * 1 );
  cudaMalloc( (void **) &blocks_d, sizeof(int) * 1 );
  cudaMalloc( (void **) &numprocs_d, sizeof(int) * 1 );
  cudaMalloc( (void **) &myid_d, sizeof(int) * 1 );
  cudaMalloc( (void **) &mypi_d, sizeof(double) * THREADS *
   MAX_BLOCKS );
```

PI for MPI+CUDA in C

```
…
 while (1) {
   if (myid == 0) {
     printf…
   }
   MPI_Barrier(MPI_COMM_WORLD);
   MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
   MPI_Bcast(&blocks, 1, MPI_INT, 0, MPI_COMM_WORLD);
   if (n == 0 || blocks > MAX_BLOCKS)
     break;
   …
   cudaMemcpy( blocks_d, &blocks, sizeof(int) * 1,
               cudaMemcpyHostToDevice );
   …
   integrate<<< blocks, THREADS >>>(n_d, blocks_d,
                                    myid_d, numprocs_d, mypi_d);
   if (blocks > 1)
     global_reduce<<< 1, 512 >>>(n_d, blocks_d, mypi_d);
   …
   cudaMemcpy( &mypi, mypi_d, sizeof(double) * 1,
               cudaMemcpyDeviceToHost );
```

PI for MPI+CUDA in C

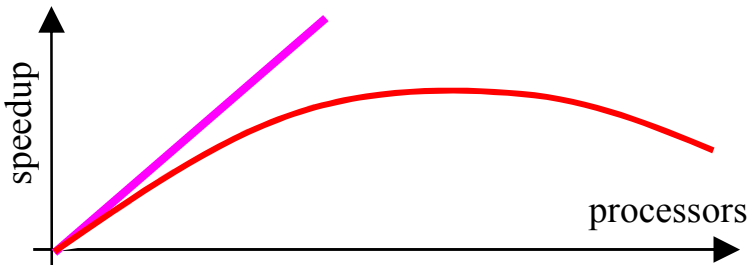
```
    MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);

    if (myid == 0) {
      printf…

  // free GPU memory
  cudaFree(n_d);
  cudaFree(blocks_d);
  cudaFree(mypi_d);

  MPI_Finalize();

  return 0;
}
```

- Up to 100 nodes, 32k GPU threads/node = 3,200,000X parallel

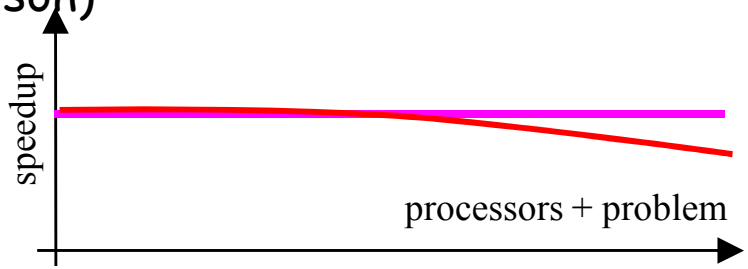
UGHH!!!

On the Limits of Speedup: Scaling

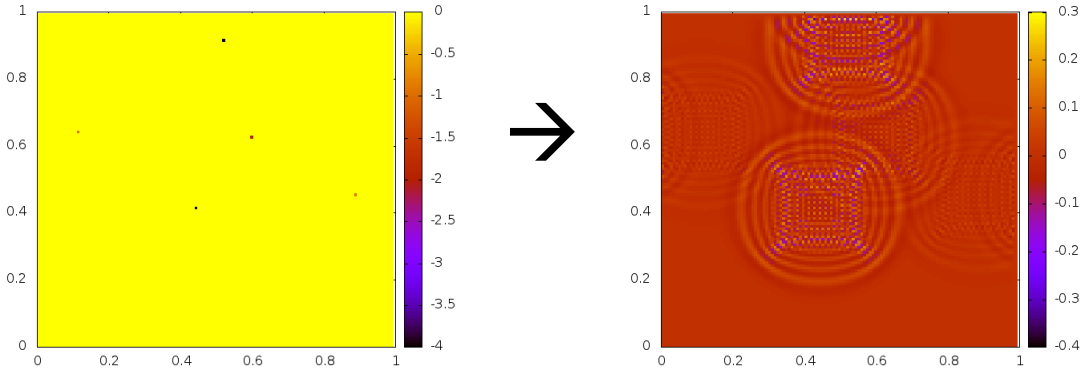
Shown here: ideal speedup/real speedup

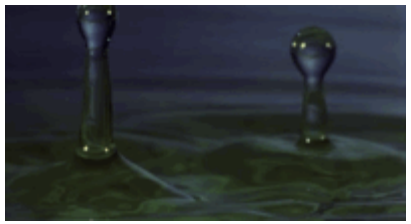


- Task (Strong) Scaling (Ahmdahl)
  - Same problem size, more cores
  - Limited by serial part
- Task+Problem (Weak) Scaling (Gustafson)
  - Increase cores+inputs @ same rate → much better scaling
  - May not fit your problem



- Ex.: Fluid Dynamics Stencil → Lake





Parallel Problem Solving on ARC (1)

- MPI+GPU short course http://moss.csc.ncsu.edu/~mueller/mpigpu/
  - OpenMP threads for multicores
  - OpenACC + CUDA for GPUs
  - MPI for multiple nodes

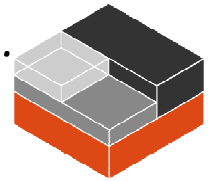
- Scientific programming: lots of libraries
  - Numerical packages: scalapack, openblas (linear algebra), fftw
  - Solver frameworks: Petsc, Tirlinos
  - Scientific data/file/IO formats: netcdf, adios, hdf5

Parallel Problem Solving on ARC (2)

- Python: slow when interpreted, fast when using native libraries
  - mpi4pi for multiple nodes
  - numpy, TensorFlow, Keras, … for GPUs

- Frameworks
  - Yarn for map-reduce (big data, out-of-core) in Java/Python/…
  - Spark (medium-data, in memory core) in Java/Python/…
  - TensorFlow, Keras, … (machine learning) in Pyhton

- Virtualization: LXD, VirtualBox, Docker containers…



- Tools for Debugging: Valgrind, Tau, Scalasca

Summary

- Your ARC Cluster@Home: What can I do with it?
- Primary purpose: Advance Computer Science Research (HPC and beyond)
  - Want to run a job over the entire machine?
  - Want to replace parts of the software stack?
- Secondary purpose: Service to sciences, engineering & beyond
  - Vision: Have domain scientists work w/ Computer Scientists on code

http://moss.csc.ncsu.edu/~mueller/cluster/arc/

- Equipment donations welcome ☺
- Ideas how to improve ARC? → let us know
  - Qs? → send to mailing list (once you have an account)
  - request an account: email ngholka<at>ncsu.edu
    - Research topic, abstract, and compute requirements/time
      - ***Must*** include your unity ID
      - NCSU Students: Advisor sends email as means of their approval
      - Non-NCSU: same + preferred username + hostname(your remote login location.