Introduction to High Performance Computing System, Programming and Applications

Department of Computer Science and Engineering
Yonghong Yan
yanyh@cse.sc.edu
http://cse.sc.edu/~yanyh

CSCE569 Parallel Computing, Spring 2018: https://passlab.github.io/CSCE569/

Contents

- High performance computing and parallel computing
  – What and why
- Measuring the performance of computers and supercomputers
- Parallel system architectures and programming
  1. Shared memory system (multi-core and multi-CPU machine)
  2. Peripheral discrete memory system (GPU accelerator)
  3. Distributed memory system (computing cluster)
- Summary

What is High Performance Computing

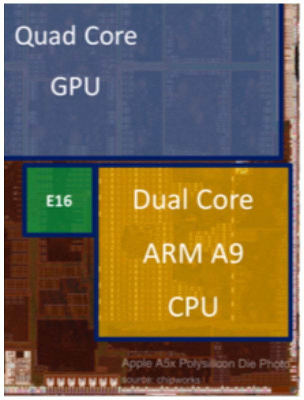
- **Aggregating computing power of multiple computing elements**
  - **Parallel processing vs sequential processing**
- **Higher performance than a typical desktop computer or workstation**
  - **Supercomputer vs computing**
- **Solving large problems in science, engineering, or business**
  - **Computational science and big-data processing vs web browser, office software, music player, etc**

*What is HPC: http://insidehpc.com/hpc-basic-training/what-is-hpc/

Parallel Computing

- HPC is what really needed *
  - Parallel computing is so far the only way to get there!!
- Parallel computing makes sense!

- Applications that require HPC
  - Many problem domains are naturally parallelizable
  - Data cannot fit in memory of one machine
- Computer systems
  - Physics limitation: has to build it parallel
  - Parallel systems are widely accessible
    - Smartphone has 2 to 4 cores + GPU now

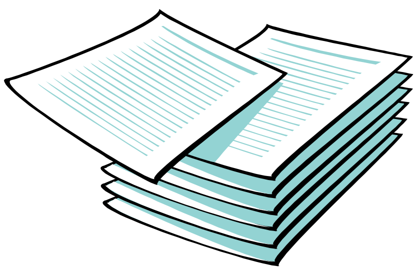


Supercomputer: http://en.wikipedia.org/wiki/Supercomputer
TOP500 (500 most powerful computer systems in the world): http://en.wikipedia.org/wiki/TOP500, http://top500.org/
HPC matter: http://sc14.supercomputing.org/media/social-media

An Example: Grading

15 questions

300 exams

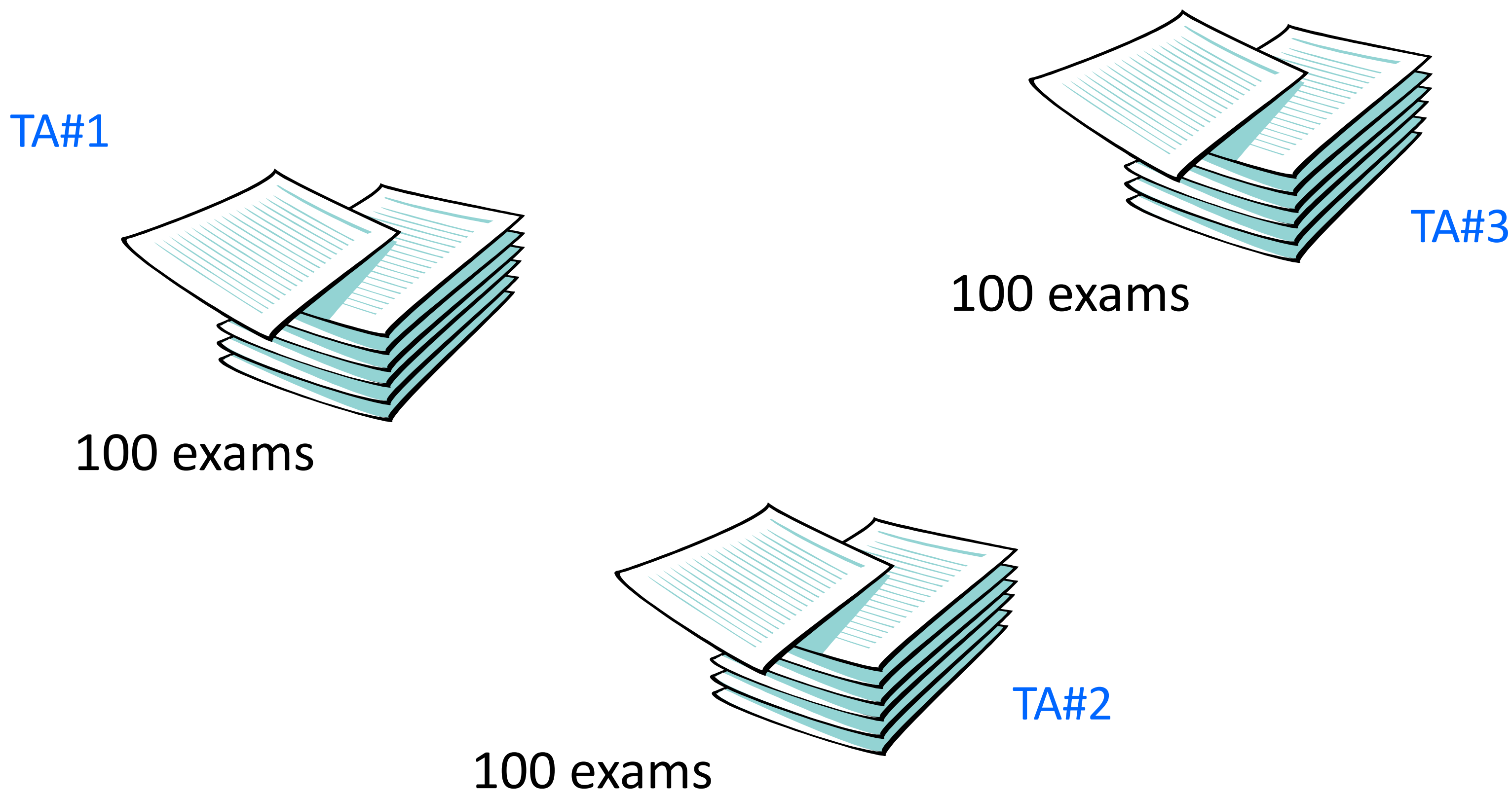


Three Teaching Assistants

- To grade 300 copies of exams, each has 15 questions

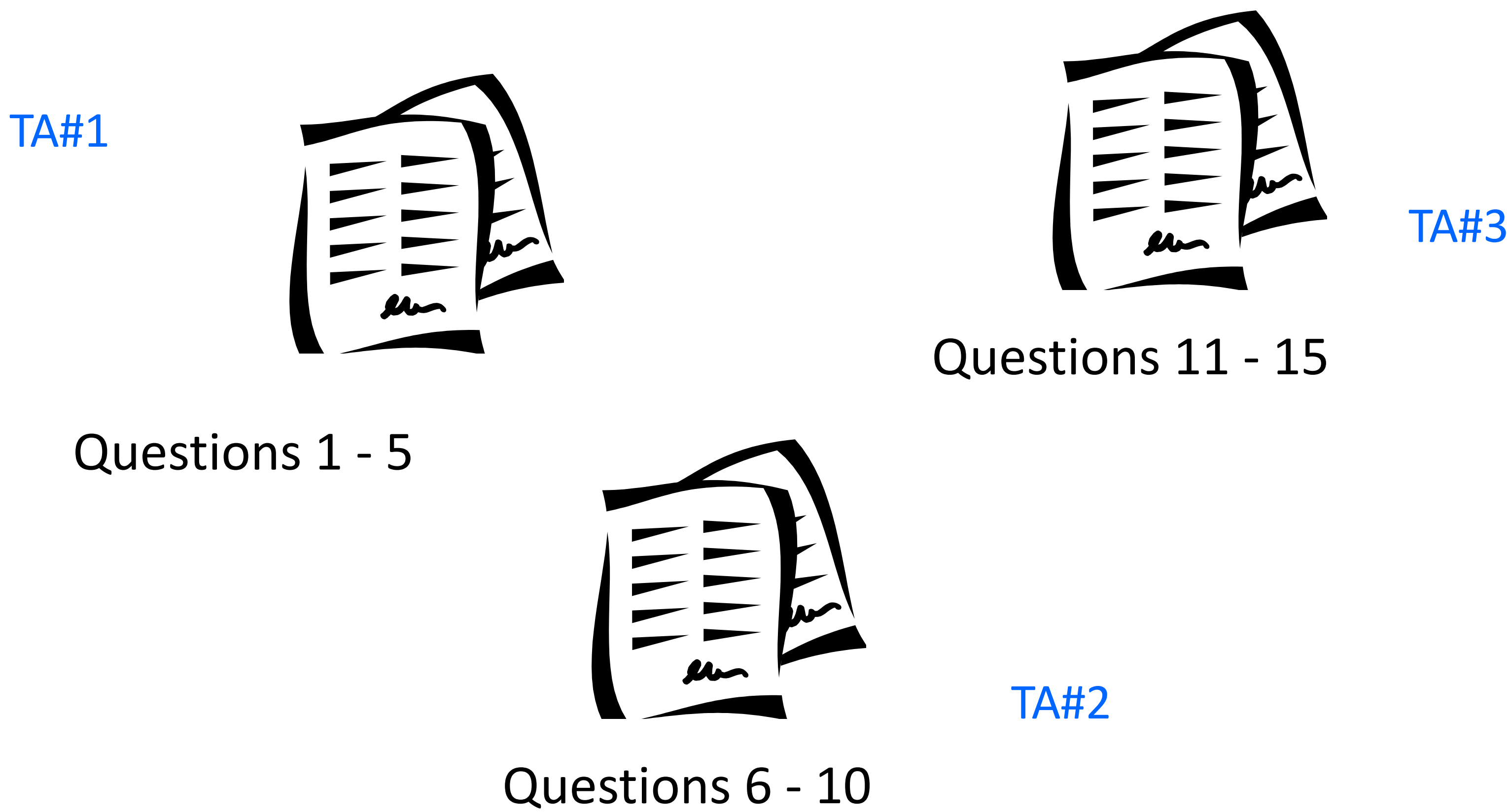
Division of Work – Data Parallelism

- Each does the same type of work (task), but working on different sheet (data)



Division of Work – Task Parallelism

- Each does different type of work (task), but working on same sheets (data)



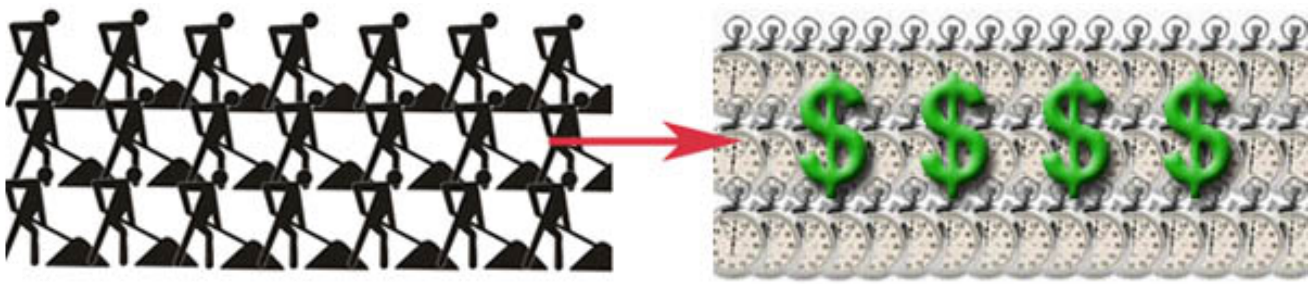
What is Parallel Computing?

- **A form of computation*:**
  - **Large problems divided into smaller ones**
  - **Smaller ones are carried out and solved simultaneously**

*http://en.wikipedia.org/wiki/Parallel_computing

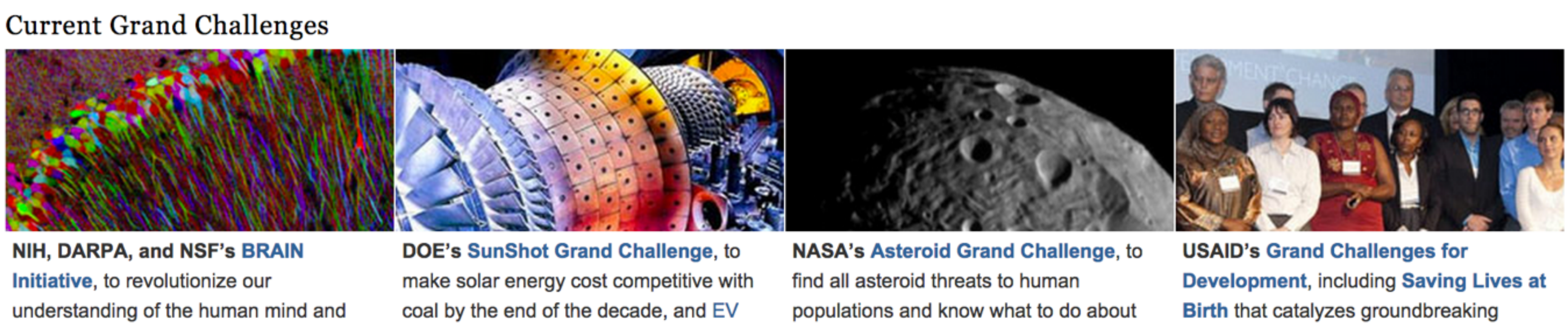
Parallel Computing

- **Save time (execution time) and money!**
  - **Parallel program can run faster if running concurrently instead of sequentially.**



Picture from: Intro to Parallel Computing: https://computing.llnl.gov/tutorials/parallel_comp

- **Solve larger and more complex problems!**
  - **Utilize more computational resources**



From "21st Century Grand Challenges | The White House", http://www.whitehouse.gov/administration/eop/ostp/grand-challenges
Grand challenges: http://en.wikipedia.org/wiki/Grand_Challenges

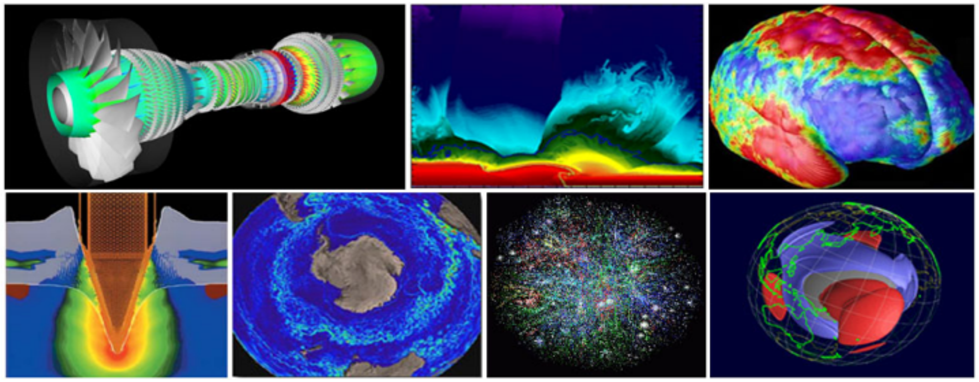
Simulation: The *Third* Pillar of Science

- **Traditional scientific and engineering paradigm:**
  1) **Do theory or paper design.**
  2) **Perform experiments or build system.**
- Limitations of experiments:
  - Too difficult -- build large wind tunnels.
  - Too expensive -- build a throw-away passenger jet.
  - Too slow -- wait for climate or galactic evolution.
  - Too dangerous -- weapons, drug design, climate experimentation.
- **Computational science paradigm:**
  3) **Use high performance computer systems to simulate the phenomenon**
     - **Base on known physical laws and efficient numerical methods.**

From slides of Kathy Yelic's 2007 course at Berkeley: http://www.cs.berkeley.edu/~yelick/cs267_sp07/

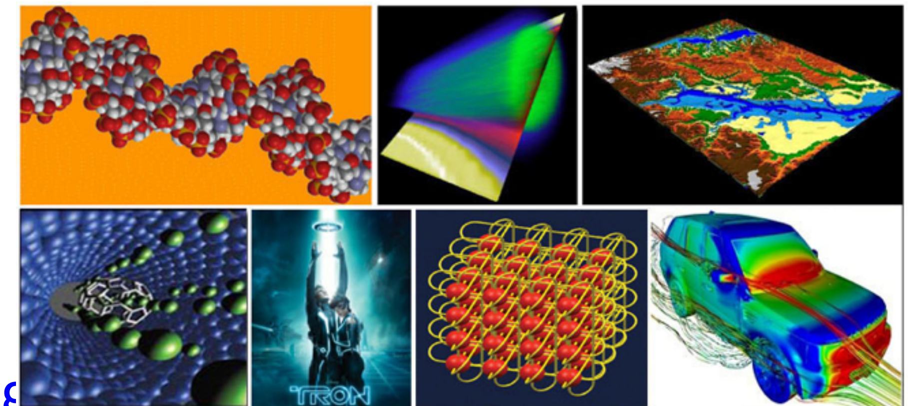
Applications: Science and Engineering

- Model many difficult problems by parallel computing
  - Atmosphere, Earth, Environment
  - Physics - applied, nuclear, particle, condensed matter, high pressure, fusion, photonics
  - Bioscience, Biotechnology, Genetics
  - Chemistry, Molecular Sciences
  - Geology, Seismology
  - Mechanical Engineering - from prosthetics to spacecraft
  - Electrical Engineering, Circuit Design, Microelectronics
  - Computer Science, Mathematics
  - Defense, Weapons

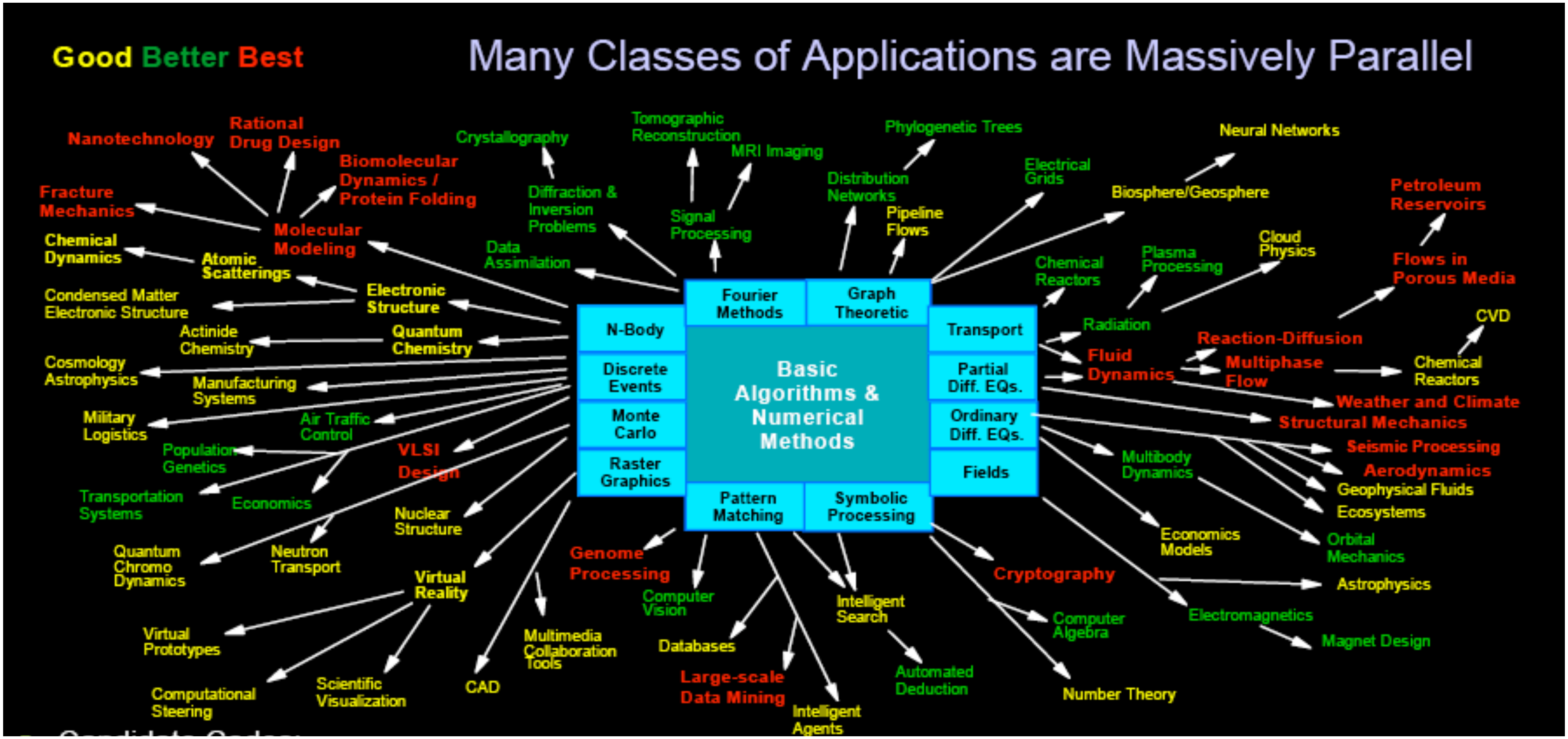


Applications: Industrial and Commercial

- Processing large amounts of data in sophisticated ways
  - Databases, data mining
  - Oil exploration
  - Medical imaging and diagnosis
  - Pharmaceutical design
  - Financial and economic modeling
  - Management of national and multi-national corporations
  - Advanced graphics and virtual reality, particularly in the entertainment industry
  - Networked video and multi-media technologies
  - Collaborative work environments
  - Web search engines, web based business services



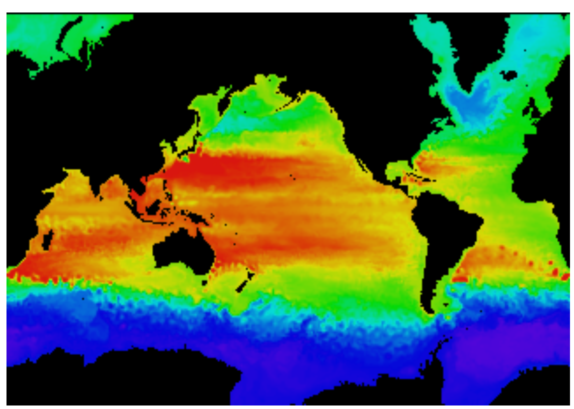
Inherent Parallelism of Applications

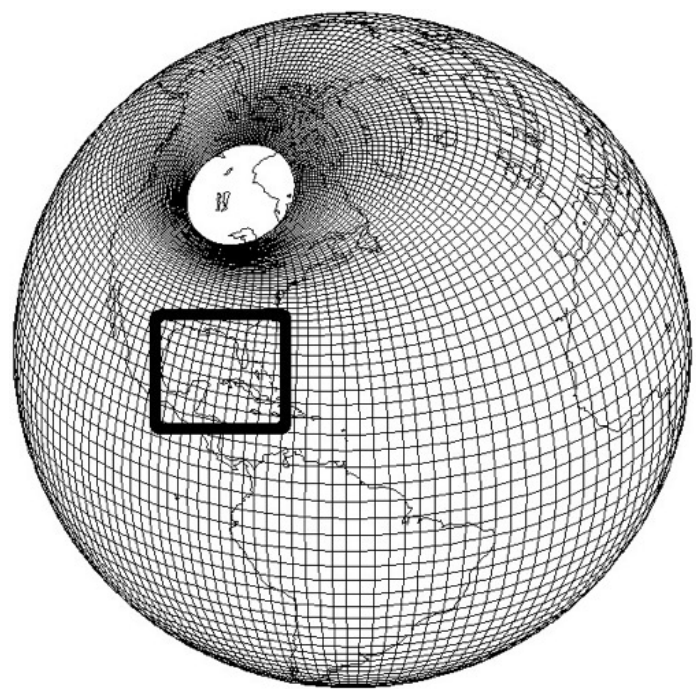


12 Dwarfs: The Landscape of Parallel Computing Research: A View from Berkeley
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf

Global Climate Modeling Problem

- **Problem is to compute:**
  - **f(latitude, longitude, elevation, time) →**
    temperature, pressure, humidity, wind velocity
- Approach:
  - *Discretize* the domain, e.g., a measurement point every 10 km
  - Devise an algorithm to predict weather at time t+dt given t
- Uses:
  - Predict major events, e.g., El Nino
  - Air quality forecasting





Units of Measure in HPC

- **Flop**: floating point operation (*, /, +, -, etc)
- **Flop/s**: floating point operations per second, written also as **FLOPS**
- **Bytes**: size of data
  - **A double precision floating point number is 8 bytes**
- Typical sizes are millions, billions, trillions…
  - Mega Mflop/s = 10^6 flop/sec Mzbyte = 2^{20} = 1048576 = ~10^6 bytes
  - Giga Gflop/s = 10^9 flop/sec Gbyte = 2^{30} = ~10^9 bytes
  - Tera Tflop/s = 10^{12} flop/secTbyte = 2^{40} = ~10^{12} bytes
  - **Peta Pflop/s = 10^{15} flop/sec Pbyte = 2^{50} = ~10^{15} bytes**
  - Exa Eflop/s = 10^{18} flop/secEbyte = 2^{60} = ~10^{18} bytes
  - Zetta Zflop/s = 10^{21} flop/secZbyte = 2^{70} = ~10^{21} bytes

- www.top500.org for the units of the fastest machines measured using High Performance LINPACK (HPL) Benchmark
  - The fastest: Summit, USA, 122.3 petaflop/s
  - The second fastest: Sunway TaihuLight, ~93 petaflop/s, the fastest till June 2018

HPC Peak Performance (Rpeak) Calculation

- **Node performance in Gflop/s = (CPU speed in GHz) x (number of CPU cores) x (CPU instruction per cycle) x (number of CPUs per node).**
  - **CPU instructions per cycle (IPC) = #Flops per cycle**
    - **Because pipelined CPU can do one instruction per cycle**
    - **4 or 8 for most CPU (Intel or AMD)**
  - **http://www.calcverter.com/calculation/CPU-peak-theoretical-performance.php**

- **HPC Peak (Rpeak) = # nodes * Node Performance in GFlops**

CPU Peak Performance Example

- Intel X5600 series CPUs and AMD 6100/6200/6300 series CPUs have 4 instructions per cycle
  Intel E5-2600 series CPUs have 8 instructions per cycle

- Example 1: Dual-CPU server based on Intel X5675 (3.06GHz 6-cores) CPUs:
  – 3.06 x 6 x 4 x 2 = 144.88 GFLOPS
- Example 2: Dual-CPU server based on Intel E5-2670 (2.6GHz 8-cores) CPUs:
  – 2.6 x 8 x 8 x 2 = 332.8 GFLOPS
  – With 8 nodes: 332.8 GFLOPS x 8 = 2,442.4 GFLOPS = 2.44 TFLOPS
- Example 3: Dual-CPU server based on AMD 6176 (2.3GHz 12-cores) CPUs:
  – 2.3 x 12 x 4 x 2 = 220.8 GFLOPS
- Example 4: Dual-CPU server based on AMD 6274 (2.2GHz 16-cores) CPUs:
  – 2.2 x 16 x 4 x 2 = 281.6 GFLOPS

https://saiclearning.wordpress.com/2014/04/08/how-to-calculate-peak-theoretical-performance-of-a-cpu-based-hpc-system/

How to Measure and Calculate Performance (FLOPS)

https://passlab.github.io/CSCE569/resources/sum.c

```
    elapsed = read_timer();
    REAL result = sum(N, X, a);
    elapsed = (read_timer() - elapsed);

    double elapsed_2 = read_timer();
    result = sumaxpy(N, X, Y, a);
    elapsed_2 = (read_timer() - elapsed_2);

    /* you should add the call to each function and time the execution */
    printf("==================================================================
    printf("\tSum %d numbers\n", N);
    printf("------------------------------------------------------------------
    printf("Performance:\t\tRuntime (ms)\t MFLOPS \n");
    printf("------------------------------------------------------------------
    printf("Sum:\t\t\t%4f\t%4f\n", elapsed * 1.0e3, 2*N / (1.0e6 * elapsed));
    printf("SumAXPY:\t\t\t%4f\t%4f\n", elapsed_2 * 1.0e3, 3*N / (1.0e6 * elapsed_2));
    return 0;
}

REAL sum(int N, REAL X[], REAL a) {
    int i;
    REAL result = 0.0;
    for (i = 0; i < N; ++i)
        result += a * X[i];
    return result;
}

/*
 * sum: a*X[]+Y[]
 */
REAL sumaxpy(int N, REAL X[], REAL Y[], REAL a) {
    int i;
    REAL result = 0.0;
    for (i = 0; i < N; ++i)
        result += a * X[i] + Y[i];
    return result;
}
```

- Calculate # FLOPs (2*N or 3*N)
  - Check the loop count (N) and FLOPs per loop iteration (2 or 3).
- Measure time to compute using timer
  - elapsed and elapsed_2 are in second
- FLOPS = # FLOPs/Time
  - MFLOPS in the example

High Performance LINPACK (HPL) Benchmark Performance (Rmax) in Top500

- *Measured* using the High Performance LINPACK (HPL) Benchmark that solves a dense system of linear equations → Ranking the machines
  - Ax = b
  - https://www.top500.org/project/linpack

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/SC/Oak Ridge National Laboratory United States	2,282,544	122,300.0	187,659.3	8,806
2	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
3	**Sierra** - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/NNSA/LLNL United States	1,572,480	71,610.0	119,193.6	
4	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 , NUDT	4,981,760	61,444.5	100,678.7	18,482

Top500 (www.top500.org), June 2018

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/SC/Oak Ridge National Laboratory United States	2,282,544	122,300.0	187,659.3	8,806
2	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
3	**Sierra** - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/NNSA/LLNL United States	1,572,480	71,610.0	119,193.6	
4	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 , NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
5	**AI Bridging Cloud Infrastructure (ABCI)** - PRIMERGY CX2550 M4, Xeon Gold 6148 20C 2.4GHz, NVIDIA Tesla V100 SXM2, Infiniband EDR , Fujitsu National Institute of Advanced Industrial Science and Technology (AIST) Japan	391,680	19,880.0	32,576.6	1,649
6	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 , Cray Inc. Swiss National Supercomputing Centre (CSCS)	361,760	19,590.0	25,326.3	2,272

Three Kinds of Parallel Systems

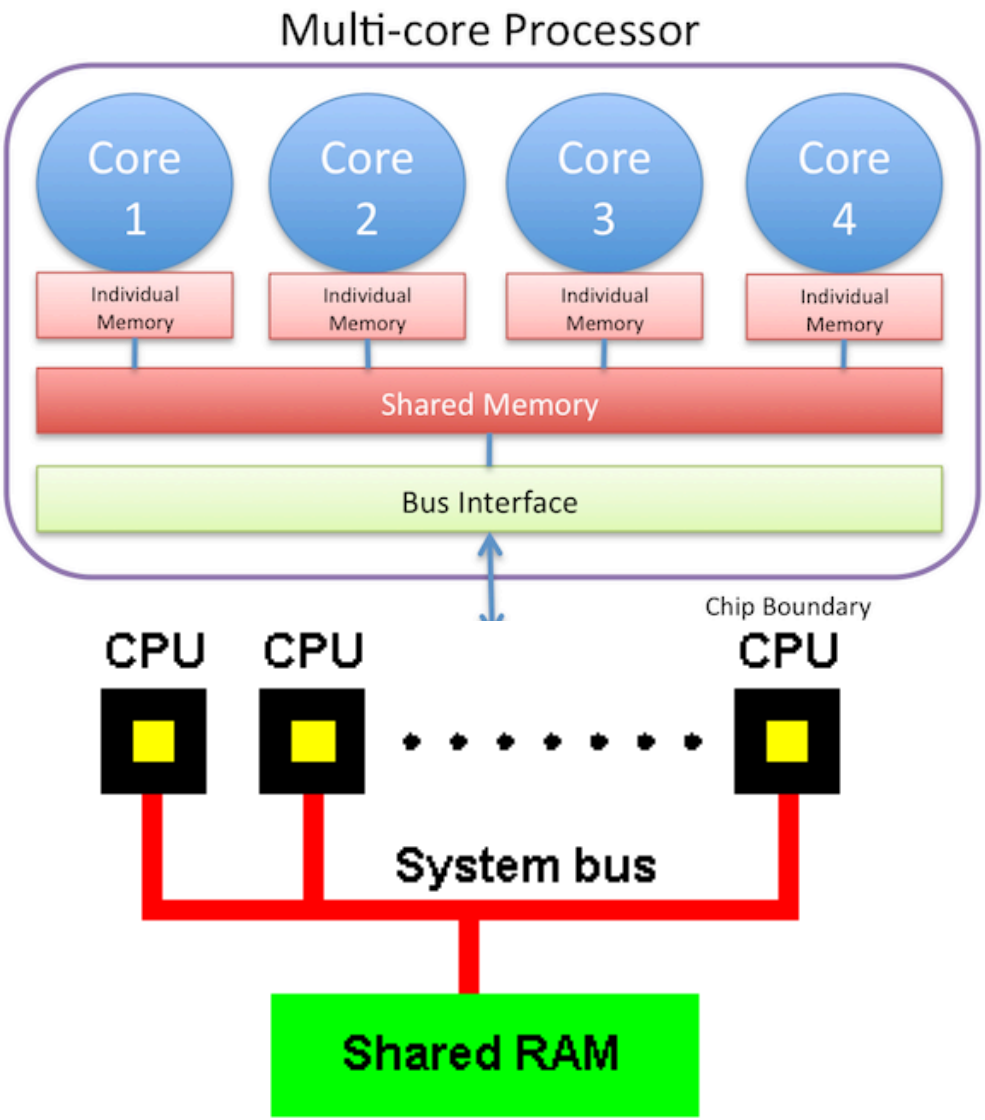
- Parallel computing itself is not hard: **A form of computation:**
  - **Large problems divided into smaller ones**
  - **Smaller ones are carried out and solved simultaneously**

- The **most difficult** issue of using HPC system, via parallel programming is to deal with **memory and data movement**
  - **Actually, most computing related problem**

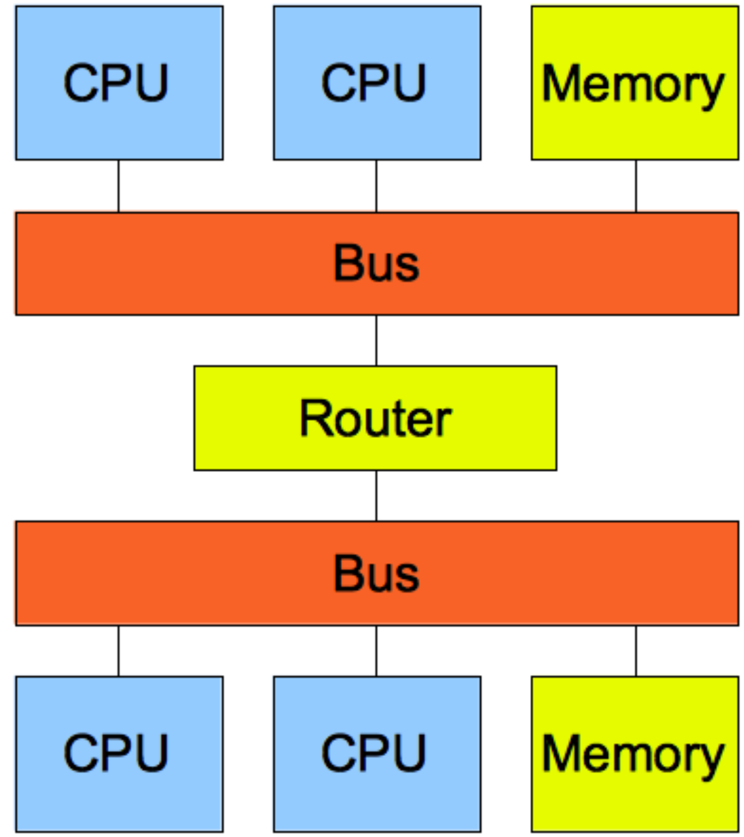
- Think of **three kinds of memory systems** → three kinds of parallel systems
  1. Shared memory system (multi-core, multi-CPU machine)
  2. Peripheral discrete memory system (GPU and accelerators)
  3. Distributed memory system (computing cluster)

  4. The combination of the above three

Shared Memory Systems



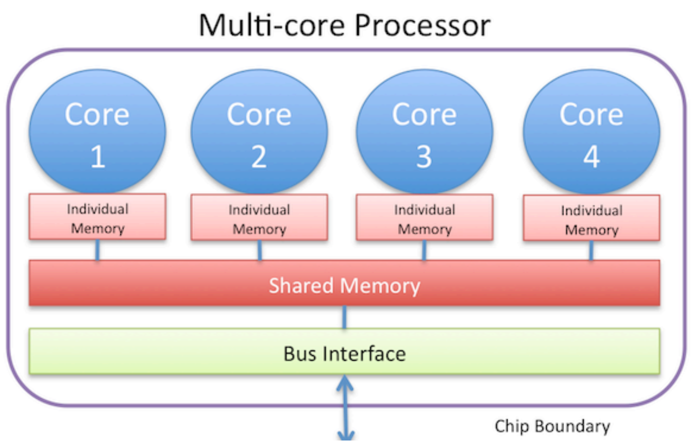
SMP: Multiple processors share RAM and system bus

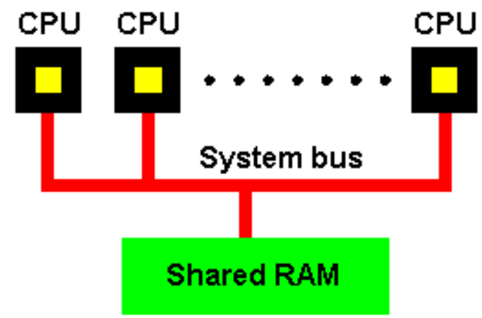


NUMA Architecture

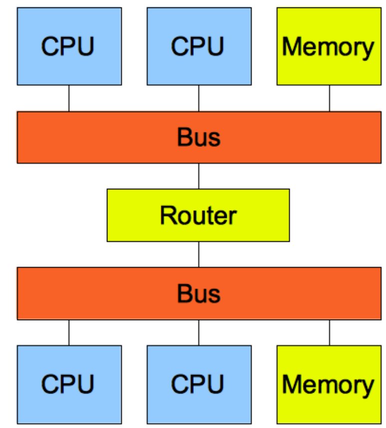
Motherboard of NUMA Architecture

Parallel Programming via Threading





SMP: Multiple processors share RAM and system bus



NUMA Architecture

- Parallel programming could be very simple
  - Performance could be very hard because of the memory system

- **Multithreading and tasking**
  - **POSIX threads and multi-processing in system level**
    - **pthread library and fork () system call**
  - **OpenMP**
  - **Cilkplus and lots of others**

OpenMP “Hello Word” Example/1

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {



          printf("Hello World\n");



   return(0);
}
```

OpenMP “Hello Word” - An Example/2

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
          printf("Hello World\n");

   } // End of parallel region

   return(0);
}
```

OpenMP “Hello Word” - An Example/3

```
$ gcc -fopenmp hello.c


$ export OMP_NUM_THREADS=2
$ ./a.out
Hello World
Hello World

$ export OMP_NUM_THREADS=4
$ ./a.out
Hello World
Hello World
Hello World
Hello World
$
```

```c
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
          printf("Hello World\n");

   } // End of parallel region

   return(0);
}
```

OpenMP “Hello Word” - An Example/4

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

int main(int argc, char *argv[]) {

#pragma omp parallel
  {
    int thread_id = omp_get_thread_num();
    int num_threads = omp_get_num_threads();

    printf("Hello World from thread %d of %d\n",
         thread_id, num_threads);
  }

   return(0);
}
```

Directives

Runtime Environment

OpenMP Worksharing Constructs

Sequential code

```
for(i=0;i<N;i++)   { a[i] = a[i] + b[i]; }
```

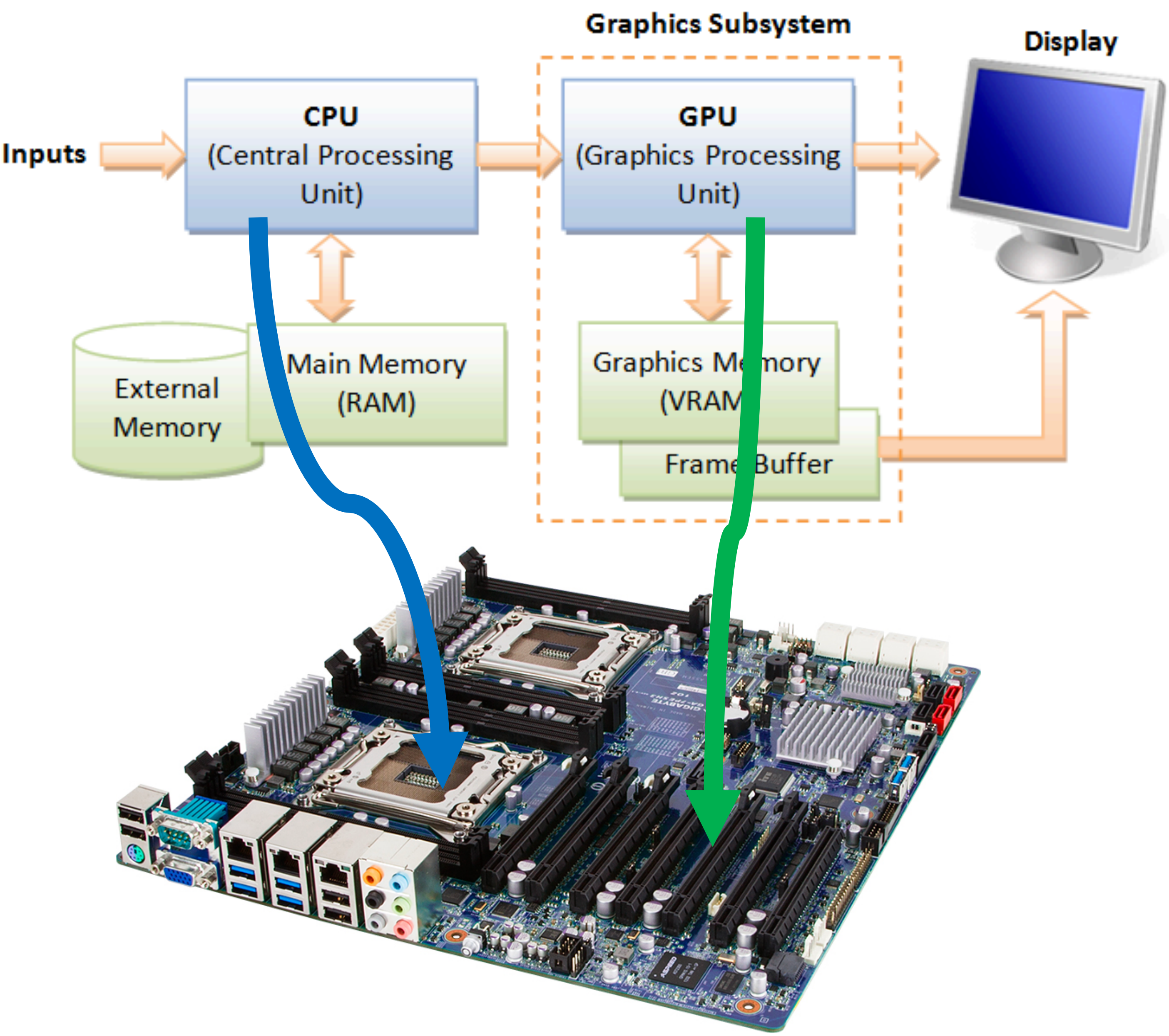
OpenMP parallel region and a worksharing for construct

```
#pragma omp parallel shared (a, b) private (i)
#pragma omp for schedule(static)
     for(i=0;i<N;i++)   { a[i] = a[i] + b[i]; }
```

OpenMP parallel region

```
#pragma omp parallel shared (a, b)

{

     int id, i, Nthrds, istart, iend;
     id = omp_get_thread_num();
     Nthrds = omp_get_num_threads();
     istart = id * N / Nthrds;
     iend = (id+1) * N / Nthrds;
     for(i=istart;i<iend;i++)   { a[i] = a[i] + b[i]; }
}
```

Discrete Memory System and Graphics Processing Unit (GPU)

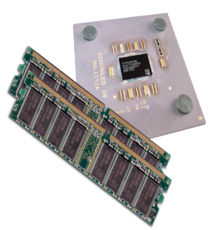


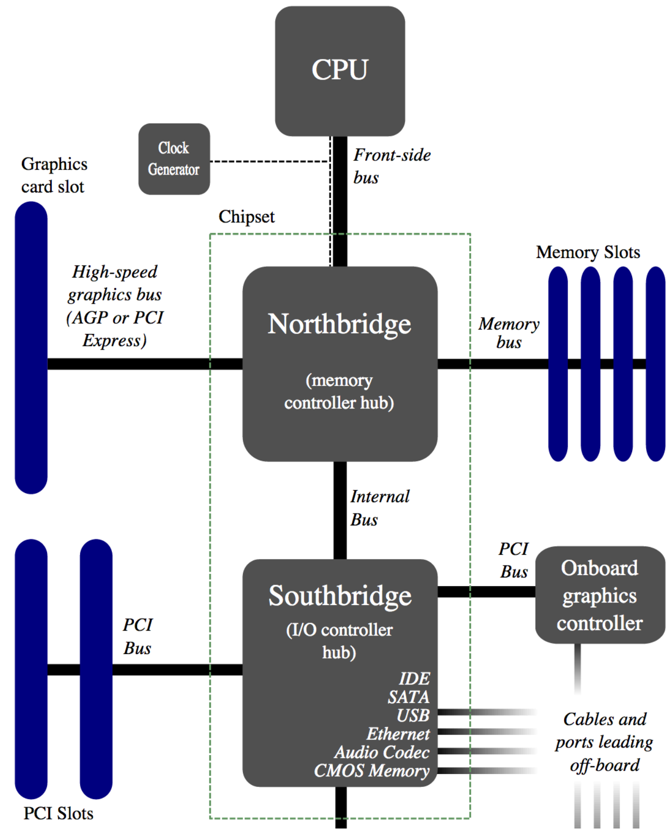
GPU Manycore Accelerators: From ~2007

- NVIDIA Tesla V100, Released May 2017
  - Total 80 SM Processors
- Cores
  - 5120 FP32 cores, 2560 FP64 cores, 640 Tensor cores
- Memory
  - 16G HBM2

GPU Computing – Offloading Computation

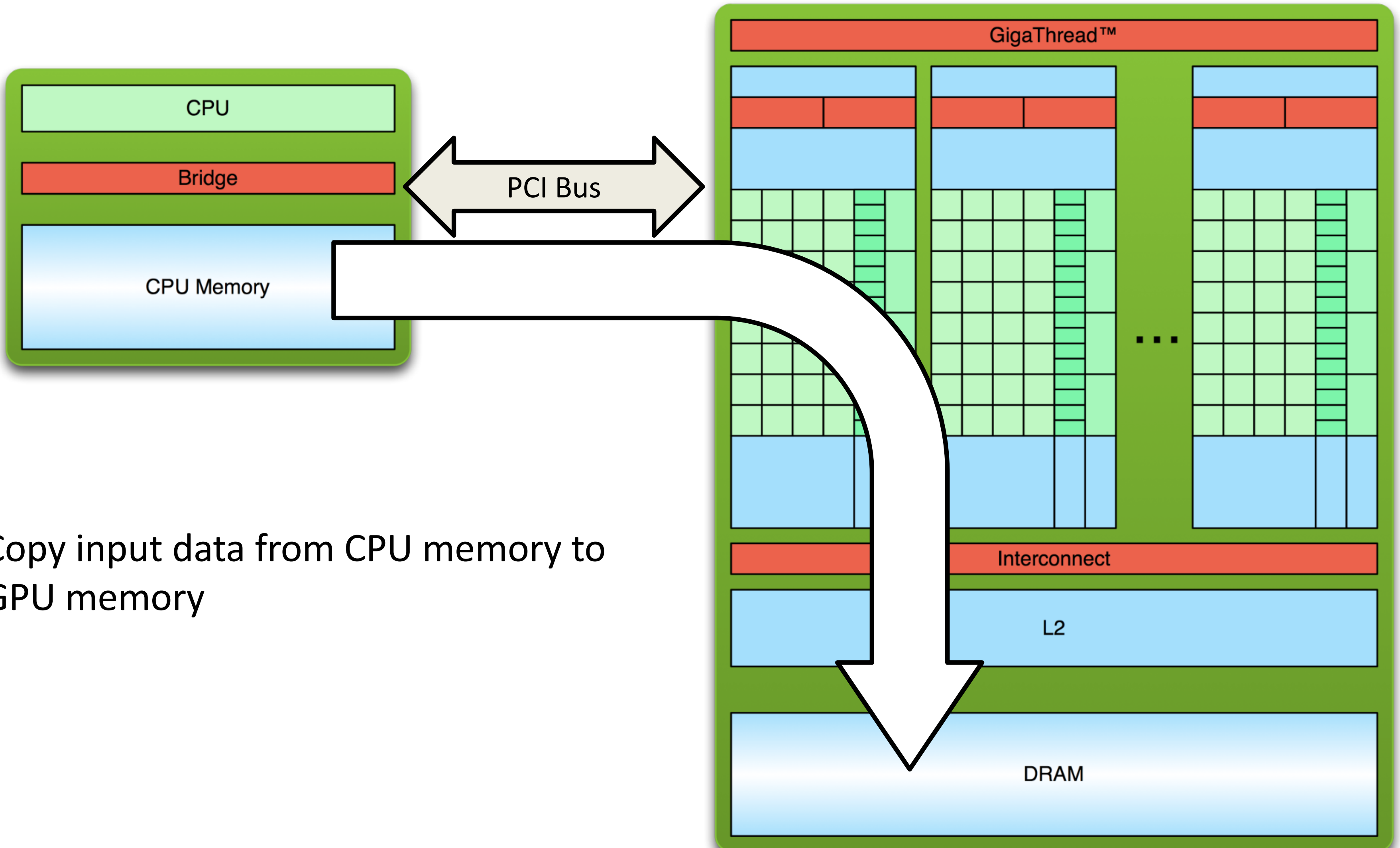
- The GPU is connected to the CPU by a reasonable fast bus (8 GB/s is typical today): PCIe





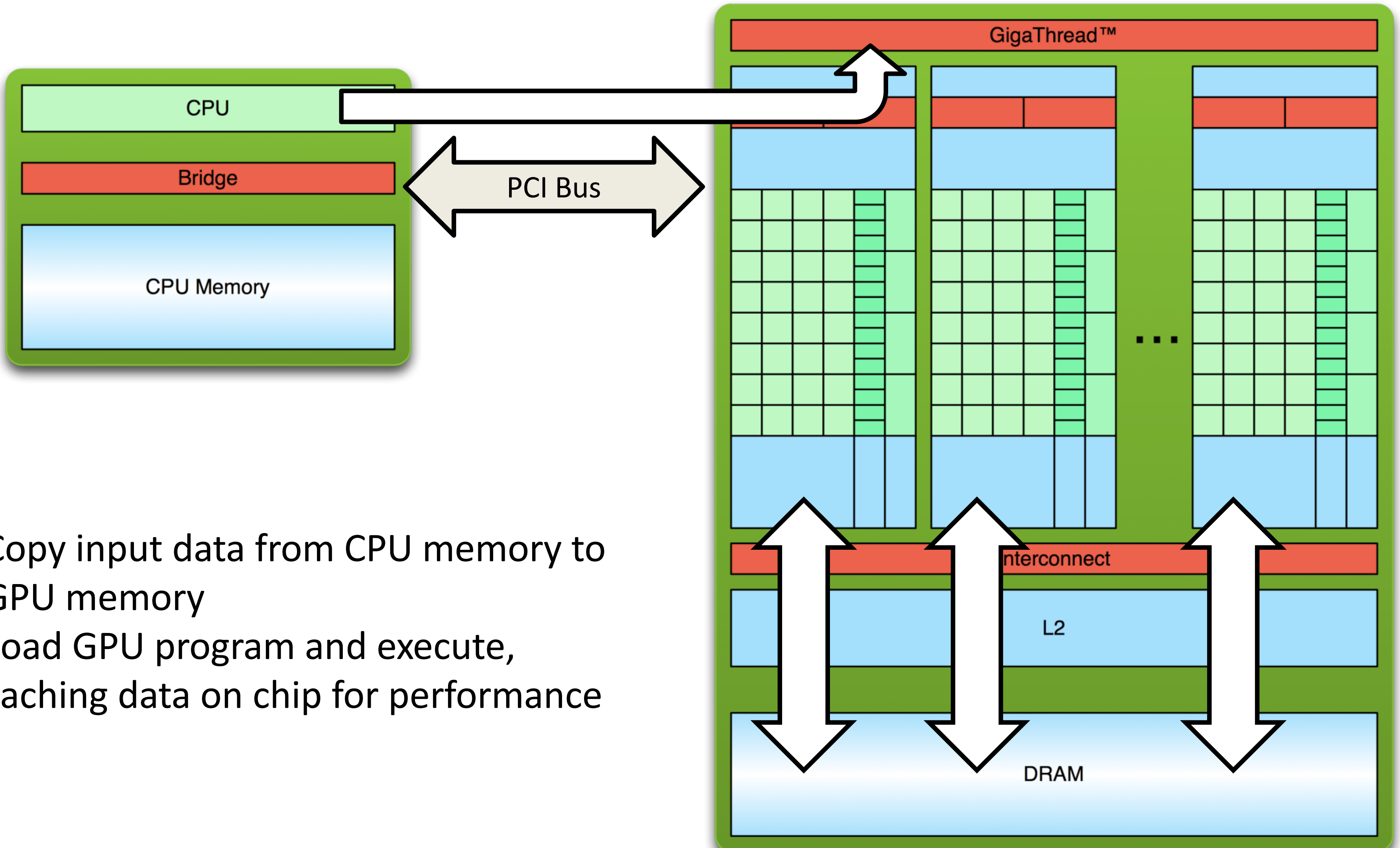
- Terminology
  - **Host**: The CPU and its memory (host memory)
  - **Device**: The GPU and its memory (device memory)

Offloading Workflow for GPU Programming



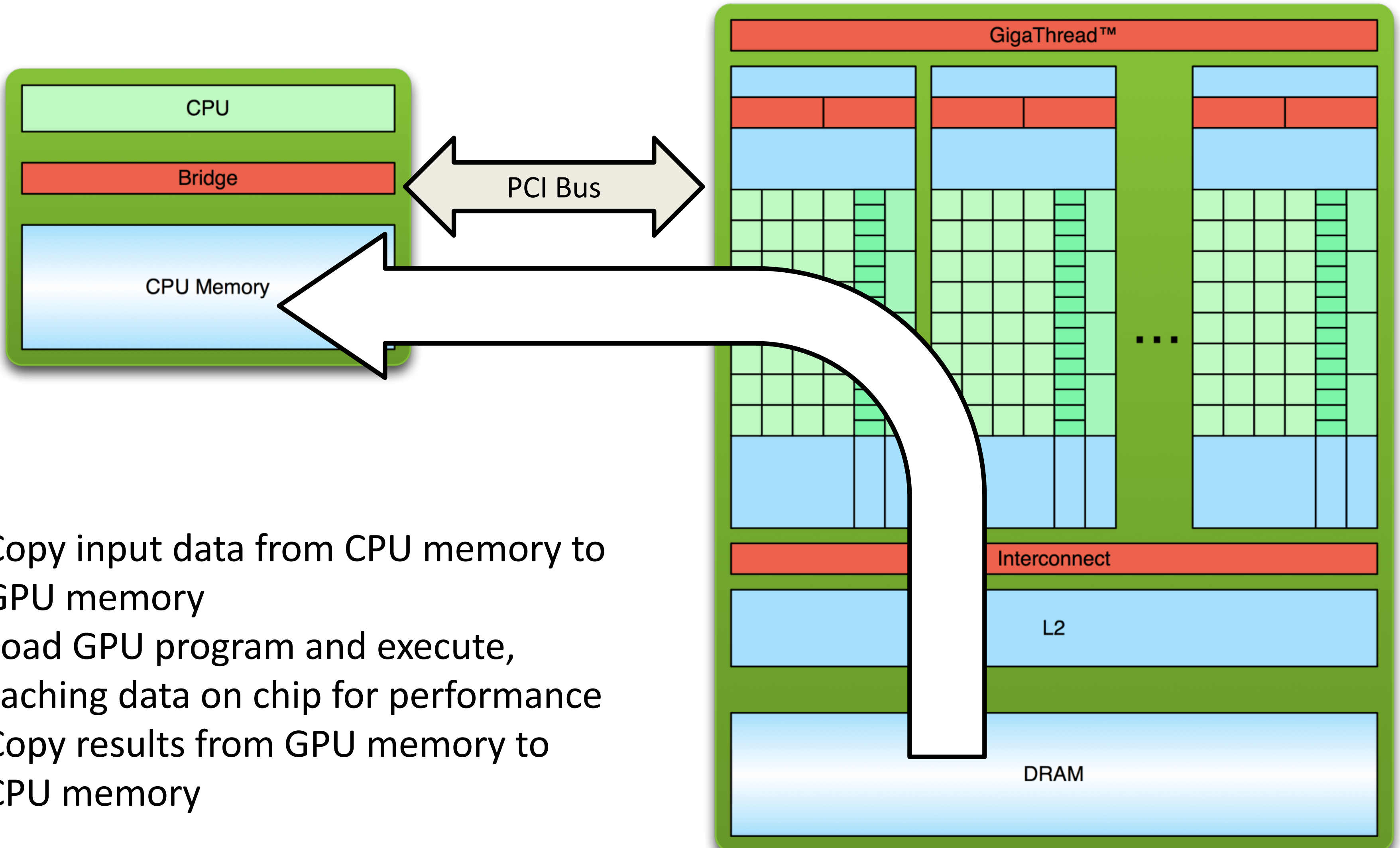
1. Copy input data from CPU memory to GPU memory

Offloading Workflow for GPU Programming



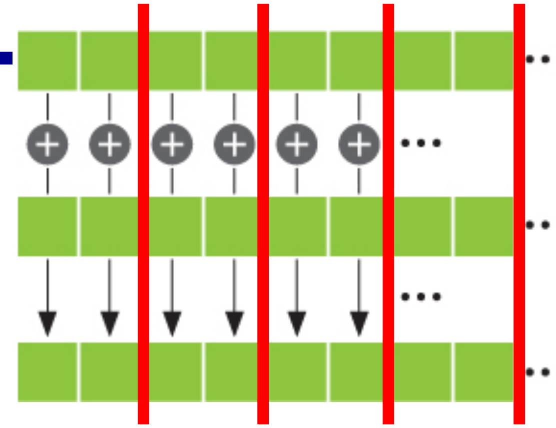
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Offloading Workflow for GPU Programming



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

AXPY Example with OpenMP: Multicore



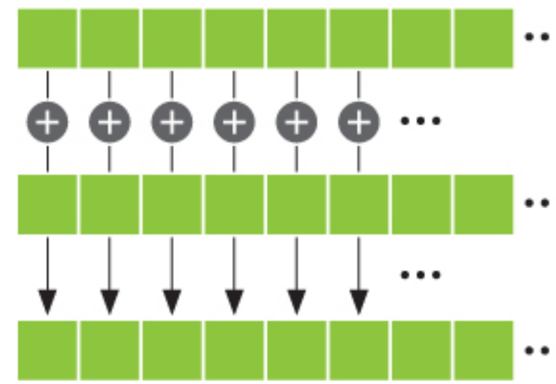
- y = α·x + y
  - x and y are vectors of size n
  - α is scalar

```
void axpy(REAL *x, REAL *y, long n, REAL a) {
  #pragma omp parallel for shared(x, y, n, a)
  for (int i = 0; i < n; ++i)
    y[i] += a * x[i];
}
```

- **Data (x, y and a) are shared**
  - Parallelization is easy

AXPY Offloading To a GPU using CUDA

```
// CUDA kernel. Each thread takes care of one element of c
__global__ void axpy(REAL *x, REAL *y, int n, REAL a) {
    int id = blockIdx.x*blockDim.x+threadIdx.x;
    if (id < n) y[id] += a * x[id];
}

int main( int argc, char* argv[] ) {

    // ... init host a, x and y
    // Allocate memory for each vector on GPU
    cudaMalloc(&d_x, size);
    cudaMalloc(&d_y, size);

    // Copy host vectors to device
    cudaMemcpy( d_x, h_x, size, cudaMemcpyHostToDevice);
    cudaMemcpy( d_y, h_y, size, cudaMemcpyHostToDevice);

    int blockSize, gridSize;
    blockSize = 1024;
    gridSize = (int)ceil((float)n/blockSize);
    axpy<<<gridSize, blockSize>>>(d_x, d_y, n, a);

    // Copy array back to host
    cudaMemcpy( h_y, d_y, size, cudaMemcpyDeviceToHost );

    // Release device memory
    cudaFree(d_x);
    cudaFree(d_y);
}
```



Memory allocation on device

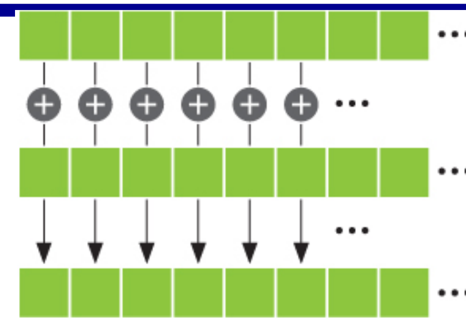
Memcpy from host to device

Launch parallel execution

Memcpy from device to host

Deallocation of dev memory

AXPY Example with OpenMP: single device



- $y = \alpha \cdot x + y$
  - x and y are vectors of size n
  - α is scalar

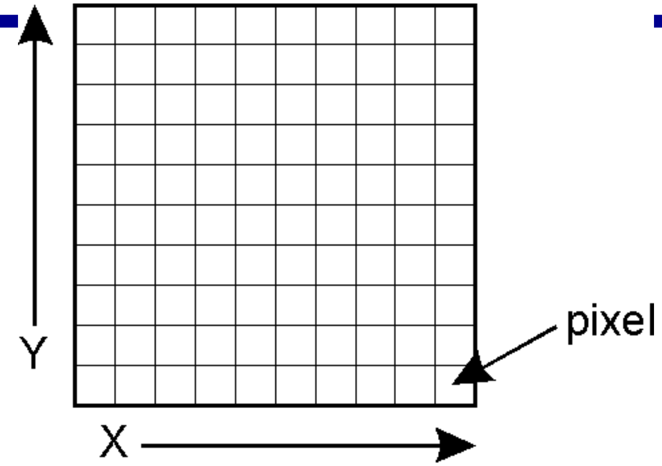
```
void axpy_ompacc(REAL* x, REAL* y, int n, REAL a) {
  #pragma omp target device (0) map(tofrom: y[0:n]) \
        map(to: x[0:n],a,n)
  #pragma omp parallel for shared(x, y, n, a)
  for (int i = 0; i < n; ++i)
    y[i] += a * x[i];
}
```

- **target** directive: annotate an offloading code region
- **map** clause: map data between host and device → moving data
  - **to**|**tofrom**|**from**: mapping directions
  - Use array region

What kinds of computation fit for *Graphics* Processing Unit (GPU)?

Image Format and Processing

- Pixels
  - Images are matrix of pixels



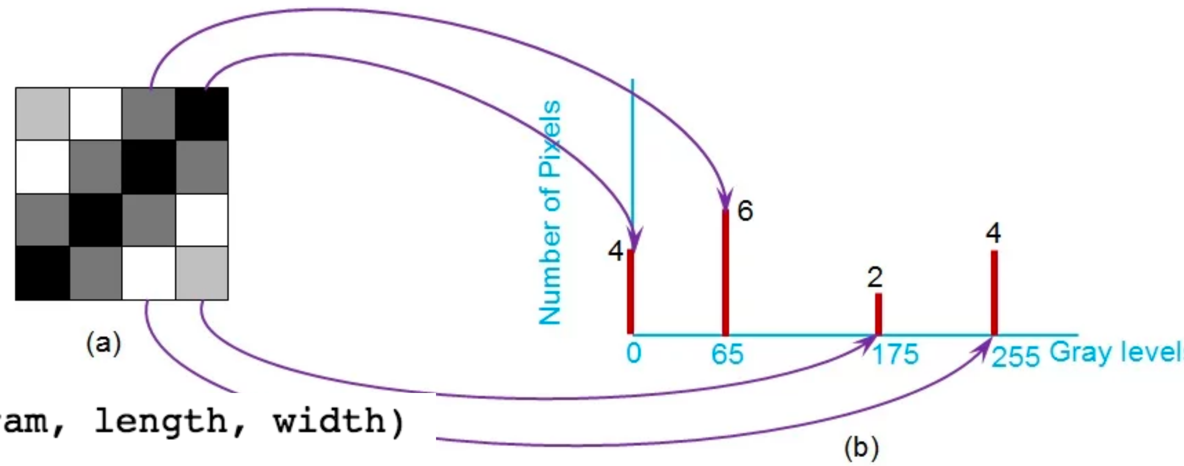
- Grayscale images
  - Each pixel value normally range from 0 (black) to 255 (white)
  - 8 bits per pixel

But the camera sees this:

194	210	201	212	199	213	215	195	178	158	182	209
180	189	190	221	209	205	191	167	147	115	129	163
114	126	140	188	176	165	152	140	170	106	78	88
87	103	115	154	143	142	149	153	173	101	57	57
102	112	106	131	122	138	152	147	128	84	58	66
94	95	79	104	105	124	129	113	107	87	69	67
68	71	69	98	89	92	98	95	89	88	76	67
41	56	68	99	63	45	60	82	58	76	74	65
20	41	69	75	56	41	51	73	55	70	63	44
50	50	57	69	75	75	73	74	53	68	59	37
72	59	53	66	84	92	84	74	57	72	63	42
67	61	58	65	75	78	76	73	59	75	69	50

Histograms of Monochrome Image

- **Same operation for every pixel**




```
calculate_histogram(image, histogram, length, width)
   int     length, width;
   short   **image;
   unsigned long histogram[];
{
   long  i,j;
   short k;
   for(i=0; i<length; i++){
      for(j=0; j<width; j++){
         k = image[i][j];
         histogram[k] = histogram[k] + 1;
      }
   }
}  /* ends calculate_histogram */
```

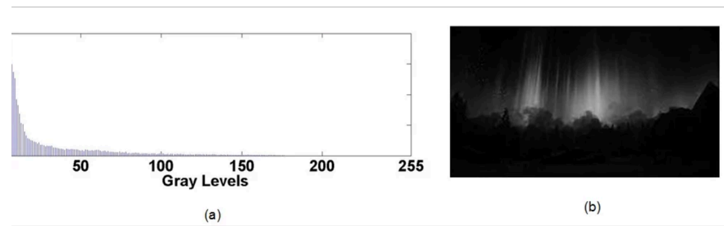


Figure 5. *Histogram of a dark image. Image by Sneha H.L.*

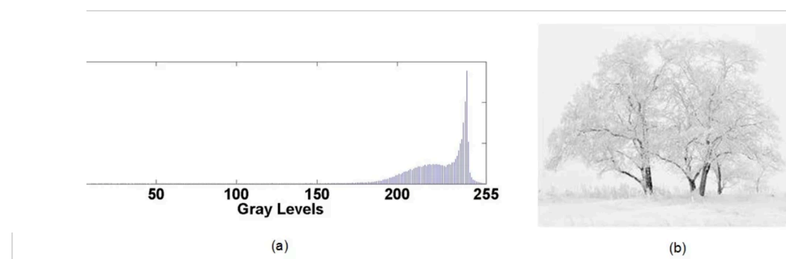
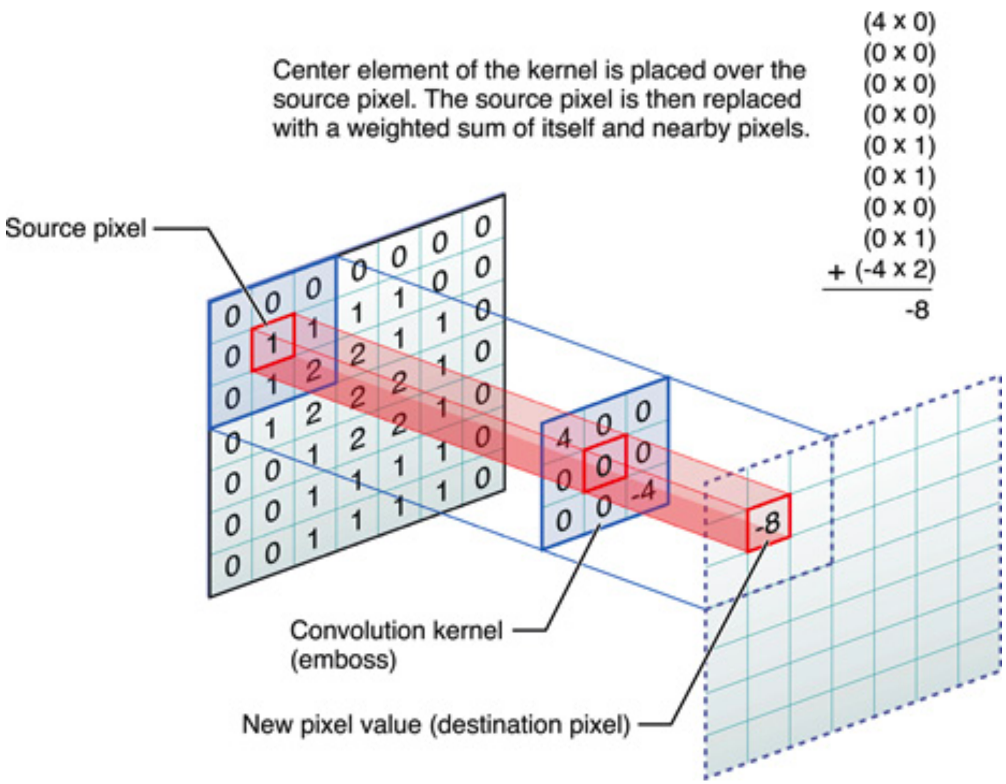


Figure 6. *Histogram of a bright image. Image by Sneha H.L.*

http://homepages.inf.ed.ac.uk/rbf/BOOKS/PHILLIPS/cips2edsrc/HIST.C

Image Filtering

- Changing pixel values by doing a **convolution** between a kernel (filter) and an image.



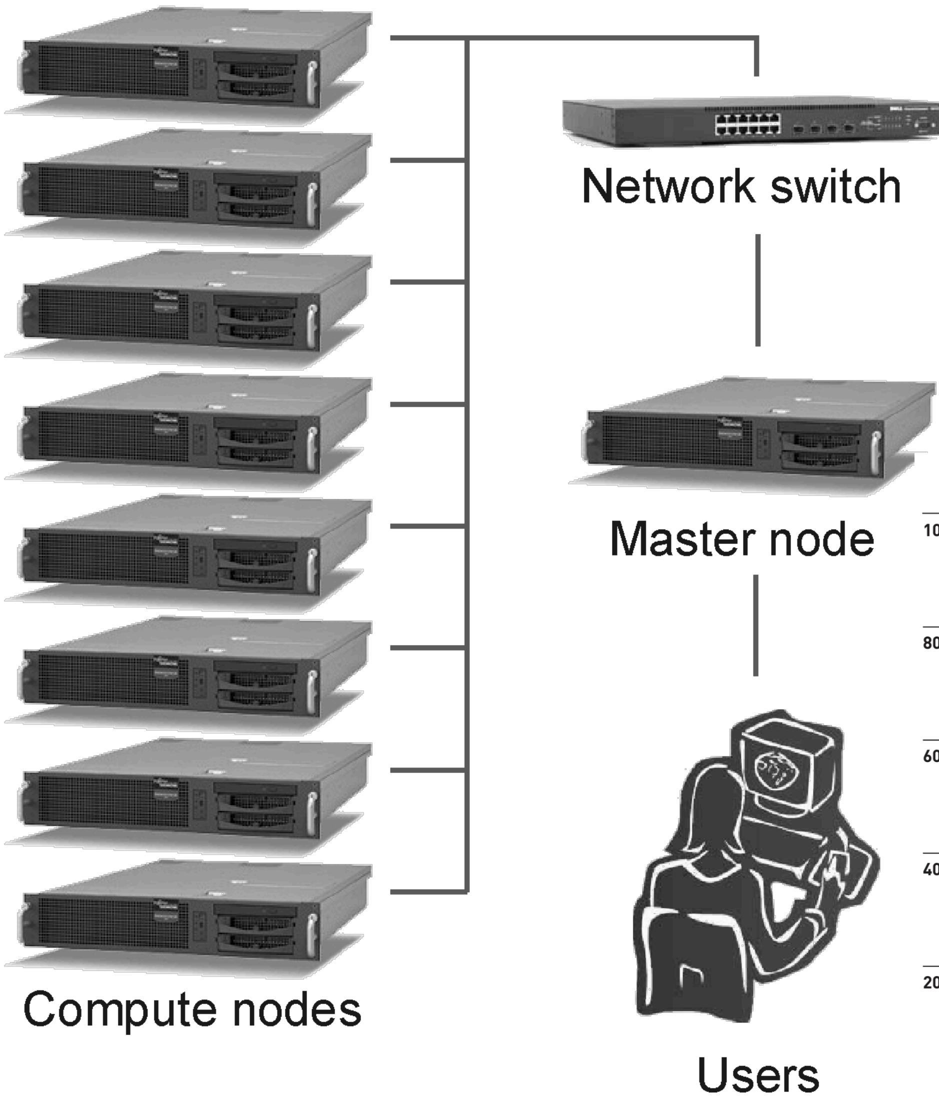

```
for(i=1; i<rows-1; i++){
   if( (i%10) == 0) printf("%d ", i);
   for(j=1; j<cols-1; j++){
      sum = 0;
      for(a=-1; a<2; a++){
         for(b=-1; b<2; b++){
            sum = sum +
                  the_image[i+a][j+b] *
                  filter[a+1][b+1];
         }
      }
      sum            = sum/d;
      if(sum < 0)    sum = 0;
      if(sum > max)  sum = max;
      out_image[i][j]    = sum;

   }  /* ends loop over j */
}  /* ends loop over i */
```

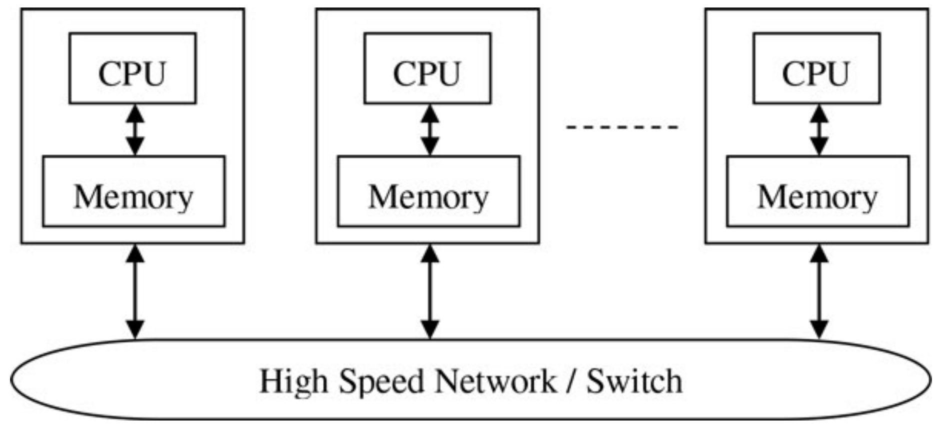
Computation that Fit for GPUs

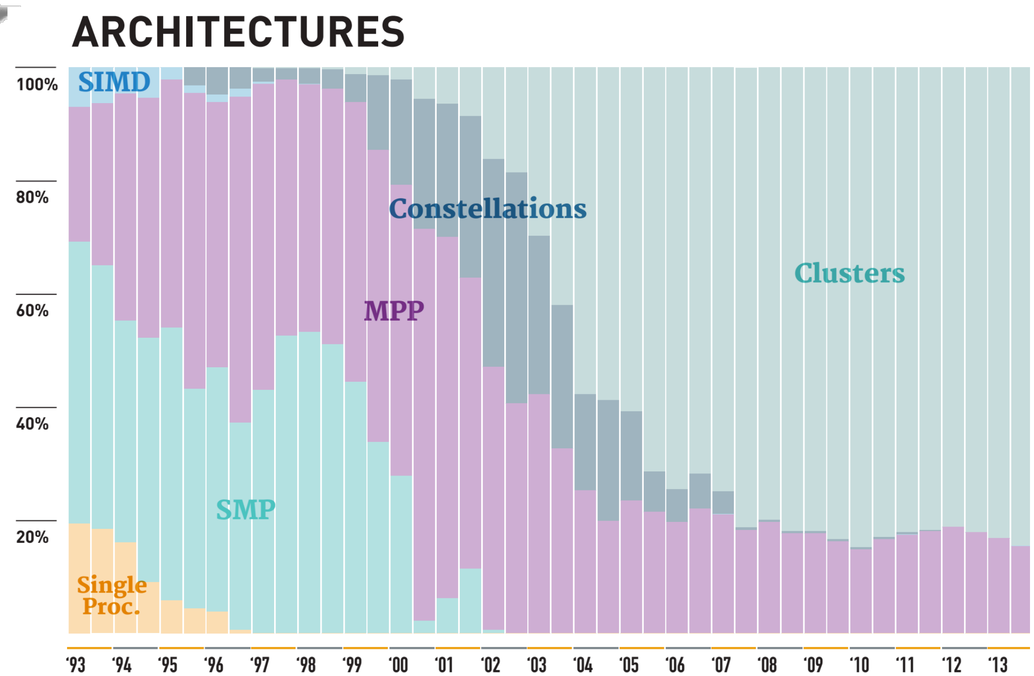
- **GPU vs CPU ←→ think of ants vs bulls**
  - GPU cores are much simpler than CPU cores
    - A single GPU core is much slower than a CPU core
  - GPU has much more number of cores than CPU
    - GPU could be much faster than a CPU

- **Same or similar operations on large number of elements**
  - **1-D, 2-D, 3-D matrices etc**
- **No data dependency between processing those elements**

Distributed Memory Systems

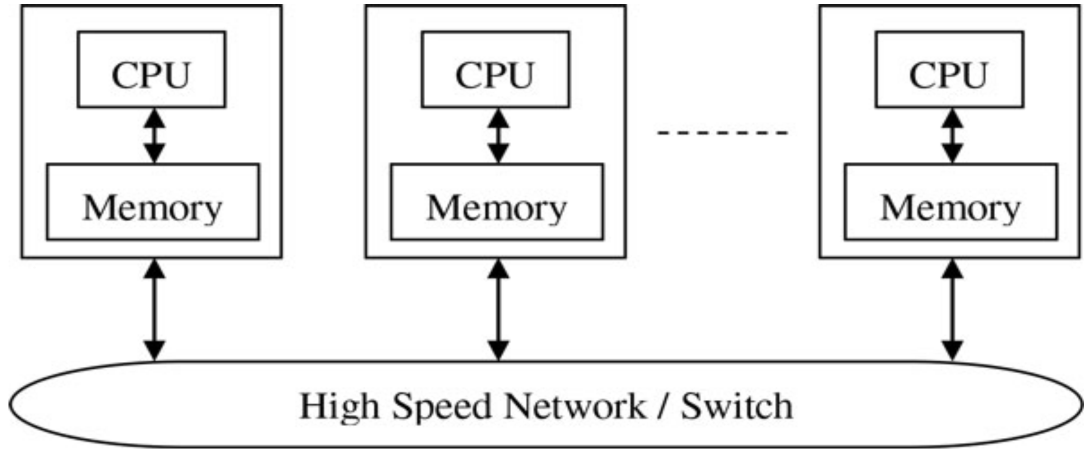






Computing Clusters

Distributed Memory System and Message Passing



- Memories are distributed
- Data have to be explicitly moved

- The standard for programming distributed memory system for HPC
  - Single Program Multiple Data (SPMD)
  - MPI (Message Passing Interface)

- Enterprise and big-data:
  - Hadoop, Spark, etc

SPMD and MPI

- SPMD: Single Program Multiple Data
- MPI: Message Passing Interface

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

int main( int argc, char *argv[] ){
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

```
I am 0 of 4
I am 2 of 4
I am 1 of 4
I am 3 of 4
```

```
mpicc mpihello.c
mpd&
mpirun -np 4 ./a.out
```

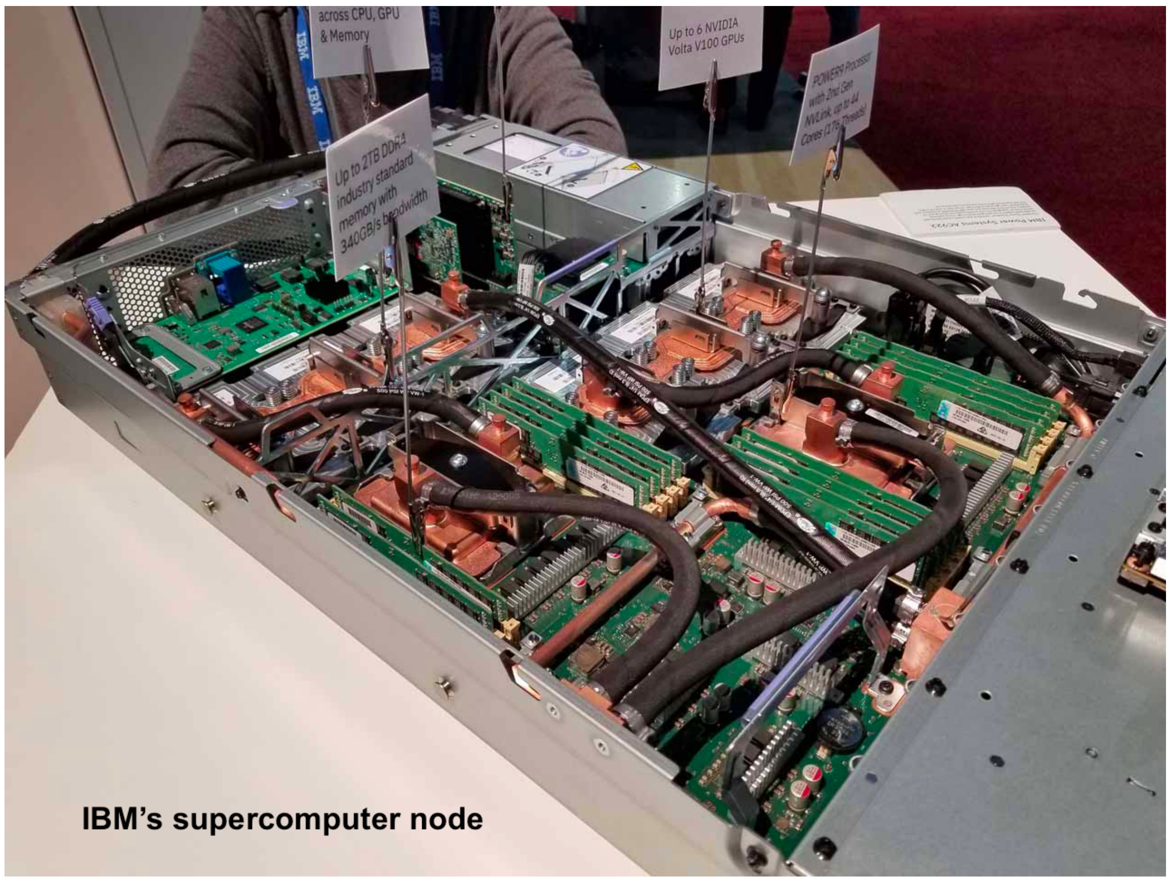
A Simple MPI Program

```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[])
{
  int rank, buf;
  MPI_Status status;
  MPI_Init(&argv, &argc);
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );

  /* Process 0 sends and Process 1 receives */
  if (rank == 0) {
    buf = 123456;
    MPI_Send( &buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  } else if (rank == 1) {
    MPI_Recv( &buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
          &status );
    printf( "Received %d\n", buf );
  }

  MPI_Finalize();
  return 0;
}
```

SPMD Model

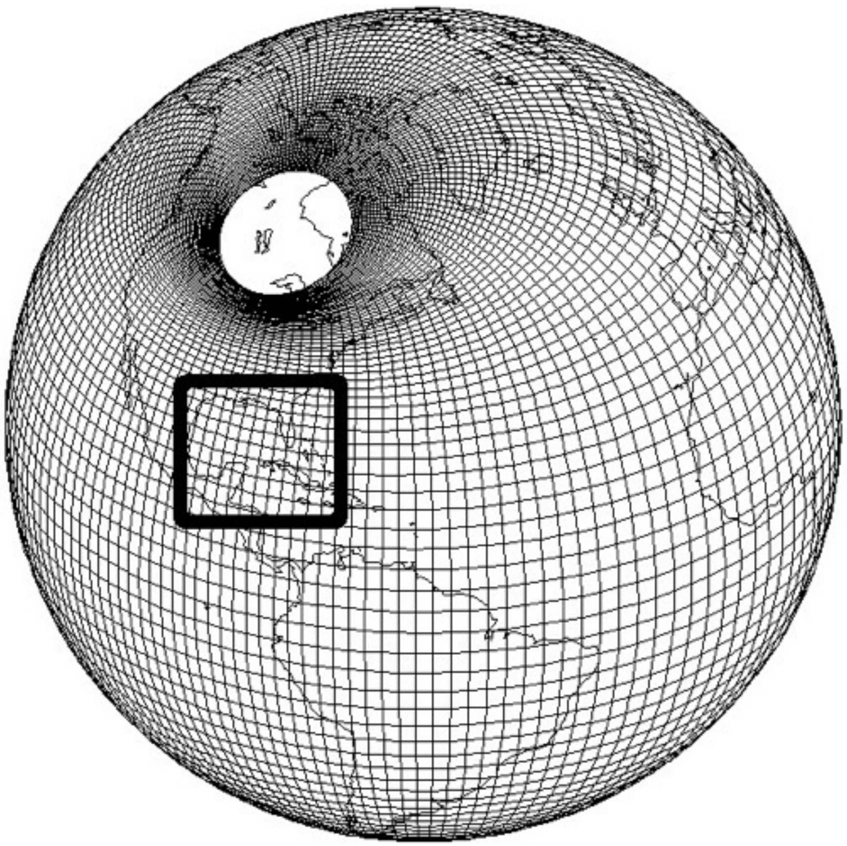
Using Two or Three Memory Systems the Same Time: The Reality

- The real systems are much more complicated
  - It is often that user need to use MPI/OpenMP/CUDA the same time



IBM’s supercomputer node

Most Computational Simulation Applications



```
MPI_Init(argc, argv); //MPI SPMD parallel
//domain decomposition
/**** the main simulation loop ****/
#pragma omp parallel ... //OpenMP threading
while (not converge && < time_steps) {
  //computing for subdomains
  #pragma omp for ... //OpenMP worksharing
  for (...)//math model-based computation
  //communication such as boundary exchange

  ...

  //optional I/O or instrumentation for
  //in-situ processing
}
```

B): MPI/OpenMP Skeleton Code of Parallel Iterative Methods for Computational Simulation

Using Domain-Specific Framework and Batch Processing

- Data processing for simple embarrassing parallelism
  - MapReduce: Hadoop and Spark
- Python/R: job or process based parallel processing
  - A completely new process will be created
- Docker and virtual machine
  - Isolated containers or VM to achieve parallel batch processing
- Resource management is the key for Python/R and docker/VM approach

Summary

- Understand the fundamental HPC helps you choose the right software and hardware for your applications

- For writing your own code
  - Parallelism is the starting point
  - Memory is the key for both correctness and performance
    - Could become very ugly

- It is just not an easy problem
  - It has never been and it seems not going to be