Section 1
High-Performance Computing for Bioinformatics

Maoying Wu

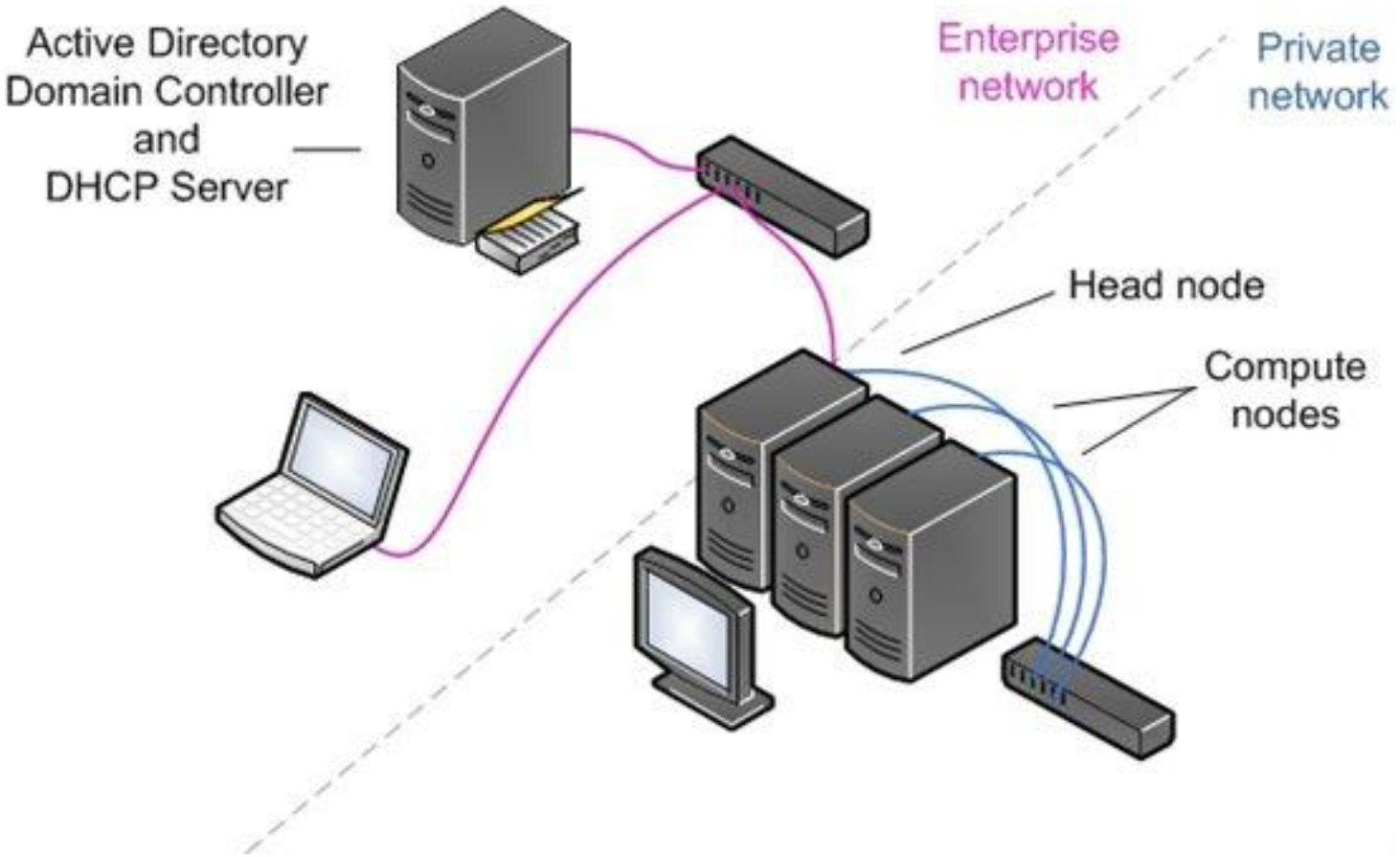
BI390 Fall 2019

Outline

- Setting Up a PC-based Computer Cluster
- Message Passing Interface (MPI) Programming
- CUDA Programming

PC-based Cluster

PC-based Clusters



Network setup

- 1 Head node: 192.168.5.10
- 5 Compute nodes: 192.168.5.11-192.168.5.15
- Netmask: 255.255.255.0
- Gateway: 192.168.5.1
- Disable the firewall and selinux
- Edit /etc/hosts to include all the nodes within the cluster.

NFS Setup

- Network File System shared within cluster
- Head node:
  - install nfs-utils and rpcbind
  - edit /etc/exports:
  - start nfs service
  - exportfs -a
- compute nodes:
  - edit /etc/fstab
  - mount -a

NIS Setup

- Network information service (NIS)
- Creation of “global” user list within a cluster (NIS Domain).
- head node:
  - service: ypserv, rpcbind
  - other settings
- compute nodes
  - service: ypbind, rpcbind

SSH setup

- Mentioned in the lab tutorial.

MPICH Setup

- Mentioned in the lab tutorial.

Torque: Portable Batch System

- A cluster's job submission system
- Allocating the job to some other node, log in as the user, and execute it
  - The script must contain cd's or absolute references to access files
- Useful Commands
  - qsub : submits a job
  - qstat : monitors status
  - qdel : deletes a job from a queue

PBS Script

PBS	Description
#PBS -N jobname	Assign a name to job
#PBS -M email_address	Specify email address
#PBS -m b	Send email at job start
#PBS -m e	Send email at job end
#PBS -m a	Send email at job abort
#PBS -o out_file	Redirect stdout to specified file
#PBS -e errfile	Redirect stderr to specified file
#PBS -q queue_name	Specify queue to be used
#PBS -l nodes=2:ppn=4	Specify MPI resource requirements
#PBS -l walltime=runtime	Set wallclock time limit

PBS Script: Example

```
#!/bin/bash
#PBS –N job_name
#PBS -o $PBS_JOBNAME.o$PBS_JOBID
#PBS -e $PBS_JOBNAME.e$PBS_JOBID
#PBS –q dque
#PBS –l nodes=4:ppn=2,walltime=02:00:00

cd $PBS_O_WORKDIR
mpirun –np 8 a.out
```

Introduction to MPI Programming

Message Passing Interface(MPI)

- A standard message passing specification for distributed memory parallel computers
  - Each processor has its own memory and cannot access the memory of other processors
  - Any data to be shared must be explicitly transmitted from one to another
- Most message passing programs use the *single program multiple data (SPMD)* model
  - Each processor executes the same set of instructions
  - Parallelization is achieved by letting each processor operation a different piece of data
  - MIMD (Multiple Instructions Multiple Data)

SPMD Example

```
int main(int argc, char **argv){
    if(process is assigned Master role){

        /* Assign work and coordinate workers and collect results */

        MasterRoutine(/*arguments*/);
     } else {  /* it is worker process */

        /* interact with master and other workers. Do the work and send results to the master*/

        WorkerRoutine(/*arguments*/);
     }
    return 0;
}
```

What we need to know?

- How many processors are working?
- What's my role in computing?
- How to send and receive data?

MPI: Basic Functions

Function	Description
int MPI_Init(int *argc, char **argv)	Initialize MPI
int MPI_Finalize()	Exit MPI
int MPI_Comm_size(MPI_Comm comm, int *size)	Determine number of processes within a comm
int MPI_Comm_rank(MPI_Comm comm, int *rank)	Determine process rank within a comm
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)	Send a message
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int src, int tag, MPI_Comm comm, MPI_Status *status)	Receive a message

Communicator

- An identifier associated with a group of processes
  - Each process has a **unique rank** within a specific communicator from **0 to (nprocesses-1)**
  - Always required when initiating a communication by calling an MPI function
- Default: **MPI_COMM_WORLD**
  - Contains all processes
- Several communicators can co-exist
  - A process can belong to different communicators at the same time

Example code: hello.c

```
#include "mpi.h”
int main( int argc, char *argv[] ) {
    int nproc, rank;
    MPI_Init (&argc,&argv); /* Initialize MPI */
    /* Get Comm Size*/
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    /* Get rank */
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    printf(“Hello World from process %d\n”, rank);
    MPI_Finalize(); /* Finalize */
    return 0;
```

How to compile?

- -I: where to find the header file
- -L: where to find the shared/static library
- mpicc mpi_code.c –o a.out
- Two widely used (and free) MPI implementations
  - MPICH (http://www-unix.mcs.anl.gov/mpi/mpich)
  - OPENMPI (http://www.openmpi.org)

Blocking (阻塞) Message Passing

- The call waits until the data transfer is done
  - The sending process waits until all data are transferred to the system buffer
  - The receiving process waits until all data are transferred from the system buffer to the receive buffer
  - Buffers can be freely reused

Blocking Message Send

MPI_Send (void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm);

• buf	Specifies the starting address of the buffer.
• count	Indicates the number of buffer elements
• dtype	Denotes the datatype of the buffer elements
• dest	Specifies the rank of the destination process in the group associated with the communicator comm
• tag	Denotes the message label
• comm	Designates the communication context that identifies a group of processes

Blocking Message Send ...

Standard (MPI_Send)	The sending process returns when the system can buffer the message or when the message is received and the buffer is ready for reuse.
Buffered (MPI_Bsend)	The sending process returns when the message is buffered in an application-supplied buffer.
Synchronous (MPI_Ssend)	The sending process returns only if a matching receive is posted and the receiving process has started to receive the message.
Ready (MPI_Rsend)	The message is sent as soon as possible.

Blocking Message Receive

MPI_Recv (void *buf, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm, MPI_Status *status);

• buf	Specifies the starting address of the buffer.
• count	Indicates the number of buffer elements
• dtype	Denotes the datatype of the buffer elements
• source	Specifies the rank of the source process in the group associated with the communicator comm
• tag	Denotes the message label
• comm	Designates the communication context that identifies a group of processes
• status	Returns information about the received message

Send and Receive: Example

```
…
if (rank == 0) {
    for (i=0; i<10; i++)  buffer[i] = i;
    MPI_Send(buffer, 10, MPI_INT, 1, 123, MPI_COMM_WORLD);
} else if (rank == 1) {
    for (i=0; i<10; i++)  buffer[i] = -1;
    MPI_Recv(buffer, 10, MPI_INT, 0, 123, MPI_COMM_WORLD,
     &status);
    for (i=0; i<10; i++)
      if (buffer[i] != i)
        printf("Error: buffer[%d] = %d but is expected to be %d\n", i,
     buffer[i], i);
 }
…
```

Non-Blocking Message Passing

- Returns immediately after the data transferred is initiated
- Allows to overlap computation with communication
- Need to be careful though
  - When send and receive buffers are updated before the transfer is over, the result will be wrong

Non-Blocking Message Passing ...

MPI_Isend (void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm, MPI_Request *req);

MPI_Irecv (void *buf, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm, MPI_Request *req);

MPI_Wait(MPI_Request *req, MPI_Status *status);

- req — Specifies the request used by a completion routine when called by the application to complete the send operation.

Blocking	MPI_Send	MPI_Bsend	MPI_Ssend	MPI_Rsend	MPI_Recv
Non-blocking	MPI_Isend	MPI_Ibsend	MPI_Issend	MPI_Irsend	MPI_Irecv

Non-Blocking Message Passing: Example

```
…
right = (rank + 1) % nproc;
left = rank - 1;
if (left < 0)        left = nproc – 1;
MPI_Irecv(buffer, 10, MPI_INT, left, 123,
    MPI_COMM_WORLD, &request);
MPI_Isend(buffer2, 10, MPI_INT, right, 123,
    MPI_COMM_WORLD, &request2);
MPI_Wait(&request, &status);
MPI_Wait(&request2, &status);
…
```

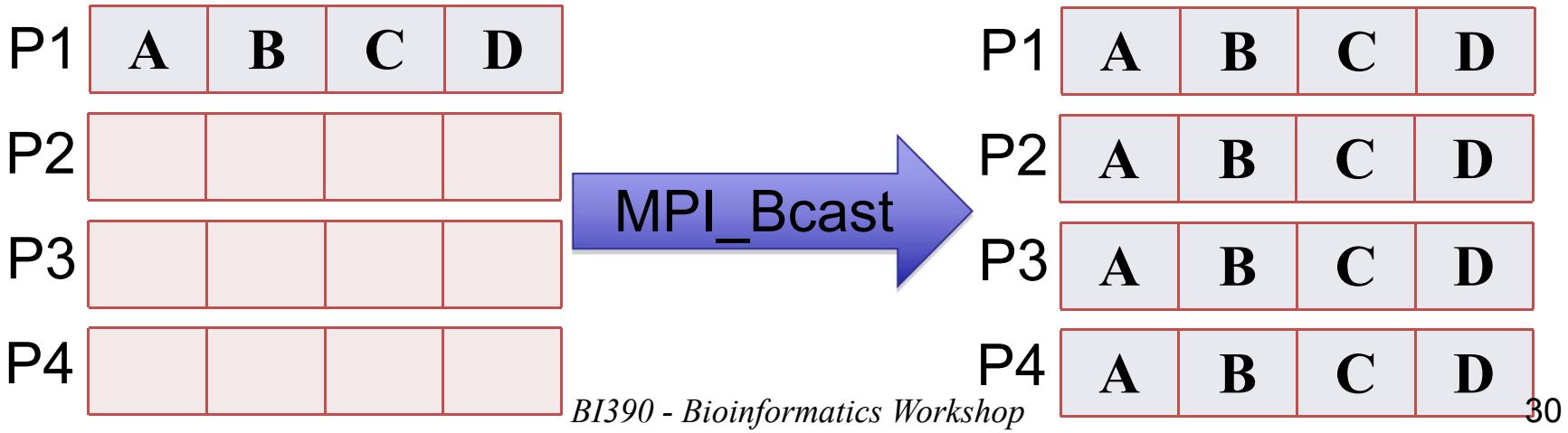
Collective Communication

- A single call handles the communication between all the processes in a communicator
- There are 3 types of collective communications
  - Data movement (e.g. MPI_Bcast)
  - Reduction (e.g. MPI_Reduce)
  - Synchronization (e.g. MPI_Barrier)

Broadcast

int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);

- One process (root) sends data to all the other processes in the same communicator
- Must be called by all the processes with the same arguments

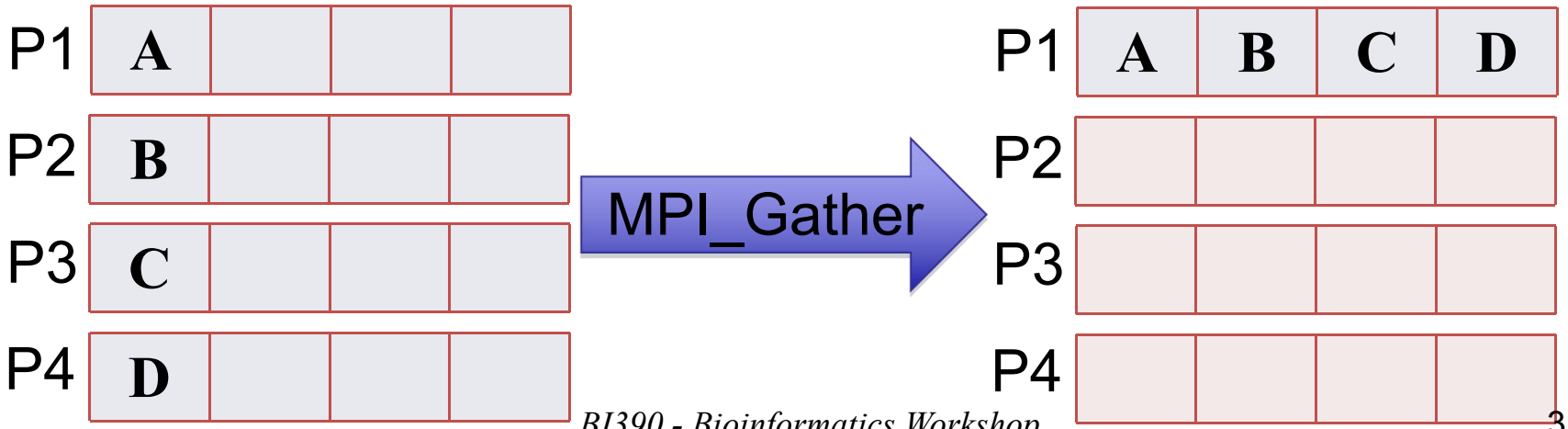


Gather

int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)

One process (root) collects data to all the other processes in the same communicator

- Must be called by all the processes with the same arguments

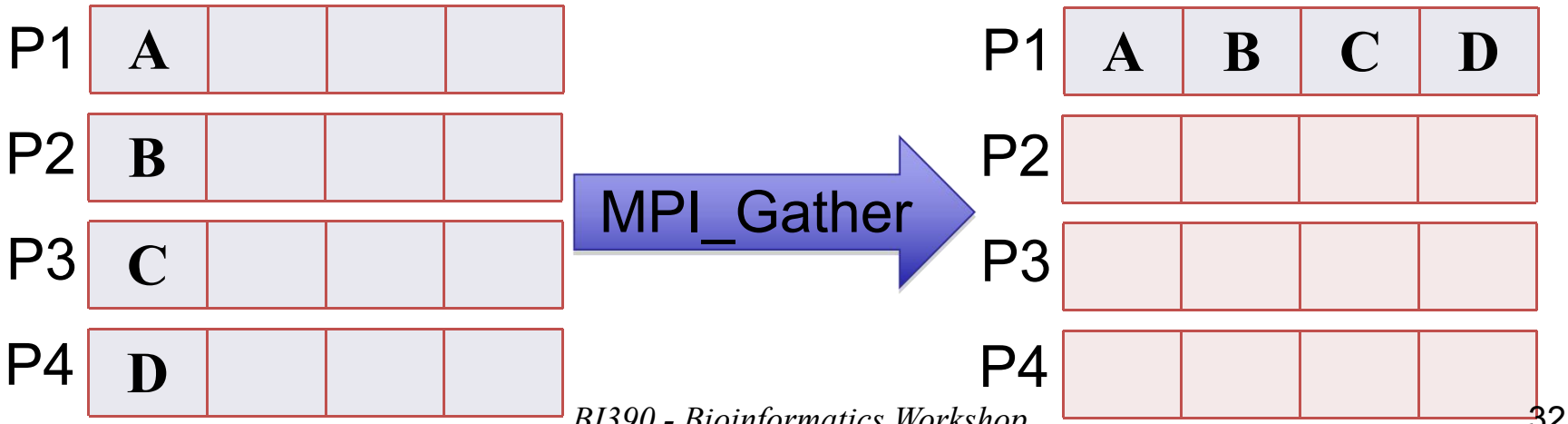


Gather to All

int MPI_Allgather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, MPI_Comm comm)

All the processes collect data to all the other processes in the same communicator

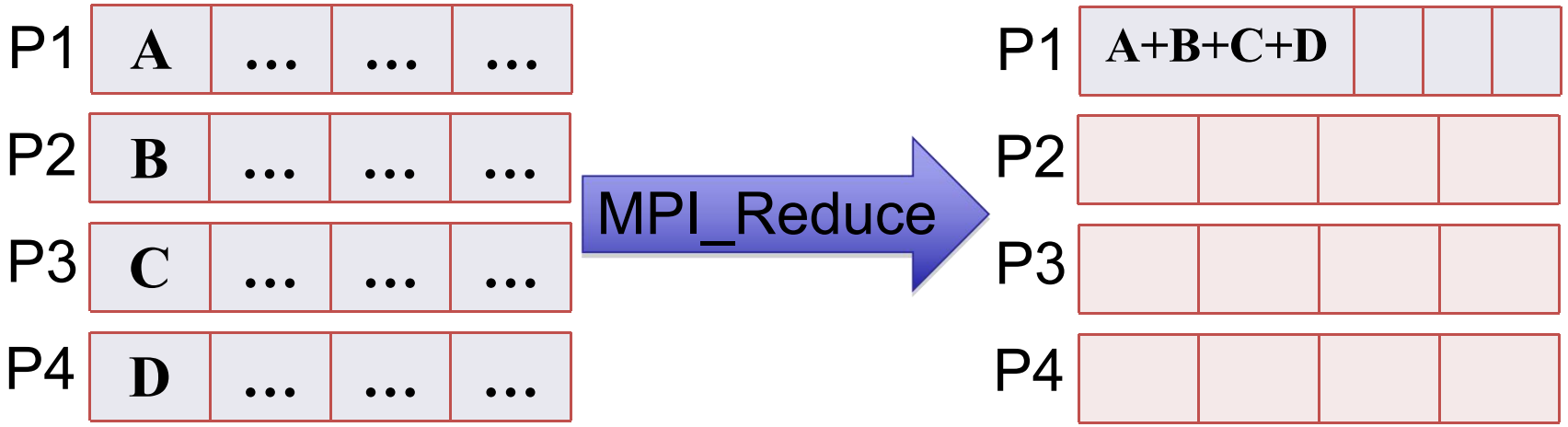
- Must be called by all the processes with the same arguments



Reduction

int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);

- One process (root) collects data to all the other processes in the same communicator, and performs an operation on the data
- MPI_SUM, MPI_MIN, MPI_MAX, MPI_PROD, logical AND, OR, XOR, and a few more
- MPI_Op_create(): User defined operator

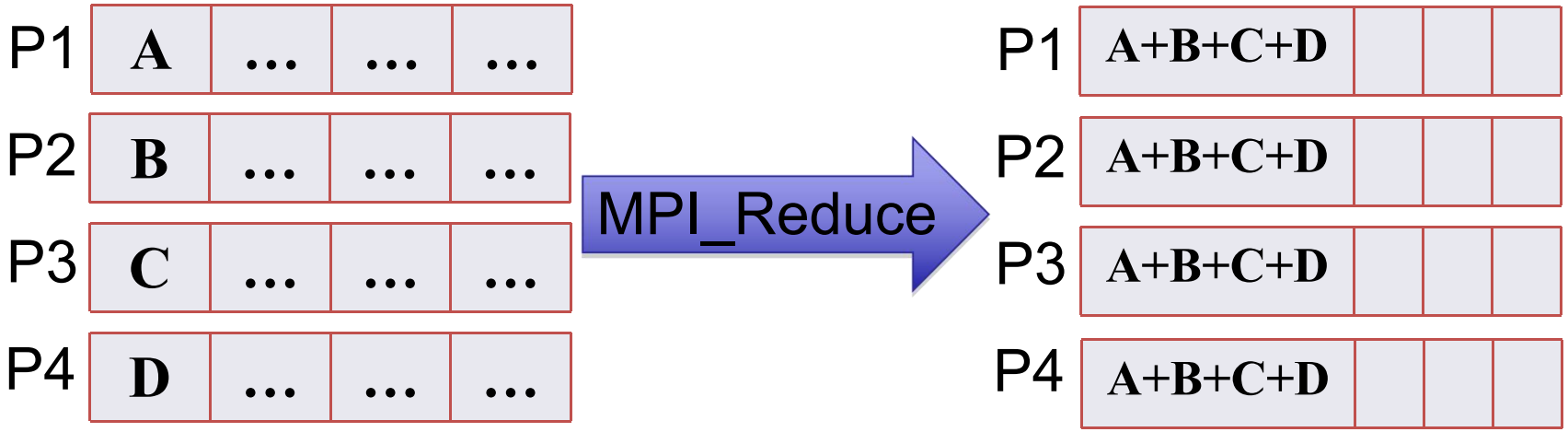


Reduction to All

int MPI_Allreduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);

All the processors collect data to all the other processes in the same communicator, and performs an operation on the data

- MPI_SUM, MPI_MIN, MPI_MAX, MPI_PROD, logical AND, OR, XOR, and a few more
- MPI_Op_create(): User defined operator



Synchronization

int MPI_Barrier(MPI_Comm comm);

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[]) {
    int rank, nprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Barrier(MPI_COMM_WORLD);
    printf("Hello, world.  I am %d of %d\n", rank, nprocs);
    MPI_Finalize();
    return 0;
}
```

How to execute MPI code?

The implementation supplies scripts to launch the MPI parallel calculation

- mpirun –np #proc a.out
- mpiexec –n #proc a.out

A copy of the same program runs on each processor core within its own process (private address space)

Communication

- through the network interconnect
- through the shared memory on SMP machines

MPI: Reference

- http://www.mpi-forum.org
- http://www.llnl.gov/computing/tutorials/mpi/
- http://www.nersc.gov/nusers/help/tutorials/mpi/intro/
- http://www-unix.mcs.anl.gov/mpi/tutorial/gropp/talk.html
- http://www-unix.mcs.anl.gov/mpi/tutorial/
- MPICH (http://www-unix.mcs.anl.gov/mpi/mpich/)
- OpenMPI (http://www.open-mpi.org/)

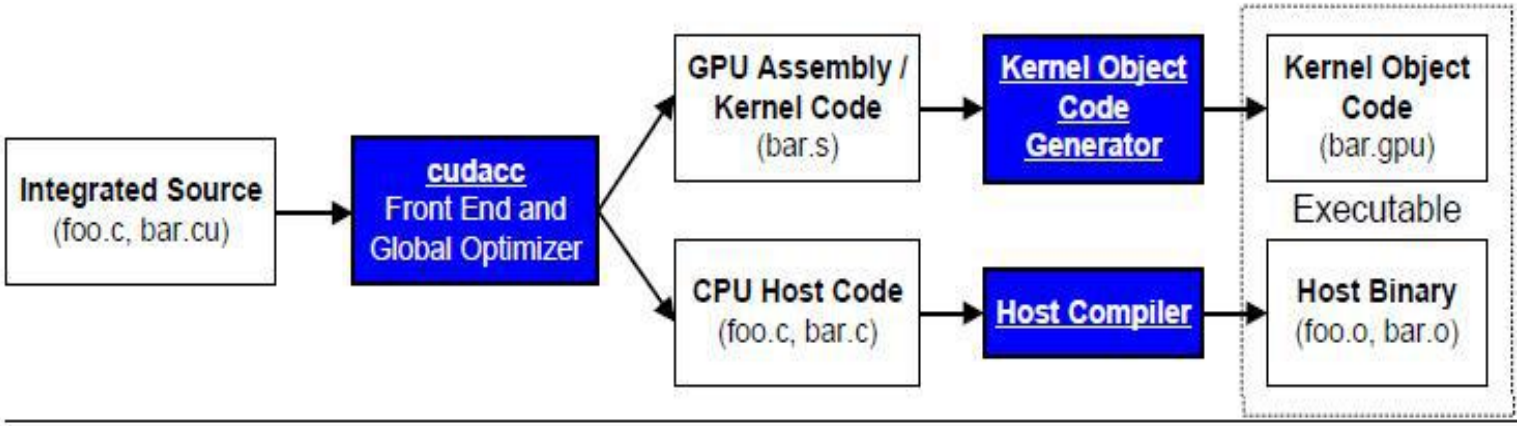
Introduction to CUDA Programming

CUDA: Overview

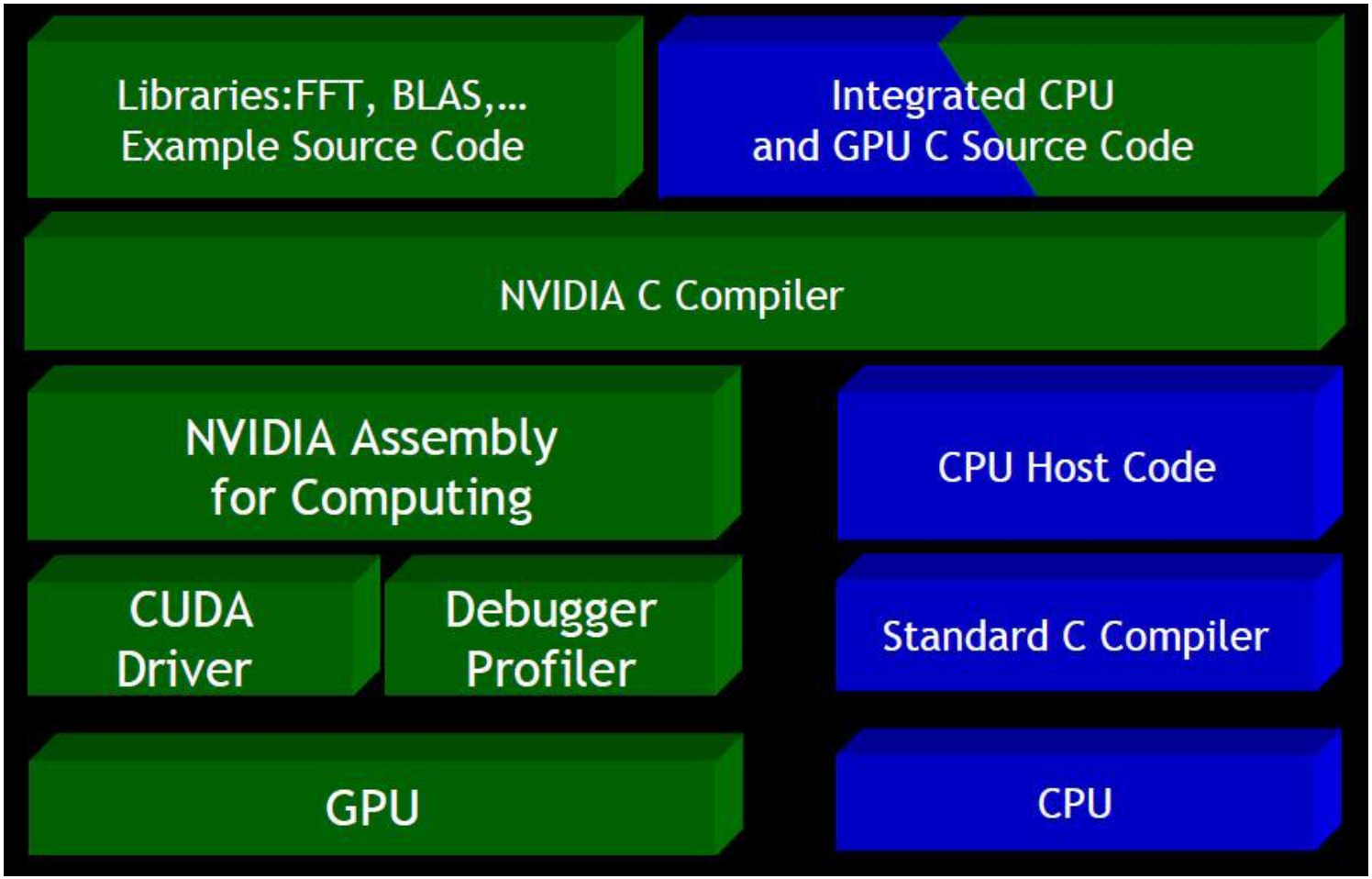
- A platform for performing massively parallel computations on graphics accelerators
- Developed by NVIDIA
- CUDA presents a unique opportunity to develop widely-deployed parallel applications
- OpenCL for AMD

CUDA Compilation

- CUDA is a programming model
- CUDA is a set of extensions to ANSI C
- CPU code is compiled by the host C compiler and the GPU code (kernel) is compiled by the CUDA compiler. Separate binaries are produced



CUDA Stack

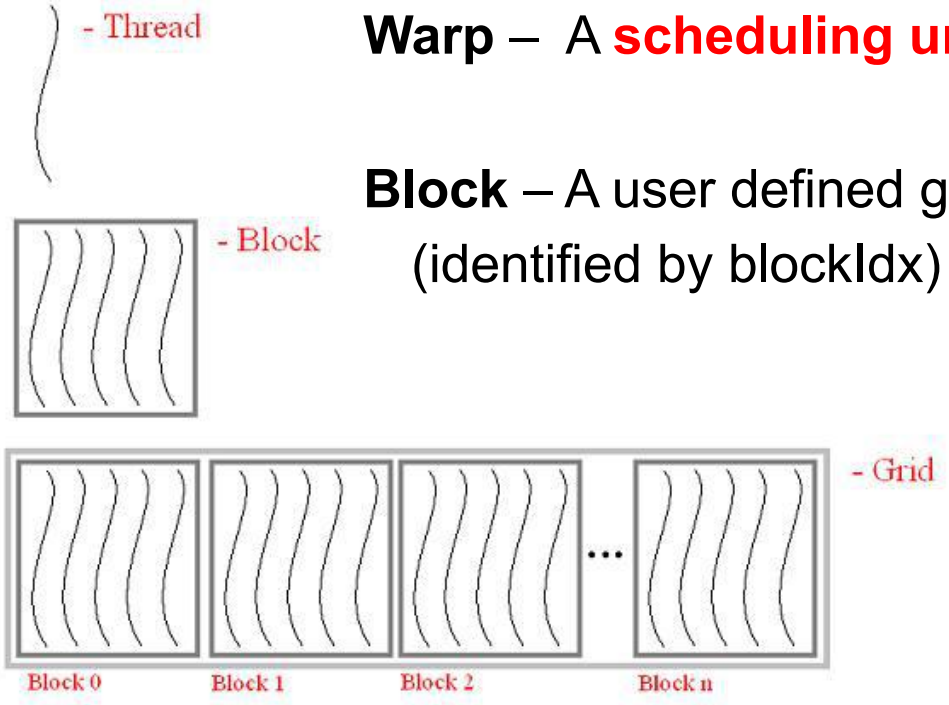

Libraries:FFT, BLAS,...
Example Source Code
Integrated CPU
and GPU C Source Code
NVIDIA C Compiler
NVIDIA Assembly
for Computing
CPU Host Code
CUDA
Driver
Debugger
Profiler
Standard C Compiler
GPU
CPU

Thread Hierarchy

Thread – Distributed by the CUDA runtime (identified by threadIdx)

Warp – A **scheduling unit** of up to 32 threads

Block – A user defined group of 1 to 512 threads. (identified by blockIdx)



Grid – A group of one or more blocks. A grid is created for each CUDA kernel function

CUDA Memory Hierarchy

- The CUDA platform has three primary memory types

  Local Memory – per thread memory for automatic variables and register spilling.

  Shared Memory – per block low-latency memory to allow for intra-block data sharing and synchronization. Threads can safely share data through this memory and can perform barrier synchronization through _ _syncthreads()

  Global Memory – device level memory that may be shared between blocks or grids

Moving Data…

CUDA allows us to copy data from one memory type to another.

This includes dereferencing pointers, even in the host's memory (main system RAM)

To facilitate this data movement CUDA provides **cudaMemcpy()**

```
cudaError_t cudaMemcpy ( void *                  dst,
                         const void *            src,
                         size_t                  count,
                         enum cudaMemcpyKind     kind
                       )
```

Parameters:

dst	- Destination memory address
src	- Source memory address
count	- Size in bytes to copy
kind	- Type of transfer

enum cudaMemcpyKind

CUDA memory copy types

Enumerator:

cudaMemcpyHostToHost	Host -> Host.
cudaMemcpyHostToDevice	Host -> Device.
cudaMemcpyDeviceToHost	Device -> Host.
cudaMemcpyDeviceToDevice	Device -> Device.

Code Example

-SAXPY in C

```
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

-SAXPY in CUDA

```
__global__ void saxpy_parallel(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)  y[i] = a*x[i] + y[i];
}
```

```
// Invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

calls the kernel, jumps to GPU

Will be explained more in depth later…

Kernel Functions

- A kernel function is the basic unit of work within a CUDA thread

- Kernel functions are CUDA extensions to ANSI C that are compiled by the CUDA compiler and the object code generator

Kernel Limitations

- There must be no recursion; there's no call stack
- There must no static variable declarations
- Functions must have a non-variable number of arguments

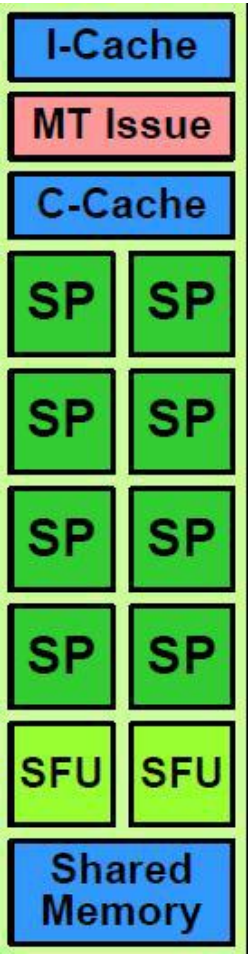
CUDA Warp

- CUDA utilizes SIMT (Single Instruction Multiple Thread)
- Warps are groups of 32 threads. Each warp receives a single instruction and “broadcasts” it to all of its threads.
- CUDA provides “zero-overhead” warp and thread scheduling. Also, the overhead of thread creation is on the order of 1 clock.
- Because a warp receives a single instruction, it will diverge and converge as each thread branches independently

CUDA Hardware

- The primary components of the Tesla architecture are:
  - Streaming Multiprocessor (The 8800 has 16)
  - Scalar Processor
  - Memory hierarchy
  - Interconnection network
  - Host interface

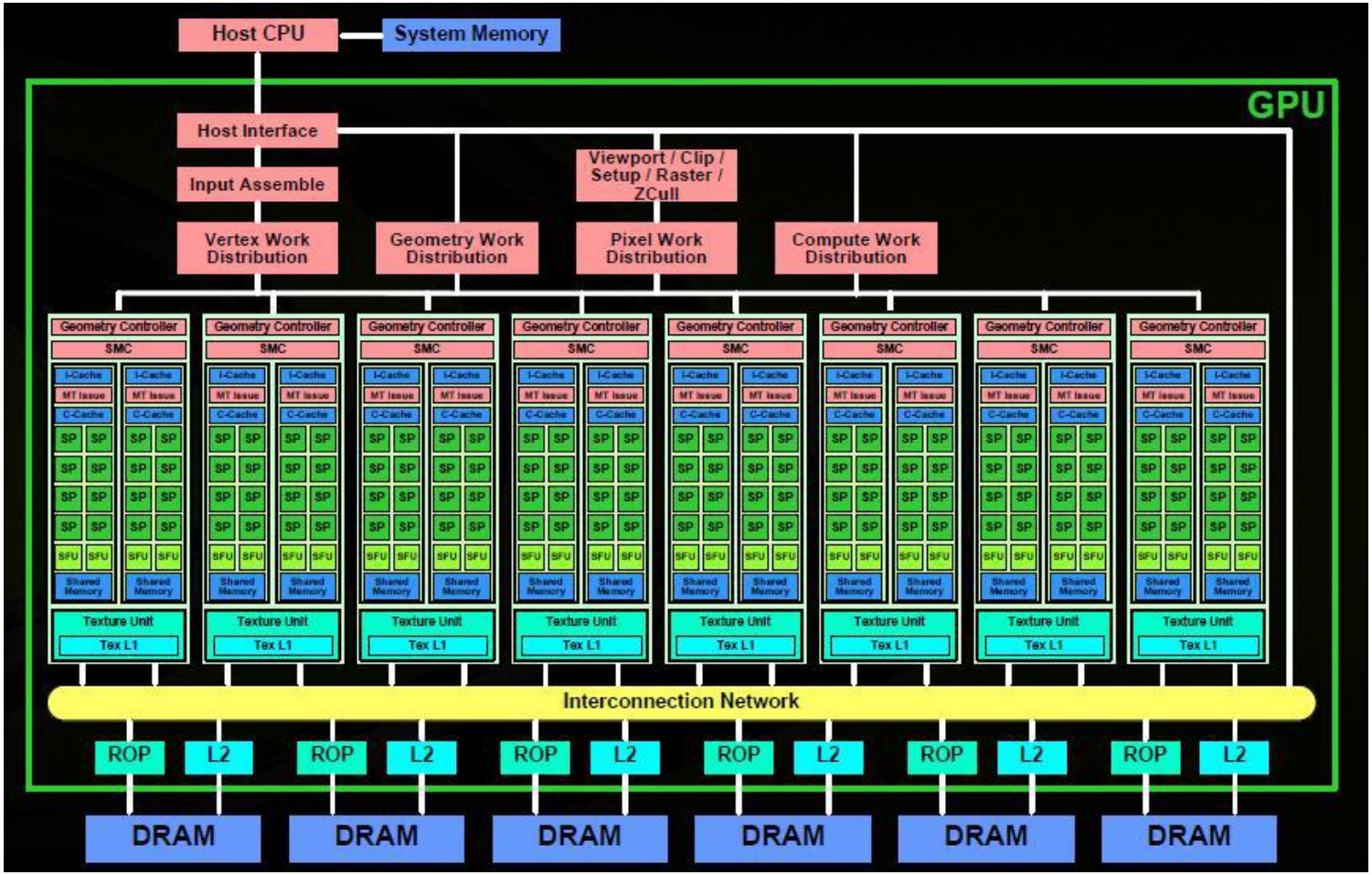
Streaming Multiprocessor (SM)



- Each SM has 8 Scalar Processors (SP)
- IEEE 754 32-bit floating point support (incomplete support)
- Each SP is a 1.35 GHz processor (32 GFLOPS peak)
- Supports 32 and 64 bit integers
- 8,192 dynamically partitioned 32-bit registers
- Supports 768 threads in hardware (24 SIMT warps of 32 threads)
- Thread scheduling done in hardware
- 16KB of low-latency shared memory
- 2 Special Function Units (reciprocal square root, trig functions, etc)

Each GPU has 16 SMs…

The GPU


Host CPU
System Memory
GPU
Host Interface
Input Assemble
Viewport / Clip / Setup / Raster / ZCull
Vertex Work Distribution
Geometry Work Distribution
Pixel Work Distribution
Compute Work Distribution
Geometry Controller
SMC
I-Cache
MT Issue
C-Cache
SP
SFU
Shared Memory
Texture Unit
Tex L1
Interconnection Network
ROP
L2
DRAM

Scalar Processor

- Supports 32-bit IEEE floating point instructions:

  FADD, FMAD, FMIN, FMAX, FSET, F2I, I2F

- Supports 32-bit integer operations

  IADD, IMUL24, IMAD24, IMIN, IMAX, ISET, I2I, SHR, SHL, AND, OR, XOR

- Fully pipelined

Code Example: Revisited

-SAXPY in C

```
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

-SAXPY in CUDA

```
__global__ void saxpy_parallel(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)  y[i] = a*x[i] + y[i];
}
```

```
// Invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

calls the kernel, jumps to GPU

Myths About CUDA

- GPUs are the only processors in a CUDA application
  - No, the CUDA platform use the CPU and GPU as the co-processor.
- GPUs have very wide (1000s) SIMD machines
  - No, a CUDA Warp is only 32 threads
- Branching is not possible on GPUs
  - Incorrect.
- GPUs are power-inefficient
  - Nope, performance per watt is quite good
- CUDA is only for C or C++ programmers
  - Not true, there are third party wrappers for Java, Python, and more