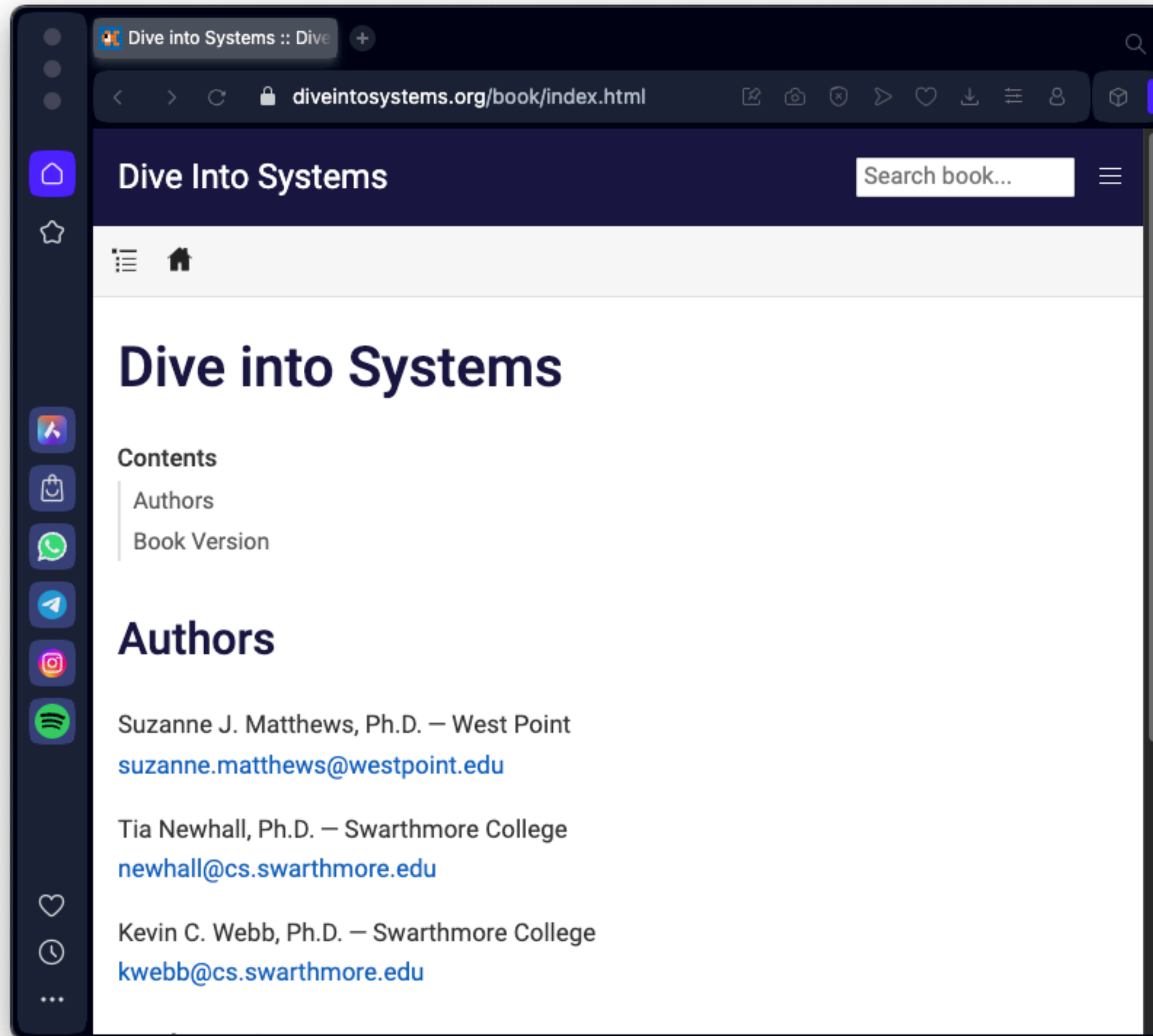
15. Looking Ahead: Other Parallel Systems and Parallel Programming Models

For COMSC 142

Sam Bowne

May 1, 2025

Free online textbook



- https://diveintosystems.org/book/index.html

Topics

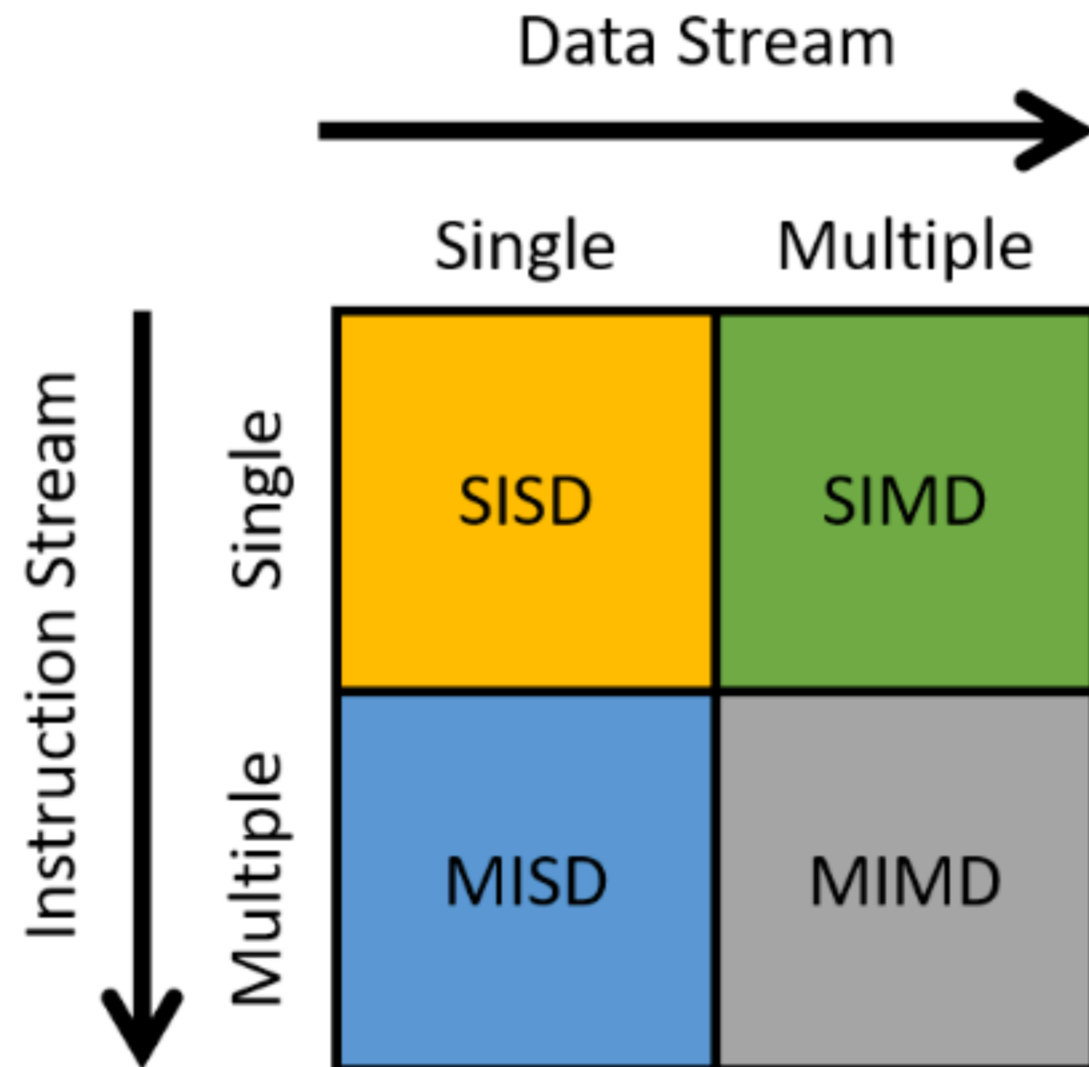
15.1. Hardware Acceleration and CUDA

15.2. Distributed Memory Systems

15.3. To Exascale and Beyond

Flynn's Taxonomy of Architecture

- **SISD**: Single instruction/ single data
- **MISD**: Multiple instruction/single data
- **SIMD**: Single instruction/multiple data
- **MIMD**: Multiple instruction/multiple data



Flynn’s Taxonomy of Architecture

- **SISD**: Single instruction/single data
  - a single control unit processing a single stream of instructions
  - most processors prior to the mid-2000s were SISD machines
- **MISD**: Multiple instruction/single data
  - rarely used anymore
- **SIMD**: Single instruction/multiple data
  - GPUs
- **MIMD**: Multiple instruction/multiple data
  - Modern multicore CPUs

15.1. Hardware Acceleration and CUDA

15.1. Heterogeneous Computing: Hardware Accelerators, GPGPU Computing, and CUDA

- **Heterogeneous computing**
  - computing using multiple, different processing units
- These processing units often have different ISAs (Instruction Set Architectures), some managed by the OS, and others not
- Typically, heterogeneous computing means support for parallel computing using the computer's CPU cores and one or more of its accelerator units such as
  - **Graphics Processing Units (GPUs)** or
  - **Field Programmable Gate Arrays (FPGAs)**

15.1.1. Hardware Accelerators

- **FPGA**
  - reprogrammable, meaning that they can be reconfigured to implement specific functionality in hardware
  - often used to prototype **application-specific integrated circuits (ASICs)**
  - typically require less power to run than a full CPU
  - May be used for
    - device controllers, for sensor data processing, for cryptography, and for testing new hardware designs

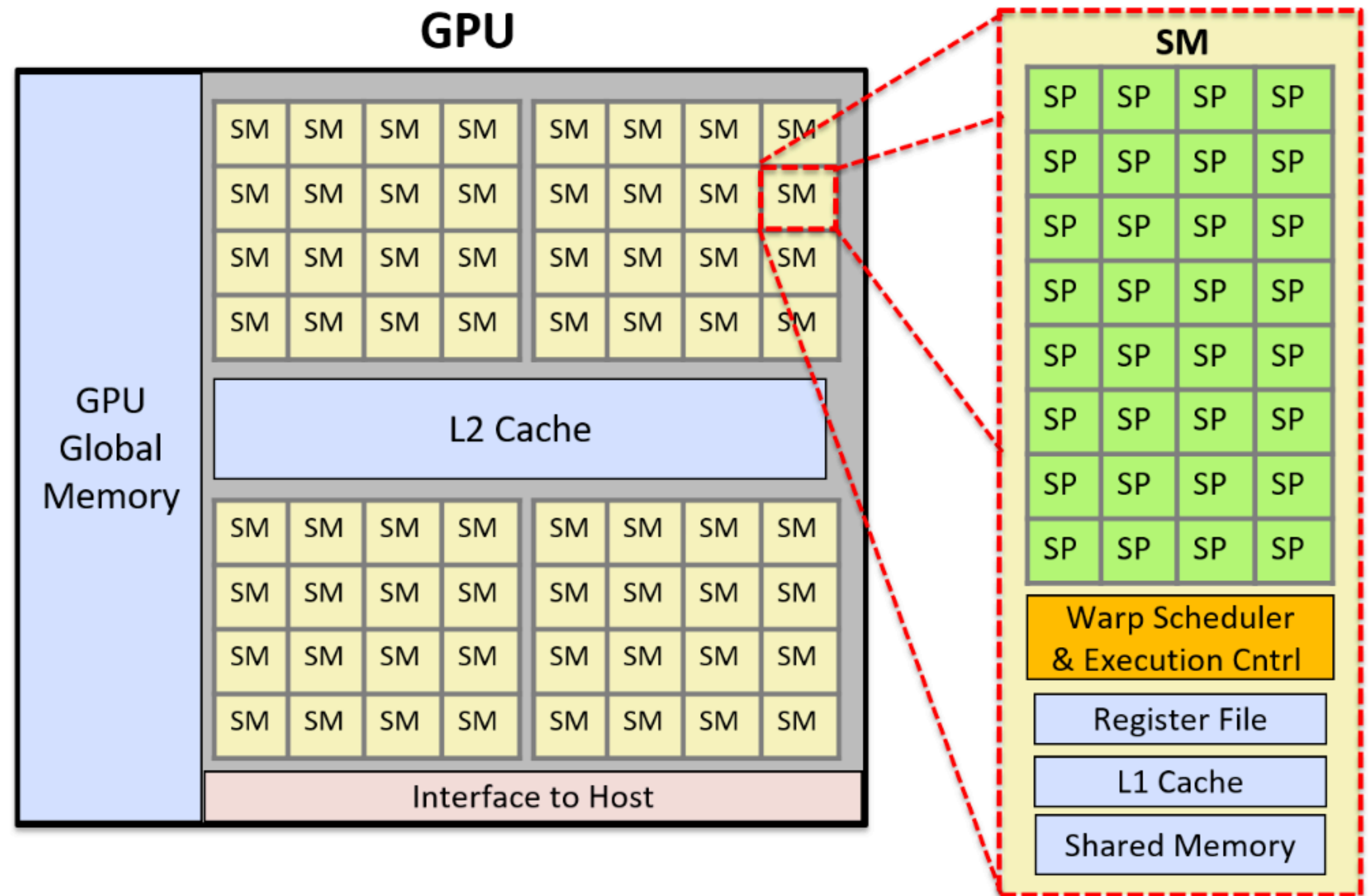
15.1.1. Hardware Accelerators

- **Cell Processors**
    - a multicore processor consisting of one general-purpose processor and multiple co-processors
    - Sony PlayStation 3 was the first Cell architecture
- **GPUs**
    - perform computer graphics computations
    - writes to a frame buffer, which delivers the data to the computer's display

15.1.2. GPU architecture overview

- Single Instruction/Multiple Thread (SIMT)

- SM: Streaming Multiprocessor
- SP: Scalar Processor
- Warp scheduler
- Warp: a set of threads
- The threads all execute the same instructions in lockstep, on different data



15.1.3. GPGPU Computing

- Applies special-purpose GPU processors to general-purpose parallel computing tasks
- Combines computation on the host CPU cores with SIMT computation on the GPU
- The host operating system does not manage the GPU's processors or memory
- CUDA library functions to explicitly allocate CUDA memory on the GPU and to copy data between CUDA memory on the GPU and host memory

15.1.4. CUDA

- **CUDA (Compute Unified Device Architecture)**
  - NVIDIA's programming interface for GPGPU computing
- **CUDA kernel function**
  - runs on the GPU
  - annotated with **global** to distinguish them from host functions
- **CUDA thread**
  - basic unit of execution in a CUDA program
  - scheduled in warps

CUDA memory functions

```
/* "returns" through pass-by-pointer param dev_ptr GPU memory of size bytes
 * returns cudaSuccess or a cudaError value on error
 */
cudaMalloc(void **dev_ptr, size_t size);

/* free GPU memory
 * returns cudaSuccess or cudaErrorInvalidValue on error
 */
cudaFree(void *data);

/* copies data from src to dst, direction is based on value of kind
 *   kind: cudaMemcpyHosttoDevice is copy from cpu to gpu memory
 *   kind: cudaMemcpyDevicetoHost is copy from gpu to cpu memory
 * returns cudaSuccess or a cudaError value on error
 */
cudaMemcpy(void *dst, const void *src, size_t count, cudaMemcpyKind kind);
```

Blocks and Grid

- CUDA threads are organized into **blocks**, and the blocks are organized into a **grid**
- **Grids** can be organized into one-, two-, or three-dimensional groupings of **blocks**
- Each thread is uniquely identified by its thread (x, y, z) position in its containing block's (x, y, z) position in the grid

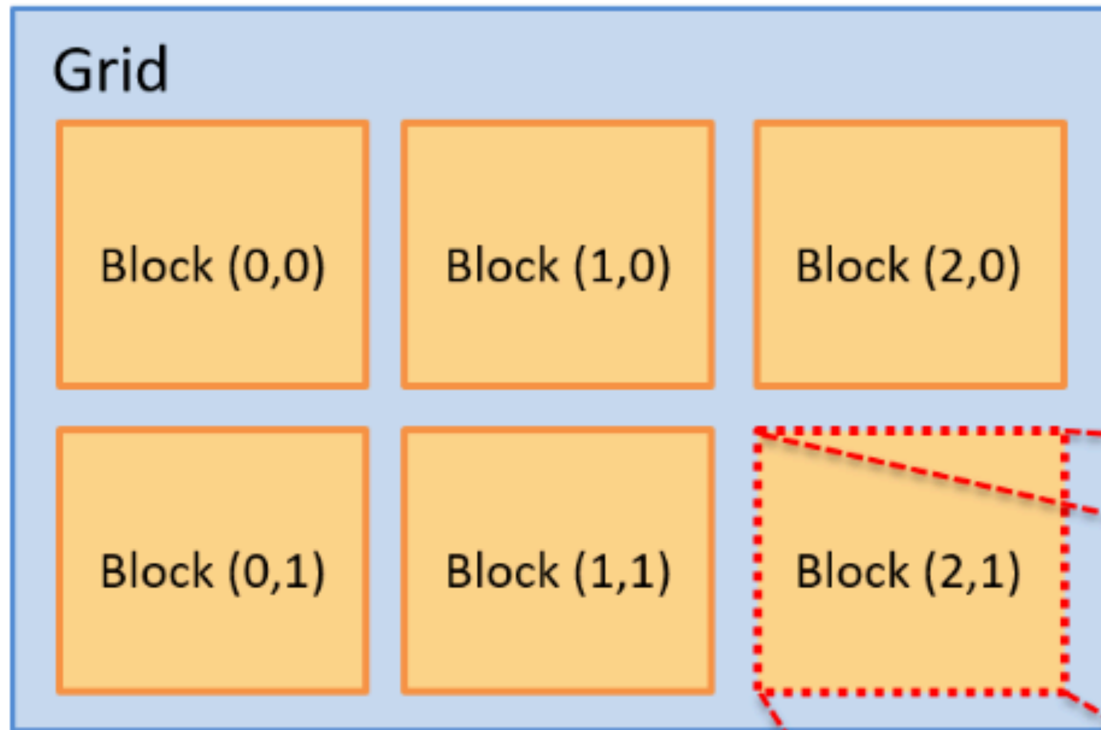
```
dim3 blockDim(16,16);   // 256 threads per block, in a 16x16 2D arrangement
dim3 gridDim(20,20);    // 400 blocks per grid, in a 20x20 2D arrangement
```

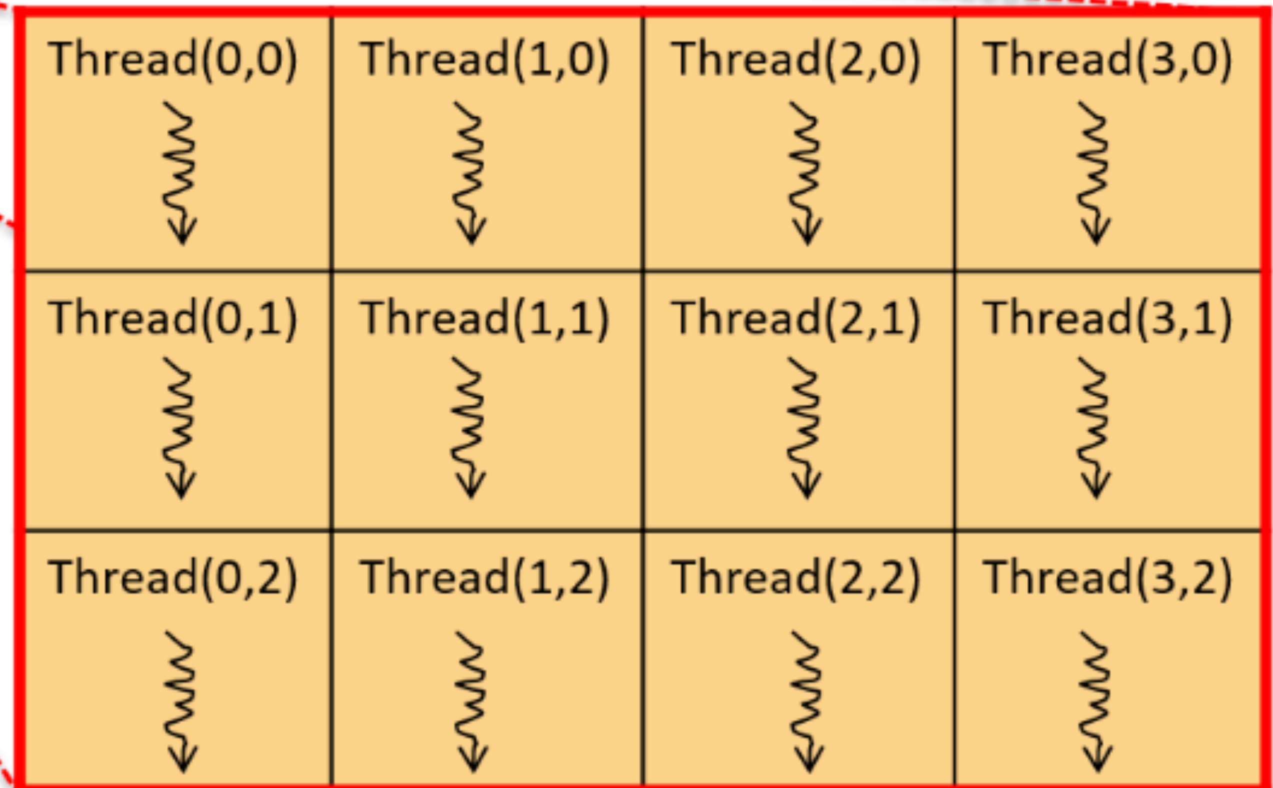
- Call a kernel function, passing parameters

```
ret = do_something<<<gridDim,blockDim>>>(dev_array, 100);
```

Blocks and Grid

```
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
```





Example CUDA Program: Scalar Multiply

```
x = a * x    // where x is a vector and a is a scalar value
```

1. Allocate host-side memory for the vector `x` and initialize it.
2. Allocate device-side memory for the vector `x` and copy it from host memory to GPU memory.
3. Invoke a CUDA kernel function to perform vector scalar multiply in parallel, passing as arguments the device address of the vector `x` and the scalar value `a`.
4. Copy the result from GPU memory to host memory vector `x`.

```
// 1. allocate host memory space for the vector (missing error handling)
vector = (int *)malloc(sizeof(int)*N);

// initialize vector in host memory
// (a user-defined initialization function not listed here)
init_array(vector, N, 7);

// 2. allocate GPU device memory for vector (missing error handling)
cudaMalloc(&dev_vector, sizeof(int)*N);

// 2. copy host vector to device memory (missing error handling)
cudaMemcpy(dev_vector, vector, sizeof(int)*N, cudaMemcpyHostToDevice);

// 3. call the CUDA scalar_multiply kernel
// specify the 1D layout for blocks/grid (N/BLOCK_SIZE)
//    and the 1D layout for threads/block (BLOCK_SIZE)
scalar_multiply<<<(N/BLOCK_SIZE), BLOCK_SIZE>>>(dev_vector, scalar);

// 4. copy device vector to host memory (missing error handling)
cudaMemcpy(vector, dev_vector, sizeof(int)*N, cudaMemcpyDeviceToHost);

// ...(do something on the host with the result copied into vector)

// free allocated memory space on host and GPU
cudaFree(dev_vector);
free(vector);
```

```
__global__ void scalar_multiply(int *array, int scalar) {

  int index;

  // compute the calling thread's index value based on
  // its position in the enclosing block and grid
  index = blockIdx.x * blockDim.x + threadIdx.x;

  // the thread's uses its index value is to
  // perform scalar multiply on its array element
  array[index] = array[index] * scalar;
}
```

CUDA Thread Scheduling and Synchronization

- All threads in a warp execute the same set of instructions in lockstep
- CUDA guarantees that all threads from a single kernel call complete before any threads from a subsequent kernel call are scheduled

15.1.5. Other Languages for GPGPU Programming

- **OpenCL**, **OpenACC**, and **OpenHMPP** are three examples of languages that can be used to program any graphics device (they are not specific to NVIDIA devices)
- **OpenCL (Open Computing Language)**
  - similar programming model to CUDA's
  - targets a wide range of heterogeneous computing platforms that include a host CPU combined with other compute units
    - could include CPUs or accelerators such as GPUs and FPGAs

15.1.5. Other Languages for GPGPU Programming

- **OpenACC (Open Accelerator)**
  - a higher-level abstraction programming model than CUDA or OpenCL
  - programmer annotates portions of their code for parallel execution
  - the compiler generates parallel code that can run on GPUs
- **OpenHMPP (Open Hybrid Multicore Programming)**
  - another language that provides a higher-level programming abstraction for heterogeneous programming

Kahoot!

Ch 15a

15.2. Distributed Memory Systems

15.2. Distributed Memory Systems, Message Passing, and MPI

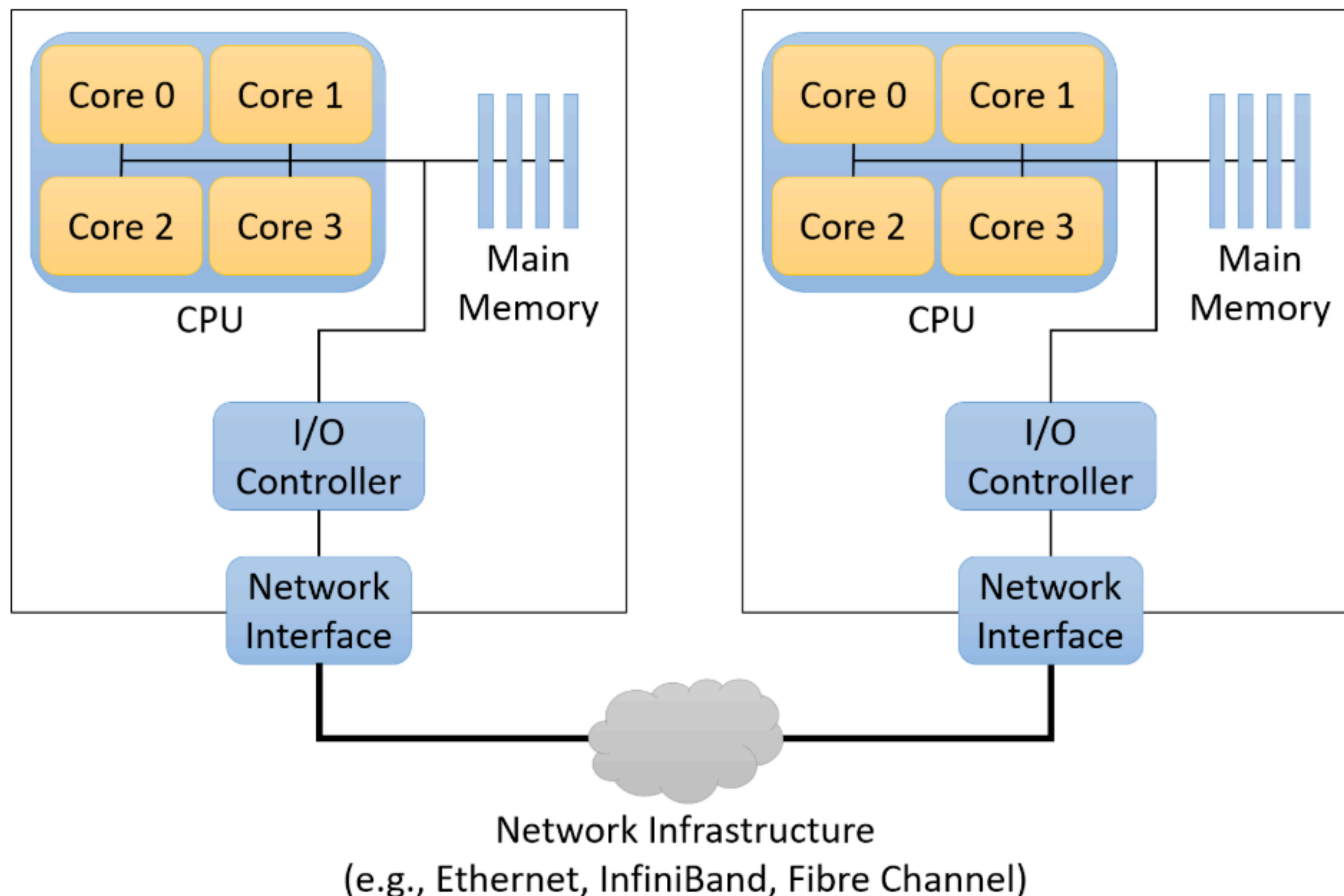
- **Distributed Memory System**
  - or **Distributed System**
  - A collection of computers working together
  - distributed systems include:
    - **Supercomputer**
    - **Commodity Off-The-Shelf (COTS) cluster**

Supercomputer

- Many compute nodes are tightly coupled to a fast interconnection network
- Each compute node contains its own CPU(s), GPU(s), and memory
- Multiple nodes might share auxiliary resources like secondary storage and power supplies

Commodity Off-The-Shelf (COTS) cluster

- Typically employ a **shared-nothing architecture**
  - each node contains its own set of computation hardware (CPU(s), GPU(s), memory, and storage)



15.2.1. Parallel and Distributed Processing Models

- Client/Server
- Pipeline
- Boss/Worker
- Peer-to-Peer

Client/Server

- Extremely common model
- Example: Web server and browser clients
- **Server** process provides a service to **clients** that ask for something to be done
- **Server process** waits at well-known address
  - to receive incoming connections from clients
  - replies to requests with either a satisfactory response or an error message

Pipeline

- Divides an application into a distinct sequence of steps
  - each of which can process data independently
  - parallel processes operate on distinct data elements
- Example: producing computer-animated films
  - many frames, each of which needs similar processing
  - adding textures or applying lighting

Boss/Worker

- **Boss** process
  - acts as a central coordinator
  - distributes work among the processes at other nodes
- Works well for problems that require processing a large, divisible input
  - such as password cracking
  - each worker gets a subset of the list of possible passwords

Peer-to-Peer

- No centralized control process
- **Peer** processes self-organize the application into a structure
    - in which they each take on roughly the same responsibilities
- Example: BitTorrent file sharing
    - each peer repeatedly exchanges parts of a file with others until they've all received the entire file
- **Peer-to-peer** applications are generally robust to node failures
- They typically require complex coordination algorithms, making them difficult to build and rigorously test

15.2.2. Communication Protocols

- Processes in a distributed memory system communicate via **message passing**
  - one process explicitly sends a message to processes on one or more other nodes
- Some applications require frequent communication
  - to tightly coordinate the behavior of processes across many nodes
- Other applications communicate to divide up a large input among processes
  - and then mostly work independently

15.2.2. Communication Protocols

- A distributed application formalizes its communication expectations by defining a **communication protocol**, specifying:
  - When a process should send a message
  - To which process(es) it should send the message
  - How to format the message
- Without a protocol, an application might fail to interpret messages properly or even **deadlock**
  - two processes: each waiting for the other to send it a message
  - neither process will ever make progress

15.2.3. Message Passing Interface (MPI)

- Defines a standardized interface to communicate in a distributed memory system
- By adopting the **MPI** communication standard, applications become **portable**
  - can be compiled and executed on many different systems
- Allows a programmer to divide an application into multiple processes
  - each process has a **rank** number
    - ranges from 0 to N-1 for an application with N processes

MPI functions

- **MPI_Comm_rank**
  - finds a process’s rank number
- **MPI_Comm_size**
  - finds how many processes are executing
- **MPI_Send**
  - sends a message
- **MPI_Recv**
  - receives a message

MPI functions

- **MPI_Bcast**
  - one process sends a message to every other process
- **MPI_Scatter**
  - divides up an array and distributes the pieces among processes
- **MPI_Gather**
  - retrieves all the data to coalesce the results

15.2.4. MPI Hello World

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

int main(int argc, char **argv) {
    int rank, process_count;
    char hostname[1024];

    /* Initialize MPI. */
    MPI_Init(&argc, &argv);

    /* Determine how many processes there are and which one this is. */
    MPI_Comm_size(MPI_COMM_WORLD, &process_count);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    /* Determine the name of the machine this process is running on. */
    gethostname(hostname, 1024);

    /* Print a message, identifying the process and machine it comes from. */
    printf("Hello from %s process %d of %d\n", hostname, rank, process_count);

    /* Clean up. */
    MPI_Finalize();

    return 0;
}
```

Running MPI Code

```
$ mpicc -o hello_world_mpi hello_world_mpi.c
```

- The **mpirun** command needs to be told
  - which computers to run processes on (--hostfile)
  - how many processes to run at each machine (-np)
  - Here, hosts.txt says to create four processes across two computers, one named lemon, and another named orange

```
$ mpirun -np 8 --hostfile hosts.txt ./hello_world_mpi
Hello from lemon process 4 of 8
Hello from lemon process 5 of 8
Hello from orange process 2 of 8
Hello from lemon process 6 of 8
Hello from orange process 0 of 8
Hello from lemon process 7 of 8
Hello from orange process 3 of 8
Hello from orange process 1 of 8
```

15.2.5. MPI Scalar Multiplication

- Scalar multiplication on an array
  - using the **boss/worker** model

```
if (rank == 0) {
    /* This code only executes at the boss. */
}
```

- **MPI Communication**
  - boss sends the scalar value and the size of the array to the workers
  - boss divides the initial array into pieces and sends a piece to each worker
  - each worker multiplies the values in its piece of the array by the scalar and then sends the updated values back to the boss

Broadcasting Important Values

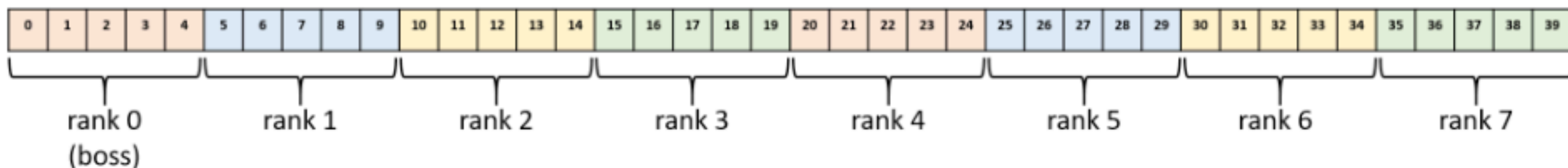
- Every process executes **MPI_Bcast**
  - but it behaves differently depending on the rank of the calling process
  - if the rank matches that of the fourth argument, the caller assumes the role of the sender
  - All other processes that call **MPI_Bcast** act as receivers

```
/* Each process determines how many processes there are. */
MPI_Comm_size(MPI_COMM_WORLD, &process_count);

/* Boss sends the total array size to every process with a broadcast. */
MPI_Bcast(&array_size, 1, MPI_INT, 0, MPI_COMM_WORLD);

/* Determine how many array elements each process will get.
 * Assumes the array is evenly divisible by the number of processes. */
local_size = array_size / process_count;
```

Distributing the Array

```c
if (rank == 0) {
    int i;

    /* For each worker process, send a unique chunk of the array. */
    for (i = 1; i < process_count; i++) {
        /* Send local_size ints starting at array index (i * local_size) */
        MPI_Send(array + (i * local_size), local_size, MPI_INT, i, 0,
                 MPI_COMM_WORLD);
    }
} else {
    MPI_Recv(local_array, local_size, MPI_INT, 0, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
}
```

Input Array



Distributed Array Pieces

Aggregating Results

```
if (rank == 0) {
    int i;

    for (i = 1; i < process_count; i++) {
        MPI_Recv(array + (i * local_size), local_size, MPI_INT, i, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
} else {
    MPI_Send(local_array, local_size, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
```

Scatter/Gather

- A simplified, easier procedure than the **for** loops in the previous two slides

```
/* Boss scatters chunks of the array evenly among all the processes. */
MPI_Scatter(array, local_size, MPI_INT, local_array, local_size, MPI_INT,
            0, MPI_COMM_WORLD);
```

```
/* Boss gathers the chunks from all the processes and coalesces the
 * results into a final array. */
MPI_Gather(local_array, local_size, MPI_INT, array, local_size, MPI_INT,
           0, MPI_COMM_WORLD);
```

15.2.6. Distributed Systems Challenges

- What if a node fails?
  - What if the network fails?
- Nodes don’t share a clock
  - Network delays may vary

15.3. To Exascale and Beyond

High-Performance Computing (HPC)

- Applications written in languages like **C**, **C++**, or **Fortran**
  - with multithreading and message passing enabled
  - with libraries such as **POSIX threads**, **OpenMP**, and **MPI**
- **High-end Data Analysis (HDA)** systems
  - process a deluge internet-based **user data** for companies like
    - Amazon, Google, Microsoft, and Facebook

HDA v. HPC

- **HPC** uses supercomputers
- **HDA** uses **data centers**

	High-End Data Analysis (HDA)	High Performance Computing (HPC)
Types of Applications	Cloud Applications; Internet Data Analysis; User Applications	Visualization, Modeling; Scientific Instrument Analysis; User Applications
Frameworks, Libraries	MapReduce (e.g. Hadoop); Apache Spark	OpenMP; MPI; CUDA, OpenCL; Numerical Libraries
Middleware	Distributed File System (HDFS, GFS); Data Storage System (HTable, BigTable); Virtual Machines / Containers	Parallel File System (e.g Lustre); Batch Scheduler (PBS, SLURM)
Operating System	Linux, Microsoft Windows	Linux
Cluster Hardware	Commodity Nodes (x86, ARM); Local Storage; Ethernet Interconnect	Specialty Nodes (x86, ARM) + GPU; Storage Area Network (SAN) + Local Storage; InfiniBand + Ethernet Interconnect

15.3.1. Cloud Computing

- Three pillars
  - **Software as a Service (SaaS)**
  - **Infrastructure as a Service (IaaS)**
  - **Platform as a Service (PaaS)**

Software as a Service (SaaS)

- Users access software in the cloud
  - Example: Gmail
- Organizations can rent service
- Services are managed completely by cloud providers
- Users don’t need to purchase or maintain servers

Infrastructure as a Service (IaaS)

- Users rent virtual machines in the cloud
  - either general purpose or preconfigured for a particular application
- Example: Amazon’s Elastic Compute Cloud (EC2)
- Users must configure applications, data, and in some cases the virtual machine’s OS itself

Platform as a Service (PaaS)

- Users develop and deploy their own web applications for the cloud
- PaaS provides a variety of languages APIs
- Examples:
  - Microsoft Azure
  - Google App Engine
  - Heroku
  - CloudBees

15.3.2. MapReduce

- Based on the mathematical operations of Map and Reduce from functional programming
- The **Map** operation applies the same function to all the elements in a collection
  - as in Python's list comprehension
- **Reduce** combines the elements, such as with sum()

```
    The typical way to perform
    scalar multiplication
'''
```

```
    for i in range(len(array)):
        array[i] = array[i] * s

    return array
```

```
    Equivalent program that
    performs scalar multiplication
    with list comprehension
```

```
    # using list comprehension
    return [multiply(x, s) for x in array]
```

The MapReduce Programming Model

- **Map** function
  - written by the programmer
  - takes an input (key, value) pair and outputs a series of intermediate (key, value) pairs
  - written to a distributed filesystem shared among all the nodes
- **Combiner**
  - defined by the MapReduce framework
  - aggregates (key, value) pairs by key, to produce (key, list(value)) pairs
- **Reduce**
  - written by the programmer
  - combines all the values to form final (key, value)
  - written to the distributed filesystem and output to the user

Word Frequency program

- Determines the frequency of each word in a large text corpus

```
void map(char *key, char *value) {
    // key is document name
    // value is string containing some words (separated by spaces)
    int i;
    int numWords = 0; // number of words found: populated by parseWords()

    // returns an array of numWords words
    char *words[] = parseWords(value, &numWords);
    for (i = 0; i < numWords; i++) {
        // output (word, 1) key-value intermediate to file system
        emit(words[i], "1");
    }
}
```

Word Frequency program

```
void reduce(char *key, struct Iterator values) {
    // key is individual word
    // value is of type Iterator (a struct that consists of
    // a items array (type char **), and its associated length (type int))
    int numWords = values.length();  // get length
    char *counts[] = values.items(); // get counts
    int i, total = 0;
    for (i = 0; i < numWords; i++) {
        total += atoi(counts[i]); // sum up all counts
    }
    char *stringTotal = itoa(total); // convert total to a string
    emit(key, stringTotal); // output (word, total) pair to file system
}
```

Word Frequency program

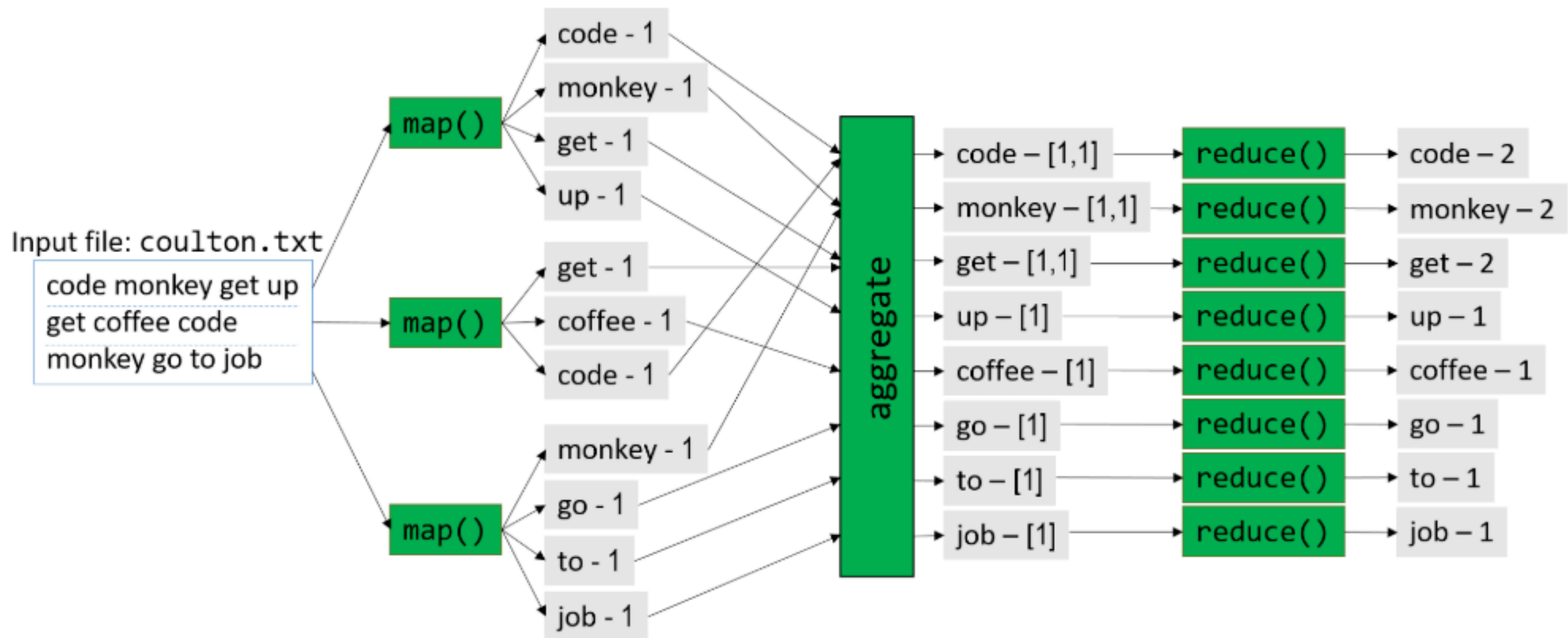


Figure 2. Parallelization of the opening lines of the song "Code Monkey" using the MapReduce framework

Fault Tolerance

- MapReduce was designed with fault tolerance in mind
- For any MapReduce run, there is one boss node and potentially thousands of worker nodes
- The chance that a worker node will fail is therefore high
- To remedy this, the boss node pings individual worker nodes periodically
    - If the boss node does not receive a response from a worker node, the boss redistributes the worker's assigned workload to a different node and re-executes the task

Hadoop and Apache Spark

- Google's implementation of MapReduce is closed source
- Yahoo! developed **Hadoop**, an open source implementation of MapReduce
  - later adopted by the Apache Foundation
- **Hadoop** project consists of an ecosystem of tools
  - Hadoop Distributed File System or HDFS
    - an open source alternative to Google File System
  - HBase
    - modeled after Google's BigTable

Hadoop and Apache Spark

- **Hadoop** limitations:
  - it is difficult to chain multiple MapReduce jobs together into a larger workflow
  - writing of intermediates to the HDFS proves to be a bottleneck, especially for small jobs (smaller than one gigabyte)
- **Apache Spark** was designed to address these issues, among others.
  - up to 100 times faster than Hadoop on some applications

15.3.3. Looking Toward the Future: Opportunities and Challenges

- Centralized computing
  - new data is produced in **edge environments**
    - near sensors and other data-generating instruments
    - on the other end of the network from commercial cloud providers and HPC systems
  - scientists and practitioners gather data
    - analyze it using a local cluster, or
    - move it to a supercomputer or data center for analysis
- This centralized computing is no longer a viable strategy as improvements in sensor technology have exacerbated the data deluge
- Including **Internet of Things (IoT)** devices

Big Data

- Aggressively summarize data at each transfer point between the edge and the cloud
- Infrastructure capable of processing, storing, and summarizing data in edge environments in a unified platform
- **Edge** (or **fog**) **computing**
- Flips the traditional analysis model of Big Data
    - instead of analysis occurring at the supercomputer or data center ("last mile”)
    - analysis occurs at the source of data production ("first mile")

Convergence

- Converge the HPC and cloud computing ecosystems
  - to create a common set of frameworks, infrastructure and tools for large-scale data analysis
- **Big Data Exascale Computing (BDEC)** working group
  - views cloud computing as a "digitally empowered" phase of scientific computing
    - in which data sources are increasingly generated over the internet
  - supercomputers and data centers are "nodes" in a very large network of computing resources
    - working in concert to deal with data flooding from multiple sources
  - Each node aggressively summarizes the data flowing to it
    - releasing it to a larger computational resource node only when necessary

New fields

- Artificial intelligence and quantum computing
- New **Domain-Specific Architectures (DSAs)**
  and **Application-Specific Integrated Circuits (ASICs)**
  - such as **TPUs (Tensor Processing Units)**
- New architectures will also lead to new languages
  - and perhaps even new operating systems

Kahoot!

Ch 15b