# CS 241 Honors
# Parallel Programming

Robert Andrews

University of Illinois Urbana-Champaign

February 28, 2017

## Writing Parallel Code

### PThreads: The Good

- Explicit management of threads
- Control what data threads receive
- Mutexes, condition variables, semaphores, and barriers are provided

### PThreads: The Bad

- It's hard to write effective, bug-free code in a short amount of time.

## OpenMP

**What is OpenMP?**

A set of compiler directives for generating parallelized code from serial code

Use OpenMP by including the OpenMP header:

```c
#include "omp.h"
```

Compile by setting the `-fopenmp` flag:

```
gcc hello.c -fopenmp
```

## Hello OpenMP!

```c
#include <stdio.h>
#include "omp.h"
int main()
{
  printf("Hello, world!\n");
}
```

How do we make this execute in parallel?

## Hello OpenMP!

```c
#include <stdio.h>
#include "omp.h"
int main()
{
  #pragma omp parallel
  {
    printf("Hello, world!\n");
  }
}
```

Output:

```
Hello, world!
Hello, world!
Hello, world!
Hello, world!
```

For now, the number of threads doesn't matter; what's important is that "Hello, world!" was printed multiple times.

## Number of Threads

Can be specified in the program or as an environment variable.

```
export OMP_NUM_THREADS=4
```

```
#pragma omp parallel num_threads(4)
omp_set_num_threads(4)
```

Available inside the program:

```
int n = omp_get_thread_num_threads();
int i = omp_get_thread_num();
```

Here `n` holds the number of threads and `i` holds a unique thread id between 0 and `n−1` inclusive.

## Sharing Work Among Threads

Suppose we have a loop in our program that looks like

```
for (int i = 0; i < N; i++)
{
  big_computation(i);
}
```

How do we split this up among many threads?

## Sharing Work Among Threads

```
#pragma omp parallel
{
  for (int i = 0; i < N; i++)
  {
    big_computation(i);
  }
}
```

Not quite; this runs all `N` iterations of the loop once in each thread.

## Sharing Work Among Threads

```
#pragma omp parallel
{
  #pragma omp for
  for (int i = 0; i < N; i++)
  {
    big_computation(i);
  }
}
```

Now the computation is split among the threads. Can also use shorthand:

```
#pragma omp parallel for
{
  for (int i = 0; i < N; i++)
  {
    big_computation(i);
  }
}
```

Note that OpenMP creates an implicit barrier at the end of a `#pragma omp for` block.

## Thread-Private Variables

By default, variables are shared among threads. Create private copies using `#pragma omp private(...)`.

```
int id = 0;
int n = omp_get_num_threads();
#pragma omp parallel private(id)
{
  // id is not necessarily 0
  id = omp_get_thread_num();
  printf("I am thread %d of %d\n", id, n);
}
// what is id here?
```

Careful: the value of `private` variables is undefined at the start and end of the parallel region!

## Thread-Private Variables

Two directives fix these undefined values:

- `firstprivate` – Variables are initialized with the last value they had before the parallel region.
- `lastprivate` – Variables keep their value from the last loop (only valid in `#pragma omp parallel for`!)

```
int n = 5;
#pragma omp parallel for firstprivate(n) lastprivate(n)
{
  // n is initialized to 5 in every thread
  for (int i = 0; i < 241; i++)
  n = i;
}
// n is guaranteed to equal 240 here
```

## Reduction

Creates a thread-private variable, initializes it, and defines how the values are combined between threads.

```
int sum = 0;
#pragma omp parallel for reduction(+:sum)
{
  for (int i = 0; i < 100; i++)
  sum += i;
}
// sum == 4950
```

Other supported reduction operators: `*`, `-`, `&`, `|`, `^`, `&&`, and `||`.

Reduction variables are initialized to the identity for the operation you are using ($0$ for `+`, $1$ for `*`, `true` for `&&`, `false` for `||`, etc.).

## Synchronization

Solving the critical section problem:

```
#pragma omp critical
{
  /* Critical section code! */
}
```

How to use barriers:

```
/* line 1 */
/* line 2 */
/* line 3 */
#pragma omp barrier
/* This line will only be executed when
every thread has finished line 3 */
```

OpenMP also provides locks; Google them if you're interested.

## Scheduling

Scheduling options let you decide how loop iterations will be distributed among worker threads.

```
#pragma omp schedule(sched [, chunk])
```

Possible scheduling schemes:

- static – Evenly divide iterations among threads. Use blocks of size chunk if provided, otherwise one block per thread.
- dynamic – Divides into blocks of size 1 or chunk; assign a block to a thread when the thread is free.
- guided – Like dynamic, but starts with large blocks and then quickly decreases block size to chunk.
- runtime – Uses whatever is in the runtime variable OMP_SCHEDULE.

Examples:

```
#pragma omp schedule(static)
#pragma omp schedule(dynamic, 16)
#pragma omp schedule(runtime)
```

## MPI

### What is MPI?

MPI (Message Passing Interface) is the standard tool used for writing parallel code running on distributed-memory systems.

Use MPI by including the MPI header:

```
#include "mpi.h"
```

Compile with the MPI compiler:

```
mpicc hello.c
```

Start the MPI daemon[1]:

```
mpd --ncpus=8 &
```

Run using `mpiexec`:

```
mpiexec -n 8 ./a.out
```

[1] This should already be running on EWS

## Hello MPI!

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

int main(int argc, char ** argv)
{
  int num_procs;
  int id;

  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);

  printf("I am process %d of %d\n", rank, num_procs);

  MPI_Finalize();
}
```

## Basic Message Passing

Basic functions:

- `MPI_Send(buff, count, MPI_type, dest, tag, Comm);`
- `MPI_Recv(buff, count, MPI_type, source, tag, Comm, &stat);`

Used to send messages between processes

**Blocking Calls**

`MPI_Send` and `MPI_Recv` return only when `buff` can be used again

## Basic Message Passing

Arguments:

- buff – Pointer to a buffer of type compatible with MPI_type
- int count – Number of items in buff
- MPI_type – Usually one of MPI_DOUBLE, MPI_INT, MPI_LONG, or MPI_FLOAT
- int source – Rank of sending process (can be MPI_ANY_SOURCE
- int dest – Rank of receiving process
- int tag – Integer identifying a tag (can be MPI_ANY_TAG *only in the receiving process*)
- MPI_COMM Comm – MPI communicator (usually MPI_COMM_WORLD)
- MPI_Status stat – Structure holding status info about the message (source, tag, error, etc.)

## Basic Message Passing

Let's pass some data back and forth!

```
// setup... assume two processes
// buffer is an int array, buffer[i] == i, len == 100
MPI_Status stat;
if (rank == 0)
{
  MPI_Send(buffer, len, MPI_INT, 1, 1, MPI_COMM_WORLD);
  MPI_Recv(buffer, len, MPI_INT, 1, 2, MPI_COMM_WORLD,
    &stat);
}
else // rank == 1
{
  MPI_Recv(buffer, len, MPI_INT, 0, 1, MPI_COMM_WORLD,
    &stat);
  MPI_Send(buffer, len, MPI_INT, 0, 2, MPI_COMM_WORLD);
}
```

## Basic Message Passing

What if we did this instead?

```
// setup... assume two processes
// buffer is an int array, buffer[i] == i, len == 100
MPI_Status stat;
if (rank == 0)
{
  MPI_Send(buffer, len, MPI_INT, 1, 1, MPI_COMM_WORLD);
  MPI_Recv(buffer, len, MPI_INT, 1, 2, MPI_COMM_WORLD,
    &stat);
}
else // rank == 1
{
  MPI_Send(buffer, len, MPI_INT, 0, 2, MPI_COMM_WORLD);
  MPI_Recv(buffer, len, MPI_INT, 0, 1, MPI_COMM_WORLD,
    &stat);
}
```

Answer: deadlock!

## MPI Collectives

**MPI_Barrier(COMM);**

Creates a barrier that all processes must arrive at before any process is allowed to continue

All collectives have an implicit barrier

**MPI_Bcast(outbuff, count, MPI_type, source, COMM);**

Takes `count` elements of type `MPI_type` from `outbuff` and broadcasts their value from the rank `source` process to all other processes

## MPI Collectives

`MPI_Reduce(inbuff, outbuff, count, MPI_type, OP, dest, COMM);`

Reduce `count` values of type `MPI_type` from `inbuff` using `OP` and store the result in `outbuff` on the rank `dest` process. Some useful reductions:

- `MPI_MAX`, `MPI_MIN`
- `MPI_SUM`, `MPI_PROD`
- `MPI_LAND`, `MPI_LOR` (logical and/or)
- `MPI_BAND`, `MPI_BOR` (bitwise and/or)

`MPI_Allreduce(inbuff, outbuff, count, MPI_type, OP, COMM);`

Same as `MPI_Reduce`, but the reduction is stored on every process

## MPI Collectives

Let's compute $\sum_{i=1}^{n-1} i$.

```
// Do MPI setup: find rank, size, etc.
int i, n;
int my_sum = 0;
int total_sum = 0;
if (rank == 0)
{
  n = atoi(argv[0]);
}
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
for (i = rank * n / size; i < (rank + 1) * n / size; i++)
{
  my_sum += i;
}
MPI_Reduce(&my_sum, &total_sum, 1, MPI_INT, MPI_SUM, 0,
    MPI_COMM_WORLD);
```

## Non-blocking Communication

- `MPI_Isend(outbuff, count, MPI_type, dest, tag, MPI_COMM, request);`
- `MPI_Irecv(inbuff, count, MPI_type, source, tag, MPI_COMM, request);`
- `MPI_Wait(request, status);`
- `MPI_Test(request, flag, status);`

Like blocking communication, but introduces a new parameter: `request`.

## Non-blocking Communication

`request` is a pointer to an `MPI_Request`.

To check if the request completed, use `MPI_Test(request, flag, status)`. `flag` will be `true` if and only if the request finished.

To wait for a request to complete, use `MPI_Wait(request, status)`.

Do not modify `outbuff` or `inbuff` until you have checked the send/receive has finished.

## Non-blocking Message Passing

Let's fix the deadlock from earlier.

```
// setup... assume two processes
// buffer is an int array, buffer[i] == i, len == 100
MPI_Status stat;
if (rank == 0)
{
  MPI_Send(buffer, len, MPI_INT, 1, 1, MPI_COMM_WORLD);
  MPI_Recv(buffer, len, MPI_INT, 1, 2, MPI_COMM_WORLD,
    &stat);
}
else // rank == 1
{
  MPI_Send(buffer, len, MPI_INT, 0, 2, MPI_COMM_WORLD);
  MPI_Recv(buffer, len, MPI_INT, 0, 1, MPI_COMM_WORLD,
    &stat);
}
```

## Non-blocking Message Passing

Let's fix the deadlock from earlier.

```
MPI_Status stat;
MPI_Request req_recv, req_send;
if (rank == 0) {
  MPI_Isend(buffer, len, MPI_INT, 1, 1, MPI_COMM_WORLD,
    &req_send);
  MPI_Irecv(buffer, len, MPI_INT, 1, 2, MPI_COMM_WORLD,
    &req_recv);
  MPI_Wait(&req_send, &status);
  MPI_Wait(&req_recv, &status);
}
else { // rank == 1
  MPI_Isend(buffer, len, MPI_INT, 0, 2, MPI_COMM_WORLD,
    &req_recv);
  MPI_Irecv(buffer, len, MPI_INT, 0, 1, MPI_COMM_WORLD,
    &req_send);
  MPI_Wait(&req_send, &status);
  MPI_Wait(&req_recv, &status);
}
```

## MPI Beyond 241 Honors

This is plenty to get started, but you can go very far down the MPI rabbit hole:

- Even more communication modes
- Extensive data type support to build larger datatypes to work with MPI
- Actually use more communicators than `MPI_COMM_WORLD`

Skim through the MPI standard[2] and see if you can find something interesting.

[2]Bill Gropp, one of the original authors of MPI, is a professor here at Illinois. Ever heard of Blue Waters? That's him.

## What is CUDA?

- CUDA a C/C++ language extension created by NVIDIA to develop code for GPU's
- Allows developers to take advantage of extreme parallelism found on GPU's
- Lots of resources at `https://developer.nvidia.com/how-to-cuda-c-cpp`

EWS Labs in Siebel support writing/compiling/running CUDA code (not sure about other labs).

If you try to run CUDA code on EWS via ssh, you're going to have a bad time.[3]

[3] Your code will appear to run and return immediately without actually doing anything.

## What is CUDA?

In CUDA, we refer to the CPU executing our code as the *host* and the GPU as the *device*. Basic CUDA computations have a simple structure:

- Set up computation in host code
  - Host and device do not share memory
  - Need to explicitly move data to device
- Run computation on device code (device functions are referred to as kernels)
- Move results back to host

## Hello CUDA!

Let's write a simple program in CUDA

```
#include <stdio.h>

__global__ void hello_kernel()
{
}

int main()
{
  hello_kernel<<<1,1>>>();
  printf("Hello, world!\n");
  return 0;
}
```

Compile with the NVIDIA compiler:

```
nvcc hello.cu
```

Run as usual:

```
./a.out
```

## Hello CUDA!

```
__global__ void hello_kernel()
{
}
```

The `__global__` qualifier marks `hello_kernel` as code executed on the device and called from the host

```
int main()
{
  hello_kernel<<<1,1>>>();
  printf("Hello, world!\n");
  return 0;
}
```

The line `hello_kernel<<<1,1>>>()` calls `hello_kernel()`, which runs on the device

The `<<<1,1>>>` will be explained later

## Function Specifiers

Three function specifiers are available in CUDA:

- `__host__` – The default; code is called from and executed on the host
- `__device__` – Code is called from and executed on the device (can combine this with `__host__` to allow for code to be executed on either the device or host)
- `__global__` – Called from the host, executed on the device (a “kernel”)

## Computing with CUDA

Let's add two numbers on the device:

```
__global__ void add(int a, int b, int* c) {
  *c = a + b;
}

int main() {
  int c;
  int* dev_c;

  cudaMalloc((void**) &dev_c, sizeof(int));

  add<<<1,1>>>(241, 374, dev_c);

  cudaMemcpy(&c, dev_c, sizeof(int),
  cudaMemcpyDeviceToHost);

  printf("241 + 374 = %d\n", c); // prints 241 + 374 = 615
  cudaFree(dev_c);
  return 0;
}
```

## Computing with CUDA

This example introduces some new CUDA calls:

```
cudaMalloc(void** devPtr, size_t count);
```

Allocates count bytes on the device and stores a pointer to this memory in *devPtr.

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

Copies count bytes from dest to src. kind is one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, and cudaMemcpyDeviceToDevice.

```
cudaFree(void* devPtr)
```

Frees memory pointed to by devPtr.

## Computing with CUDA

```
__global__ void add(int a, int b, int * c)
{
  *c = a + b;
}
```

The host and device do not share memory, so we cannot directly return the sum `a + b` from this kernel. To get around this, we use `cudaMemcpy`.

Make sure you keep track of which pointers refer to device memory and which refer to host memory!

A pointer returned by `cudaMalloc` cannot be dereferenced on the host.

## Blocks and Threads in CUDA

CUDA uses blocks and threads as its units of parallelism.

- Each block runs a copy of the kernel
- Each block consists of a number of threads
- Threads within a block execute in groups (called “warps”) of 32
- Threads in a warp execute in lockstep – minimize divergent code in a warp!
- These are not your CPU’s threads! GPU threads are *very* cheap, so use lots of them

There are hardware-imposed limits on the number of blocks and threads used:

- A kernel can be run by at most 65,535 blocks
- A block can contain up to 512 threads

We’ll address how to get around this when you need more than 33,553,920 threads.

## Blocks and Threads in CUDA

Let's look at an example that uses blocks and threads:

```
__global__ void vector_add(int *a, int *b, int *c)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  c[index] = a[index] + b[index];
}
```

`blockIdx`, `blockDim`, and `threadIdx` are automatically available to us. If we wanted to use 2048 blocks with 512 threads, call this with

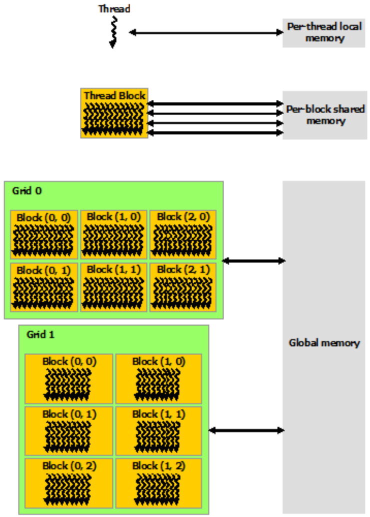
```
vector_add<<<2048, 512>>>(dev_a, dev_b, dev_c);
```

CUDA also supports 2D grids of blocks and 2D/3D blocks of threads!

```
dim3 blocks(128, 128);
dim3 threads(16, 16, 8);
vector_add<<<blocks, threads>>>(dev_a, dev_b, dev_c);
```

Access with `blockIdx.y`, `threadIdx.z`, etc.

## CUDA Memory Hierarchy



Source: NVIDIA Developer Zone CUDA Toolkit Documentation.

## CUDA Memory Hierarchy

By default, variables in kernels will be in per-thread local memory. The following specifiers are available:

- `__device__` – Global memory (also where `CudaMemcpy` and `CudaMalloc` live)
- `__constant__` – Same as global memory, but cannot be modified
- `__shared__` – Per-block shared memory
- `__local__` – Thread-local memory

Shared memory allows for communication between threads in a block

Can be useful for implementing a reduction

## Beyond 33,553,920 Threads

How can we do a vector add when each vector has 1 billion elements? Pass the vector size (more generally, the problem size) to the device!

```
__global__ void vector_add(int *a, int *b, int *c, int n)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  while (index < n)
  {
    c[index] = a[index] + b[index];
    index += blockDim.x * gridDim.x;
  }
}
```

## CUDA Barriers

A barrier for all threads in a block is as simple as one line!

```
__global__ void some_kernel()
{
  // code...
  __syncthreads();
  // more code...
}
```

## Atomics in CUDA

CUDA provides nice functionality for atomic operations:

```
__global__ void some_kernel(int *buffer)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  for (int i = index; i < index + 10; i++)
  {
    atomicAdd(&buffer[i], 1);
  }
}
```

Atomic operations include `atomicAdd`, `atomicSub`, `atomicMax`, `atomicMin`, `atomicExch`, and more!

## If you want to learn more...

The department offers a multitude of courses on parallel programming:

- CS 483: Applied Parallel Programming (offered in the Fall)
- CS 484: Parallel Programming (offered in the Spring)
- CS 498 MS3: Parallel Algorithms (offered Spring 2017)
- CS 598 EVS: Communication Cost Analysis of Algorithms (offered Fall 2016)
- CS 598 LVK: Parallel Programming (offered Fall 2016)
- CS 598 WG: Extreme Computing (offered Spring 2015 and 2016)

## Parallel Computation

What follows are slides from a previous iteration of this lecture that focused more on the theory of parallel computation. Definitions, models, and basic techniques relevant to parallel computation are covered. These extra slides are provided for those who are interested in approaching parallel computation from a mathematical viewpoint.

Disclaimer: these slides have not been updated since Fall 2016 and as such may contain errors.

## Definitions

Let $T(p)$ be the time a program takes to run on $p$ processors.

### Speedup

The *speedup* $S(p)$ of a program on $p$ processors is defined as

$$S(p) = \frac{T(1)}{T(p)}.$$

### Efficiency

The *efficiency* $E(p)$ of a program on $p$ processors is defined as

$$E(p) = \frac{S(p)}{p} = \frac{T(1)}{p \cdot T(p)}.$$

## Speedup Limitations

Suppose a program takes $t_s$ time to run the serial part of the code and $t_p$ time to run the parallel part of the code on a single processor. In an ideal world, we expect the running time on $p$ processors $T(p)$ to be

$$T(p) = t_s + \frac{t_p}{p}.$$

In particular, this gives us a lower bound on the running time. If we take the limit as $p \to \infty$, we get

$$\lim_{p \to \infty} T(p) = t_s.$$

This implies that parallelizing a program can never make it run in less than $t_s$ time.

## Speedup Limitations

Consider an ideal program which has no serial part, i.e. $t_s = 0$. Then the running time on $p$ processors is

$$T(p) = \frac{t_p}{p},$$

which results in a speedup of

$$S(p) = \frac{T(1)}{T(p)} = t_p \frac{t_p}{p} = p.$$

*Question*: Can we ever do better than this?

Surprisingly, **yes**! Superlinear speedup can occur due to caching effects, but this requires efficient hardware in addition to parallel code.

## Scaling

When discussing the performance of parallel code, there are two distinct notions of scaling: **strong scaling** and **weak scaling**.

### Strong Scaling

*Strong scaling* refers to how the execution time of the program behaves when we **vary the number of processors** and **fix the problem size**.

"We have lots of money, how well does our code perform as we throw more CPUs/GPUs at it?"

Looking at strong scaling can provide insight on how the overhead for parallelism (e.g. communication, scheduling) grows with the number of processors.

*Question*: If we plot execution time against the number of processors, what do we see in an ideal world? What do we see in the worst-case scenario?

## Scaling

### Weak Scaling

*Weak scaling* refers to how the execution time of the program behaves when we **fix the ratio of problem size to processors** and **vary the problem size**.

"If each processor does the same amount of work, how well does our code handle larger problem sizes?"

Looking at weak scaling lets us compare the rates of growth of meaningful computation and parallelism overhead.

*Question*: If we fix the ratio of problem size to processors and plot execution time against the number of processors, what do we see in an ideal world? In the worst-case scenario?

## Models of Parallel Computation

We can classify models of parallelism based on how instructions and data flow through the program. This is sometimes known as *Flynn's Taxonomy*:

| | Single Instruction | Multiple Instructions |
|---|---|---|
| Single Data | SISD | MISD |
| Multiple Data | SIMD | MIMD |

This system of classification is typically applied at the hardware level, but we can also consider programs which make use of these models of parallelism.

## Models of Parallel Computation

### SISD – Single Instruction, Single Data

In SISD, the program contains a single instruction stream and operates on a single data stream. The SISD paradigm *makes no use of parallelism!*

*Examples*: Any serial program, single-core processors.

### MISD – Multiple Instruction, Single Data

In MISD, the same set of data is operated on by different instruction streams. MISD architecture is uncommon, as SIMD or MIMD tend to lend themselves better to parallel computation.

*Examples*: (Arguably) the human brain, neural networks.

## Models of Parallel Computation

### SIMD – Single Instruction, Multiple Data

In SIMD, the same set of instructions are used to operate simultaneously on different streams of data. In particular, instructions execute in lockstep: if a branch is encountered, all processing units will follow all taken branches, performing no-ops on the branches they would not have taken otherwise.

*Examples*: GPUs, x86 SSE registers, mapping a function to data.

### MIMD – Multiple Instruction, Multiple Data

MIMD is the most general model of parallel computation: multiple different instruction streams are allowed to operate on multiple data streams. Processors have complete freedom to operate independtly from one another in both instructions executed and data operated on.

*Example*: Multi-core processors, web servers, parallel sorting.

## Models of Parallel Computation

When working in a MIMD architecture, we have to consider how the various processors access the data they operate on.

### Shared Memory

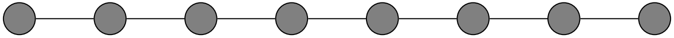
In the *shared memory* model, all processors have access to the same memory space. Communication occurs by reading and writing from this shared memory. Synchronization objects, such as locks, are typically needed to ensure correctness of the program.

### Distributed Memory

In the *distributed memory* model, each processor has its own private memory space. Processors communicate and share data by passing messages to one another over some shared network. The topology (i.e. structure) of this network can play an important role in determining the performance of your code.

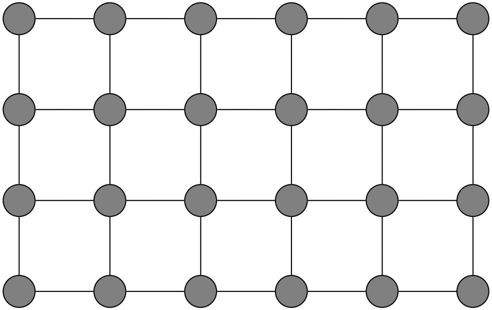
## Example Network Topologies

Path:



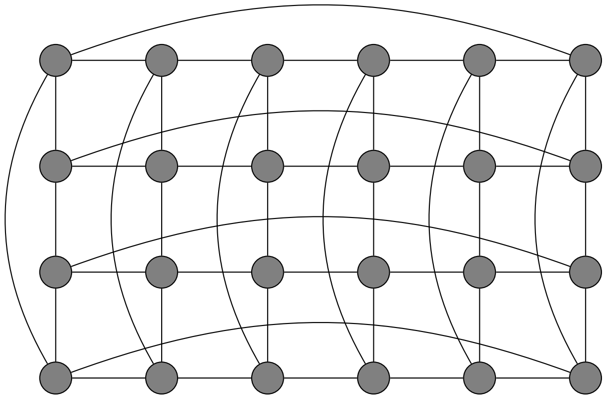
## Example Network Topologies

Grid:



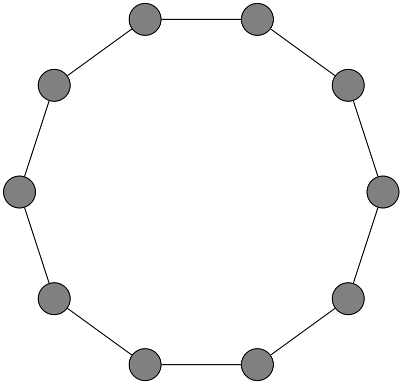
# Example Network Topologies

Torus (i.e. a grid with wrap-around):



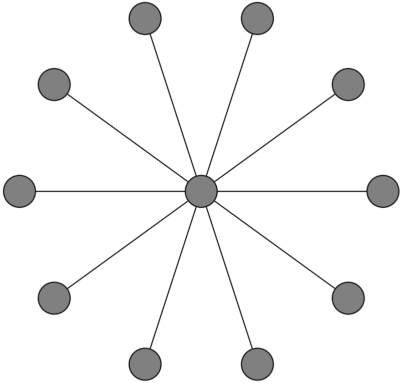
## Example Network Topologies

Ring:



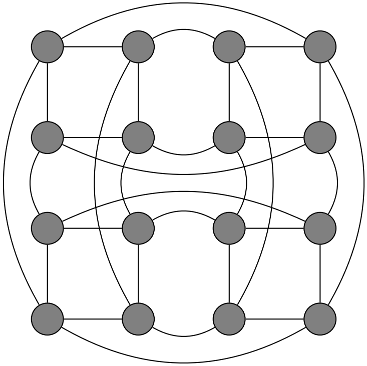
## Example Network Topologies

Star:



## Example Network Topologies

Hypercube (of dimension 4, generalizes to higher dimensions):



## Mapping

Let's revisit the problem of mapping a function to data:

**Mapping Problem**

Given a list $[a_1, a_2, \ldots, a_n]$ and a function $f$, compute the list $[f(a_1), f(a_2), \ldots, f(a_n)]$.

*Question*: Assume computing $f(a_i)$ takes approximately the same amount of time for all $i \in \{1, \ldots, n\}$. How can we efficiently solve this problem?
*Naïve approach*: Spawn one thread for each $a_i$ and compute $f(a_i)$ in that thread.

*Question*: Under what circumstances does the naïve approach show good performance? Good performance?
*Answer*: The naïve approach will perform well exactly when spawning $n$ threads is cheap.

## Mapping

*Question*: Suppose you are solving this on a CPU with $k$ cores, where $k \ll n$. How can we can avoid the costs of context switching $n$ threads?

*Answer*: Spawn exactly $k$ threads, assigning $n/k$ mapping tasks to each thread.

This demonstrates a common theme in parallel programming: an optimal solution often must take into consideration the hardware on which it will run.

## Histogram

### Histogram Problem

Fix some alphabet $\Sigma$ and consider a string $w = a_1 a_2 \cdots a_n$ where each $a_i \in \Sigma$, i.e. $w$ has length $n$ and each character in $w$ is taken from the alphabet $\Sigma$. For each $c \in \Sigma$, compute the number of times $c$ appears in $w$.

*Example*: Let $\Sigma = \{\mathsf{a}, \mathsf{b}, \ldots, \mathsf{z}\}$ and $w = \mathsf{foobar}$. We should return that f, b, a, and r each occur once in $w$, o occurs twice in $w$, and all other characters never occur in $w$.

*Question*: How can we efficiently solve this on a machine with $k$ cores?
*First Solution*: Spawn a thread for each character in $\Sigma$ and compute the number of times that character appears in $w$.

This solution is less than ideal, since each thread takes $\Theta(n)$ time to finish. With the overhead required to use parallelism, this is worse than the $\Theta(n)$ serial solution.

## Histogram

To solve this problem efficiently, we need the notion of a reduction.

### Reduction

Suppose we have $k$ threads and the $i^{th}$ thread contains some value $a_i$. Given some associative[a] and commutative[b] binary operation $\square \in \{+, \cdot, \&\&, ||, \max, \min, ...\}$, a *reduction* computes the value

$$a_1 \square a_2 \square \cdots \square a_k$$

and stores it in the memory of one or more threads.

[a] $a \square (b \square c) = (a \square b) \square c$
[b] $a \square b = b \square a$

## Histogram

*Better solution*: Spawn $k$ threads, one for each core. Split the string $w$ into $k$ parts (of roughly equal size) and assign each part to one thread. In each thread, compute a partial histogram of the corresponding substring of $w$. Perform a summation reduction on the partial histograms to obtain the final histogram.

*Analysis*: If we assume that the threads run simultaneously with minimal overhead, then we incur $\Theta(n/k)$ time to compute each partial histogram and $O(k)$ time to perform the reduction, so we compute the histogram in $O\left(\frac{n}{k} + k\right)$ time.

*Question*: Can we do better than $O(k)$ time for a reduction?

## Faster Reductions

Suppose we have values $a_1, \ldots, a_k$, where $a_i$ is in the memory of thred $i$, and we wish to compute $a_1 \square a_2 \square \cdots \square a_i$. We can parallelize the reduction process itself to compute a reduction in $O(\log k)$ time (assuming sufficient hardware).

*Fast Reductions*: Let $b_i = a_{2i-1} \square a_{2i}$, i.e., $b_1 = a_1 \square a_2$, $b_2 = a_3 \square a_4$, etc. We can compute each of the $b_i$ simultaneously, so we can compute $b_1, \ldots, b_m$ in $O(1)$ time. We repeat this process, halving the size of our data set at each step. We reduce our data set to a single element in at most $\log_2 k$ steps, so the entire reduction takes $O(\log k)$ time.

## Matrix-Vector Multiplication

Consider the problem of repeatedly multiplying a vector by some matrix.

**Matrix-Vector Multiplication Problem**

Given an integer $m$, an $n \times n$ matrix $A$ and a vector $v$ of length $n$, compute $A^m v$.

The serial algorithm takes $O(n)$ time to compute a single entry in $O(Av)$,so computing $Av$ takes $O(n^2)$ time, thus computing $A^m v$ takes $O(mn^2)$ time.

*Question*: How can we make use of parallelism to do better?

## Matrix-Vector Multiplication

*Proposed Solution*: Suppose we have $k$ cores at our disposal. Spawn $k$ threads and have each thread compute $n/k$ entries of $Av$ in $O(n^2/k)$ time, so we can compute $A^m v$ in $O(mn^2/k)$ time.

*Remark*: This algorithm is **not** correct! To compute even one entry of $Av$, we need to have all values of $v$. For example, to compute any entries of $A^m v$, we need to know $A^{m-1}v$ in its entirety.

## Matrix-Vector Multiplication

*Corrected Solution*: At the end of each step, each thread must broadcast the $n/k$ values it has computed to the other $k-1$ threads before the next step in the computation may be carried out.

*Analysis*: Assuming broadcasting a single element takes $O(1)$ time[4], this broadcast takes $O(n)$ time. From this, a single step of the algorithm takes $O(n^2/k + n)$ time, so the entire algorithm runs in $O(mn^2/k + mn)$ time.

In practice, a *barrier* is needed between each step of the computation. This ensures that each thread finishes its work and broadcasts its results to all other threads before the next step of the computation begins.

[4] This may not be true, depending on the details of the machine you are using

## Task Scheduling

Let's take a deeper look at how we assign tasks to threads.

*Example*: Suppose we working on a machine with 4 cores and want to compute the sum of the first $k$ integers[5] for $k = 1, 2, \ldots, 8$. If we spawn 4 threads and assign to the $i^{th}$ thread $k = 2i - 1$ and $k = 2i$, then we get the following running times:

| Thread | Running time |
|---|---|
| 1 | 4s |
| 2 | 16s |
| 3 | 36s |
| 4 | 64s |

*Problem*: Threads 1 and 2 spend a lot of time waiting around for threads 3 and 4 to finish.

[5]Pretend you don't know the formula $\sum_{i=1}^{k} = \frac{k(k+1)}{2}$ and instead use a `for` loop

## Task Scheduling

This is an example of the more general problem of load balancing across threads. There are several potential solutions to this problem:

- Use a more intelligent mapping of problems to threads. In the previous problem, if we map $k = 1$ and $k = 8$ to thread 1, $k = 2$ and $k = 7$ to thread 2, etc., we instead get

  | Thread | Running time |
  |---|---|
  | 1 | 37s |
  | 2 | 31s |
  | 3 | 27s |
  | 4 | 25s |

  We still spend some time waiting around, but not nearly as much as in our previous solution.

## Task Scheduling

- Use a *dynamic* scheduler: give each thread one sub-problem to work on at a time, assigning a new sub-problem to a thread when it completes its current sub-problem. This results in less time spent idling in each thread, but at the additional cost of performing the scheduling.
- Use a *guided* scheduler: this behaves like a dynamic scheduler, but each thread is initially assigned a large block of sub-problems. As threads become free, we assign sub-problems in smaller blocks, eventually approaching a block size of 1, i.e. reverting to the dynamic scheduler. This gives us the benefit of similar finishing times while attempting to avoid overhead in scheduling the initial sub-problems.

It is up to *you* to determine the best scheduling policy for your problem.

## Conway's Game of Life

*To think about at home*: How would you parallelize Conway's Game of Life?[6] To simplify the problem, try simulating an $n \times n$ grid rather than the infinite grid on which the Game of Life actually takes place. Assume cells outside your $n \times n$ grid are always dead.

[6] `https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life`