Process

An instance of the computer that is being executed. These are its components: - Executable machine language program. - Block of memory. - Descriptor of the OS resources allocated to it. - Security info. - Information about the state of the process.

Threading

- They allow programmers to divide their programs into independent tasks.
- A stream of instructions that can be scheduled to run independently of its main program.
- The hope is that when one thread blocks because it is waiting for resources, the other can run.

Processes vs Threads

- Threads exist within a process; they're like the children of the process.
- A process has at least one thread.
- If a process has more than one thread, it is multithreaded.
- Starting a thread within a process is known as **forking**.
- Terminating a thread is known as **joining**.
- Both threads and processes are units of execution or **tasks**.
- **Processes do not share memory** (each gets its own block of memory from the system).
- **Threads within a process share memory** (since they are children of the process, they have access to its resources).
- Data stored in a process's memory can be **shared or private**:
  - If **private**, only the thread that owns it can use it.

Shared Memory

- Allows processors to have access to a global address space.
- Multiple processes can operate independently but share the same memory resources.
- Changes in a memory location affected by one task are visible to others.

Uniform Memory Access (UMA) vs Non-Uniform Memory Access (NUMA)

- The time to access all memory locations is the same for all cores.
- A memory location a core is directly connected to can be accessed faster than a memory location that must be accessed through another chip.

Task Scheduling

- A **scheduler** is a program that uses a **scheduling policy** to decide which process should run next.
- It uses a selection function. The selection function considers:
  - Resources the process requires. Time the process has been waiting. The process's priority.
- Scheduling policy should try to optimize:
  - **Responsiveness** of interactive processes, **Turnaround time**, **Resource utilization**, **Fairness**

Non-Preemptive Scheduling Policies

- Each task runs to completion before the next one can run.
- **First In First Out (FIFO)**, **Shortest-Job-First (SJF)**.

Preemptive Scheduling Policies

- **Round-Robin**: Each task is assigned a fixed time before it is required to give way to the next task and move back to the queue.
- **Earliest-Deadline-First**. **Shortest Remaining Time First**

Key Terms

- **Shared resource**: A resource available to all processes in the concurrent program.
- **Critical section**: Sections of code within a process that require access to shared resources. Cannot be executed while another process is in a corresponding section of code.
- **Condition synchronization**: A mechanism ensuring that a process does not proceed until a certain condition is satisfied.

- **Starvation**: A situation where a runnable process is overlooked indefinitely by the scheduler.

Dead or Alive(lock)

Concurrent programs must be safe, meaning the program cannot enter a bad state, and alive meaning they have to make progress Two problems that can occur with concurrency: - **Deadlock**: A process is waiting for a shared resource that will never be available (e.g., another process is waiting for this process to act). - **Livelock**: Multiple processes continuously change state in response to each other without making progress.

Conditions for Deadlock

For deadlock to occur, **four conditions must hold**: 1. **Mutual Exclusion**: The program involves a shared resource protected by mutual exclusion. 2. **Hold While Waiting**: A process can hold a resource while waiting for others. 3. **No Preemption**: The OS cannot force a process to deallocate a resource it holds. 4. **Circular Wait**: P1 is waiting for a resource held by P2, and P2 is waiting for a resource held by P1.

Preventing Deadlock

To prevent deadlock, **prevent at least one of the four conditions** from occurring.

POSIX Threads

A **POSIX thread** is a thread associated with a process's shared resources. Each thread has its own: **Stack**, **Program counter**, **Registers**, **Thread ID**

Races

A **race condition** occurs when the **parent process exits before its child threads complete**. This does not allow enough time for child threads to finish execution. - Best fix for race conditions: use mutual exclusions and join the threads

Task Parallelism vs. Data Parallelism

- Task Parallelism shares the tasks among each core ie on core does the tasks on all data
- Data parallelism shares the data among each core

```
// Task Parallelism
#pragma omp parallel num_threads(4)
{
  int id = omp_get_thread_num();
  printf("T%d:A\n", id);
  printf("T%d:B\n", id);

  if (id == 0)
    printf("T0:special task\n");

  if (id == 1)
    printf("T1:special task\n");

  if(id == 2)
    printf("T2:special task\n");
}

// Data Parallelism
#pragma omp parallel num_threads(2)
{
  int id = get_thread_num()
  int my_a = id * 3;  \\ where you want the thread to start doing work
  int my_b = id * 3 + 3; \\ where it should stop doing work

  printf("T%d will process indexes %d to ");

  for (int index = my_a; index < my_b; index++)
```

```
    printf("do work\n");
}
return 0;
```

Race Conditions

A **race condition** occurs when multiple threads **simultaneously access and modify shared data**, leading to **unpredictable behavior**.

```
#pragma omp parallel
{
    global_sum += my_sum; // Potential race condition
}
```

To prevent this, we use **mutual exclusion** techniques.

Barriers

Barriers ensure that all threads reach a synchronization point before continuing execution.

Types of Barriers:

Implicit Barriers - Automatically added at the end of parallel regions. **Explicit Barriers** - Defined using `#pragma omp barrier`.

```
#pragma omp parallel
{
    compute_part();
    #pragma omp barrier // Ensures all threads finish before proceeding
    finalize_part();
}
```

Barrier Limitations:

- All threads must encounter the barrier.
- Conditional execution may lead to **illegal barriers**.

`nowait` Clause

Using `nowait` allows threads **to skip synchronization** when it is unnecessary, improving performance.

```
#pragma omp single nowait
{
    expensive_task();
}
// Other threads continue execution without waiting.
```

OpenMP Mutual Exclusion Mechanisms:

1. **Critical Directive** - Ensures exclusive execution.
2. **Atomic Directive** - Ensures atomic updates to a shared variable.
3. **Locks** - Explicit locking mechanisms.

Mutual exclusions

Mutual exclusion ensures that only **one thread at a time** accesses a critical section.

```
// critical: protects critical section, kinda slow, especially if overused
// Allows **simultaneous execution** of **different** critical sections.
#pragma omp critical
{
    shared_var += local_val;
}
// Named Critical Sections
#pragma omp critical(name1)
```

```
x = compute_x();
#pragma omp critical(name2)
y = compute_y();

// atomic
// only supports simple operations `x++`, `x--`, `x += expr`, `x = x + expr`
#pragma omp atomic
sum += value;

// locking
#include <omp.h>
static omp_lock_t mylock;

int main() {
    omp_init_lock(&mylock);

    #pragma omp parallel
    {
        omp_set_lock(&mylock);
        critical_section();
        omp_unset_lock(&mylock);
    }

    omp_destroy_lock(&mylock);
    return 0;
}
```

Key Lock Functions: - `omp_init_lock(&lock);` - `omp_set_lock(&lock);` - `omp_unset_lock(&lock);` - `omp_destroy_lock(&lock`

When to Use Which?

Atomic, Single-variable updates (fastest). **Critical**, Protects complex code sections. **Locks**, Fine-grained control over execution. **Avoid Mixing** different mutual exclusion methods. **Fairness is NOT guaranteed -** Some threads may starve. **Avoid Nesting** critical sections (deadlocks possible).

Shared Variables

- Exist in **one memory location**, accessible by all threads.

```
int x = 5;
#pragma omp parallel
{
    // All threads access the same x
}
int y = 5;
#pragma omp parallel private(y)
{
    // Each thread gets its own y (uninitialized)
}
int z = 5;
#pragma omp parallel firstprivate(z)
{
    // Like `private`, but initialized with the original value.
    // Each thread gets its own z, initialized to 5
}
```

Default Clause

Sets the default scope for all variables.

```
int x = 0, y = 0;
#pragma omp parallel num_threads(4) default(none) private(x) shared(y)
```

```
{
    x = omp_get_thread_num();
    #pragma omp atomic
    y += x;
}
```

Reductions

Reduction operations allow threads to **aggregate results** safely without manual synchronization.

Reduction Examples

```
// syntax
#pragma omp parallel reduction(<operator> : <variable list>)
int sum = 0;
#pragma omp parallel reduction(+:sum)
{
    sum += omp_get_thread_num();
}

// Multiple Variables
int x = 10, y = 10;
#pragma omp parallel reduction(+:x, y)
{
    x = omp_get_thread_num();
    y = 5;
}
```

Reduction Operations

`+` Summation. `*` Multiplication. `&` Bitwise AND. `|` Bitwise OR. `^` Bitwise XOR. `&&` Logical AND. `||` Logical OR. max/min

Parallel Summation with Reduction

Instead of using a **critical section**, reductions optimize aggregation.

```
double global_sum = 0;
#pragma omp parallel num_threads(4) reduction(+:global_sum)
{
    global_sum += compute_value(omp_get_thread_num());
}
```

Area Under a Curve (Trapezoidal Rule)

Using **reduction** to integrate a function:

```
double global_result = 0.0;
#pragma omp parallel num_threads(4) reduction(+:global_result)
{
    global_result += Local_trap(a, b, n);
}
printf("Approximate area: %f\n", global_result);
```

Work-Sharing Constructs

- Used to distribute work among threads inside a parallel region.
- **Types:**
  - `for` – Divides loop iterations across threads.
  - `single` – Assigns work to a single thread.
  - `sections` – Splits tasks into sections executed by different threads.
- There is an **implied barrier** at the exit unless `nowait` is specified.

Parallel For

- Loop iterations are divided across threads dynamically.
- The loop variable is **private** by default.
- The execution order is **non-deterministic**.

Syntax Options:

Inside an existing parallel region: "'c #pragma omp for for(i = start; i < end; i += step) { // Loop body }

#pragma omp parallel for for(i = start; i < end; i += step) { // Loop body } "'

Data Dependency & Loop-Carried Dependencies

- Parallel loops should avoid **loop-carried dependencies** (when one iteration depends on results from another).

```c
fibo[0] = fibo[1] = 1;
#pragma omp parallel for
for (i = 2; i < n; i++)
    fibo[i] = fibo[i-1] + fibo[i-2];
```

Reduction in Parallel Loops

- Reduction avoids data races when accumulating results.
- **Example: Summing values in an array**

```c
double sum = 0.0;
#pragma omp parallel for reduction(+:sum)
for (i = 0; i < n; i++)
    sum += array[i];
```

Assigning Work to a Single Thread

- Use `#pragma omp single` for operations that should only be done once.

```c
#pragma omp parallel
{
    printf("Hi from T%d\n", omp_get_thread_num());
    #pragma omp single
    printf("One Hi from T%d\n", omp_get_thread_num());
}
```

Only one thread will execute the `single` block.

Parallel Sections

- `#pragma omp sections` allows different sections of code to be executed by different threads.

```c
#pragma omp parallel sections
{
    #pragma omp section
    {
        printf("Section 1 executed by thread %d\n", omp_get_thread_num());
    }
    #pragma omp section
    {
        printf("Section 2 executed by thread %d\n", omp_get_thread_num());
    }
}
```

- There is an **implicit barrier** at the end of the sections unless `nowait` is used.

Loop Scheduling

- The `schedule` clause determines how loop iterations are assigned to threads.
- `static`, Equal chunks assigned at compile time. `dynamic`, Threads take chunks dynamically. `guided`, Starts with large chunks, then reduces. `auto`, Compiler decides the best method.

```
#pragma omp parallel for schedule(dynamic,2)
for(int i = 0; i<8; i++)
    printf("T%d: %d\n", omp_get_thread_num(), i);
```

Ordered Iterations

- Ensures that iterations follow a strict order when needed.

```
#pragma omp for ordered schedule(dynamic)
for(int i=0; i<100; i++) {
    f(a[i]); // Can run in parallel
    #pragma omp ordered
    g(a[i]); // Runs in order
}
```

Parallel Matrix Multiplication

```
#pragma omp parallel for collapse(2)
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++) {
        C[i][j] = 0;
        for (k = 0; k < N; k++)
            C[i][j] += A[i][k] * B[k][j];
    }
```

Finding the Maximum Value

```
int max_parallel(int *arr){
    int i, m = arr[0];
    #pragma omp parallel for reduction(max:m)
    for (i = 0; i < N; i++)
        if (m < arr[i])
            m = arr[i];
    return m;
}
```

Producer-Consumer Model

```
void produce() {
    while (i < NUM_ITEMS) {
        #pragma omp critical(one)
        if (!full) {
            put(item);
            i++;
        }
    }
}

void consume() {
    while (j < NUM_ITEMS) {
        #pragma omp critical(two)
        if (!empty) {
            get();
            j++;
        }
    }
}
```

Ensures only one thread modifies shared data at a time.

Performance Metrics

- **Response Time** The time taken to complete one task
- **Throughput** is the number of tasks completed per unit of time
- **CPU Time breakdown**
  - User time: Time the CPU spends running the user's code, System time: Time CPU spent running the OS's code, Wait time: Time spent waiting for I/O or other services ### Instruction-Level Metrics
- **IPS (Instructions Per Second):** Approximate speed of CPU execution.
- **CPI (Cycles Per Instruction):** CPU Time = (CPI × Instruction Count) / Clock Rate ### Overhead in Parallelism
- **Overhead includes:**
  - Thread creation/destruction, Synchronization, Communication, Waiting due to load imbalance or mutual exclusion ### Speedup and Efficiency formulas:

  ```
  Speedup = T_serial / T_parallel
  Efficiency = E = S / p       // p is the number
  ```

  As p increases then E decreases due to overhead. If the problem size increases then both Speedup (S) and Efficiency (E) increase. due to less overhead ### Amdahl's Law S is the max speedup, r is the percentage/fraction of the program that is parallelizable, p is the number of cores. r/p is also the parallel speedup

  ```
  S = 1 / ((1 - r) + (r/p))
  S = 1 / ((1 - r) + r) // as p approaches infinity
  ```

  #### Gustafson's law

  This formula is for scalable/large problem sizes. If 'r' or the parallelizable portion is 100% then S = P **Strong scalability** if E remains constant as p increases (that means the problem size is fixed). **Weak scalability** if E remains constant as both p and problem size increase. ### CUDA **Latency** is the time taken to complete one task. **Throughput** is the number of tasks completed per unit of time. ### CPU vs. GPU Architecture (Feature, CPU, GPU), (Control logic, Complex, Simple), (Threads, Few, Thousand), (Memory bandwidth, Lower, higher), (Latency, Optimized, Higher), (User Case, Serial Work, Parallel Work) GPU uses SIMD (single instruction multiple data). That's why GPU's are optimized for parallelism. Host is the CPU, Device is the GPU, Kernel is the function run on the device (executed in parallel by many threads). A grid is a collection of blocks, a block is a collection of threads ### Function Qualifiers (Qualifier, Runs on, Callable from), (**global**, Device, Host), (**device**, Device, Device), (**host**, Host, Host), (**host device**, Both, Both) ### cudaDeviceSynchronize()

- CUDA and CPU code are asynchronous by default.
- Use `cudaDeviceSynchronize()` to wait until all launched kernels finish.
- Useful for **timing kernel execution**. ### Thread Organization
- Threads are organized into 1D/2D/3D blocks.
- Blocks are organized into 1D/2D/3D grids.
- Each thread/block has its own ID:
- **Dimension Variables:** ### Launch Configuration
- To compute total threads and blocks:

  ```
  // computing thread idx
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  // 2d grid
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;

  int nthreads = 256;
  ```

```
int nblocks = (N + nthreads - 1) / nthreads;
vectorAdd<<<nblocks, nthreads>>>(...);
// block fimension
dim3 blockDim(16, 16);
//blocks in grid
dim3 gridDim((width+15)/16, (height+15)/16);
//thread indexing
int y = blockIdx.y * blockDim.y + threadIdx.y;
int x = blockIdx.x * blockDim.x + threadIdx.x;
// Always check bounds!
if (x < width && y < height) {
    // safe access
}
```

Matrix Multiplication: One Block

- Threads in a **single block** compute a matrix `P = M × N`
- Each thread computes one element in result `P`.
- Size limited to **32x32**, i.e. 1024 threads max per block.

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int width) {
    int r = threadIdx.y;
    int c = threadIdx.x;
    if (r < width && c < width) {
        float value = 0;
        for (int k = 0; k < width; k++)
            value += d_M[r*width + k] * d_N[k*width + c];
        d_P[r*width + c] = value;
    }
}
```

- Threads perform computation on shared data. ### Memory Types in CUDA (Type, Scope, Speed, Notes), (**Registers**, Thread, Very Fast, Private), (**Shared**, Block, Fast, Shared across threads in a block), (**Global**, All grids, Slow, Accessible by all threads), (**Constant**, All grids, Fast (cached), Read-only, limited size (64 KB)), (**Local**, Thread, Slow, (cached), Used when registers spill) ### Memory Optimization Guidelines

1. **Minimize Host-Device Transfers:** Batch small transfers, Keep intermediate structures on the device
2. **Use Fast Memory Types:** (Register, Thread), (Shared, Fast), (Constant, Grid), (Global, Slow), (Local, Slow)
3. **Reduce Global Memory Traffic:** Use **tiling**: load data into shared memory, compute, write back, Coalesced access patterns ### Tiling and Shared Memory Example

```
__shared__ float shrArr[128];
int idx = threadIdx.x;
shrArr[idx] = arr[idx];
__syncthreads();
// process shrArr[idx]
```

Synchronization Example Fix

```
array[i] = array[i-1]; // data race!
//fix:
__syncthreads();
int temp = array[i-1];
__syncthreads();
array[i] = temp;
//atomic sections
__global__ void increment_atomic(int* x) {
    atomicAdd(x, 1);
}
//critical sections
__device__ void lock() {
    while (atomicCAS(mutex, 0, 1) != 0);
    __threadfence();
}
__device__ void unlock() {
    atomicExch(mutex, 0);
    __threadfence();
}
```

Coalesced Global Memory Access

Access pattern matters! - Best: contiguous access pattern (stride 1) - Avoid: Strided access, Random access, Misaligned blocks ### Access Pattern Examples

```
x = A[i];           // Coalesced    x = A[2 * i];        // Strided
x = A[128 - i];     // Strided      A[A[i]] = 7;         // Random
```

Memory Location Summary

(type, memory, lifetime, speed), (int x, Register, Thread, Very fast), (int arr[10], Local, Thread, Slow), (**shared**, Shared, Block, Fast), (**device**, Global, Application, Slow), (**constant**, Constant, Application, Fast (cached)) ### L1 and L2 Caches - L1: Per SM, fast but **not coherent** - L2: Shared across GPU, coherent - Reads use both caches; writes go through L2 only ### Constant Memory Limited (64 KB), cached, read-only from device, Great for parameters or lookup tables ### Local Memory Private to thread, Physically located in global memory (but cached), Used when registers are insufficient

Distributed Memory Programming

CUDA, and OpenMP are for shared memory parallel programming. MPI is for distributed memory programming - **MPI (Message Passing Interface)**: Standard for writing programs that run on multiple machines. - Faster than Spark but Spark is better for data management.

Parallel Trapezoidal Rule

```
my_a = a + my_rank * my_n * h;
my_sum = Trap(my_a, my_b, my_n, h);
if (my_rank != 0)
    MPI_Send(&my_sum, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
else {
    total_sum = my_sum;
    for (source = 1; source < comm_sz; source++) {
        MPI_Recv(&my_sum, 1, MPI_DOUBLE, source, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        total_sum += my_sum;
    }
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#define BLOCK_SIZE 1024
__global__ void gpu_sqr(int *d_in, int *d_out, int size) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < size) {
        d_out[idx] = d_in[idx] * d_in[idx];
    }
}
void cpu_sqr(int *data_in, int *data_out, int size) {
    for (int i = 0; i < size; ++i) {
        data_out[i] = data_in[i] * data_in[i];
    }
}
#define CHECK_CUDA(call)                                                        \
    {                                                                           \
        cudaError_t err = call;                                                 \
        if (err != cudaSuccess) {                                          \
            fprintf(stderr, "CUDA error at %s:%d: %s\n", __FILE__, __LINE__,     \
                    cudaGetErrorString(err));                              \
            exit(EXIT_FAILURE);                                            \
        }                                                                  \
    }
int main() {
    const int N = 10000;
    int *h_in, *h_out; int *d_in, *d_out;
    dim3 grid;
    h_in = (int*)malloc(N * sizeof(int));
    h_out = (int*)malloc(N * sizeof(int));
    for (int i = 0; i < N; ++i) {
        h_in[i] = i;
    }
    CHECK_CUDA(cudaMalloc((void**)&d_in, N * sizeof(int)));
    CHECK_CUDA(cudaMalloc((void**)&d_out, N * sizeof(int)));
    CHECK_CUDA(cudaMemcpy(d_in, h_in, N * sizeof(int), cudaMemcpyHostToDevice));
    grid.x = (N + BLOCK_SIZE - 1) / BLOCK_SIZE;
    gpu_sqr<<<grid, BLOCK_SIZE>>>(d_in, d_out, N);
    CHECK_CUDA(cudaGetLastError());
    CHECK_CUDA(cudaMemcpy(h_out, d_out, N * sizeof(int), cudaMemcpyDeviceToHost));
    for (int i = 0; i < 10; ++i) {
        printf("h_out[%d] = %d\n", i, h_out[i]);
    }
    free(h_in);
    free(h_out);
    CHECK_CUDA(cudaFree(d_in));
    CHECK_CUDA(cudaFree(d_out));
    return 0;
}
```