MPI - Message Passing Interface. Library with its own version of c compiler "mpicc". Each process in MPI has a rank, numbered starting at 0.

Distributed memory, process-based.

MPI_Reduce - MPI function with several predefined reduction operations including: Maximum, Minimum, Sum, Product, Log. Result ONLY returned to the root process. Example MPI_REDUCE call from poker:

MPI_Reduce(&localStraightFlushes, &straightFlushes, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

[INPUT DATA] [OUTPUT DATA][count][Datatype] [Operator] [destination process][which processes]

MPI_AllReduce - Exact same as MPI_REDUCE, except returns result to all processes not just root.

MPI_Broadcast - The Same Data belonging to a single process is sent to all of the processes.

MPI_Bcast(cnt, 1, MPI_INT, 0, MPI_COMM_WORLD); // Broadcast the number of trials to all processes

[data to send][num of elements in buffer][datatype][rank of root process][communicator to other processes]

MPI_Gather - gathers data from all processes in a communicator (root) and collects it at the root process.

*int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype,void *recvbuf, int recvcount, MPI_Datatype recvtype,int root, MPI_Comm comm);*

[data from each proc.][num of element rec'd][data type][pointer to root buffr][number of elements each proc. sends][data type][rank of root][comm]

MPI_Scatter - Sends Different data from one process to all other processes in a communicator. Splits the elements in sendbuf into chunks of size sendcount, and sends it's process unique data. Ex below giving parts of array to different procs.

MPI_Scatter(sendbuf, 2, MPI_INT, recvbuf, 2, MPI_INT, 0, MPI_COMM_WORLD);

[entire array to be scattered][each proc gets 2 ints][data type of data][name of array][each proc gets 2 ints][rank 0 sending][comm]

Reduction - Form of aggregation, where multiple results are combined into a single value. Ex: when summing, adding process 1 and process 2 sum into a total sum

Synchronization in MPI - int MPI_Barrier(MPI_Comm comm); //will block all processes in the communicator until all processes have called it

Communication Styles: Point 2 Point - processes exchange messages explicitly between pairs. (MPI Send, Recv)

Collective Communication: A whole group of whole processes participates together. (Bcast, Scatter, Gather)

Concatenation - Form of aggregation where results are merged, for example merging two sorted sub-arrays

I/O - Process 0 always reads input, and then send values to other processes. All processes/mpicommworld can output. For fully parallel All processes perform I/O at once — no rank 0 bottleneck.

Deadlock - To avoid race conditions (where two threads modify the same data at once)), we use locks, When 2 or more processes are paused waiting for a receive. *A waiting for B, and B waiting for A.*

Blocking - Wait until receive By default, most "normal" MPI communication operations are blocking unless they have an "I" prefix (which stands for *Immediate* or *non-blocking*)**.**

Block Scheduling - Dividing data into equal chunks, so each process gets the same amount of data, collects results at the end into one process for total result. Works best when the amount of work is the same, it would be bad when one process has to do numbers 1-10 and one process has to do numbers 10000-100000 when checking if prime.

Loop Splitting - Each process gets the next piece of data (ex: P1 does 1,4,7,10, P2 2,5,8,11, P3 3,6,9,12). This works best when the size of data is not evenly dispersed, every process has a similar run time.

Self Scheduling - There is a master process and minion processes, minions tell the master when they are idle, and the master gives them more work to do. More communication means more overhead, but it is a dynamic scheduling method so it is good when the difficulty and shape of data for a problem is not known beforehand.

Monte CARLO- Each process independently does random trials → **no shared data, no synchronization → no race conditions**.

Threads - can be thought of as a "light-weight" process. shared memory, fast communication/shared variables, one thread can kill all, needs synchronization, Limited to one machine

Processes - Separate memory, Slow communication, one process does not affect others,synchronization usually not needed, Scales across multiple machines (e.g. MPI)

Speedup - How many times faster the parallel version is, time serial/ time parallel

Task vs Data Parallelism - Task parallelism partitions tasks among cores, such as one processor decoding video frames, another applying filters, and a final encoding frames. Data parallelism focuses on splitting up data, example splitting up and summing parts of an array.

Caching - CPU has a small fast memory next to it - the cache. We keep what we need there using Spatial Locality: Accessing a nearby location, and Temporal Locality: We will likely access the same memory again

Virtual Memory - Manages physical addresses more efficiently. Virtual memory is divided into pages, which is mapped to real physical memory. Helps to keep only what's needed in memory, idle stuff elsewhere.

Pipelining - Allows for different steps to happen simultaneously. Example: When instance 1 is done with fetching, it begins to get decoded and instance 2 also starts to get fetched.

Fosters method-Partitioning,communication, Aggregation, mapping

Race condition- A race condition happens when two or more threads or processes access shared data at the same time, and the final result depends on the timing

Critical section-A critical section is just a *region of code* where shared data is accessed or modified. A critical section is *not supposed* to be accessed by more than one thread at the same time.

Synchronization- synchronization prevents a critical section from a race condition

```c
// mpidemo.c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    // Initialize the MPI environment: sets up processes and the default communicator (MPI_COMM_WORLD)
    MPI_Init(&argc, &argv);
    int rank, size;

    // MPI_Comm_rank: "Who am I?"
    //   - Assigns to 'rank' the ID of this process within the communicator.
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // MPI_Comm_size: "How many of us are there?"
    //   - Assigns to 'size' the total number of processes in the communicator.
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // We'll scatter 4 integers to each rank (so total length = 4*size)
    int chunk = 4;
    int N = chunk * size;

    // Example use of broadcast:
    //   - Rank 0 already knows 'chunk' and 'N'; we broadcast them so *every* rank
    //     has the same config values (even though they could compute N themselves here).
    MPI_Bcast(&chunk, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&N,     1, MPI_INT, 0, MPI_COMM_WORLD);

    int *sendbuf = NULL;
    if (rank == 0) {
        // Only the root allocates and fills the big array we'll scatter
        sendbuf = (int*)malloc(N * sizeof(int));
        for (int i = 0; i < N; ++i) sendbuf[i] = i + 1;   // 1..N
    }

    // Each rank receives exactly 'chunk' integers from the root
    int *recvbuf = (int*)malloc(chunk * sizeof(int));

    // MPI_Scatter:
    //   - Root divides 'sendbuf' into 'size' chunks and sends one chunk to each rank.
    //   - Non-root ranks receive into 'recvbuf'.
    MPI_Scatter(sendbuf, chunk, MPI_INT, recvbuf, chunk, MPI_INT, 0, MPI_COMM_WORLD);

    // Local work on each process: sum its own chunk
    int local_sum = 0;
    for (int i = 0; i < chunk; ++i) local_sum += recvbuf[i];

    // MPI_Reduce:
    //   - Combines all 'local_sum' values across ranks using the operation MPI_SUM.
    //   - The result is placed in 'global_sum' on the root (rank 0).
    int global_sum = 0;
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // Root prints the result and then shares it with everyone
    if (rank == 0) {
        printf("[root] Global sum of 1..%d is %d\n", N, global_sum);
    }

    // MPI_Bcast again:
    //   - Root broadcasts 'global_sum' so every rank can see/print/use it.
    MPI_Bcast(&global_sum, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Now every rank can report what it saw/computed
    printf("Rank %d of %d: local_sum=%d, global_sum=%d\n", rank, size, local_sum, global_sum);

    free(recvbuf);
    if (rank == 0) free(sendbuf);

    // Cleanly shut down the MPI environment
    MPI_Finalize();
    return 0;
}
```

Sums Numbers 1-16 in 4 different parallel processes. Build with mpicc -o mpidemo mpidemo.c
Run with mpirun -np 4 ./mpidemo. Always have #include <mpi.h>, MPI_Init, MPI_Finalize

Operation Value	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

MPI Reduce Operations

$$T_{parallel} = 0.9 \times T_{serial} / p + 0.1 \times T_{serial} = 18 / p + 2$$

Tparallel = Tserial/num processors(P) + Overhead, parallelized part/ P + unparallelized part.

Efficiency- how well the processors are being utilized. If E=1 → perfect efficiency, E= speedup/ num P

Scalability - MPI_Wtime() measures scalability,The closer efficiency (E) is to 1, the better the scalability. Strong scalability-fixed problem size, weak scalability fixed work per process

keep the problem the same size add processors=strong, change problem size add processors weak.

Pthreads: POSIX threads c library allow us to create parallel shared memory programs, with a lot of control compared to openMP.
Using Pthreads: #include <pthread.h>, define a thread with **pthread_t thread1;**
Pthread_create syntax: pthread_create(threadname, NULL, Method to call, values);
Pthread_join syntax: pthread_join(thread name, NULL); //
Pthread_exit: ensure this is being called as "Pthread_exit(NULL)" at the end of methods being used in the threads.
Pthread program steps: First add include statement, then define all pthreads. Then call pthread_create on all of them, only after they are all created call pthread_join on all of them.

CISC372FinalPractice > C ThreadExample.c > ...

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

// Function that each thread will run
void* print_message(void* arg) {
    printf("Hello from thread!\n");
    pthread_exit(NULL);
}

int main() {
    pthread_t thread1, thread2;

    // Create two threads
    pthread_create(&thread1, NULL, print_message, NULL);
    pthread_create(&thread2, NULL, print_message, NULL);

    // Wait for both threads to finish
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    printf("Both threads have finished executing.\n");

    return 0;
}
```

This example on the left creates two threads which each do a print statement, and then exit. They utilize pthread_join, create and exit correctly.
This example below Is a code snippet showing how mutexes work with threads. With mutexes, it is important to call the lock before a critical section and after the critical section call unlock.

```c
int count = 0;
pthread_mutex_t lock_var; // Assume this is initialized

void* increment_counter(void* arg) {

    // 1. Acquire the lock BEFORE reading the shared variable
    pthread_mutex_lock(&lock_var);

    // --- CRITICAL SECTION START ---
    int temp = count;
    temp = temp + 1;
    count = temp;
    // --- CRITICAL SECTION END ---

    // 2. Release the lock AFTER updating the variable
    pthread_mutex_unlock(&lock_var);

    return NULL;
}
```

```c
// 1. Declare the barrier globally so all threads can see it
pthread_barrier_t my_barrier;

void* worker_thread(void* arg) {
    long id = (long)arg;

    printf("Thread %ld is working...\n", id);

    // --- BARRIER ---
    // 2. Threads wait here until the required number arrive
    pthread_barrier_wait(&my_barrier);
    // ---------------

    printf("Thread %ld has passed the barrier!\n", id);
    return NULL;
}

int main() {
    pthread_t threads[3];

    // 3. Initialize: &barrier, attributes (NULL), count (3)
    pthread_barrier_init(&my_barrier, NULL, 3);

    // Create 3 threads
    for(long i = 0; i < 3; i++) {
        pthread_create(&threads[i], NULL, worker_thread, (void*)i);
    }

    // Join threads
    for(int i = 0; i < 3; i++) {
        pthread_join(threads[i], NULL);
    }

    return 0;
}
```

The code on the left is an example of using barriers. First the barrier is initialized, then you include the barrier in the method being used by the threads. When one thread reaches the barrier, it pauses until all other threads hit it.
Gang: In OpenACC a gang represents a set of parallel operations as defined by the parallel construct. Gangs work independently of each other and may not coordinate (think thread blocks).
Vector: Standard SIMD type processing. Each thread works on one element of the vector.
Worker: Worker parallelism sits between vector and gang levels. A gang consists of 1 or more workers, each of which operates on a vector of some length.
Cache: Within a gang the OpenACC model exposes a cache memory, which can be used by all workers and vectors within the gang

Open ACC: Very simple solution for parallelizing our C code, and doesn't require nvidia GPU or anything. If the computer is not compatible, the compiler just ignores it. Very simple to use as it is not much code added, just annotations.

#Pragma acc kernels: "Here is a block of code, please try to parallelize it". Parallelizes whatever it can, probably won't be perfectly optimal but still will be better than the serial solution.

#Pragma acc parallel: "I know what I'm doing, run this in parallel now." Controlling what is being parallelized.

We can also add clauses to our pragmas - two correctness clauses are

Private(var1, var2, var3) - specifies that each loop iteration requires its own copy of the listed variables

Reduction(operator:variable) - similarly to the private clause in that a private copy of the affected variable is generated for each loop iteration, but reduction goes a step further to reduce all of those values into a single value based on the operator. **private copies into one final result, which is returned from the region.**

Atomic We can specify that an instruction is "Atomic" to deal with race conditions between threads within a gang

copy (var1, var2): causes a copy to the device of the listed variables, and copies them back after the region

copyin(var1,var2): does not copy back to host

copyout(var1,var2): reserves space, but doesn't copy back at the end

create(var1,var2): creates space for the variables, but does no copying

present(var1,var2): informs openACC that the variables are already present on the device

deviceptr:(var1,var2): indicates the variables were created on the device outside of OpenACC (ex: cudaMalloc)

Loop splitting: use the parallel loop keyword

Welcome | AccSerial.c U

CISC372FinalPractice > AccSerial.c > matrix_mult_serial(float *, floa

```c
#include <stdio.h>
#define N 1024

void matrix_mult_serial(float *A, float *B, float *C) {
    // Three nested loops
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            float sum = 0.0f;
            for (int k = 0; k < N; k++) {
                sum += A[i*N + k] * B[k*N + j];
            }
            C[i*N + j] = sum;
        }
    }
}
```

AccKernels.c U

CISC372FinalPractice > AccKernels.c > ...

```c
// OpenACC Kernels Version
#include <stdio.h>
#define N 1024

void matrix_mult_kernels(float *A, float *B, float *C) {

    // "Compiler, here is the data. Please analyze and accelerate this region."
    #pragma acc kernels copyin(A[0:N*N], B[0:N*N]) copyout(C[0:N*N])
    {
        // We do NOT need to add loop directives here.
        // The compiler sees independent iterations and does it for us.
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                float sum = 0.0f;
                for (int k = 0; k < N; k++) {
                    sum += A[i*N + k] * B[k*N + j];
                }
                C[i*N + j] = sum;
            }
        }
    } // End of kernels region
}
```

AccParallel.c U

CISC372FinalPractice > AccParallel.c > ...

```c
// OpenACC Parallel Loop Version
#include <stdio.h>
#define N 1024

void matrix_mult_parallel(float *A, float *B, float *C) {

    // 1. Manage Data
    #pragma acc data copyin(A[0:N*N], B[0:N*N]) copyout(C[0:N*N])
    {
        // 2. Define Parallel Region AND Work Sharing
        // "Launch threads immediately and share the work of these loops."
        // collapse(2) turns the 2D grid (i,j) into a 1D linear list of tasks.
        #pragma acc parallel loop collapse(2)
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {

                float sum = 0.0f;
                // This inner k-loop runs sequentially inside each thread
                for (int k = 0; k < N; k++) {
                    sum += A[i*N + k] * B[k*N + j];
                }
                C[i*N + j] = sum;
            }
        }
```

CISC372FinalPractice > CUDA > testcuda.cu > main()

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

#define N 1000000  // Size of the arrays (1 million elements for this example)

int main() {
    // Declare pointers for arrays A, B, and C.
    int *A, *B, *C;

    // Allocate memory for the arrays on the host (CPU)
    // These arrays will hold integers, and each array will contain N elements.
    A = (int*)malloc(N * sizeof(int));
    B = (int*)malloc(N * sizeof(int));
    C = (int*)malloc(N * sizeof(int));

    // Initialize arrays A and B with some values.
    // For simplicity, we'll set array A to be the integers 0 to N-1.
    // Array B will be the even integers (2, 4, 6, ...).
    for (int i = 0; i < N; i++) {
        A[i] = i;        // Array A holds values 0, 1, 2, ..., N-1
        B[i] = i * 2;    // Array B holds values 0, 2, 4, ..., 2*(N-1)
    }

    // Use OpenACC to perform the computation on the GPU.
    // The `data` pragma manages the movement of arrays between host and device memory.
    #pragma acc data copyin(A[0:N], B[0:N]) copyout(C[0:N])
    {
        // Parallelize the loop using OpenACC's `parallel loop` directive.
        // This will offload the loop execution to the GPU, where each thread will compute one element of array C.
        // Each thread will compute the sum of A[i] and B[i] for a specific value of i.
        #pragma acc parallel loop
        for (int i = 0; i < N; i++) {
            C[i] = A[i] + B[i];  // Add corresponding elements of A and B, storing result in C
        }
    }

    // After the parallel computation finishes, array C contains the sum of arrays A and B.
    // Let's print the first 10 elements of array C to verify that the computation worked.
    printf("First 10 elements of array C:\n");
    for (int i = 0; i < 10; i++) {
        printf("%d + %d = %d\n", A[i], B[i], C[i]);
    }

    // Free the allocated memory for arrays A, B, and C.
    // It's important to free memory after use to prevent memory leaks.
    free(A);
    free(B);
    free(C);

    return 0;  // End of program
}
```

Cuda Kernel- add _global_ then add `int index = blockIdx.x * blockDim.x + threadIdx.x; Then change i with index`

_syncthreads-using shared memory to ensure data is written before reading, only syncs threads on the same block

-syntax launched from host for cuda-my_kernel<<<numBlocks, threadsPerBlock>>>(args...);

Matrix multiplication multiple dimension- dim 3 used for 2d kernel

Threadid- range 0 to blockdim -1, blockdim.x

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

Softmax- needs multiple passes over the input, so the number of global memory accesses is proportional to the number of elements (O(N)), usually around a few reads and writes per element, making it largely memory-bandwidth-limited rather than compute-limited.

_syncthreads-using shared memory to ensure data is written before reading, only syncs threads on the same block

CuBLAS - cuda basic linear algebra sub-programs - include library with #include <cublas_v2.h> . Link with the cublas library with – nvcc <obj files.o> -o <output file> -lcublas. For really big matrices, CuBLAS wins.

CuRAND - generates by pseudo-random and quasi-random numbers. Initialize:curand_init(seed, sequence, offset, &state); Kernel to generate random numbers in an array:

```
__global__ void fillRandomKernel(float* array, long count,float min, float max){
    long position=gridDim.x*blockIdx.y*blockDim.x+blockIdx.x*blockDim.x+threadIdx.x;
    curandState state;
    curand_init((unsigned long long)clock() + position, 0, 0, &state);
    array[position]= curand_uniform(&state)*(max - min) + min;
}
```

```
__global__ void addVectors(int *a, int *b, int *c, int n) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;//index is the global thread index
    if (index < n) {//if the index is less than the number of elements in the array, then add the elements
        c[index] = a[index] + b[index];//add the elements
    }
}
// Serial version - runs on CPU
void addVectorsSerial(int *a, int *b, int *c, int n) {
    for (int i = 0; i < n; i++) {//for each element in the array, add the elements
        c[i] = a[i] + b[i];//add the elements
    // ========== CUDA CODE SECTION ==========
    // Allocate device memory
    cudaMalloc((void**)&d_a, n * sizeof(int));//allocate memory for the array
    cudaMalloc((void**)&d_b, n * sizeof(int));//allocate memory for the array
    cudaMalloc((void**)&d_c, n * sizeof(int));//allocate memory for the array
    // Copy data from host to device
    cudaMemcpy(d_a, h_a, n * sizeof(int), cudaMemcpyHostToDevice);//copy the data from the host to the device
    cudaMemcpy(d_b, h_b, n * sizeof(int), cudaMemcpyHostToDevice);//copy the data from the host to the device
    // Launch kernel
    int threadsPerBlock = 256;//number of threads per block
    int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;//number of blocks per grid
    addVectors<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, d_c, n);
    // Copy result back from device to host
    cudaMemcpy(h_c, d_c, n * sizeof(int), cudaMemcpyDeviceToHost);//copy the data from the device to the host
    // Free device memory
    cudaFree(d_a);//free the memory for the array
    cudaFree(d_b);//free the memory for the array
    cudaFree(d_c);//free the memory for the array
```

OPENMP-Atomic-simple operation fast execution
Parallel for-use this to do loop splitting- run iterations of a for loop at the same time using multiple threads
Cuda blocks are like open mp teams- threads in a block= threads in a team
distribute - decides loop iterations each team gets if you have 4 teams and n=1000 team 1=0-249, team 2=250-599
Pragma- here's a special instruction, if the compiler understands it then run it, if not dont run
Open mp vs acc- open mp is mostly for cpu but does both and open acc is made for gpu only
Simt- sig threads, single instruction- multiple threads
Open acc kernels vs mp parallel for- open acc looks at the whole thing for gpu, mp cpu unless target still only looks at 1 loop, you decide what's parallel
#pragma omp parallel for schedule(dynamic, 2)-shows how its called

```
    // ❌ WRONG: Data race - multiple threads write to same variable
void bad_reduction(double *a, int n) {
    double sum = 0.0;
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        sum += a[i];  // DATA RACE! Multiple threads modifying sum
    }
}


// ✅ CORRECT: Use reduction clause
void good_reduction_cpu(double *a, int n) {
    double sum = 0.0;
    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < n; i++) {
        sum += a[i];  // Each thread has private copy, combined at enc
    }
```

```
// PARALLEL REGION:
//   #pragma omp parallel
//   {
//       // Code executed by all threads
//   }
// CRITICAL SECTION:
//   #pragma omp critical
//   {
//       // Only one thread executes at a time
//   }
// VARIABLE SCOPING:
//   shared(var)      - All threads share same variable
//   private(var)     - Each thread has private copy (uninitialized)
//   firstprivate(var) - Each thread has private copy (initialized from original
//   lastprivate(var)  - Each thread has private copy (last value copied back)
//   default(shared)   - Default: variables are shared
//   default(none)     - Must explicitly specify all variable scoping
//
// SCHEDULE CLAUSES:
//   schedule(static)           - Divide into equal chunks at compile time
//   schedule(static, chunk)    - Each chunk has 'chunk' iterations
//   schedule(dynamic)          - Threads grab chunks as they finish
//   schedule(dynamic, chunk)   - Dynamic with specified chunk size
//   schedule(guided)          - Chunk size decreases as work progresses
//   schedule(guided, chunk)    - Guided with minimum chunk size
//   schedule(auto)             - Let OpenMP decide
//   schedule(runtime)          - Set via OMP_SCHEDULE environment variable
//
// ODD-EVEN TRANSPORT SORT:
//   - Alternates between even and odd phases
//   - Even phase: compare pairs (0,1), (2,3), (4,5), ...
//   - Odd phase: compare pairs (1,2), (3,4), (5,6), ...
//   - Parallelizable because comparisons in each phase are independent
//   - Uses critical section to safely update global sorted flag
```

```
double dot_product(double *a, double *b, int n) {
    double sum = 0.0;
    // target - offload to GPU
    // teams - create thread teams
    // distribute - distribute iterations across teams
    // parallel for - parallelize loop within teams
    // reduction(+:sum) - combine partial sums
    // map(to: ...) - copy input arrays to GPU
    // map(tofrom: sum) - copy sum to/from GPU
    #pragma omp target teams distribute parallel for \
        map(to: a[0:n], b[0:n]) \
        map(tofrom: sum) \
        reduction(+:sum)
    for (int i = 0; i < n; i++) {
        sum += a[i] * b[i];
    }
    return sum;
}
double dot_product(double *a, double *b, int n) {
    double sum = 0.0;
    // Serial loop - computes dot product sequentially
    for (int i = 0; i < n; i++) {
        sum += a[i] * b[i];
    }
    return sum;
int main() {
    const int n = 1000;

    // Allocate and initialize arrays
    float *a = (float*)malloc(n * sizeof(float));
    float *b = (float*)malloc(n * sizeof(float));
    float *c = (float*)malloc(n * sizeof(float));

    for (int i = 0; i < n; i++) {
        a[i] = i;
        b[i] = i * 2.0f;
    }
    // Example: Vector addition on GPU
    vec_add(a, b, c, n);
    // Print first few results
    printf("Vector addition results (first 5):\n");
    for (int i = 0; i < 5; i++) {
        printf("c[%d] = %.1f\n", i, c[i]);
    }

    // Example: Dot product on GPU
    double *da = (double*)malloc(n * sizeof(double));// Allocate da
    double *db = (double*)malloc(n * sizeof(double));

    for (int i = 0; i < n; i++) {
        da[i] = i;// Initialize da with values 0, 1, 2, ..
        db[i] = i;// Initialize db with values 0, 1, 2, ..
    }
    double result = dot_product(da, db, n); // Use these arrays for
    printf("\nDot product result: %.2f\n", result);
    // Cleanup
    free(a);
    free(b);
    free(c);
    free(da);
    free(db);

    return 0;
```

loop work-sharing constructs: The schedule clause

Schedule Clause	When To Use
STATIC	Pre-determined and predictable by the programmer
DYNAMIC	Unpredictable, highly variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead

Least work at runtime : scheduling done at compile-time

Most work at runtime : complex scheduling logic used at run-time

loop worksharing constructs: The schedule clause

- **The schedule clause affects how loop iterations are mapped onto threads**
  - schedule(static [,chunk])
    - Deal-out blocks of iterations of size "chunk" to each thread.
  - schedule(dynamic[,chunk])
    - Each thread grabs "chunk" iterations off a queue until all iterations have been handled.
  - schedule(guided[,chunk])
    - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size "chunk" as the calculation proceeds.
  - schedule(runtime)
    - Schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library … for OpenMP 3.0)

GPU Memory in a nutshell

- Register
  - Data stored is only visible to the thread that wrote it, lasts for the lifetime of the threads
- Local Memory
  - Same scope as register but performs slower since the memory is off-chip
- Shared Memory
  - Shared between all streaming processors in a multiprocessor.
  - Fast memory like registers
  - Can be L1 cached
- Global Memory
  - All threads have access to this memory
  - Largest volume of memory
  - Very high latency
  - Non-cacheable
- Constant and Texture Memory
  - Read-only memory
  - Slow and Can be cached
  - Beneficial to only special type of applications
  - Reside on-chip

Variable Type Qualifiers in CUDA

- __device__:
  - May be used in conjunction with __constant__ or __shared__
  - When not specified, variable is in global memory space of device
  - Has application lifetime
  - Accessible to any thread in grid
  - Accessible to host through api calls
  - May be used with __managed__ to directly access from host.
- Nothing: thread-local variable
- __constant__:
  - Variable resides on device in constant memory
  - Has application lifetime
  - Accessible to any thread in grid
  - Accessible to host through api calls
- __shared__:
  - Variable resides in shared memory space of one thread block
  - Has lifetime of thread block
  - Only accessible to threads in the block

Key CUDA Points

- a kernel call specifies the number of blocks and number of threads per block
  - the indexing of each can be organized in 1d, 2d, or 3d
  - use whatever is convenient for your application (still a set of threads)
  - the number of threads per block is limited (e.g., 1024)
  - the number of blocks can be very high
- blocks must be entirely independent of each other
  - cannot assume blocks will run concurrently
  - if you need to share information between blocks, use multiple kernel calls
- threads within a block all run concurrently
  - shared memory
  - all the usual concurrency techniques can be used

```
278  gcc -fopenmp openmpcombined.c -o openmpcombined
279  gcc -fopenmp -O3 openmpcombined.c -o openmpcombined
280  # Windows PowerShell
281  $env:OMP_NUM_THREADS=4
282  ./openmpcombined
283
284  nvcc simple_cuda.cu -o simple_cuda
285  nvcc testcuda.cu -o testcuda
286  nvcc cudatwo.cu -o cudatwo
287  nvcc -lcublas -lcurand cudatwo.cu -o cudatwo
288  cublas linker flag
```

```
70  // DIRECTIVE REFERENCE
71  // ================================================================================
72  //
73  // CPU Directives:
74  //   #pragma omp parallel for          - Parallelize loop on CPU
75  //   #pragma omp parallel for reduction(op:var) - Parallel with reduction
76  //
77  // GPU Directives:
78  //   #pragma omp target                - Offload code block to GPU
79  //   #pragma omp target teams           - Create thread teams on GPU
80  //   #pragma omp target teams distribute - Distribute iterations to teams
81  //   #pragma omp target teams distribute parallel for - Full GPU parallelization
82  //
83  // Clauses:
84  //   map(to: arr[0:n])      - Copy array TO GPU (input)
85  //   map(from: arr[0:n])    - Copy array FROM GPU (output)
86  //   map(tofrom: arr[0:n])  - Copy array both directions (input/output)
87  //   reduction(op:var)      - Combine results from all threads
88  //   collapse(k)            - Collapse k nested loops into one
```

Speedup = T serial / T parallel

CISC372 Final – Post Midterm One Pager/ Table of Contents

POSIX Threads (pthreads)- Create thread:

```
pthread_create(&t, NULL, worker, arg);
```

- Join (wait):

```
pthread_join(t, NULL);
```

- Mutex:

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock(&m); /* critical */ pthread_mutex_unlock(&m);
```

- Barrier: all threads wait until all arrive.- Semaphore: counting lock; sem_wait (down), sem_post (up).

OpenMP (CPU)- Parallel region:

```
#pragma omp parallel { /* each thread */ }
```

- Parallel loop:

```
#pragma omp parallel for
for (int i=0;i<n;i++) work(i);
```

- Critical vs atomic:

```
#pragma omp critical
  sum += a[i];
#pragma omp atomic
  sum += a[i];
```

- Reduction:

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

- schedule(static|dynamic[,chunk]) – load balance control.- Odd/even sort: even phase swap(0,1),(2,3)…; odd phase swap(1,2),(3,4)…; each phase parallel.

CUDA Basics- Kernel:

```
__global__ void add(const float *a,const float *b,float *c,int n){
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  if (idx<n) c[idx]=a[idx]+b[idx];
}
```

- Launch: add<<<numBlocks, blockSize>>>(d_a,d_b,d_c,n);- Memory:

```
cudaMalloc(&d_a, bytes); cudaMemcpy(d_a,h_a,bytes,cudaMemcpyHostToDevice);
cudaMemcpy(h_c,d_c,bytes,cudaMemcpyDeviceToHost); cudaFree(d_a);
```

- Indexing 2D (matrix):

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

- Shared memory reduction (idea):

```
extern __shared__ float s[];
s[tid] = local_sum; __syncthreads();
for (int stride = blockDim.x/2; stride>0; stride>>=1){
  if (tid<stride) s[tid]+=s[tid+stride];
  __syncthreads();
}
```

OpenACC- Offload loop:

```
#pragma acc data copyin(a[0:n]) copyout(c[0:n])
{ #pragma acc parallel loop
  for (int i=0;i<n;i++) c[i]=2.0f*a[i]; }
```

- Data clauses: copy, copyin, copyout, create, present.- Reduction:

```
#pragma acc parallel loop reduction(+:sum)
```

- atomic:

```
#pragma acc atomic
```

OpenMP GPU Offload- Basic pattern:

```
#pragma omp target teams distribute parallel for \
        map(to:a[0:n]) map(from:c[0:n])
for (int i=0;i<n;i++) c[i]=2.0f*a[i];
```

- map(to:), map(from:), map(tofrom:) – control host↔device copies.- collapse(k) – flatten k nested loops for more parallel work.- GPU reduction:

```
double sum=0
```