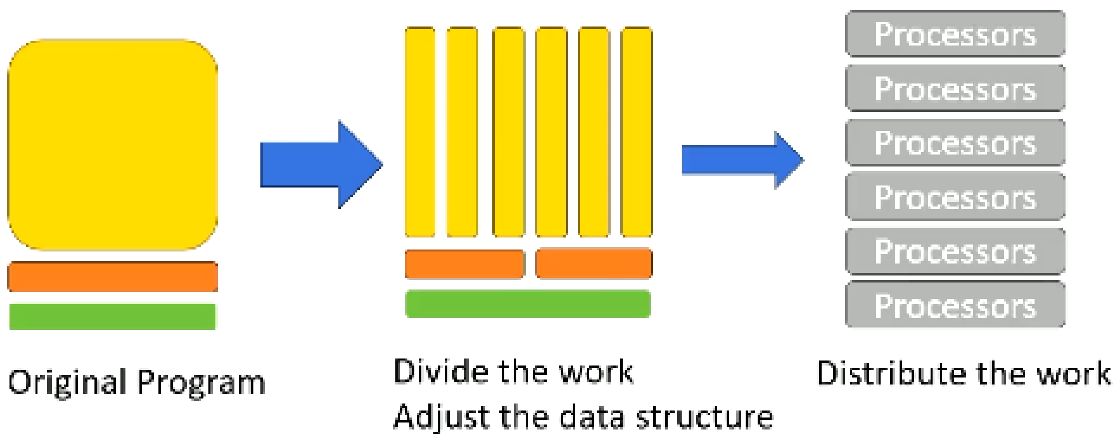
Parallel Programming

Parallel Programming Patterns

Learning Objective

1. Provide an overview of different parallel programming paradigms

How to Create a Parallel Application



How to Create a Parallel Application

Steps to Write a Parallel Program

1. Discover concurrency
   - Identify opportunities for parallelism
2. Structuring the algorithm
   - Organize the algorithm to harness concurrency effectively
3. Implementation
   - Implement the algorithm in a suitable programming environment
4. Execution and optimization
   - Execute and fine-tune the code on a parallel system for optimal performance

Parallel Programming Patterns

1. Master/Worker Pattern
   - Master process or thread manages a pool of worker process/threads and a task queue
   - Workers execute tasks concurrently dequeing tasks from the shared task queue
   - Suitable for "embarrassingly parallel problems" where tasks vary
2. SPMD Pattern
   - Single program, multiple data
   - All processing elements execute the same program in parallel, each with its dataset
   - Widely used in GPU programming
3. Loop Parallelism Pattern
   - Loops are common and excellent candidates for parallelism
   - Many loops involve repetitive, independent iterations suitable for parallel execution
   - Each task is the same
4. Fork/Join Pattern

    - Combines serial and parallel processing
    - Parents tasks for new task for their completion before continuing
    - Often used in programs with a single entry point
5. Pipeline Pattern
    - Resembles a CPU pipeline
    - Each parallel processor handles different stages of a task
    - Ideal pattern for processing data streams
    - Examples:
        - Signal processing
        - Graphics pipeline
        - Compression workflows: decompression -> work -> compression

Summary

1. Introduced the SPMD concept, a fundamental parallel programming approach
2. Covered key parallel programming patterns: Master-worker, loop parallelism, SPMD, pipeline parallelism

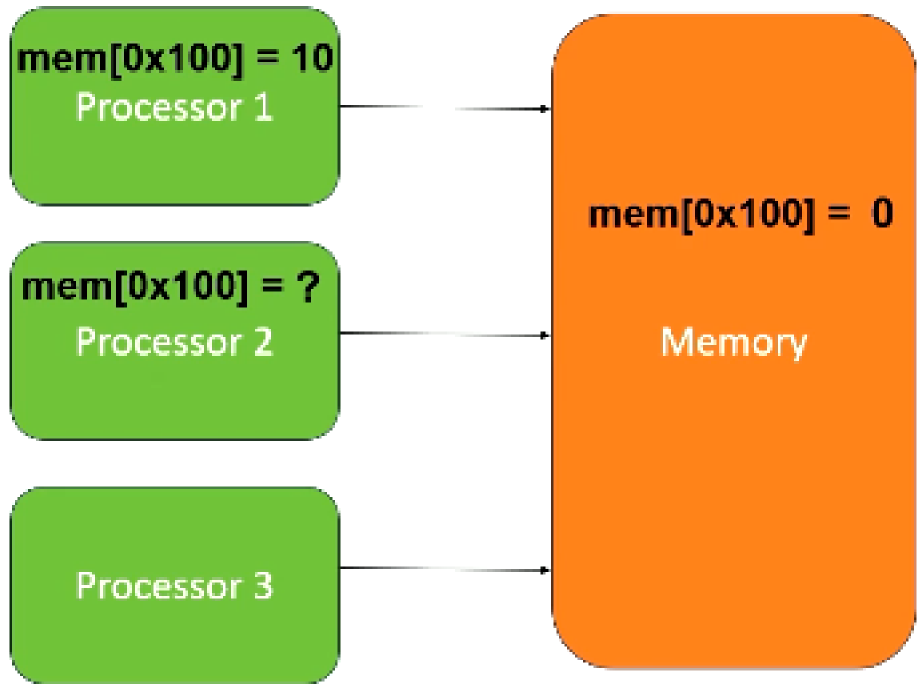
OpenMP vs MPI (Part 1)

Learning Objectives:

1. Explain the fundamental concepts of shared memory programming
2. Describe key concepts essential for shared memory programming
3. Explore the primary components of OpenMP programming

Programming with Shared Memory

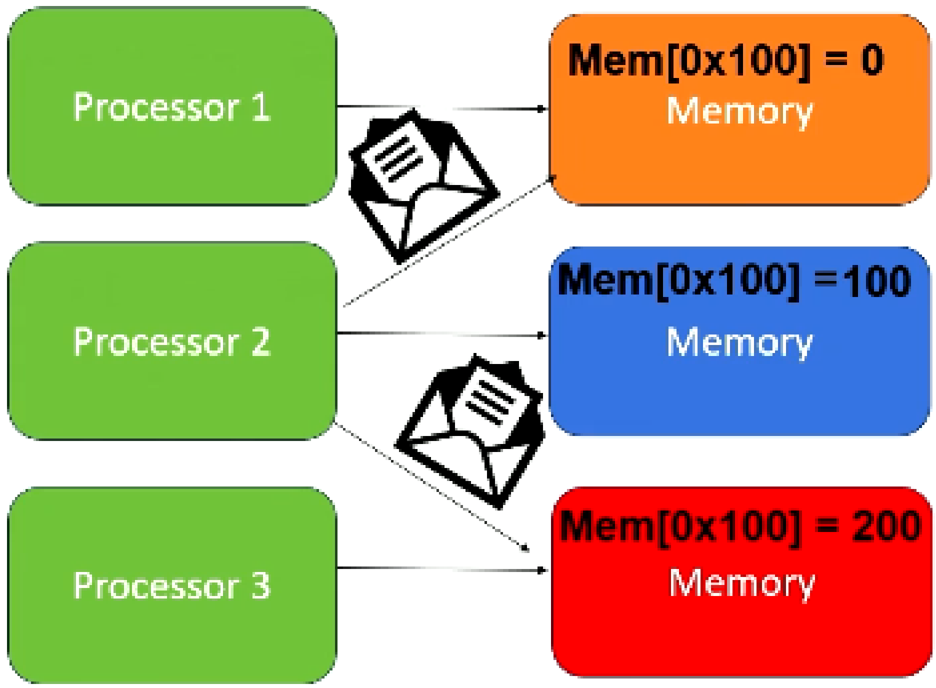
1. All processors can access the same memory
    - Proc 2 can observe updates made by Proc 1 by simply reading values from shared memory



Shared Memory

Programming with Distributed Memory

1. Distributed memory systems have each processor with its own memory space
    - To access data in other memory space, processors send a message
    - Proc 2 requests messages from Proc 1 and Proc 3



Distributed Memory

Overview of OpenMP

1. Open standard for parallel programming on SMP
2. Collection of compiler directives, library routines, and environment variables to write parallel programs

Key Tasks for Parallel Programming

1. Parallelization
2. Specifying thread counts
3. Scheduling
4. Data sharing
5. Synchronization

What is a Thread?

1. An entity that runs on a sequence of instructions
2. A thread has its own register and stack memory
3. Is a thread equivalent to a core?
   - No, thread is a software concept. One CPU core could run multiple threads or a single thread
4. The meaning of thread is different on CPUs and GPUs

Data Decomposition

1. Split data into two cores
2. Modern CPUs typically execute multiple threads in one core but for simplicity, we assume one core executes one thread in our example

Example: Vector Sum

1. Manual parallelization process
   - Create two threads
   - Split the array into two and give 1/2 array to each thread
   - Merge two partial sums
2. Mutex Operation
   - What if both threads try to update the total sum?
     - Sum would either be 100 or 200 instead of 300
   - We need to prevent both threads from updating the total sum variable in memory
   - Sum variable is shared data
   - Updating shared variable is critical section of code
   - Mutex (mutual exclusion) ensures only one thread can access critical section of code
   - Lock: Acquire a mutex to enter critical section
   - Unlock: Release a mutex after finishing the critical section; others are allowed to access the critical section
3. Low Level Programming for Vector Sum
   - Programmer has to specify:
     - p-thread: low-level programming to interface with threads
     - Thread create
     - Thread join
     - Mutex (lock)
4. Vector sum in OpenMP

```
#include <iostream>
#include <omp.h>

int main() {
    const int size = 1000;
    int data[size];
    int sum = 0;
    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < size; ++i) {
        sum += data[i];
    }
    std::cout << "Sum: " << sum << std::endl;
}
```

OpenMP

1. Compiler directive
   - Works for C/C++/Fortran (widely used in HPC)
   - Compiler replaces directives with calls to runtime library
   - Library function handles thread create/join
2. #pragma omp directive [clause [clause] ...]
   - Directives are the main OpenMP construct: pragma omp parallel for
   - Clauses provide additional information: reduction (+:sum)
   - Reduction is commonly used

Summary

1. Learned the basic concepts of OpenMP programming, particularly the reduction operation
2. Learned how to program vector sum using OpenMP

OpenMP vs MPI (Part 2)

Learning Objectives

1. Extend your understanding of the concept of scheduling in OpenMP
2. Describe key components of OpenMP and MPI programming

How Many Threads?

1. In OpenMP, the number of threads can be set
   - by environment variable: OMP_NUM_THREADS
   - the omp_set_num_threads() function within the code

Scheduling

1. Consider a vector sum example with a 1M-size vector and 5 threads but only 2 cores. We'll explore how to manage work distribution in such scenarios
   - We probably want to give 200K elements to each thread
   - This works well if each thread can make the same progress
   - But what if each thread progresses differently?

Static Scheduling/Dynamic Scheduling

1. Give 200K elements to each thread
   - Static
2. Give 1 element to ecah thread and come back for more work when done
   - Dynamic
3. Give 1000 elements to each thread and come back for more work when done
   - Dynamic
   - Strikes a balance between 1 and 2
4. Initially give 1000, but afterwards vary size based on completion time
   - Adapts to the runtime conditions
   - Guided scheduling: Chunk size varies over time
5. Dynamic scheduling can adopt run time effect
   - Maybe some threads got scheduled to an old machine, etc.

Data Sharing

1. Private vs shared data
   - Partial sum is private data and total sum is shared data
2. Thread synchronization
   - Barrier
   - Critical Section
   - Atomic

Barrier

1. Barrier: #pragma omp barrier
2. Synchronization point that all participating threads wait until it is reached
   - Sorting, then update

Critical Section

1. Critical section: #pragma omp critical [name]
2. A critical section should be updated only by one thread
   - Incrementing a counter

Atomic

1. Atomic: #pragma omp atomic
2. To ensure some tasks are done atomically
   - Atomically: Either the work is all done or nothing; no partial work
   - Incrementing a counter requires loading the counter value, adding, and storing
3. Atomic operation can be done with mutex or hardware might support atomic operation natively

Parallel Sections

1. Work can be done in parallel but not within a loop
2. How can we express this?
   - Sections directive
   - work1 and work2 will be executed in parallel
   - Combined with other constructions: ordered, single
3. The section directive can be used with various programming patterns

```
#pragma omp parallel sections {
    #pragma omp section
    {
        // work1
    }
    #pragma omp section
    {
        // work2
    }
}
```

Example of Parallel Sections: Ordered

```
#pragma omp parallel
{
    #pragma omp for ordered
    for (int i = 0; i < 5; i++) {
        #pragma omp ordered
        {
            // this block of code will be executed in ordered
            printf("Thread %d is doing iteration %d\n", omp_get_thread_num(), i);
        }
    }
}
```

Example of Parallel Sections: Single

```
#pragma omp parallel
{
    #pragma omp single
    {
        // this block of code will be executed by only one thread
        printf("This is a single thread task\n");
    }
    // other parallel work
}
```

Summary

1. Introduced several key concepts in parallel programming
2. Mutex, critical sections, barrier operations

Programming with MPI

Learning Objectives

1. Describe fundamental concepts of distributed memory parallel programming
2. Gain understanding of MPI (Message Passing Interface) programming

Why Study OpenMP and MPI?

1. OpenMP and MPI are crucial because CUDA programming combines shared memory and distributed memory approaches
2. Some memory regions are shared among all cores, while others are not visible

MPI Programming

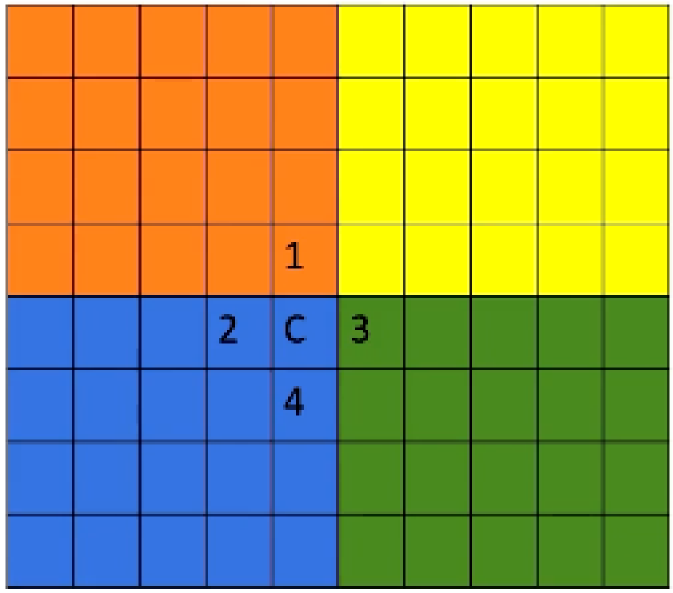
1. MPI stands for message passing interface, a communication model for parallel computing
2. Example:
    - Two processes want to communicate with each other
    - Process 0 sends an integer value to process 1 using MPI_send()
    - Process 1 receives the value sent by process 0 using MPI_recv()

Broadcasting

1. MPI_bcast() broadcasts data from one process to all other processes

Stencil Operations

1. Stencil operations are common in HPC, involving computations with neighboring data
    - c = (1 + 2 + 3 + 4) / 4
    - Repeat this computation for all elements
    - Parallel programming with 4 processes
    - In MPI, each process can access only its own areas
    - How can we compute ‘c’, requiring access to two other memory regions?



Stencil Operations

Communicating Boundary Information

1. Boundary element needs to be communicated
    - Using messages to send boundary data to other processes

Summary

1. Message passing is the key component of MPI programming
2. Importance of message passing demonstrated through stencil operations