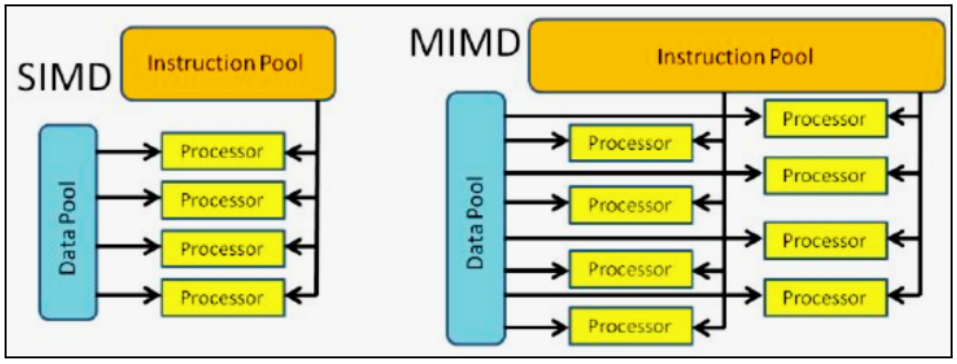
Q1.

a) To parallelize a computational problem, **how** would you decide to use shared memory or distributed memory paradigm. Explain.

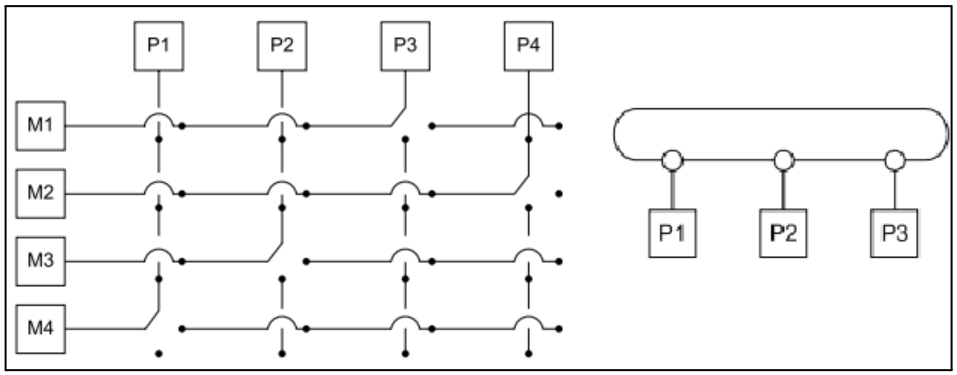
Shared Memory: is used when tasks share a single address space e.g. multicore CPUs or GPUs. It offers low-latency communication but is limited by memory size and scalability. Suitable for thread-level (fine-grained) parallelism (e.g. OpenMP). **Distributed Memory:** is for clusters where each process has its private memory and communicates via message-passing (e.g., MPI). It scales well for large problems but incurs higher communication overhead. Solve big data problems on large-scale systems (data centers).

b) Consider Flynn's taxonomy configurations: SIMD and MIMD. **Illustrate** how they are used in distributed memory paradigms?



MIMD (Multiple Instruction, Multiple Data): enables processors to execute independent instructions on separate data. Thus supporting asynchronous tasks and diverse workloads, such as in MPI-based applications. SIMD (Single Instruction, Multiple Data): is rarely used due to the lock step nature of execution.

c) **Illustrate** internal organization of **interconnects** used for both shared and distributed memory systems separately.



Shared Memory Systems: Use a bus-based interconnection or crossbar switches for communication between processors and shared memory. Buses are simple and cost-effective but can become bottlenecks. Crossbars provide higher bandwidth by enabling simultaneous connections. Distributed Memory Systems: Employ point-to-point interconnects such as single and multiple rings. These ensure scalability and low-latency communication between nodes with private memory.

d) **Analyze** the limitations of Amdahl's Law and the advantages of Gustafson's Law in parallel computing. **Give a numerical example** and explain terms used in both equations.

Amdahl's Law limits speedup by emphasizing the impact of serial portions of a task. Speedup S is $S = \frac{1}{(1-P)+\frac{P}{N}}$, where P is the parallel fraction, and N is the number of processors. Limitation: For large N, S is capped by $(1 - P)$. Example: For $P = 0.9$ and $N = 10$, $S = 5.26$.

Gustafson's Law scales workload with processors, redefining speedup S_G as $S_G = N - (1 - P) \cdot N$, focusing on effective parallel execution. Example: With $P = 0.9$ and $N = 10$, $S_G = 9.1$, showing better scalability.

e) **What** is the impact of false sharing on program execution? **Illustrate** with a dry run of a small C code snippet.

False sharing occurs when multiple threads access independent variables that reside on the same cache line, leading to unnecessary cache coherence traffic. This degrades performance due to frequent invalidation and reloading of cache lines, even when threads operate on separate data.

```c
#define SIZE 64  // Assume cache line size is 64 bytes
float arr[SIZE];
#pragma omp parallel for
for (int i = 0; i < 4; i++) {
    // Variables arr[0], arr[16], ... share the same cache line
    arr[i * 16] = i;
}
```

A four thread system, each thread is assigning value to a different array element. However, due to the fact that all of them fit in the same cache line, cache coherence protocols will invalidate the cache line.

f) **How** do you measure the execution time of a parallel section in both OpenMP and MPI C code, applying insights from Assignment #1 and the semester project?

To measure execution time in OpenMP, use omp_get_wtime() to capture timestamps before and after the parallel section. For MPI, use MPI_Wtime() in a similar manner. Ensure accurate timing by placing these calls outside synchronization or communication overheads. In both cases, calculate the elapsed time as the difference between end and start timestamps, and use MPI_Reduce in MPI to gather timings across processes for analysis, considering load imbalance and communication costs.

Q2 Part (a)

Create two parallel tasks: The first task increments an index variable i and adds a random number to array A[i]. The second task reads the value at A[i] into a variable z and decrements i. Ensure synchronization to maintain correct results.

```c
#pragma omp parallel {
    #pragma omp single { // First task
        #pragma omp task {
            #pragma omp atomic
            i++;  // Increment index i
            A[i] = rand() % 100;  // Add random number to A[i]
        }
    }
    #pragma omp single { // Second task
        #pragma omp task {
            int z = A[i];  // Read value at A[i] into z
            #pragma omp atomic
            i--;  // Decrement index i
        }
    }
}
```

Q2 Part b

Write optimal code to create five threads where one thread reads data from a global array B (containing 500K floating-point numbers), and then all threads compute the average value by distributing the computation.

```c
#pragma omp parallel num_threads(5) {
    int id = omp_get_thread_num();
    float local_sum = 0.0;
    // can't be done using #pragma omp master or single
    if (id == 0) { //reads data from B. N=500000
        for (i = 0; i < N; i++) A[i] = B[i];
    }
    #pragma omp barrier // other threads waits
    #pragma omp for
        for (i = 0; i < N; i++) {
            local_sum += A[i];
        }
    #pragma omp atomic
    sum += local_sum;
}
float average = sum / N;
```

Q2 part c

Distribute 100K elements of array C equally to 5 processes equally. Each process computes the sum of its portion and stores the result in a global array at the index matching its process ID, using only mpi_send and mpi_recv.

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
int elements_per_process = ARRAY_SIZE / NUM_PROCESSES; // 100000 / 5. No fraction
float local_sum = 0;
if (rank == 0) { // master
    for (int i = 1; i < NUM_PROCESSES; i++) { // own chunk not sent by master
        MPI_Send(&C[i * elements_per_process], elements_per_process, MPI_FLOAT, i, 0, MPI_COMM_WORLD);
    }
    for (int i = 0; i < elements_per_process; i++) local_sum += B[i];
    G[0] = local_sum;
    for (int i = 1; i < NUM_PROCESSES; i++) { // recieve other sums in global array
        MPI_Recv(&G[i], 1, MPI_FLOAT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
} else { //slaves
    MPI_Recv(B, elements_per_process, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    for (int i = 0; i < elements_per_process; i++) local_sum += B[i];
    MPI_Send(&local_sum, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
}
MPI_Finalize();
```

Q2 Part d

Use MPI_Scatter to distribute 5**0** float elements of array D among 10 processes **equally**, then use MPI_Gather to collect the average of 5 elements from each process into a global array E of size 10.

```
#define ARRAY_SIZE 50 // Because 5 elements needed for each process
#define NUM_PROCESSES 10
#define ELEMENTS_PER_PROCESS (ARRAY_SIZE / NUM_PROCESSES) // = 5
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
if (size != NUM_PROCESSES) { // Ensure MPI_COMM_WORLD = 10
    if (rank == 0) printf("Error: This program requires %d processes.\n", NUM_PROCESSES);
       MPI_Finalize();
       return -1;
}
// A batch of 5 elements distributed to all 10 processes
MPI_Scatter(D, ELEMENTS_PER_PROCESS, MPI_FLOAT,
   local_data, ELEMENTS_PER_PROCESS, MPI_FLOAT, 0, MPI_COMM_WORLD);
for (int i = 0; i < ELEMENTS_PER_PROCESS; i++) local_sum += local_data[i];
local_sum /= ELEMENTS_PER_PROCESS;
// average in local_sum will be gather from each of the 10 processes in E
MPI_Gather(&local_sum, 1, MPI_FLOAT, E, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
MPI_Finalize();
```

Q3 part a

HDFS mitigates Namenode failure with the following mechanisms:

- Checkpointing and Journal Logs: The Namenode persistently stores metadata in a filesystem (FS) image (snapshot) and journal logs (recent changes). On reboot, it merges them to restore state. This minimizes recovery time after a Namenode reboot by reducing the amount of log replay needed.
- Checkpointing node: Periodically fetches and merges FS image and journal logs to minimize recovery time. This reduces the primary Namenode's workload and ensures an updated backup. While it aids recovery, the Secondary Namenode is not a failover mechanism.
- Backup (Secondary) Namenode: This high availability option uses active-standby Namenode pairs with shared storage. On failure, a standby Namenode takes over seamlessly.

Q3 part b

```
virtual void Map(const MapInput& input) {
    const string& line = input.value();
    size_t zip_start = 0;
    size_t zip_end = line.find(',');
    // Extract zip_code and disease_ID. Can use blackbox functions
    string zip_code = line.substr(zip_start, zip_end - zip_start);
    size_t disease_start = line.rfind(',') + 1;
    string disease_id = line.substr(disease_start);
    // Emit key-value pair (zip_code:disease_ID, 1)
    string key = zip_code + ":" + disease_id;
    Emit(key, "1");
}
virtual void Reduce(ReduceInput* input) {
    int64 total_count = 0;
    // Iterate over all values for the current key
    while (!input->done()) {
        total_count += StringToInt(input->value());
        input->NextValue();
    }
    // Emit the final count for the current key
    Emit(IntToString(total_count));
}
```

Q3 part c

Figure # 1

```
{'zip_code': 23456, 'CNIC': 'ID5432', 'age': 45, 'disease_ID': 102}
{'zip_code': 12345, 'CNIC': 'ID7891', 'age': 62, 'disease_ID': 101}
{'zip_code': 34567, 'CNIC': 'ID2345', 'age': 35, 'disease_ID': 103}
{'zip_code': 23456, 'CNIC': 'ID8901', 'age': 28, 'disease_ID': 101}
{'zip_code': 12345, 'CNIC': 'ID3456', 'age': 71, 'disease_ID': 102}
```

Diagram:

Use your discretion to validate the diagram drawn; it should fit the following description and output values shown below. Figure # 1 shows data stored on a HDFS data node. Each line is assumed as a split. Five mappers will process each line and emit keys on the same node. The key will be sent to different reducers after the shuffle and sort phase (this involves processing on the mapper and each reducer's nodes), each reducer output will be saved on HDFS.

```
Map Phase Output:
('23456:102',1)('12345:101',1)('34567:103',1)('23456:101',1)('12345:102',1)

Shuffle and Sort Phase Output:
('12345:101',[1])('12345:102',[1])('12345:102',[1])('12345:101',[1])('34567:103',[1])

Reduce Phase Output:
for 12345 [('12345:101',1)('12345:102',1)]
for 23456 [('23456:101',1)('12345:102',1)]
for 34567 [('34567:103',1)]
```

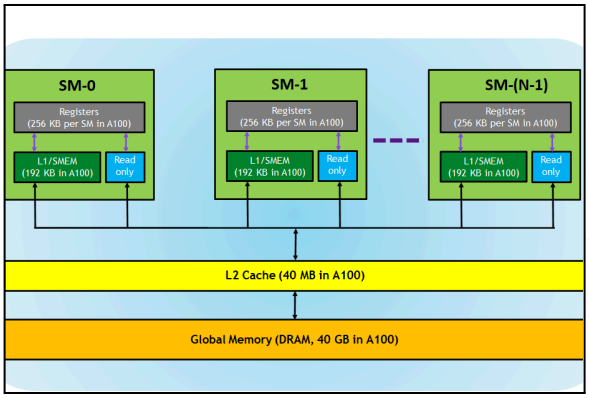
Q4. part a

```cuda
#include <cuda_runtime.h>
#include <stdio.h>
#include <stdlib.h>
#define N 1024  // Array size (can be adjusted)
// CUDA kernel for element-wise computation of Z = A + (B * C)
__global__ void calcZ(float *A, float *B, float *C, float *Z, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x; // GPU thread index
    if (idx < n) { // this if is necessary
        Z[idx] = A[idx] + (B[idx] * C[idx]);
    }
}
int main() {
    float *h_A, *h_B, *h_C, *h_Z; float *d_A, *d_B, *d_C, *d_Z;
    size_t bytes = N * sizeof(float);
    // Allocate host and device memories using malloc and cudaMalloc(); 8 calls
    // Copy host arrays to device. deduct marks if cudamemcpy d_Z);
    cudaMemcpy(d_A, h_A, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_C, h_C, bytes, cudaMemcpyHostToDevice);
    int threadsPerBlock = 256; // Configure kernel launch parameters
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    // Launch the CUDA kernel
    calcZ<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, d_Z, N);
    // Copy results back to host
    cudaMemcpy(h_Z, d_Z, bytes, cudaMemcpyDeviceToHost);
    // Free device memroy using cudaFree();
    // Free CPU memory using free()
    return 0;
}
```

Explanation of CUDA Calls

1. **cudaMalloc**: Allocates memory on the GPU device for arrays A, B, C, and Z. Also, cudaFree().
2. **cudaMemcpy**:
    - Transfers data between host and device.
    - Direction is specified with cudaMemcpyHostToDevice for inputs and cudaMemcpyDeviceToHost for results.
3. **Kernel Launch (computeZ<<<blocksPerGrid, threadsPerBlock>>>)**:
    - The grid is divided into blocksPerGrid blocks, each containing threadsPerBlock threads.
    - Each thread computes one element of Z.

Q4 Part b.



The CUDA kernel executes on NVIDIA GPU architecture by assigning each thread to compute one element of Z. Threads are organized into blocks, and blocks form a grid. Each Streaming Multiprocessor (SM) schedules and executes threads in groups called warps (32 threads). Threads access data from global memory. The kernel's thread indices (blockIdx, threadIdx) map threads to specific data elements. SMs handle threads independently, leveraging parallelism to process large arrays efficiently.

----------(X)----------