**Parallel and Distributed Computing**

**Task 3**

|  |  |
| --- | --- |
| **Total Marks: 25** | **Date: 5-8-25** |
| **Name: Muarij shakeel** | **Reg. No: l1f21bscs0030** |
| **Section: h1** | |

|  |  |  |  |
| --- | --- | --- | --- |
| **CLO** | **CLO STATEMENT** | **Bloom’s Taxonomy Level** | **PLO** |
| 3 | Analyze complex problem with shared memory programming with OpenMP | C4 | P3 |

### **Question # 1:**

**Consider a processor that implements both pipelining and superscalar execution. Explain a scenario were increasing the instruction issue width (i.e., making the processor more superscalar) results in no performance gain or even performance degradation. What architectural and program-related factors cause this behavior, and how can modern CPUs mitigate it?**

**Increasing the instruction issue width (e.g., from 4-way to 8-way superscalar) may not improve performance if the program lacks sufficient instruction-level parallelism (ILP). For example, in a loop with tight data dependencies (e.g., a[i+1] = a[i] + 1), instructions cannot execute in parallel due to data hazards, leaving wider issue slots unused. Performance might degrade due to resource contention (e.g., limited execution units, register ports) or front-end bottlenecks.**

**Key Factors:**

**Data/Control Hazards: Dependencies or branches limit parallelism.**

**Resource Contention: Insufficient ALUs, register ports, or memory bandwidth.**

**Front-End Bottlenecks: Fetch/decode stages cannot supply enough instructions.**

**Memory Latency: Cache misses stall the pipeline, negating wider issue benefits.**

**Mitigation in Modern CPUs:**

**Out-of-Order Execution (OoO): Dynamically reorder instructions to exploit ILP.**

**Speculative Execution: Predict branches and prefetch instructions.**

**Register Renaming: Eliminate false dependencies.**

**Advanced Caching: Reduce memory latency with prefetching and larger caches.**

### **Question # 2:**

### Given a NUMA-based multi-core system, describe how improper memory allocation by the operating system or a programmer can lead to non-deterministic and degraded performance. Compare this scenario with UMA and COMA architectures, and explain how each memory model deals with such allocation challenges.

In NUMA, memory access latency depends on the node’s proximity. Improper allocation (e.g., allocating memory on a remote node) causes high latency and bandwidth contention, leading to non-deterministic performance. For example, a thread running on Node 1 accessing data from Node 2 incurs remote memory latency.

**Comparison with UMA/COMA:**

UMA (Uniform Memory Access): All memory accesses have uniform latency (e.g., via a shared bus). Allocation location is irrelevant, but scalability is limited.

COMA (Cache-Only Memory Architecture): Data migrates automatically to the requesting node’s cache. Poor allocation may cause thrashing if data frequently migrates.

**Mitigation:**

NUMA-aware allocation: OS/programmer explicitly places data near compute threads.

COMA: Automatic migration via hardware-managed caches, but coherence overheads increase with mobility.

### **Question # 3:**

### In a COMA-based system, a distributed application frequently migrates large data blocks among nodes due to changing computation locality. Over time, performance worsens and memory access times become inconsistent.

### **Task:** Why can dynamic migration of data blocks in COMA lead to performance variability? How do COMA systems manage data replication and coherence compared to NUMA or UMA, and what challenges arise with heavy data mobility?

Frequent data migration increases coherence overhead (directory updates, invalidations) and network traffic, leading to inconsistent access times. Heavy migration can also cause thrashing, where data is repeatedly moved without reuse.

**COMA vs. NUMA/UMA:**

COMA: Data is dynamically replicated/migrated to local caches. Coherence is managed via directories.

NUMA: Data location is fixed; remote access incurs latency but avoids migration overhead.

UMA: Centralized memory with uniform latency but limited scalability.

**Challenges in COMA:**

Directory Scalability: Tracking data locations in large systems is complex.

Coherence Traffic: Frequent migrations increase network congestion.

Memory Bloat: Redundant copies may waste cache capacity.

### **Question # 4: Solve**

### A 2-D mesh interconnection network is built with 64 nodes. Calculate its diameter, bisection width, arc connectivity, and cost. How does it compare to a 2-D wraparound topology of the same size?

**i. 2-D Mesh (64 Nodes)**

Diameter: Longest path = 14 (e.g., corner-to-corner: 7 rows + 7 columns).

Bisection Width: 8 links (split along the middle column).

Arc Connectivity: 2 (minimum links to disconnect the network).

Cost: Total links = (rows × (cols−1)) + (cols × (rows−1)) = 8×7 + 8×7 = 112.

**2-D Wraparound (Torus):**

Diameter: 8 (4 hops in each dimension).

Bisection Width: 16 (split along two dimensions).

Cost: 128 links (each node connects to 4 neighbors).

### For a hypercube with N=1024 nodes: Compute the node degree, diameter, bisection width, arc connectivity, and cost.

Dimension: log₂(1024) = 10.

Node Degree: 10 (each node connects to 10 others).

Diameter: 10 (max hops between any two nodes).

Bisection Width: 512 (split into two 512-node cubes).

Arc Connectivity: 10 (node degree = connectivity).

Cost: (N × degree)/2 = (1024 × 10)/2 = 5120 links.

### A 3-D cube topology has 512 nodes. Calculate: diameter, bisection width, and cost.

Diameter: 21 (7 hops in each dimension).

Bisection Width: 64 (split along one dimension).

Cost: Total links = 3×8×8×7 = 1344.

### **Question # 5:**

### Write a multithreaded program to compute the **sum of a large array** using T threads.

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define T 4 // Number of threads

#define SIZE 1000000

int array[SIZE];

int partial\_sums[T] = {0};

typedef struct {

int start;

int end;

int thread\_id;

} ThreadArgs;

void\* compute\_sum(void\* arg) {

ThreadArgs\* args = (ThreadArgs\*) arg;

int sum = 0;

for (int i = args->start; i < args->end; i++) {

sum += array[i];

}

partial\_sums[args->thread\_id] = sum;

free(arg);

return NULL;

}

int main() {

pthread\_t threads[T];

int chunk\_size = SIZE / T;

// Initialize array

for (int i = 0; i < SIZE; i++) array[i] = 1;

// Create threads

for (int i = 0; i < T; i++) {

ThreadArgs\* args = malloc(sizeof(ThreadArgs));

args->start = i \* chunk\_size;

args->end = (i == T-1) ? SIZE : args->start + chunk\_size;

args->thread\_id = i;

pthread\_create(&threads[i], NULL, compute\_sum, args);

}

// Join threads

for (int i = 0; i < T; i++) pthread\_join(threads[i], NULL);

// Compute total

int total = 0;

for (int i = 0; i < T; i++) total += partial\_sums[i];

printf("Total sum: %d\n", total);

return 0;

}

Good Luck