**A blue and black logo

AI-generated content may be incorrect.A logo with a star and a letter u

AI-generated content may be incorrect.NATIONAL UNIVERSITY**

*of Computer and Emerging Sciences- Chiniot Faisalabad Campus*

**CS-3006 Parallel and Distributed Computing Assignment # 6**

**Submission Guidelines:**

* **No handwritten submissions Due Date: May 11, 2025**
* Code Submission: Submit your **code** as a separate file, clearly labeled with your

name and assignment details.

* Report Submission: Provide a **separate report** detailing your approach, algorithm,

parallelization strategy, and any challenges faced during implementation.

* Submission Format: Submit both the code and report online using

**rollno\_section\_assignmentno** format on GC. **(No handwritten submissions)**

* Deadline: Ensure that your submission is made before the specified deadline**. Late**

**submissions will not be accepted.**

* **Due Date: May 11, 2025**

**Task 1:**

Write a C program that:

* Creates n² threads (where n is a constant like 4) to compute each cell of a matrix product C = A × B.
* Each thread computes a single element C[i][j].
* Use mutexes to ensure that no two threads write to the same cell at the same time.  
  *(Hint: Implement a per-cell mutex array.)*

**Task 2:**

Design a producer-consumer model using:

* One producer thread
* Multiple consumer threads
* Condition Variables for synchronization (no busy waiting!)

The producer generates integers and consumers consume them into a shared array buffer of size 10.  
Implement:

* Blocking behavior when buffer is full (producer waits)
* Blocking behavior when buffer is empty (consumers wait)

**Task 3:**

A task queue dispatches worker threads to process tasks. If a thread encounters a fatal error (e.g., invalid task), all threads must cancel immediately and release resources.

Code:

pthread\_mutex\_t queue\_lock;

Queue task\_queue;

void\* worker(void\* arg) {

Task\* task = malloc(sizeof(Task));

pthread\_cleanup\_push(free, task); // Cleanup handler

while (1) {

pthread\_mutex\_lock(&queue\_lock);

if (queue\_empty(task\_queue)) {

pthread\_mutex\_unlock(&queue\_lock);

continue;

}

task = dequeue(task\_queue);

pthread\_mutex\_unlock(&queue\_lock);

if (validate(task) == ERROR) {

pthread\_cancel(pthread\_self()); // Trigger cancellation

}

process(task);

}

pthread\_cleanup\_pop(1);

return NULL;

}

Tasks:

* Identify the resource leak in this code if a thread is canceled while holding queue\_lock.
* Fix the code using pthread\_cleanup\_push and pthread\_cleanup\_pop to ensure:
  + The mutex is always released.
  + task memory is freed.
* Explain how cancellation points (e.g., pthread\_mutex\_lock) interact with cleanup handlers.

**Task 4:**

In a multithreaded server, each order placed by a user is handled by a separate thread.

* How would you ensure that the shared "inventory database" (e.g., stock of ingredients) is updated correctly without double-selling items?
* Would you use fine-grained locks (per-item) or a global lock (whole inventory)? Why?

**Task 5:**

Online Shopping Cart Checkout:  
Multiple users are checking out items at the same time, and an inventory system must ensure item counts are decremented correctly.

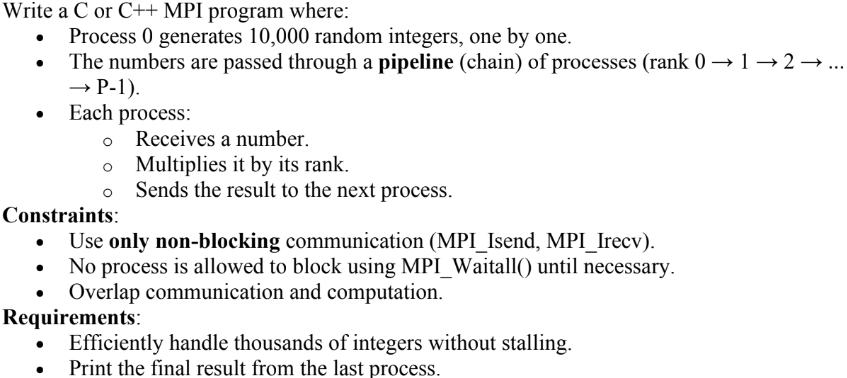
* Would you use pthread\_mutex\_lock or pthread\_mutex\_trylock in the checkout flow? Why?
* What would happen if you relied solely on trylock and the lock was often busy?

**Task 6:**

In a system like Origin 2000 (Distributed Shared Memory), memory locality matters.

* Explain why *remote memory access* is costlier than *local access*.
* How does NUMA (Non-Uniform Memory Access) architecture impact shared memory thread performance?

**Task 7:**

****