Assignment 4: MPI & Go & CUDA

Bar-Ilan CS 89-3312
Parallel Systems Programming

January 4, 2026

1 Part 1: MPI

In this part you will complete two small MPI programming tasks. Submit **two separate programs**: one for **Parallel Prefix Sum** and one for **Parallel Matrix Multiply**.

1.1 Task 1: Parallel Prefix Sum

1.1.1 Goal

Compute an **inclusive prefix sum** across MPI ranks using point-to-point communication.

1.1.2 Problem Definition

Let P be the number of MPI processes and rank $r \in \{0, \ldots, P-1\}$. Each process starts with:

$$x_r \leftarrow r$$

Compute on every rank:

$$y_r \;=\; \sum_{k=0}^{r} x_k$$

1.1.3 Requirements

1. Must work for any $P \geq 1$.
2. Must use `MPI_Send` and `MPI_Recv` (do **not** use `MPI_Scan`).
3. **Do not** compute the prefix by locally summing values from 0 to r (e.g., using a loop). You must obtain the prefix using communication between processes (i.e., passing values between ranks with MPI).
4. Each rank prints exactly one line:

   `rank=<r> x=<x_r> prefix=<y_r>`

1.2 Task 2: Parallel Matrix Multiply

1.2.1 Goal

Implement distributed matrix multiplication

$$C = A \cdot B$$

using MPI with 1D row-block distribution.

1.2.2 Starter Code: `matrix.h` and `matrix.c`

You are given `matrix.h` and `matrix.c`. They provide allocation and deterministic random generation for integer matrices. You **must use** this library to generate A and B.

1.2.3 Input (Command-Line Arguments)

Your program must receive **three** command-line arguments:

```
mpirun -np <P> ./matmul <N> <seedA> <seedB>
```

- N — matrix dimension ($N \times N$)
- `seedA` — seed used to generate matrix A
- `seedB` — seed used to generate matrix B

1.2.4 Computation and Output

- Rank 0 generates A using `seedA` and B using `seedB`.
- The program computes C in parallel.
- Rank 0 must hold the full C matrix at the end.
- Rank 0 prints a checksum of C:

  ```
  checksum(C)=<value>
  ```

1.2.5 Minimum Required Implementation (1D Row-Block Distribution)

Let P be the number of processes. Partition rows of A (and C) across processes:

$$\text{rows}(r) = \left\lfloor \frac{rN}{P} \right\rfloor \ldots \left\lfloor \frac{(r+1)N}{P} \right\rfloor - 1$$

Each rank:

1. Receives its block of rows of A from rank 0.
2. Receives the full matrix B (e.g., `MPI_Bcast`).
3. Computes its block of rows of C locally using the standard triple-loop algorithm.
4. Sends the local block of C back to rank 0 (e.g., `MPI_Gatherv`).

1.2.6 Correctness Requirements

1. Must work for any $N \geq 1$ and any $P \geq 1$.
2. Must handle cases where N is not divisible by P.
3. Rank 0 must hold the full C matrix at the end.

1.3 Submission Requirements (Part 1 — MPI)

Submit the following source files:

- `prefix_sum_sendrecv.c`
- `matmul_mpi.c`

2 Part 2: Go Concurrency

2.1 Food Delivery System Simulation

2.1.1 Goal

Implement a concurrent food delivery simulation using Go. The goal of this task is to practice:

- goroutines and channels
- **fan-in** (multiple producers into one stream)
- **fan-out** (dispatching work to multiple workers)
- bounded concurrency using tokens
- clean shutdown of concurrent pipelines

2.1.2 Overview

The system simulates a food delivery platform with:

- Restaurants that generate food orders
- A dispatcher that routes orders
- Delivery zones that process orders concurrently

All communication must be done using **channels**. You must not use HTTP, networking, or external services.

2.1.3 Command-Line Interface

Your program must accept the following command-line arguments:

```
go run . --n <N> --restaurants <R> --zones <Z> \
        --tokens <t1,t2,...,tZ> --seedA <seedA> --seedB <seedB>
```

Note: Look on the end of part 2 for notes.

- `N`: total number of orders
- `R`: number of restaurants
- `Z`: number of delivery zones
- `tokens`: number of concurrent deliveries allowed per zone
- `seedA`: seed for order generation (deterministic)
- `seedB`: seed for processing delays (deterministic)

2.1.4 Orders

Each order must have:

- a unique `orderId` in $\{0, \ldots, N-1\}$
- a `restaurantId`
- a `foodType` in $\{0, \ldots, Z-1\}$

Orders must be generated deterministically using `seedA`.

2.1.5 System Architecture

Restaurants (Producers)

- There are R restaurant goroutines.
- Together, they must generate exactly N orders.
- Each restaurant sends orders to the dispatcher.

Dispatcher (Fan-in / Fan-out)

- Collects orders from all restaurants (fan-in).
- Routes each order to the delivery zone corresponding to its `foodType` (fan-out).
- When all restaurants are done, closes all zone channels.

Delivery Zones (Workers)

- There are Z zones.
- Each zone processes orders concurrently.
- Each zone has a fixed number of **tokens** limiting the maximum number of concurrent deliveries.
- An order may only be processed if a token is available.
- After processing, the token must be returned.

Processing time should include a small randomized delay based on `seedB`.

2.1.6 Output Format

All output must be printed to **stdout** using the following strict line-based format:

```
CREATED order=<id> restaurant=<rid> type=<zone>
DISPATCHED order=<id> zone=<zone>
STARTED order=<id> zone=<zone>
COMPLETED order=<id> zone=<zone>
DONE total=<N>
```

- Each order must be `CREATED`, `DISPATCHED`, and `COMPLETED` exactly once.
- `STARTED` and `COMPLETED` events must respect token limits.
- The final line must be exactly:

  `DONE total=<N>`

2.1.7 Required Data Structures

You must implement the following data structures (you may add extra fields if you want, but you must keep these fields and names).

```
type Order struct {
    OrderID int // unique in [0..N-1]
    RestaurantID int // in [0..R-1]
    FoodType int // zone index in [0..Z-1]
}
```

Event You must represent system events using the following structure. Events are used only for printing the required output format.

```
type Event struct {
    Kind string // "CREATED" | "DISPATCHED" | "STARTED" | "COMPLETED" | "DONE"
    OrderID int
    RestaurantID int
    Zone int
}
```

Tokens per Zone To limit concurrency per zone, you must use a token channel for each zone:

```
// tokens[z] is a buffered channel with capacity = tokensCount[z]
tokens := make([]chan struct{}, Z)
```

A delivery in zone `z` may start only after receiving a token from `tokens[z]`, and it must return the token when finished.

Token Limits (from input) The maximum number of concurrent deliveries per zone is provided by the command-line argument `-tokens t1,t2,...,tZ`. For zone z, the token limit is t_z.

Your implementation must enforce:

$$\#\{\text{orders STARTED but not COMPLETED in zone } z\} \leq t_z$$

2.1.8 Correctness Requirements

1. Exactly N orders are completed.
2. No order is completed more than once.
3. Orders are dispatched to the correct zone.
4. Token limits are never exceeded.
5. The program terminates cleanly (no deadlocks, no goroutine leaks).

2.1.9 Restrictions

- Do **not** use HTTP, networking, files, or databases.
- Do **not** use global variables for synchronization.
- Use channels and goroutines for all communication.

2.1.10 Submission

Submit the following files:

- `main.go`
- Any additional `.go` files you created

2.1.11 Recommended File Structure and Submission

You may implement the solution in a single file, but we strongly recommend splitting the code into multiple files for clarity.

Required Your project must include:

- `main.go` — contains `package main` and `func main()`, parses command-line arguments, and runs the simulation.

Recommended (you may create these files)

- `types.go` — defines `Order` and `Event` structs.
- `restaurant.go` — restaurant goroutines (order generation).
- `dispatcher.go` — fan-in / fan-out dispatcher.
- `zone.go` — zone workers and token enforcement.
- `logger.go` — printing events in the required format.

You may create additional `.go` files if you want.

How we run your code We will run your submission only through:

```
go run . --n <N> --restaurants <R> --zones <Z> \
        --tokens <t1,t2,...,tZ> --seedA <seedA> --seedB <seedB>
```

Therefore:

- Your program must be runnable with `go run .`
- It must not require any manual input during runtime.
- All output must be printed to `stdout` in the required format.

What does "printed to stdout" mean? All required output must be printed to the terminal using standard printing functions such as `fmt.Print`, `fmt.Println`, or `fmt.Printf`. This output is called **stdout** (standard output) and it is what we capture when we run your program.

Do **not** write output to files, do **not** use HTTP servers, and do **not** require interactive input.

Important: Go modules (`go.mod`) We will run your Go submission using:

```
go run . --n <N> --restaurants <R> --zones <Z> \
        --tokens <t1,t2,...,tZ> --seedA <seedA> --seedB <seedB>
```

Therefore, your `go_files` directory must contain a `go.mod` file. If you create the project from scratch, generate it by running (inside `go_files`):

```
go mod init <any_name>
```

Go build flag `-p` (important on the BIU server) The flag `-p` belongs to the **Go tool** (`go run` / `go build`), and it controls **how many packages are compiled in parallel**.

- `-p 1` means: compile packages **one at a time** (slower, but uses fewer processes/threads).
- This flag affects **compilation only**. It does **not** change your program's goroutines or runtime concurrency.
- This is **not related to MPI** (do not confuse it with `mpirun -np`).

On the BIU server, if you get compilation errors such as `resource temporarily unavailable`, run:

```
go run -p 1 . --n <N> --restaurants <R> --zones <Z> \
        --tokens <t1,t2,...,tZ> --seedA <seedA> --seedB <seedB>
```

3 Part 3: CUDA Matrix Operations

3.1 Objective

In this assignment, you will implement two essential matrix operations using **CUDA**. The goal is to gain hands-on experience with parallel computing on the GPU by leveraging **threads** and **blocks** for efficient computation.

3.2 How to Run CUDA Online

You may use the following online platform to write and run your CUDA code:

- **LeetGPU**

  LeetGPU allows you to compile and execute CUDA programs without having a local GPU.

3.3 Running CUDA Locally (Optional)

If you have access to a local machine with an NVIDIA GPU, you may run your CUDA code locally. However, make sure that your solution also runs correctly on LeetGPU.

3.4 The Assignment

You are given two matrices A and B of size 4×4, represented as 1D arrays:

```
int A[16] = {1, 2, 3, 4,
             5, 6, 7, 8,
             9, 10, 11, 12,
             13, 14, 15, 16};

int B[16] = {2, 4, 6, 8,
             10, 12, 14, 16,
             18, 20, 22, 24,
             26, 28, 30, 32};
```

3.4.1 Task 1: Matrix Addition

Implement a CUDA program that performs **matrix addition**:

$$C = A + B$$

Each CUDA thread should compute exactly **one element** of the resulting matrix.

3.4.2 Task 2: Matrix Multiplication

Implement a CUDA program that performs **matrix multiplication**:

$$C = A \cdot B$$

Each CUDA thread should compute exactly **one element** of the resulting matrix.

3.5 Example Output for Matrix Addition

```
Matrix A:
1  2  3  4
5  6  7  8
9 10 11 12
13 14 15 16

Matrix B:
2  4  6  8
10 12 14 16
18 20 22 24
26 28 30 32

Matrix C (A + B):
3  6  9 12
15 18 21 24
27 30 33 36
39 42 45 48
```

4 Submission Guidelines

1. The assignment may be completed individually or in pairs.
2. Submission is through **submit**.
3. Make sure your CUDA code compiles and runs without errors or warnings on **LeetGPU**.
4. Make sure your MPI and Go programs run without errors on the **BIU server**.
5. At the top of every submitted file, include a comment with your ID number and full name. For example:

   ```
   // 123456789 - Israela Israeli
   ```

6. The submitted zip file (`ass4.zip`) must include the following files:
   - `prefix_sum_sendrecv.c`
   - `matmul_mpi.c`
   - A directory named `go_files` containing all source files for the Go part
   - `addition.cu`
   - `multiplication.cu`
7. **Deadline: 25/1**

   Note: An additional week has been granted for this assignment; therefore, grace days not be used.