DESIGN DOCUMENT

Ritu Basumatary (2024471)
Arja Kaur Anand (2024104)

Project: SimpleMultithreader (C++11 header-only, Pthreads), assignment 5

1. Objective (short)

Implement a header-only `simple-multithreader` that exposes two `parallel_for` APIs (1D and 2D) accepting C++11 lambdas and using POSIX threads. Each `parallel_for` call must create exactly `numThreads` active threads (main thread + `numThreads-1` pthreads), run the work, print execution time, and terminate threads on return.

2. High-level design

- **Header-only:** All implementation in `simple-multithreader.h` so user programs include the header only.
- **APIs:**
    - ```
      void parallel_for(int low, int high,
      std::function<void(int)>&& lambda, int numThreads);
      ```
    - ```
      void parallel_for(int low1, int high1, int low2, int high2,
      std::function<void(int,int)>&& lambda, int numThreads);
      ```
- **Threading model:** No thread pool. For each API call:
    - Create `numThreads - 1` pthreads.
    - Use main thread to execute the remaining partition.
    - Join all created pthreads before return.
- **Work distribution:** Static contiguous chunking — split iteration range(s) into `numThreads` contiguous subranges (balanced by `base + 1` to the first `rem` threads when remainder exists).

- **2D mapping:** Flatten `(i,j)` to single index `flat` in `[0, rows*cols)`; map back via `i = flat / cols + low1`, `j = flat % cols + low2`.

- **Timing:** Use `std::chrono::steady_clock` to compute and print elapsed milliseconds per `parallel_for` call.

3. Key implementation details

- **Lambda sharing:** Wrap provided lambda in `std::shared_ptr<std::function<...>>` so it can be safely referenced by all threads.

- **Thread argument:** Small `struct` allocated on heap for each pthread containing `start_idx`, `end_idx`, lambda pointer, and for 2D `low1`, `low2`, `width2`. The pthread frees the struct at the end.

- **Entry functions:** Two pthread entry functions (`thread_entry_1d` and `thread_entry_2d`) that iterate the assigned range and call the lambda. Exceptions are caught and swallowed (do not propagate across pthread boundary).

- **Error handling:**

    - Validate `numThreads` (fallback to 1) and ranges (`low >= high` is no-op).

    - Check `pthread_create` return and on failure join previously created threads and throw `std::runtime_error`.

    - For very large flattened sizes exceeding `INT_MAX` throw `runtime_error`.

- **Modularity:** Reusable helper functions for splitting ranges and timestamping; minimal duplication between 1D and 2D logic.

4. Performance & correctness considerations

- **Static chunking** is simple and low-overhead; it is suitable for uniform-cost iterations (vector addition). For highly skewed iteration costs (irregular 2D work), dynamic scheduling would be better — not implemented because assignment forbids thread pools/queues.

- **Memory safety:** Each thread only writes/read its partition. No shared mutable state inside the header other than the lambda pointer.

- **Scalability:** Overhead of creating threads per `parallel_for` call is acceptable for coarse-grained work; for many short calls the overhead may dominate.

5. Build & test

- **Build (WSL / Linux):** `g++ -std=c++11 -pthread vector.cpp -o vector_test` and `g++ -std=c++11 -pthread matrix.cpp -o matrix_test`

- **Run examples:** `./vector_test 4 48000000` and `./matrix_test 4 1024`

- Tests provided by instructor (vector & matrix) are used unchanged.

6. Files included

- `simple-multithreader.h` — header-only implementation

- `vector.cpp` — 1D vector-add test program (uses `parallel_for(0, size, lambda, numThreads)`)

- `matrix.cpp` — 2D allocation / multiplication test (uses both APIs)

- `Makefile` — build targets for `vector_test` and `matrix_test`

7. Contribution (equal split)

Work was divided **equally** between the two group members; each performed an equal share of design, implementation, testing, and documentation.

Ritu

- Implemented the core header (`simple-multithreader.h`) including:

    - Thread argument structs, pthread entry functions, static range splitting, 1D & 2D `parallel_for` implementations, timing, and error checks.

- Integrated lambda-sharing and memory-safety measures.

- Wrote tests and validated vector/matrix examples on WSL; recorded timings.

- Wrote build instructions and prepared Makefile.
- Performed debugging and resolved build issues on WSL/VS Code

Arja

- Reviewed and refined the threading model, API signatures, and chunking logic.

- Implemented example programs (`vector.cpp`, `matrix.cpp`) or verified instructor examples compile and run unchanged.
- Integrated lambda-sharing and memory-safety measures.