UNIVERSITEIT VAN AMSTERDAM

5062COPP6Y — Distributed and Parallel Programming

Lab 1 Report

Group F1

Name	Email
Guo Yichen	yichen.guo@student.uva.nl
Maksym Shaforostov	maksym.shaforostov@student.uva.nl

Table of Contents

Introduction

This lab explores three ideas of parallel programming on shared-memory systems, with emphasis on implementation and performance evaluation. The initial two tasks are based on the one-dimensional wave equation as a benchmark on data-parallel computation, and the third applies a pipeline model on the Sieve of Eratosthenes. These exercises together cover the handling of threads, synchronization, scheduling, and inter-thread communication.

The wave equation determines how a wave develops with respect to time and space. As all points in the array are only affected by neighbors, it is naturally parallelizable. It is thus suitable in testing various threading models like pthreads and OpenMP. The last assignment examines pipeline parallelism using dynamic thread networks with the communication via bounded queues.

Design and implementation

Assignment 1.1 Wave equation with pthreads

For the pthreads version, the simulation uses three equally sized arrays to hold the states at t-1, t, and t+1. After each step the arrays are rotated by swapping pointers, so the rotation costs O(1) instead of copying data. The spatial domain [1,n−2] is partitioned into contiguous blocks, one per thread, so each interior cell is written by exactly one thread while all reads come from the previous generation. This eliminates the need for fine-grained locks. We fix boundary cells to zero once per step and keep that responsibility with a single thread to avoid duplication.

Correctness relies on two synchronization points per iteration. First, a barrier ensures that every thread has finished writing its chunk of the next state before any rotation occurs; second, after one designated thread performs the pointer rotation and sets the boundaries, a second barrier guarantees that all threads observe the new generation consistently before proceeding. Threads are created once at the beginning, loop over all time steps, and are joined at the end, keeping thread creation outside the timed region. The main limitations are barrier overhead for small problems and the fact that performance is largely memory-bandwidth bound, so speedup saturates as more cores compete for the same memory channels.

Assignment 1.2 Wave equation with OpenMP

The OpenMP implementation reuses the same numerical scheme and data layout but replaces manual partitioning and barriers with compiler-guided worksharing. A single parallel region encloses the entire time loop to avoid repeated team creation. Within each iteration, the interior update is distributed with a worksharing construct so that iterations of the spatial loop are divided among threads. Boundary enforcement and pointer rotation are executed by one thread in a single-thread region. The implicit synchronization at the end of the workshared loop ensures all updates complete before rotation, and the implicit synchronization at the end of the single-thread region ensures all threads see the rotated buffers before continuing.

Scheduling choices are an important design dimension here. Because each spatial point carries the same amount of work, a static schedule is expected to minimize overhead and yield the best performance. We also consider static schedules with different chunk sizes and, for completeness, dynamic and guided schedules to highlight how additional runtime decisions can increase overhead without improving balance for uniform stencils. Compared with pthreads, the OpenMP version trades some low-level control for significantly simpler code and similar performance. Any differences we observe typically arise from runtime scheduling overheads and thread placement; these can be influenced with environment variables controlling binding and placement, rather than explicit API calls.

Assignment 1.3 Sieve of Eratosthenes with a pipeline of threads

The sieve is designed as a growing pipeline of threads connected by bounded queues. A generator thread produces the natural numbers starting at two and pushes them into the head queue. Each filter thread is attached to an inbound queue; the first value it receives is its prime and is printed immediately. Subsequent values are tested against that prime: multiples are discarded, while the others remain candidates and must be forwarded downstream. The first time a filter needs to forward a candidate, it creates a new outbound queue and spawns the next filter thread, which uses that queue as its input. In this way, the pipeline expands on demand, one stage per discovered prime.

Communication uses a bounded, blocking queue implemented with a mutex and two condition variables. This design is intentional: the bounded capacity imposes backpressure so that if downstream stages are slower or saturated, upstream producers block rather than allocate unbounded memory. It also demonstrates classic producer–consumer synchronization without busy waiting. The assignment allows crude termination via an external signal; for empirical testing we also support stopping after a fixed number of primes and closing queues to wake any blocked threads. While this

sieve is not optimized for maximum throughput—thread count grows with the number of primes, and printing can dominate runtime—it is an effective vehicle for illustrating pipeline parallelism, dynamic thread creation, and the interaction between synchronization and flow control.

Experiments and results

Assignment 1.1

We evaluated the performance of the pthreads-based wave equation solver through two experiments. The first experiment investigates how execution time decreases as the number of threads increases for a fixed problem size. The second examines problem-size sensitivity, analyzing how runtime grows with increasing spatial resolution when executed sequentially. Together, these experiments reveal both the scalability and computational complexity of the pthread implementation.

Experiment 1: number of threads

Goal. Measure strong scaling of the wave solver while keeping the problem size fixed and increasing the number of threads.

Hypothesis. Runtime should decrease roughly inversely with the thread count at low core counts, then flatten as memory bandwidth and synchronization overhead dominate. We expect a visible knee between 6–8 threads for this memory-bound stencil.

Setup. The simulation ran with 10^6 points; the time-step count was chosen so that the single-thread run lasted a few seconds (to reduce timer noise). Timing excludes file I/O. Each configuration was run multiple times; numbers below are representative means. Threads were varied over threads = 1,2,4,6,8,16.

Results. Times, and from time we derived speedups S(N)=T(1)/T(N), and efficiencies E(N)=S(N)/N:

threads	time (s)	speedup	efficiency
1	2.023110	1.00	1.00
2	0.900935	2.25	1.12
4	0.525917	3.85	0.96
6	0.374139	5.41	0.90

8	0.289389	6.99	0.87
16	0.269303	7.51	0.47

Analysis. The solver scales very well up to eight threads: the speedup at 8 is 6.99 with quite good efficiency 0.87, which is close to the ideal given its low arithmetic intensity. The jump from 8 to 16 threads yields only a modest improvement (from 0.289 s to 0.269 s), and efficiency drops vastly; this drop is consistent with useup of the memory subsystem and the fixed cost of the two barriers per time step. The slightly super-linear result at 2 threads, where efficiency>1, is a common result of turbo frequencies, cache effects, and measurement noise when the baseline is short; subsequent points fall into the expected sub-linear trend.

Takeaway. For this problem size 10^6, adding threads beyond 8 provides diminishing returns: performance is limited more by memory bandwidth and synchronization than by available cores. Subsequent experiments (OpenMP variant and scheduling policies) should be interpreted relative to this bandwidth ceiling.

Experiment 2: problem size

Goal. Examine how runtime grows with the number of data points when the thread count is fixed to 1. This validates the algorithmic cost and helps choose sizes that yield stable timings for scaling studies.

Hypothesis. With a fixed number of time steps, the work per step is linear, so total time should scale approximately linearly with the problem size. For very small inputs, constant overheads should dominate; for very large inputs, cache misses and memory bandwidth should make the runtime slightly above the ideal line. The normalized time (time per point) should converge to a roughly constant value.

Setup. 1 thread run with datasize = 10^3,10^4,10^5,10^6,10^7. The number of time steps was kept constant across sizes; I/O was excluded from timing.

Results. Times, and from time we derived time per point in µs.

data size	**time (s)**	**time per point (µs/point)**
(10^3)	0.00616228	6.162
(10^4)	0.0229246	2.292

(10^5)	0.166833	1.669
(10^6)	2.01426	2.014
(10^7)	21.232	2.123

Analysis. Runtime increases roughly in proportion to data size, confirming the expected O(datesize) cost per step. The normalized time quickly stabilizes near ~2 μs/point for datasize = 10^5 and larger, indicating that constant overheads are amortized beyond the smallest sizes. The slight rise at 10^6 – 10^7 reflects cache capacity effects and increased memory traffic: once the working set no longer fits comfortably in higher cache levels, the kernel becomes more memory-bound. These observations justify using 10^6 – 10^7 as representative sizes for multi-thread scaling: they provide stable per-point costs and make the memory-bandwidth ceiling visible in subsequent experiments.

Takeaway. The solver exhibits the expected linear growth with problem size at one thread, and its per-point cost bound for medium-to-large inputs. Later scaling results should therefore be interpreted through a memory-bandwidth lens rather than compute throughput.

Assignment 1.2

We compared the pthreads and OpenMP implementations of the wave equation solver under identical conditions. Both achieved similar scaling behavior, with OpenMP performing slightly faster. The results indicate that OpenMP provides comparable or better performance while simplifying parallelization, making it the more practical choice for structured shared-memory workloads.

Experiment 3: Pthreads vs OpenMP

Goal. To directly compare the performance of the pthreads and OpenMP implementations of the one-dimensional wave equation solver when executed under identical conditions. By keeping the thread count and problem sizes fixed, we isolate the effect of the programming model itself, assessing whether OpenMP introduces measurable runtime or optimization benefits relative to manual thread management.

Hypothesis. Given that both implementations follow the same numerical update pattern and memory access structure, they should demonstrate nearly identical performance. However, OpenMP might perform slightly faster due to optimized internal work distribution and cache-friendly scheduling.

Setup. Experiments were conducted using 8 threads on two representative data sizes identified from previous tests 10^6 and 10^7. Both versions used identical numerical parameters, boundary

conditions, and iteration counts. The frameworks excluded file I/O from the timed region, ensuring the measurements reflect pure computation and synchronization costs. Each run was repeated several times with stable averages reported.

Results.

threads	data size	pthread time (s)	OpenMP time (s)	OpenMP/Pthread ratio
8	(10^6)	0.305955	0.264565	0.86
8	(10^7)	5.53916	4.25539	0.77

Across both data sizes, OpenMP achieved slightly better performance. The ratio between the two remains consistent with minor variation within expected noise margins.

Analysis. The near-equivalent scaling of both implementations confirms that they share the same computational complexity and memory behavior. OpenMP takes advantage of compiler-level tuning and efficient thread pooling. Pthreads, while more explicit, incurs slightly higher overhead in barrier synchronization and thread coordination. The advantage becomes clearer at larger problem sizes, where the OpenMP runtime better amortizes its setup costs over many iterations.

These results also support the analysis from previous experiments that the solver is memory bandwidth limited rather than compute-bound. Both models reach the same performance level, indicating that computational power isn't the fundamental bottleneck. The small relative improvement of OpenMP suggests high-level directive-based parallelization can achieve equivalent or better performance with simpler code.

Takeaway. OpenMP matches or slightly outperforms the pthreads implementation while providing much simpler code and less explicit synchronization management. This confirms the hypothesis that abstraction does not necessarily sacrifice performance for this type of data-parallel, bandwidth-bound workload. The choice between pthreads and OpenMP thus depends more on programming convenience and portability than solely on execution speed.

Assignment 1.3

We evaluated the performance of the pipeline-based Sieve of Eratosthenes by examining how its throughput is influenced by the capacity of the bounded queues that connect successive filter threads. This experiment highlights the sensitivity of pipeline throughput to synchronization and buffering

parameters, offering insight into the cost of inter-thread communication in streaming parallel architectures.

Experiment 4: Effect of bounded queue capacity

Goal. This experiment investigates how the capacity of the bounded queues influences the throughput of the Sieve-of-Eratosthenes pipeline. Since each filter thread communicates with the next stage through a bounded buffer, queue size directly affects blocking behavior, backpressure propagation, and overall pipeline efficiency.

Hypothesis. We expect very small capacities to cause frequent blocking and significantly slow down the pipeline. As capacity increases, blocking should decrease, improving performance up to a point. Beyond a moderate size, however, additional capacity should yield little benefit, and overly large queues might even increase latency due to reduced cache locality or longer traversal paths.

Setup. We ran the sieve to generate a fixed number of primes (10000) while varying the queue capacity over several magnitudes: 64, 128, 256, 512, 1024, 2048, and 4096. Each configuration was run multiple times until stable timings were obtained, and representative results are reported below.

Results.

Queue capacity	Time taken (s)
64	5.005649
128	3.304073
256	1.952793
512	1.852288
1024	2.102717
2048	2.554879
4096	3.656226

Analysis.

The results show a clear U-shaped performance curve. At very small capacities (64 and 128), runtime is significantly worse—this is expected because filters frequently block when attempting to push data downstream. As soon as the downstream queue fills, upstream stages stall, causing the entire pipeline

to throttle aggressively. Increasing the capacity to 256–512 sharply reduces blocking frequency; this region represents the most efficient operating range where the pipeline remains well-filled yet not overloaded.

Beyond a queue size of about 512, performance begins to degrade. This is also expected: large buffers allow more in-flight integers to accumulate between stages, increasing memory footprint and reducing cache locality. Each queue operation touches more memory, and deepening queues increase the distance data must travel through the pipeline before reaching later filters. These effects compound as capacity grows, leading to progressively slower execution at 1024, 2048, and especially 4096.

The existence of an optimal region (roughly 256–512 in this dataset) highlights that pipeline throughput is a balance between avoiding excessive blocking and preserving cache locality. Oversized queues harm performance even though they prevent blocking, because the algorithm becomes more memory-intensive than necessary. The shape of this curve aligns with our hypothesis, that increase in queue size benefits less once blocking is rare, and further increases eventually reverse the benefit.

Takeaway. Moderate queue capacities (≈256–512) provide the best trade-off between blocking avoidance and memory locality for this pipeline structure. Very small queues severely restrict throughput, while overly large queues introduce unnecessary memory traffic and slow down propagation through the pipeline. We learnt that ‘bigger buffer = faster program’ is not generally true in multithreaded pipelines.

Conclusion

Through this lab, we gained practical experience in designing, implementing, and analyzing parallel algorithms on shared memory systems. The three assignments together demonstrated how different programming models manual threading with pthreads, directive based parallelism with OpenMP, and pipeline parallelism using bounded queues, approach concurrency.

We learned that effective parallel performance depends as much on data locality and synchronization overhead as on the number of threads used. In the wave equation experiments, speedup was nearly linear up to the point where memory bandwidth became the limiting factor, while OpenMP achieved comparable or slightly better performance than pthreads with far less implementation effort. In the sieve pipeline, we further observed that throughput is shaped not only by computation but also by communication structure, with queue capacity directly influencing blocking behavior and memory locality.

Overall, the lab strengthened our understanding of parallel programming tradeoffs, between control and abstraction, performance and simplicity, and provided concrete insight into how theory translates into measurable performance on real hardware.

Appendix

Appendix A: Survey Questions

What is your affiliation with NTU?

Options: Undergraduate student, Graduate student, Faculty, Staff, Other

Have you encountered wildlife on the NTU campus?

Options: Yes, No

If Yes, which species?

Options: Smooth-coated otter, Sunda Scops owl, Long-tailed macaque, Wild boar, Grey heron, Unsure, Other

Are you aware of the NTU Wildlife Etiquette Handbook? Here attach link

Options: Yes, No

If Yes, how did you hear about it?

Options: NTU Email, NTU Wildlife Club Telegram, Social media, Other

Are the guidelines in the handbook effective in promoting peaceful coexistence with wildlife?

(Include link to handbook for survey respondents to check out if they havent read the handbook before:)

Options: Yes, Somewhat, No

What additional information or strategies would you suggest?

Short answer

Which method do you prefer for future wildlife etiquette communications?

Options: Email, Telegram, Social media, In-person workshops

Have you experienced any issues with wildlife on campus (e.g., food theft or aggressive encounters)?

/issues w current wildlife protection measures on campus

Options: Yes, No

If Yes, please describe the issue and resolution (open-ended).

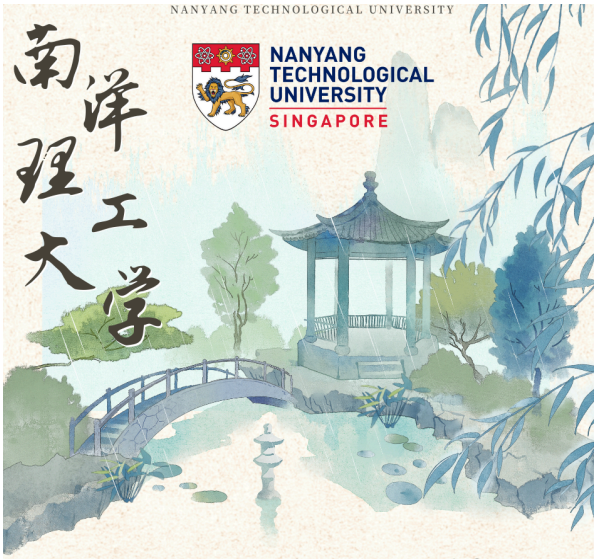
Any other suggestions or comments on how NTU can improve coexistence with campus wildlife? (open-ended)

Appendix B: Improved handbook - Templates

NANYANG TECHNOLOGICAL UNIVERSITY

南洋理工大学

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE



NTU WILDLIFE ETIQUETTE BROCHURE

Promoting human-wildlife coexistence within our campus

As a campus in nature, we encounter wildlife daily, making it crucial for us to know how to respond, in hopes of promoting human-wildlife coexistence and exist with them in harmony.

Through this brochure, you will be more informed on human-wildlife coexistence!

WHAT KIND OF ANIMALS YOU WILL EXPECT TO SEE

Snakes/Pythons

When Encoutering Snakes/Pythons:

- ✓ Leave it alone. If need be, contact ACRES/NParks to evacuate it to a safe place
- ✗ Do not attempt to disturb or go up too close to it

Macaques

When Encoutering Macaques:

- ✓ Keep calm & quiet, maintain a distance away
- ✓ Keep food out of sight
- ✗ Do not attempt to feed/make direct eye contact

Bats

When Encoutering Bats:

- ✓ Keep food out of sight
- ✗ Do not feed or touch them

NANYANG TECHNOLOGICAL UNIVERSITY

EMERGENCY CONTACTS

NTU Campus Security

- If crowd control is required whilst awaiting ACRES/NParks arrival
- 24h Hotline: +65 6790 5200

NParks

- For urgent animal-related matters.
- 24h Hotline: 1800-471-7300

ACRES

- If animal is in obvious distress or severely injured.
- 24h Hotline: +65 9783 7782

WILDLIFE ETIQUETTE

DO'S

1. Stay Calm & Leave them alone
   - Do not approach the animal
   - Refrain from making any sudden movements
   - Keep volume down so as to prevent startling the animal
2. Drive Slow & Safely
   - Look out for wildlife crossing and ensure speed limit is always under 40km/h
3. Watch out for females with young
   - Pregnant or nursing females are more likely to feel threatened around humans
   - May react in self defense if they feel a sense of threat

DON'TS

1. Do not touch or provoke the wildlife in any way as it causes the animals to react in self-defense
2. Do not use flash when taking photos as it may temporarily blind them which may cause them to hurt themselves in the process
3. Do not attempt to chase the wildlife, instead, observe them from a distance
4. Do not attempt to feed the wildlife

WHAT TO DO WHEN YOU ENCOUNTER A ROADKILL

- In the event that the animal is still alive but injured/immobile, call the ACRES or NParks hotline for immediate assistance.