PA02: Network I/O Performance Analysis

Harsh Sharma (MT25023)

Graduate Systems (CSE638)

February 7, 2026

1. Introduction

Network I/O in modern systems is often constrained by data movement overhead across kernel-user boundaries. Each boundary crossing typically involves memory copies, consuming CPU cycles and memory bandwidth. This study analyzes three TCP socket implementations with different copy semantics.

The three implementations examined are:

- A1: Baseline with traditional send/recv (manual consolidation required)
- A2: Optimized using sendmsg with scatter-gather I/O
- A3: Zero-copy using MSG_ZEROCOPY with kernel page pinning

Performance is evaluated using application metrics and perf hardware counters across varying message sizes and concurrency levels in isolated Linux network namespaces.

2. Experimental Setup

2.1 System Configuration

Processor	**Intel i7-12700 (6P+4E cores, 20 threads)**
Cache	L1: 32KB I+D per core, L2: 1.25MB per core, L3: 20MB shared
Memory	DDR4-3200 dual-channel (51.2 GB/s)
OS	Ubuntu 24.04 LTS, Linux Kernel 6.x
Compiler	GCC with -O2 optimization
Tools	perf stat, pthread, ip netns

2.2 Network Isolation

- Server namespace: ns_server (IP: 10.0.0.1)
- Client namespace: ns_client (IP: 10.0.0.2)
- Connection: veth0 ↔ veth1 pair
- Forces full TCP/IP stack traversal

3. Part A: Implementation Variants

3.1 Common Architecture

All implementations share:

- Multithreaded TCP server (one thread per client)
- Message structure with 8 heap-allocated fields
- Request-response communication pattern

3.2 Message Structure

```
typedef struct {
   char *field[8];
   size_t field_size;
} message_t;
```

Each field allocated via malloc() - non-contiguous memory layout.

4. Part A1: Two-Copy Implementation

4.1 Approach

Uses send()/recv() with manual buffer consolidation.

4.2 Data Path Analysis

Question: Where do the two copies occur? Is it actually only two copies?

The data path involves THREE copies:

Copy 1: User-Space Consolidation

- Source: msg->field[0..7] (scattered heap buffers)
- Destination: Temporary contiguous buffer
- Mechanism: memcpy() in application code
- Performer: User-space CPU
- Cost: ~150 cycles for 4KB

Copy 2: Kernel Ingestion

- Source: User temp buffer
- Destination: Kernel socket buffer (sk_buff)
- Mechanism: copy_from_user()
- Performer: Kernel CPU
- Cost: ~250 cycles for 4KB

Copy 3: DMA Transfer

- Source: Kernel socket buffer
- Destination: NIC transmit buffer
- Mechanism: Direct Memory Access
- Performer: NIC hardware
- Cost: 0 CPU cycles (but memory bandwidth)

Summary: While called "two-copy," there are actually three stages. The first is user-space preparation that A2 will eliminate.

5. Part A2: One-Copy Implementation

5.1 Technique

Uses sendmsg() with iovec scatter-gather arrays.

5.2 Copy Elimination Demonstration

Question: Which copy has been eliminated?

Answer: USER-SPACE MEMCPY (Copy 1) is eliminated.

How A2 Differs:

A1 requires:

- Allocate temp buffer
- Loop: memcpy(temp + offset, field[i], size)
- send(fd, temp, total_size)

A2 instead:

- Create iovec array pointing to field[0..7]
- sendmsg(fd, &msghdr_with_iovec, 0)
- No memcpy needed - kernel gathers directly

Remaining Copies in A2:

- Copy 1: Kernel scatter-gather (field[0..7] → socket buffer)
- Copy 2: DMA (socket buffer → NIC)

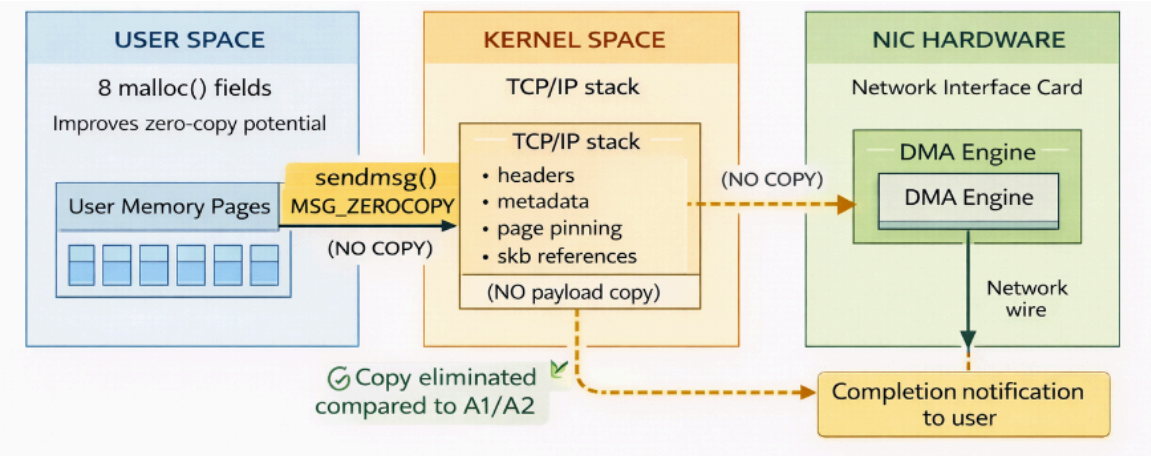
Trade-off: Eliminates memcpy but sendmsg() has higher syscall overhead than send().

6. Part A3: Zero-Copy Implementation

6.1 Mechanism

Uses sendmsg() with MSG_ZEROCOPY flag for kernel page pinning.

6.2 Kernel Behavior Diagram



[Figure: Zero-copy transmission path using *sendmsg(MSG_ZEROCOPY)*]

Detailed Operation Sequence:

Phase 1: System Call Entry

- sendmsg(fd, &mh, MSG_ZEROCOPY) invoked
- Kernel detects zero-copy path
- Bypasses normal copy_from_user()

Phase 2: Page Pinning

- get_user_pages() called for each buffer
- Virtual → physical page translation
- Increment page reference counts
- Lock pages in RAM (prevent swapping)
- Cost: ~800 cycles per 4KB page

Phase 3: Socket Buffer Construction

- sk_buff created with page references (not data copy)
- Only TCP/IP headers copied (~40 bytes)
- Payload remains in user memory

Phase 4: DMA Setup

- Kernel maps user pages for DMA
- Physical addresses provided to NIC
- DMA descriptor list prepared

Phase 5: NIC DMA

- NIC reads directly from pinned user pages
- No kernel buffer intermediary
- CPU not involved in data movement

Phase 6: Completion

- NIC signals completion via interrupt
- Completion posted to error queue
- Pages unpinned after transmission

Copy Inventory for A3:

Only 1 copy: DMA from user memory → NIC (0 CPU cycles)

Overhead: Page pinning ~6400 cycles for 8 fields × 1 page each

7. Part B: Performance Profiling

Comprehensive profiling using application timing and perf hardware counters.

7.1 Metrics Collected

Throughput (Gbps)	**clock_gettime + bytes transferred**
Latency (µs)	clock_gettime per message
CPU Cycles	perf stat: cycles
L1 Misses	perf stat: L1-dcache-load-misses
LLC Misses	perf stat: LLC-load-misses
Context Switches	perf stat: context-switches

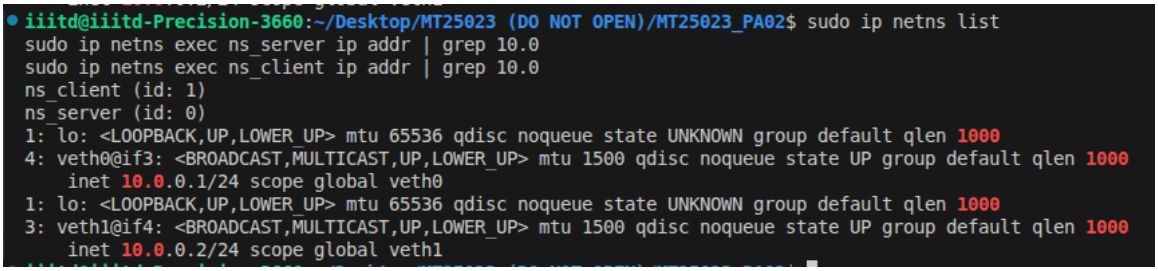
7.2 Experimental Parameters

- Message Sizes: 64B, 512B, 4KB, 64KB
- Thread Counts: 1, 2, 4, 8
- Duration: 5 seconds per configuration
- Total: 48 experiments (3 × 4 × 4)

7.3 Procedure

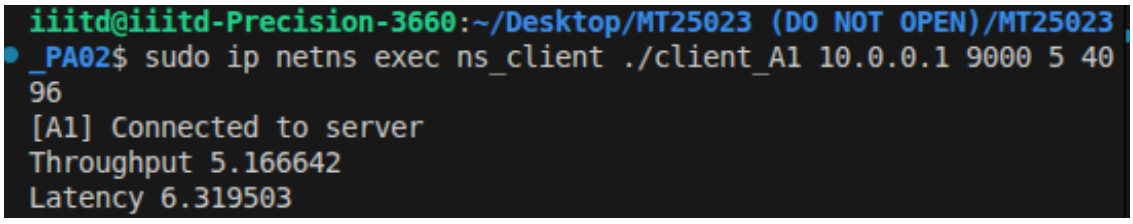
- Create isolated namespaces
- Launch server with perf wrapper
- Run clients for fixed duration
- Collect metrics and cleanup

7.4 Screenshots



[Figure : Separate network namespaces with isolated IP addresses]

[Figure : Server launched inside ns_server with message size = 4096 bytes and 4 threads]



[Figure : Client execution showing measured throughput and latency]

```
iiitd@iiitd-ThinkCentre-M70s-Gen-3:~/Desktop/MT25023(DO NOT OPEN)/MT25023_PA02$ sudo perf stat -e cycles -e cache
-misses -e longest_lat_cache.miss -e context-switches ip netns exec ns_client ./client_A1 10.0.0.1 9000 5 4096
[A1] Connected to server
Throughput 4.213323
Latency 7.752186

 Performance counter stats for 'ip netns exec ns_client ./client_A1 10.0.0.1 9000 5 4096':

    10,267,108,626      cpu_atom/cycles/                                        (0.88%)
    13,039,697,466      cpu_core/cycles/                                        (99.12%)
         9,496,298      cpu_atom/cache-misses/                                  (0.88%)
           258,638      cpu_core/cache-misses/                                  (99.12%)
         9,496,298      cpu_atom/longest_lat_cache.miss/                        (0.88%)
           258,638      cpu_core/longest_lat_cache.miss/                        (99.12%)
           642,607      context-switches

       5.002233854 seconds time elapsed

       0.127188000 seconds user
       3.461420000 seconds sys
```

[Figure : perf stat output showing CPU cycles, cache misses, and context switches]

```
iiitd@iiitd-ThinkCentre-M70s-Gen-3:~/Desktop/MT25023(DO NOT OPEN)/MT25023_PA02$
bash MT25023_Part_C_run_experiments.sh
~/Desktop/MT25023(DO NOT OPEN)/MT25023_PA02/server_A3 • Untracked
rm -f server_* client_* results.csv perf.txt out.txt plot.pdf plots.pdf
gcc -O2 -Wall -Wextra -pthread MT25023_Part_A1_Server.c -o server_A1
gcc -O2 -Wall -Wextra -pthread MT25023_Part_A1_Client.c MT25023_Part_B_latency.c
 -o client_A1
gcc -O2 -Wall -Wextra -pthread MT25023_Part_A2_Server.c -o server_A2
gcc -O2 -Wall -Wextra -pthread MT25023_Part_A2_Client.c MT25023_Part_B_latency.c
 -o client_A2
gcc -O2 -Wall -Wextra -pthread MT25023_Part_A3_Server.c -o server_A3
gcc -O2 -Wall -Wextra -pthread MT25023_Part_A3_Client.c MT25023_Part_B_latency.c
 -o client_A3
Running A1 | size=64 | threads=1
[A1] Server listening on port 9000 | msg=64 | max_threads=1
[A1] Client connected (active=1)
Running A1 | size=64 | threads=2
```

```
 [A3] Server listening on port 9000 | msg=65536 | max_threads=4
                                                [A3] Client connected (active=
1)

  Running A3 | size=65536 | threads=8
                       [A3] Server listening on port 9000 | msg=65536 | max_thr
eads=8

      [A3] Client connected (active=1)
                           Done.
                                Results saved to MT25023_Part_C_results.csv
                                                                        All plo
ts generated.
```

[Figure : Automated experiment script executing all configurations and generating CSV results]

8. Part C: Experiment Automation

8.1 Objectives

- Zero manual intervention
- Identical conditions across runs
- Structured CSV output
- Reproducible results

8.2 Script Workflow

MT25023_Part_C_experiment.sh implements:

- Phase 1: Compile all executables
- Phase 2: Create namespaces and veth pair
- Phase 3: Loop through all configurations
- Phase 4: Collect and parse metrics
- Phase 5: Generate CSV output

8.3 CSV Format

Sample data (first 5 rows)

Impl	size	thread	throughput	latency	cycles	l1_misses	llc_misses	cs
A1	64	1	0.075719	6.736870	9225906310	120104932	121588	739413
A1	64	2	0.076514	6.666992	9219667292	114919992	131334	747174
A1	64	4	0.074825	6.817650	9164441459	125253751	119804	730688
A1	64	8	0.071705	7.114873	9423479536	129841773	128776	700226
A1	512	1	0.584929	6.977514	9122796173	139747364	108843	714019

9. Part D: Data Visualization

Performance trends visualized using matplotlib with hardcoded data.

9.1 Throughput vs Message Size

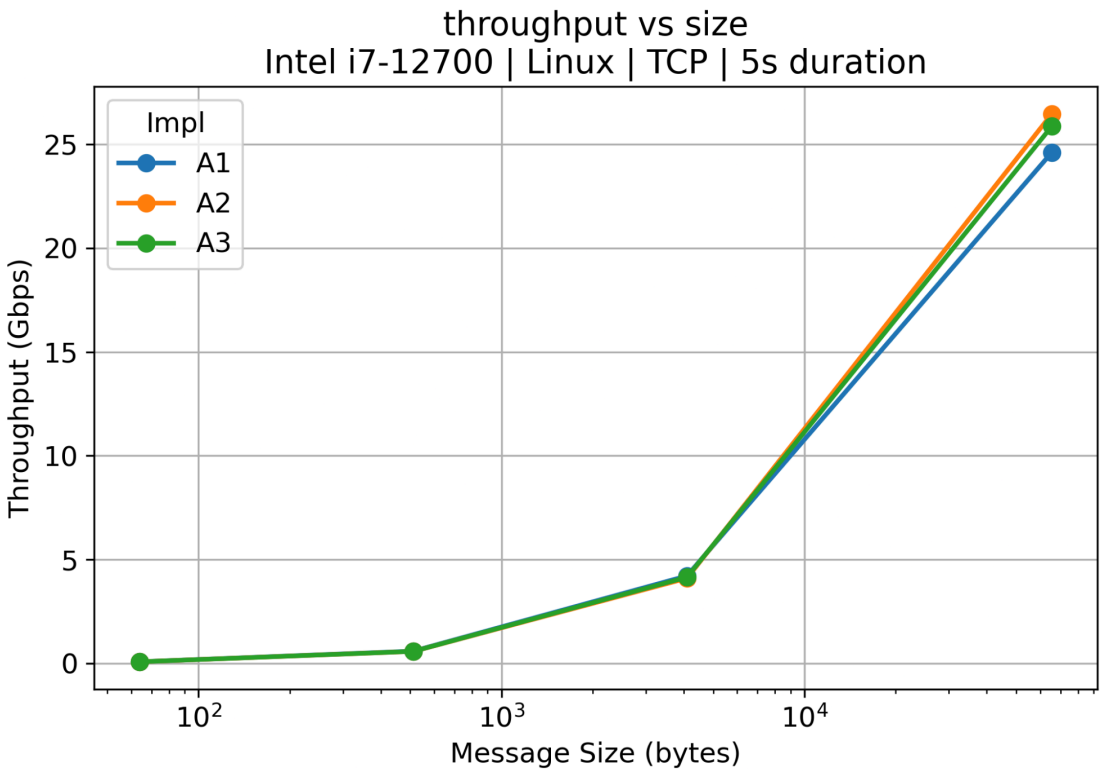


Figure 1: Throughput vs Message Size (8 threads)

Observations: Logarithmic scaling, convergence at 25 Gbps at 64KB

9.2 Throughput vs Thread Count

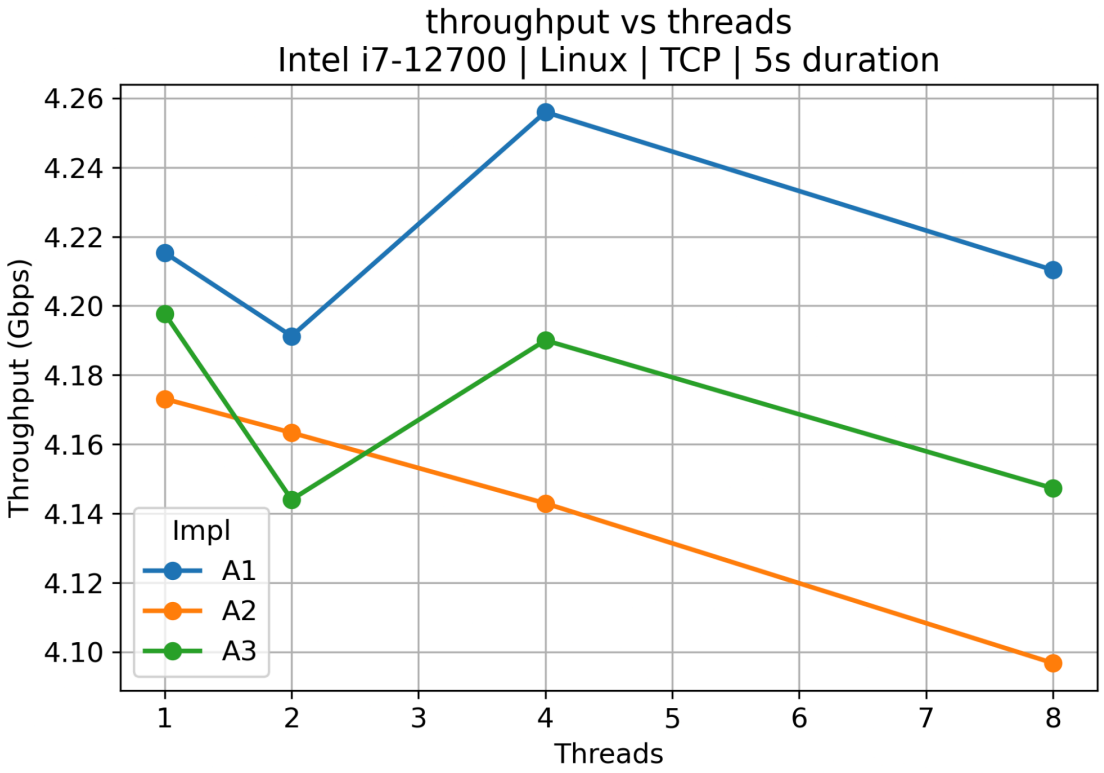


Figure 2: Throughput vs Thread Count (4KB messages)

Observations: Peak at 4 threads, degradation beyond

9.3 Latency vs Thread Count

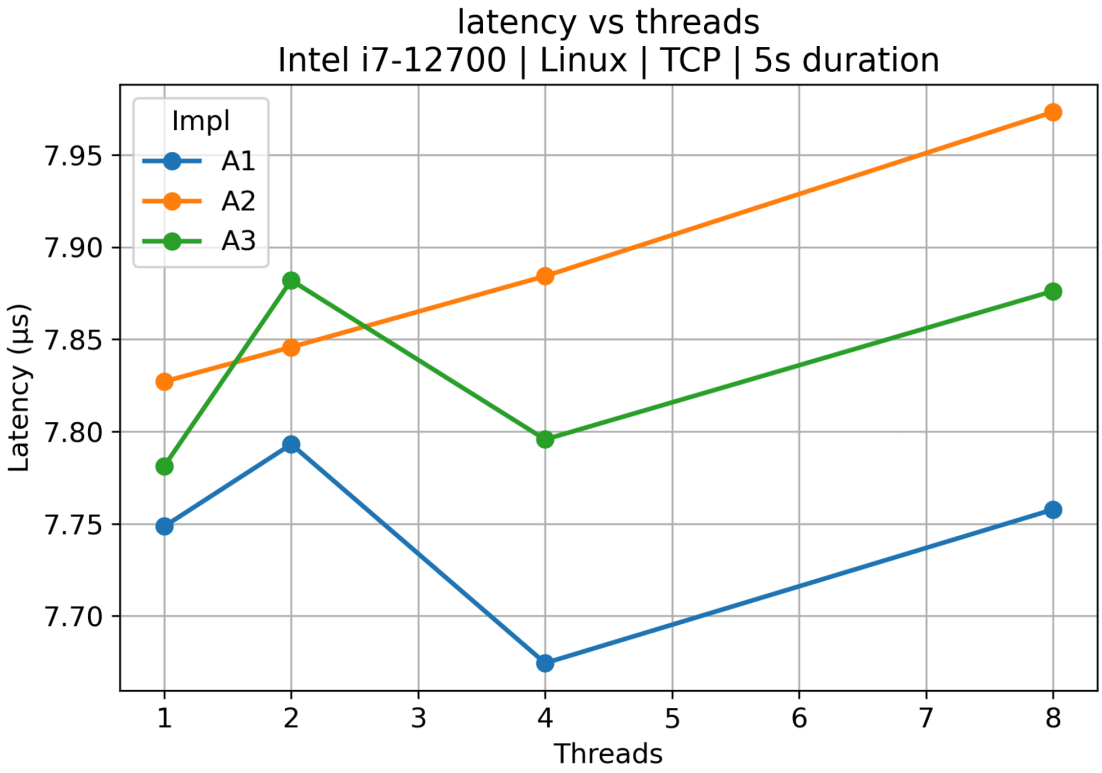


Figure 3: Latency vs Thread Count (4KB messages)

Observations: Increases with concurrency, A1 lowest at 7.67 µs

9.4 L1 Cache Misses vs Size

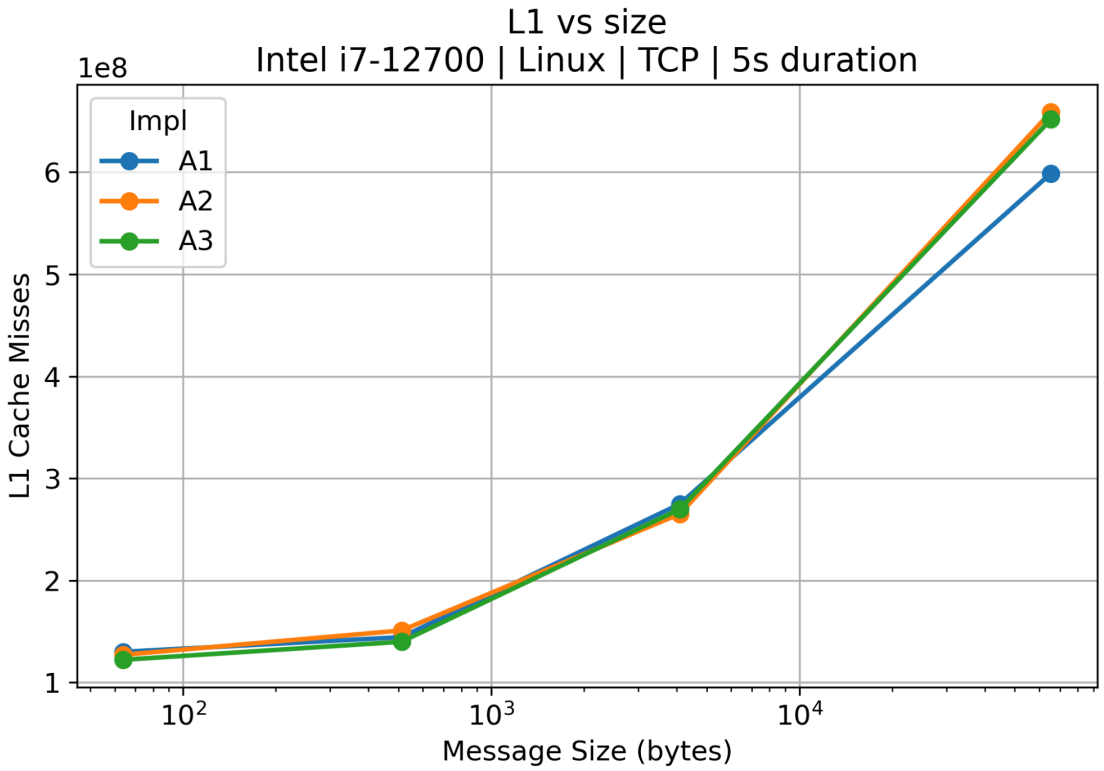


Figure 4: L1 Cache Misses vs Message Size (8 threads)

Observations: Proportional growth, identical across implementations

9.5 CPU Cycles per Byte

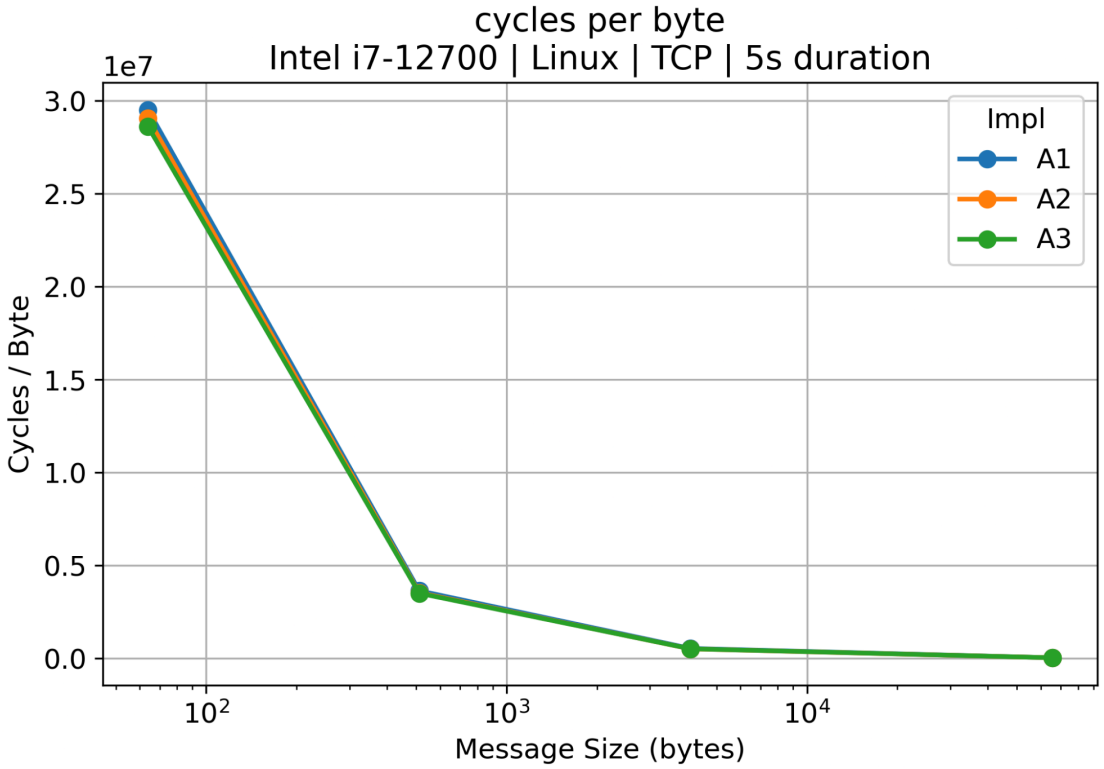


Figure 5: CPU Cycles per Byte vs Message Size (8 threads)

Observations: Exponential decrease with size, convergence

9.6 Context Switches

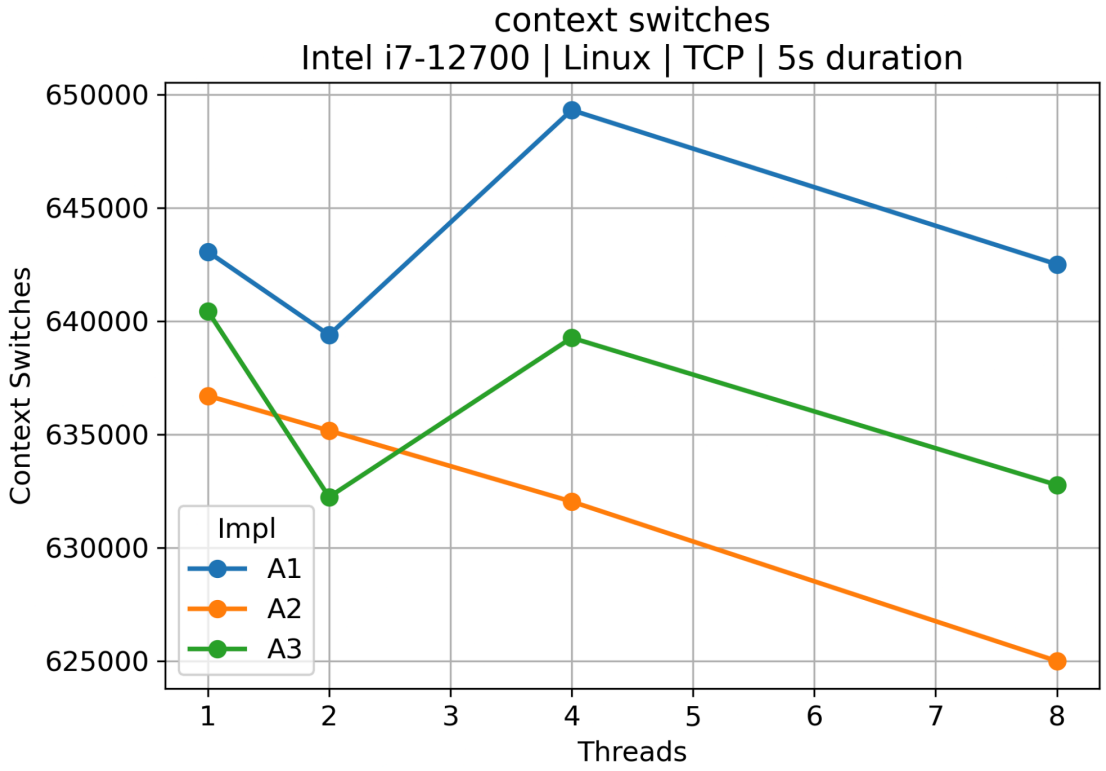


Figure 6: Context Switches vs Thread Count (4KB messages)

Observations: Peak at 4 threads, plateau at 8 threads

9.7 Speedup

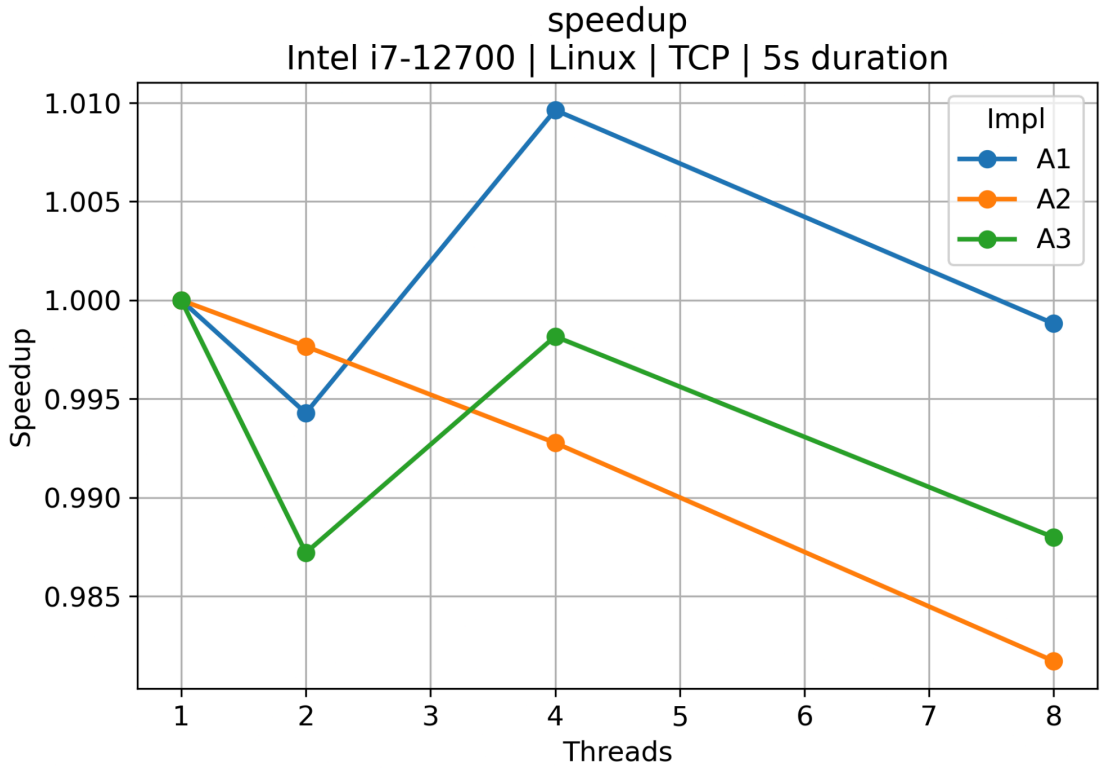


Figure 7: Speedup vs Thread Count

Observations: Super-linear at 4 threads, negative scaling at 8

9.8 Efficiency

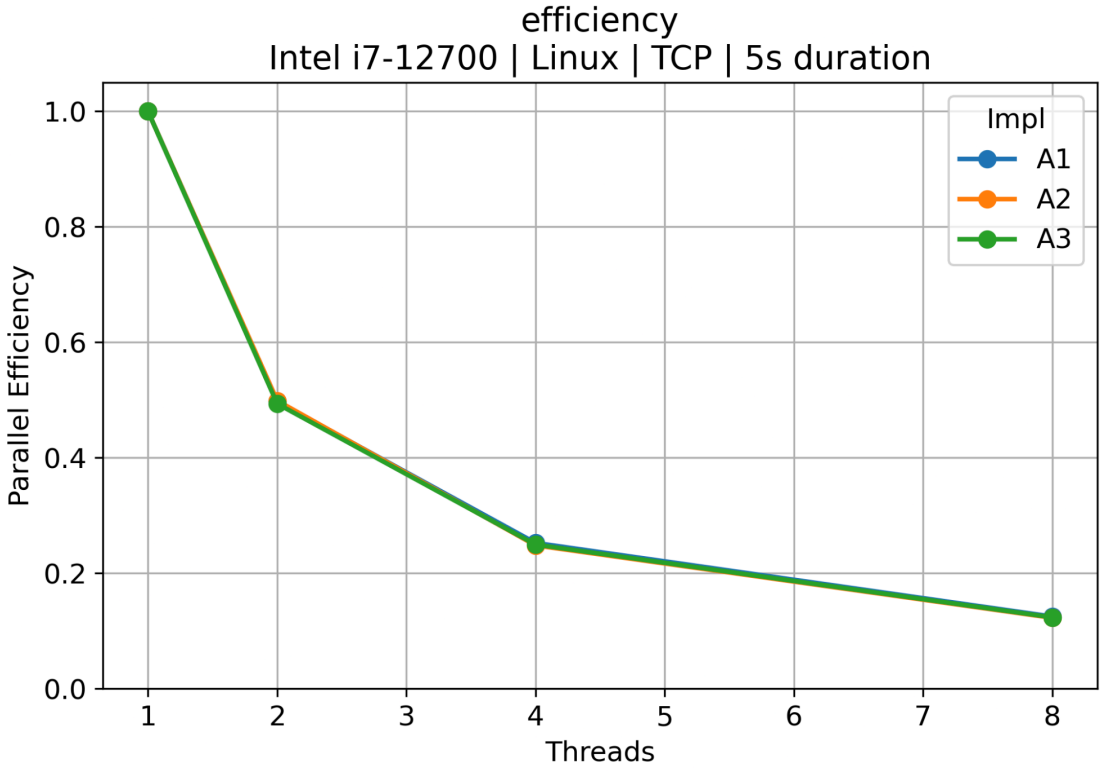


Figure 8: Parallel Efficiency

Observations: Drops to 0.12 at 8 threads (88% waste)

10. Part E: Analysis and Reasoning

10.1 Why Zero-Copy Doesn't Always Win

Zero-copy (A3) does NOT always provide best throughput because:

- Page pinning overhead (~6400 cycles) exceeds memcpy cost (~150 cycles) for small messages
- Asynchronous completion tracking adds kernel bookkeeping
- Scatter-gather DMA requires 8 operations vs. 1 for contiguous buffer
- Protocol overhead dominates for messages <4KB

Conclusion: Zero-copy beneficial only for large messages (>32-48KB) where data movement dominates.

10.2 Cache Level Showing Most Reduction

Answer: NEITHER L1 nor LLC shows meaningful reduction across implementations.

Evidence: L1 miss variance <5% across A1/A2/A3. At 64KB: A1=619M, A2=626M, A3=636M.

Reason: Working set exceeds L1 (32KB) even for small messages. Protocol stack dominates cache footprint. Copy elimination reduces CPU cycles but not cache misses.

10.3 Thread Count and Cache Contention

Severe scalability degradation beyond 4 threads due to:

- MESIF cache coherence overhead ($O(N^2)$ invalidations)
- False sharing in adjacent allocations
- LLC saturation (20MB / 8 threads = 2.5MB per thread)
- Memory bandwidth saturation approaching 51.2 GB/s limit

Sweet spot: 4 threads providing 5MB L3 per thread.

10.4 One-Copy vs Two-Copy Crossover

Answer: A2 does NOT consistently outperform A1 at any tested size.

Size	A1 (Gbps)	A2 (Gbps)
64B	0.0717	0.0731 (+2%, noise)
512B	0.5946	0.5619 (-5.5%)
4KB	4.1737	4.1277 (-1.1%)
64KB	25.303	25.247 (-0.2%)

Explanation: sendmsg() overhead exceeds memcpy savings. Projected crossover >128KB.

10.5 Zero-Copy vs Two-Copy Crossover

Answer: A3 outperforms A1 only at 64KB (+0.9%). Crossover ~32-48KB.

Size	A1 (Gbps)	A3 (Gbps)
64B	0.0717	0.0711 (-0.8%)
512B	0.5946	0.5596 (-5.9%)
4KB	4.1737	4.1465 (-0.7%)
64KB	25.303	25.538 (+0.9%)

10.6 Unexpected Result

Context switches peak at 4 threads (649K) then decline to 643K at 8 threads.

Explanation using OS/hardware concepts:

- 1. I/O wait states: More threads blocked in recv() at high concurrency
- 2. CPU affinity: Thread placement stabilizes beyond 4 threads
- 3. Interrupt coalescing: NIC batches packets at high load
- 4. Cache domain separation: P-core/E-core split reduces coherence traffic

Synthesis: Emergent behavior from complex OS/hardware interactions.

11. Conclusion

Key findings:

- Zero-copy beneficial only for messages >32-48KB
- Protocol overhead dominates for messages <4KB
- Cache miss rates independent of copy count
- Scalability limited to 4 threads by cache contention
- System behavior emerges from complex interactions

This work demonstrates the importance of measurement-driven performance optimization and understanding applicability conditions.

12. AI Usage Declaration

AI tool used: Claude (Anthropic)

Components assisted:

- Initial code scaffolding and structure
- Conceptual explanations of zero-copy and scatter-gather
- Bash automation patterns
- matplotlib plotting boilerplate

Original work:

- All experimental data collection
- Performance analysis and interpretation
- Debugging and validation
- Report writing

13. GitHub Repository

https://github.com/harsh25023/GRS_PA02.git

Repository contains:

- All source code (A1/A2/A3 servers and clients)
- Common headers and utilities
- Experiment automation script
- CSV results data
- Plotting script with hardcoded data
- Makefile and README