Parallel & Distributed Computing
Lab/Lecture Handout: Cache Coherence in SMP (UMA)

Topics: Coherence Problem & Solutions, Ping-Pong (True & False Sharing), Profiling with perf

1. Learning Outcomes

After completing this handout, students should be able to:

- Explain the cache coherence problem in SMP/UMA systems and why it occurs.
- Describe coherence solutions: snooping vs directory, write-invalidate vs write-update, MESI-family protocols.
- Identify and explain cache-line *ping-pong* due to *true sharing* and *false sharing.*
- Implement simple pthread programs that trigger ping-pong behavior.
- Use Linux `perf` to measure performance impact and interpret key metrics.

2. Prerequisites

- Linux machine (Ubuntu/Debian recommended) with `gcc`, `pthread`, and `perf`.
- Basic knowledge of caches (L1/L2/L3) and threads.
- Two or more CPU cores available.

3. Background: SMP (UMA) and the Coherence Problem

3.1. SMP (UMA)

In a classic Shared-Memory Multiprocessor (SMP) with Uniform Memory Access (UMA), all cores share the same physical memory, and each core typically has private caches (L1/L2) and sometimes a shared last-level cache (LLC).

3.2. What is the cache coherence problem?

When multiple cores cache the same memory location (more precisely, the same **cache line**), each core may hold a copy. If one core writes to that line, other cores may still hold stale copies. Without coherence, subsequent reads could observe old values.

3.3. Why it happens (the real villain: cache lines)

Coherence granularity is almost always a **cache line** (often 64 bytes). This means:

- Two different variables inside the same line can still interfere.
- Independent writes can trigger invalidations *even if variables are logically unrelated.*

4. Solutions to Cache Coherence

4.1. Policy-level solutions

- **Write-invalidate (most common):** A writer invalidates other cached copies, then writes with exclusive ownership.
- **Write-update (less common):** A writer broadcasts updated data to other caches (can cause high traffic).

4.2. Mechanism-level solutions

- **Snooping-based coherence:** Caches monitor a shared interconnect (bus/ring). Requests are broadcast; caches respond/invalidate accordingly. Good for small/medium core counts.
- **Directory-based coherence:** A directory tracks which caches share each line. Invalidation is targeted (not broadcast). Scales better for many cores/NUMA.

4.3. Protocol-level solutions (MESI family)

A coherence protocol defines valid transitions for cache-line states. The classic MESI states:

- **M (Modified):** Only this cache has the newest data; memory is stale.
- **E (Exclusive):** Only this cache has it; memory matches.
- **S (Shared):** Multiple caches may have it; memory matches.
- **I (Invalid):** Not valid in this cache.

Key idea: Before writing, a core must obtain an exclusive/modified state, forcing other cached copies to become invalid (or otherwise ensuring they do not remain stale).

5. The Ping-Pong Effect (Coherence Traffic)

5.1. True sharing ping-pong

Two threads repeatedly write the **same variable**. The cache line must migrate between cores, causing frequent invalidations and ownership transfers.

5.2. False sharing ping-pong

Two threads write **different variables**, but those variables happen to be in the **same cache line**. Coherence still operates at cache-line granularity, so the line bounces anyway.

6. Lab Part A: Program Examples (pthreads)

6.1. Build settings (recommended)

Use symbols + frame pointers for better profiling:

```
gcc -O2 -g -fno-omit-frame-pointer -pthread -std=c11 file.c -o prog
```

6.2. A0: Very simple programs (3 variants)

This section provides minimal pthread codes for three cases:

1. True sharing (same variable)
2. False sharing (different variables, same cache line)
3. Fixed false sharing (padding, different cache lines)

6.2.1. Variant 1 (Very Simple): True sharing (same shared variable)

Idea: Both threads increment the **same counter**. This forces the cache line containing that counter to bounce between cores.

Listing 1: Very simple true sharing: both threads update the same counter

```c
#include <pthread.h>
#include <stdio.h>
#include <stdatomic.h>
```

```

#ifndef ITERS
#define ITERS 200000000UL
#endif

static atomic_ulong shared_counter = 0;

void* worker(void* arg) {
  (void)arg;
  for (unsigned long i = 0; i < ITERS; i++) {
    atomic_fetch_add_explicit(&shared_counter, 1, memory_order_relaxed)
        ;
  }
  return NULL;
}

int main(void) {
  pthread_t t0, t1;

  pthread_create(&t0, NULL, worker, NULL);
  pthread_create(&t1, NULL, worker, NULL);

  pthread_join(t0, NULL);
  pthread_join(t1, NULL);

  printf("shared_counter = %lu (expected %lu)\n",
         (unsigned long)atomic_load(&shared_counter),
         2UL * ITERS);
  return 0;
}
```

6.2.2. Variant 2 (Very Simple): False sharing (adjacent counters)

Idea: Each thread updates its own counter, but the counters are adjacent in memory and can share one cache line.

Listing 2: Very simple false sharing: two different counters but adjacent in memory

```
#include <pthread.h>
#include <stdio.h>
#include <stdatomic.h>

#ifndef ITERS
#define ITERS 200000000UL
#endif

typedef struct {
  atomic_ulong a;   // thread 0 updates
  atomic_ulong b;   // thread 1 updates
} counters_t;

static counters_t c = {0, 0};

void* t0(void* arg) {
  (void)arg;
  for (unsigned long i = 0; i < ITERS; i++) {
    atomic_fetch_add_explicit(&c.a, 1, memory_order_relaxed);
  }
```

```c
  return NULL;
}

void* t1(void* arg) {
  (void)arg;
  for (unsigned long i = 0; i < ITERS; i++) {
    atomic_fetch_add_explicit(&c.b, 1, memory_order_relaxed);
  }
  return NULL;
}

int main(void) {
  pthread_t p0, p1;

  pthread_create(&p0, NULL, t0, NULL);
  pthread_create(&p1, NULL, t1, NULL);

  pthread_join(p0, NULL);
  pthread_join(p1, NULL);

  printf("a = %lu, b = %lu\n",
         (unsigned long)atomic_load(&c.a),
         (unsigned long)atomic_load(&c.b));
  return 0;
}
```

6.2.3. Variant 3 (Very Simple): Fixed false sharing (padding)

Idea: Insert padding so `a` and `b` are on different cache lines (typical 64 bytes).

Listing 3: Very simple fixed false sharing: padding separates cache lines

```c
#include <pthread.h>
#include <stdio.h>
#include <stdatomic.h>

#ifndef ITERS
#define ITERS 200000000UL
#endif

typedef struct {
  atomic_ulong a;
  char pad[64 - sizeof(atomic_ulong)];
  atomic_ulong b;
} counters_t;

static counters_t c = {0, {0}, 0};

void* t0(void* arg) {
  (void)arg;
  for (unsigned long i = 0; i < ITERS; i++) {
    atomic_fetch_add_explicit(&c.a, 1, memory_order_relaxed);
  }
  return NULL;
}

void* t1(void* arg) {
  (void)arg;
  for (unsigned long i = 0; i < ITERS; i++) {
    atomic_fetch_add_explicit(&c.b, 1, memory_order_relaxed);
  }
  return NULL;
}

int main(void) {
  pthread_t p0, p1;

  pthread_create(&p0, NULL, t0, NULL);
  pthread_create(&p1, NULL, t1, NULL);

  pthread_join(p0, NULL);
  pthread_join(p1, NULL);

  printf("a = %lu, b = %lu\n",
         (unsigned long)atomic_load(&c.a),
         (unsigned long)atomic_load(&c.b));
  return 0;
}
```

6.3. A1: True-sharing ping-pong (strict alternation)

What it demonstrates: Two threads increment the same shared counter, alternating turns. This strongly forces cache-line ownership transfers.

Listing 4: True-sharing ping-pong (turn-based alternation)

```c
#define _GNU_SOURCE
#include <pthread.h>
#include <stdio.h>
#include <stdint.h>
#include <time.h>
#include <stdatomic.h>

#ifndef ITERS
#define ITERS 50000000UL
#endif

static atomic_int turn = 0;       // 0 -> thread0, 1 -> thread1
static atomic_uint shared = 0;    // same variable updated by both
    threads

static inline uint64_t ns_now(void) {
  struct timespec ts;
  clock_gettime(CLOCK_MONOTONIC, &ts);
  return (uint64_t)ts.tv_sec*1000000000ULL + (uint64_t)ts.tv_nsec;
}

void* t0(void* arg) {
  (void)arg;
  for (unsigned long i=0; i<ITERS; i++) {
    while (atomic_load_explicit(&turn, memory_order_acquire) != 0) { }
    atomic_fetch_add_explicit(&shared, 1u, memory_order_relaxed);
    atomic_store_explicit(&turn, 1, memory_order_release);
  }
  return NULL;
}
```

```

void* t1(void* arg) {
  (void)arg;
  for (unsigned long i=0; i<ITERS; i++) {
    while (atomic_load_explicit(&turn, memory_order_acquire) != 1) { }
    atomic_fetch_add_explicit(&shared, 1u, memory_order_relaxed);
    atomic_store_explicit(&turn, 0, memory_order_release);
  }
  return NULL;
}

int main(void) {
  pthread_t a,b;
  uint64_t st = ns_now();
  pthread_create(&a,NULL,t0,NULL);
  pthread_create(&b,NULL,t1,NULL);
  pthread_join(a,NULL);
  pthread_join(b,NULL);
  uint64_t en = ns_now();
  printf("shared=%u (expected %lu)\n", atomic_load(&shared), 2UL*ITERS)
      ;
  printf("time=%.3f s\n", (en-st)/1e9);
  return 0;
}
```

6.4. A2: False sharing ping-pong (unpadded)

What it demonstrates: Each thread updates its own counter, but both counters are adjacent and likely share one cache line.

Listing 5: False sharing (unpadded) – likely same cache line

```
#define _GNU_SOURCE
#include <pthread.h>
#include <stdio.h>
#include <stdint.h>
#include <time.h>
#include <stdatomic.h>

#ifndef ITERS
#define ITERS 50000000UL
#endif

typedef struct {
  atomic_uint x;  // thread0 updates
  atomic_uint y;  // thread1 updates
} counters_t;

static counters_t c = {0,0};

static inline uint64_t ns_now(void) {
  struct timespec ts;
  clock_gettime(CLOCK_MONOTONIC, &ts);
  return (uint64_t)ts.tv_sec*1000000000ULL + (uint64_t)ts.tv_nsec;
}

void* t0(void* arg){
  (void)arg;
```

```c
  for(unsigned long i=0;i<ITERS;i++)
    atomic_fetch_add_explicit(&c.x, 1u, memory_order_relaxed);
  return NULL;
}

void* t1(void* arg){
  (void)arg;
  for(unsigned long i=0;i<ITERS;i++)
    atomic_fetch_add_explicit(&c.y, 1u, memory_order_relaxed);
  return NULL;
}

int main(void){
  pthread_t a,b;
  uint64_t st = ns_now();
  pthread_create(&a,NULL,t0,NULL);
  pthread_create(&b,NULL,t1,NULL);
  pthread_join(a,NULL);
  pthread_join(b,NULL);
  uint64_t en = ns_now();
  printf("x=%u y=%u\n", atomic_load(&c.x), atomic_load(&c.y));
  printf("time=%.3f s\n", (en-st)/1e9);
  return 0;
}
```

6.5. A3: False sharing fix (padding/alignment)

Goal: Place y on a different cache line from x. (Typical cache line size is 64 bytes.)

Listing 6: False sharing fixed via padding (typical 64-byte line)

```c
typedef struct {
  atomic_uint x;
  char pad[64 - sizeof(atomic_uint)];
  atomic_uint y;
} counters_t;

static counters_t c = {0, {0}, 0};
```

6.6. A4: Pin threads to different cores (CPU affinity)

Why: Pinning reduces scheduler noise and makes ping-pong more repeatable.

Listing 7: False sharing + CPU affinity (pinning)

```c
#define _GNU_SOURCE
#include <pthread.h>
#include <stdio.h>
#include <stdint.h>
#include <time.h>
#include <stdatomic.h>
#include <sched.h>
#include <unistd.h>
#include <string.h>

#ifndef ITERS
#define ITERS 50000000UL
#endif
```

```c

typedef struct { atomic_uint x; atomic_uint y; } counters_t;
static counters_t c = {0,0};

static inline uint64_t ns_now(void){
  struct timespec ts; clock_gettime(CLOCK_MONOTONIC, &ts);
  return (uint64_t)ts.tv_sec*1000000000ULL + (uint64_t)ts.tv_nsec;
}

static void pin_to_cpu(int cpu_id){
  cpu_set_t set; CPU_ZERO(&set); CPU_SET(cpu_id, &set);
  int rc = pthread_setaffinity_np(pthread_self(), sizeof(set), &set);
  if(rc!=0) fprintf(stderr,"Warning: affinity cpu=%d failed: %s\n",
                    cpu_id, strerror(rc));
}

typedef struct { int cpu_id; int which; } arg_t;

void* worker(void* arg){
  arg_t* a=(arg_t*)arg;
  pin_to_cpu(a->cpu_id);
  if(a->which==0){
    for(unsigned long i=0;i<ITERS;i++)
      atomic_fetch_add_explicit(&c.x,1u,memory_order_relaxed);
  }else{
    for(unsigned long i=0;i<ITERS;i++)
      atomic_fetch_add_explicit(&c.y,1u,memory_order_relaxed);
  }
  return NULL;
}

int main(void){
  int ncpu=(int)sysconf(_SC_NPROCESSORS_ONLN);
  if(ncpu<2){ fprintf(stderr,"Need >= 2 CPUs.\n"); return 1; }

  pthread_t t0,t1;
  arg_t a0={.cpu_id=0,.which=0};
  arg_t a1={.cpu_id=1,.which=1};

  uint64_t st=ns_now();
  pthread_create(&t0,NULL,worker,&a0);
  pthread_create(&t1,NULL,worker,&a1);
  pthread_join(t0,NULL);
  pthread_join(t1,NULL);
  uint64_t en=ns_now();

  printf("Pinned CPU0/CPU1: x=%u y=%u time=%.3f s\n",
         atomic_load(&c.x), atomic_load(&c.y), (en-st)/1e9);
  return 0;
}
```

7. Lab Part B: Profiling and Evaluation with Linux perf

7.1. B1: Install perf tools

Ubuntu/Debian:

```
sudo apt update
sudo apt install -y linux-tools-common linux-tools-generic
# On some systems, you may need the exact kernel tools package:
# sudo apt install -y linux-tools-$(uname -r)
```

7.2. B2: If perf needs permissions

For a lab machine (temporary change):

```
sudo sysctl -w kernel.perf_event_paranoid=1
sudo sysctl -w kernel.kptr_restrict=0
```

Note: These are security-sensitive settings; revert after the lab if needed.

7.3. B3: Baseline timing comparison

Run each variant 3 times and record average runtime:

```
./pp_unpadded
./pp_unpadded
./pp_unpadded

./pp_padded
./pp_padded
./pp_padded
```

7.4. B4: Quick performance counters with perf stat

Goal: Quantify overhead differences between unpadded vs padded.

```
sudo perf stat ./pp_unpadded
sudo perf stat ./pp_padded
```

Useful metrics to watch

Interpretation depends on CPU, but typically track:

- **cycles, instructions, IPC** (instructions per cycle)
- **cache-references, cache-misses** (overall caching behavior)
- **context-switches, cpu-migrations** (noise; should reduce with pinning)
- **task-clock** (runtime)

Expected observation: The unpadded false-sharing version often shows worse runtime and poorer efficiency due to cache-line bouncing.

7.5. B5: Intel-focused deep dive: perf c2c (Cache-to-Cache)

On many Intel systems, `perf c2c` can highlight cache-line contention (ownership transfers, HITM-like behavior).

```
sudo perf c2c record -- ./pp_unpadded
sudo perf c2c report
```

Repeat for padded:

```
sudo perf c2c record -- ./pp_padded
sudo perf c2c report
```

What to look for in the report:

- A small number of cache lines dominating "cache-to-cache" transfers.
- Reduced contention hotspots after padding.
- If symbols are available, it may point to the data object or code path.

7.6. B6: AMD variant notes (perf support varies)

On AMD, deep cache-to-cache analysis may depend on IBS support and kernel/CPU generation. If `perf c2c` is limited, use:

- `perf stat` for quantitative comparison (still very effective for teaching).
- `perf record` + `perf report` to understand where CPU time goes (less direct for false sharing).
- Optional (outside this handout): AMD uProf's cache analysis features can be used for false-sharing detection in a GUI/CLI workflow.

7.7. B7: Reduce measurement noise (recommended)

- Use thread pinning (affinity) and keep the system idle.
- Close heavy background apps.
- Repeat runs and take median/average.
- Ensure the program is compiled with `-O2` for realistic behavior.

8. Evaluation: How to Decide "Padding Helped"

8.1. Required evidence

Students should provide:

1. Runtime comparison (unpadded vs padded), with at least 3 runs each.
2. `perf stat` summary for both variants.
3. (Intel systems) `perf c2c report` screenshots or captured output showing reduced hot cache-line contention.

8.2. Suggested table format

Variant	Time (s)	cycles	cache-misses	cpu-migrations
Unpadded (false sharing)				
Padded (fixed)				

9. Lab Tasks (What students must do)

1. Compile and run the **false sharing unpadded** program; record time.
2. Modify it to **padded** version; record time.
3. Enable **CPU affinity**; compare stability and runtime.
4. Run `perf stat` on both; capture metrics.
5. (Intel) Run `perf c2c record/report` on both; identify the contended cache line(s).
6. Write a short conclusion: *Why does padding help? Why is coherence per cache line?*

10. Common Pitfalls (and fixes)

- **No speedup seen:** Possibly both threads ran on the same core. Use affinity.

- **High variability:** Background processes or CPU frequency scaling. Repeat runs; pin threads; keep system idle.
- **perf permission errors:** Adjust `perf_event_paranoid` temporarily (lab admin policy permitting).
- **Confusing conclusion:** Emphasize: *false sharing occurs even if variables are different, because they share a cache line.*

11. Mini-Quiz (in-class check)

1. Why can two independent counters still slow each other down on an SMP?
2. In write-invalidate, what happens to other cached copies when a core writes?
3. What changes when we add padding between two counters?

End of Handout