Parallel Computing

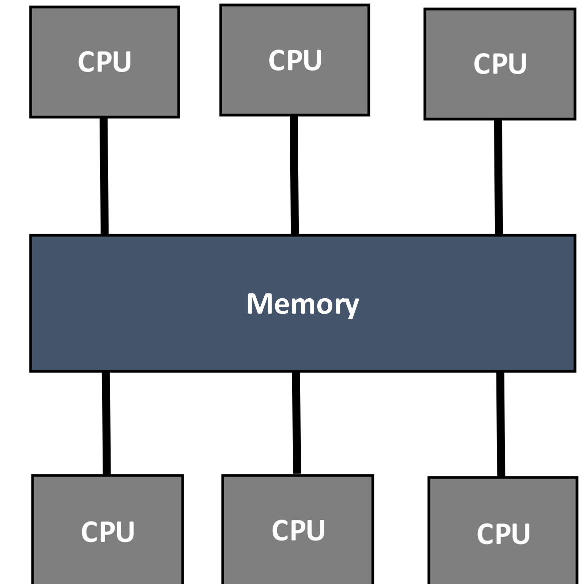
IT00CD91-3005, Spring 2025

Lecture 4: Shared Memory Programming II

Shared Memory Programming

Shared memory programming with

- `pthreads`
- `std::threads`
- `OpenMP`



OpenMP popular directives

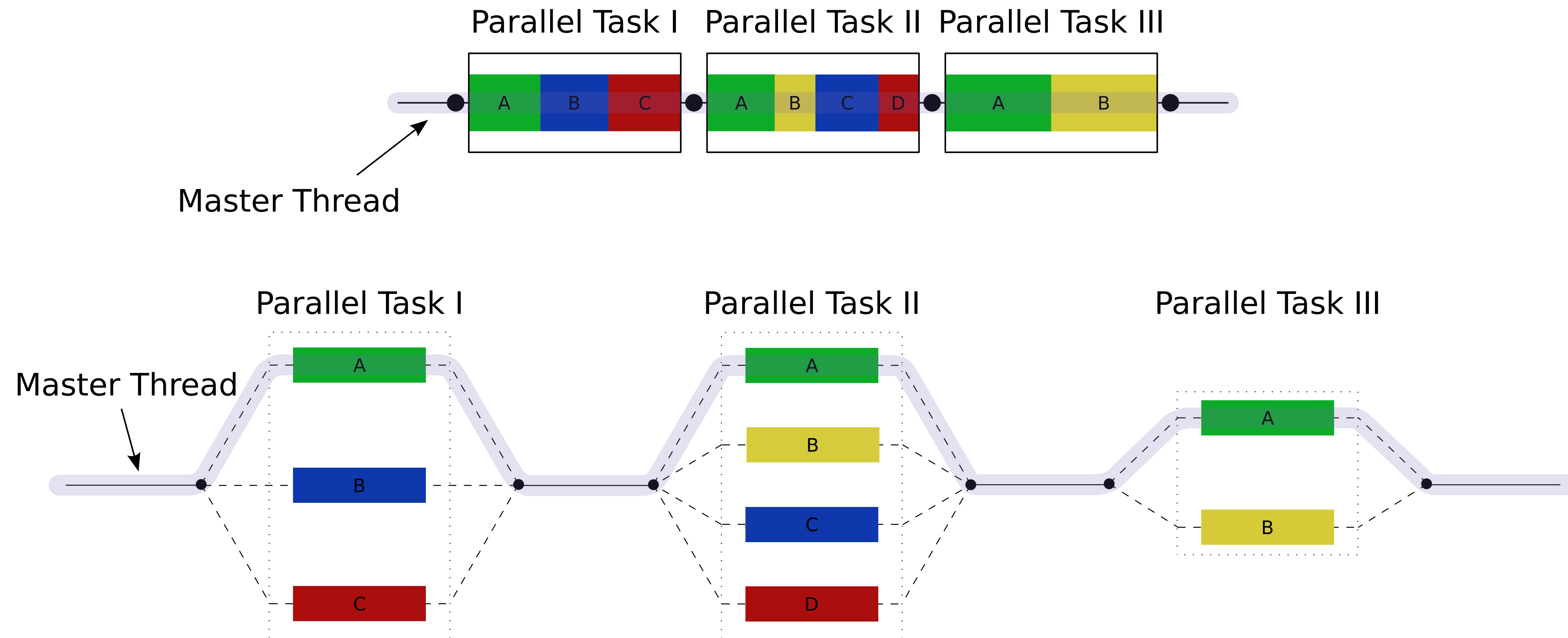
OpenMP pragma, function, or clause	Concepts
#pragma omp parallel	Creates a parallel region where a team of threads is spawned.
int omp_get_thread_num() int omp_get_num_threads()	Returns the ID of the current thread Returns the total number of threads in the current parallel region
double omp_get_wtime()	Returns wall-clock time, used to measure performance
setenv OMP_NUM_THREADS N	Internal control variables. Setting the default number of threads with an environment variable
#pragma omp barrier #pragma omp critical	All threads wait at this point before continuing Only one thread at a time can enter this block
#pragma omp for #pragma omp parallel for	Used inside a parallel region to distribute loop iteration Shortcut: creates parallel region and distribute loop in one line.
reduction(op:list)	Reductions of values across a team of threads
schedule(dynamic [,chunk]) schedule (static [,chunk])	Threads grab chunks as the finish Divides loop iterations evenly among threads
private(list) firstprivate(list) shared(list)	Each thread gets its own uninitialized copy of variables in the list Each thread gets its own copy initialized with the value from the master thread Variables in the list are shared across all threads
nowait	Removes the implicit barrier at the end of a for, section, or single block
#pragma omp single	Code inside runs only by one thread, others skip it but wait
#pragma omp task #pragma omp taskwait	Creates a deferred unit of work (task) to be run by any thread Waits until all previously created tasks in this thread have completed.

OpenMP programming model

Fork– Join Parallelism

Mater (main) thread spawns a team of threads as needed.

Parallelism added incrementally until the performance goals are met



Example: dot product

- Given two vectors a and b, the dot product is defined as

$$x = \sum_{i=0}^{N-1} a[i].\,b[i]$$

- How to write a program to compute x serially?

Example: dot product

- Given two vectors a and b, the dot product is defined as

$$x = \sum_{i=0}^{N-1} a[i].\, b[i]$$

- How to write a program to compute x serially?

```
for (int i = 0; i < N; ++i) {
    dot += A[i] * B[i];
}
```

Example: dot product

- Given two vectors a and b, the dot product is defined as

$$x = \sum_{i=0}^{N-1} a[i].b[i]$$

- How to write a program to compute x serially?

```
for (int i = 0; i < N; ++i) {
    dot += A[i] * B[i];
}
```

Ideas for parallelizing dot product program with multi-threaded programming?

Example: dot product - pthreads

```
std::vector<float> A(N, 1.0f);
std::vector<float> B(N, 2.0f);
float partial_sum[NUM_THREADS];
```

```
void* dot_product_worker(void* arg) {
    int tid = *(int*)arg;
    int chunk_size = N / NUM_THREADS;
    int start = tid * chunk_size;
    int end = (tid == NUM_THREADS - 1) ? N : start + chunk_size;

    float sum = 0.0f;
    for (int i = start; i < end; ++i) {
        sum += A[i] * B[i];
    }
    partial_sum[tid] = sum;
    return nullptr;
}
```

Example: dot product - `pthreads`

```cpp
std::vector<float> A(N, 1.0f);
std::vector<float> B(N, 2.0f);
float partial_sum[NUM_THREADS];
```

```cpp
void* dot_product_worker(void* arg) {
    int tid = *(int*)arg;
    int chunk_size = N / NUM_THREADS;
    int start = tid * chunk_size;
    int end = (tid == NUM_THREADS - 1) ? N : start + chunk_size;

    float sum = 0.0f;
    for (int i = start; i < end; ++i) {
        sum += A[i] * B[i];
    }
    partial_sum[tid] = sum;
    return nullptr;
}
```

```cpp
pthread_t threads[NUM_THREADS];
int ids[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; ++i) {
    ids[i] = i;
    pthread_create(&threads[i], nullptr, dot_product_worker, &ids[i]);
}

for (int i = 0; i < NUM_THREADS; ++i) {
    pthread_join(threads[i], nullptr);
}

float dot = 0.0f;
for (int i = 0; i < NUM_THREADS; ++i) {
    dot += partial_sum[i];
}
```

Example: dot product – `std::threads`

```cpp
std::vector<float> A(N, 1.0f);
std::vector<float> B(N, 2.0f);
float partial_sum[NUM_THREADS];
```

```cpp
void dot_product_worker(int tid) {
    int chunk_size = N / NUM_THREADS;
    int start = tid * chunk_size;
    int end = (tid == NUM_THREADS - 1) ? N : start + chunk_size;

    float sum = 0.0f;
    for (int i = start; i < end; ++i) {
        sum += A[i] * B[i];
    }
    partial_sum[tid] = sum;
}
```

```cpp
std::thread threads[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; ++i) {
    threads[i] = std::thread(dot_product_worker, i);
}

for (int i = 0; i < NUM_THREADS; ++i) {
    threads[i].join();
}

float dot = 0.0f;
for (int i = 0; i < NUM_THREADS; ++i) {
    dot += partial_sum[i];
}
```

Numerical Integration to compute Pi

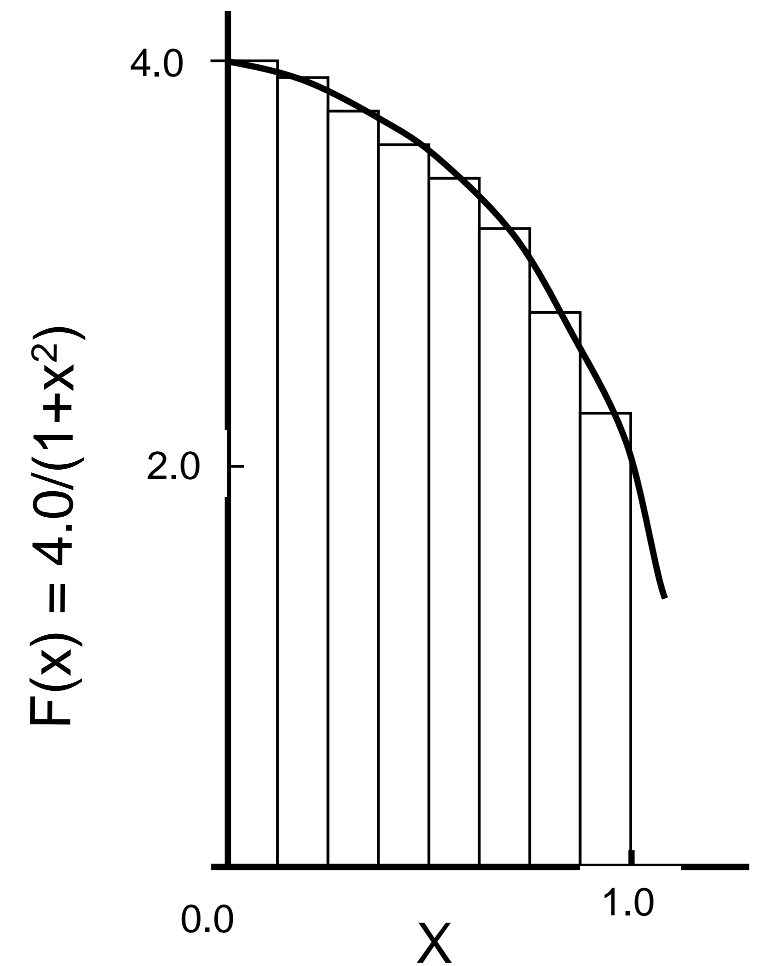
Mathematically we know that

$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi$$

We can approximate the integral as a sum of rectangles.

$$\sum_0^N f(x_i)\Delta x = \pi$$

Where each rectangle has width Δx and height $f(x_i)$ at the middle of interval i



Serial Program

Serial Program

```
int main() {
    long num_steps = 1000000;
    double step = 1.0 / num_steps;
    double sum = 0.0;

    for (long i = 0; i < num_steps; ++i) {
        double x = step * (i + 0.5);
        sum += 4.0 / (1.0 + x * x);
    }

    double pi = sum * step;
    return 0;
}
```

$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi$$

$$\sum_0^N f(x_i)\Delta x = \pi$$

Serial Pi Program – wall clock time

```
#include <cstdio>
#include <omp.h>

int main() {
    long num_steps = 1000000;
    double step = 1.0 / num_steps;
    double sum = 0.0;

    double start_time = omp_get_wtime();

    for (long i = 0; i < num_steps; ++i) {
        double x = step * (i + 0.5);
        sum += 4.0 / (1.0 + x * x);
    }

    double pi = sum * step;

    double end_time = omp_get_wtime();
    printf("Computed Pi: %.15f\n", pi);
    printf("Wall clock time = %.6f seconds\n", end_time - start_time);

    return 0;
}
```

The library routine `get_omp_wtime()` is used to find the elapsed "wall clock time" for blocks of code

Wall Clock Time (WCT):

The elapsed time between the start and end of a block, including all delays and waiting (e.g. I/O)

CPU Time:

The total time the CPU actively spends processing the program excluding idle time and I/O waits

Which one we are more interested in?

Shared Memory Systems and Memory Inconsistency

Latency vs Bandwidth

Memory accesses (load / store) have two costs

Latency - The time it takes from when data is requested or sent by the source until the destination begins to receive the first byte.

Bandwidth – the rate (bytes/sec) at which the destination receives data after it has started to receive the first byte.

Let l to be the latency and b to be the bandwidth (byte/second), then the time it takes to transmit n bytes is

$$\tau = l + \frac{n}{b}$$

How to improve memory latency?

One solution is using cache memories.

A cache is a collection of memory locations that a CPU can access more quickly than it can access main memory.

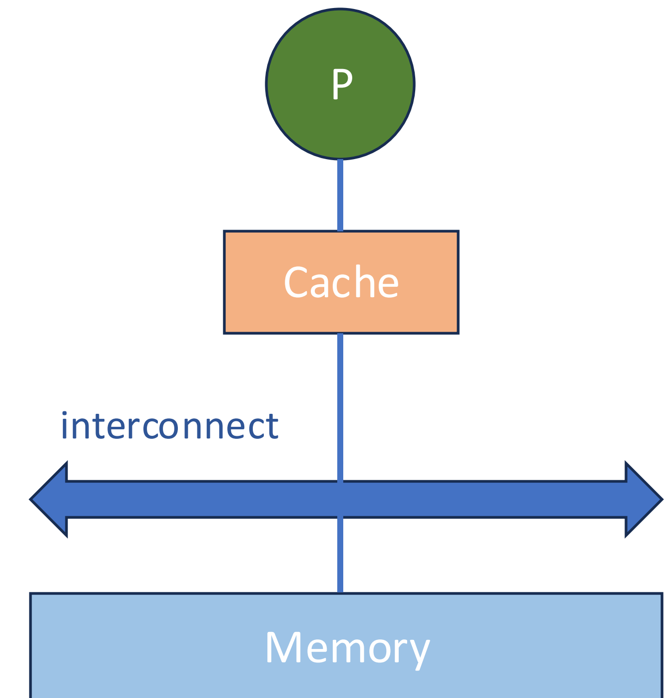
Faster but smaller than the main memory.

Improves memory latency.

Caches are managed by hardware (i.e. hidden from software)

Cache hit - when the requested data is already in the cache.

Cache miss – when the requested data is not in the cache, requiring retrieval from the main memory



Cache Memory

Which data should be stored in the cache?

Arrays are allocated as blocks of contiguous memory locations.

The read of `z[i]` is immediately followed by a read of z[i+1].

Locality - the principle that programs tend to access data and instructions in predictable patterns.

- **Spatial locality – accessing things nearby previous accesses.**
- **Temporal locality – reusing an item that was previously accessed.**

To exploit locality, the memory access will operate on blocks of data (64 bytes) instead of individual data items. These blocks are called cache lines.

```
float z[1000];

sum = 0.0;
for (i = 0; i < 1000; i++)
    sum += z[i];
```

When `z[0]` is read the system might read `16 floats` (64 bytes) – `z[0],…,z[15]` from memory into cache.

Cache Inconsistency

Writing to cache makes the cache value different from the main memory.

How to handle:

- **Write through - write to cache and main memory at the same time**
- **Write back – mark the cache line as dirty, write to memory only when the line is replaced.**

Take aways

Caches are fast memory locations to reduce CPU – Memory bottleneck.

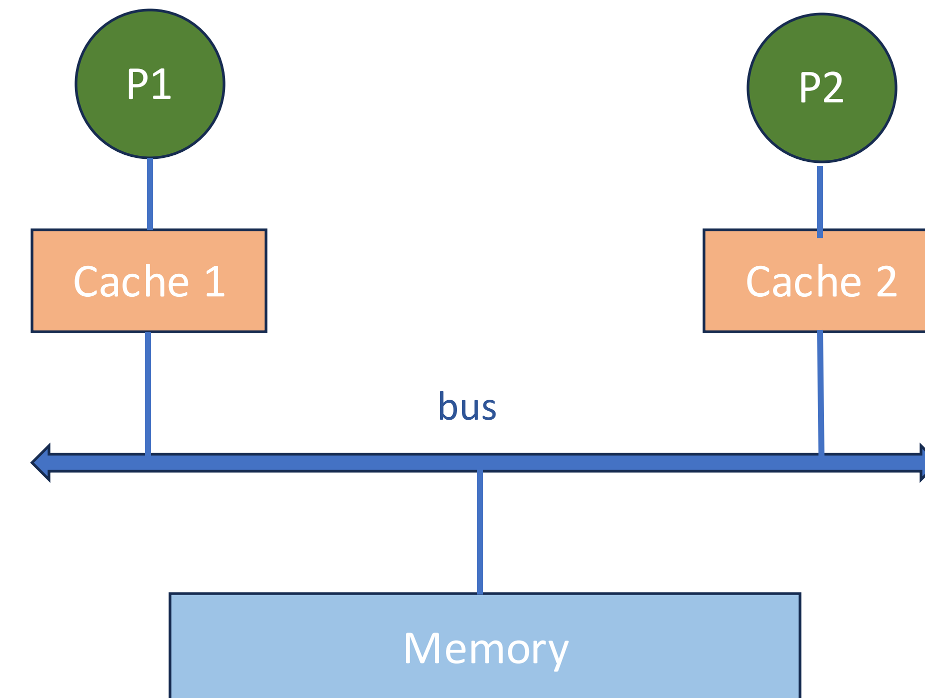
Data is stored in cache using the principle of temporal and spatial locality.

The data between memory and caches are transferred in cache lines.

In High – Performance Computing (HPC), it is critical to write cache aware programs (Assignment 2)

Caching in Shared Memory Systems

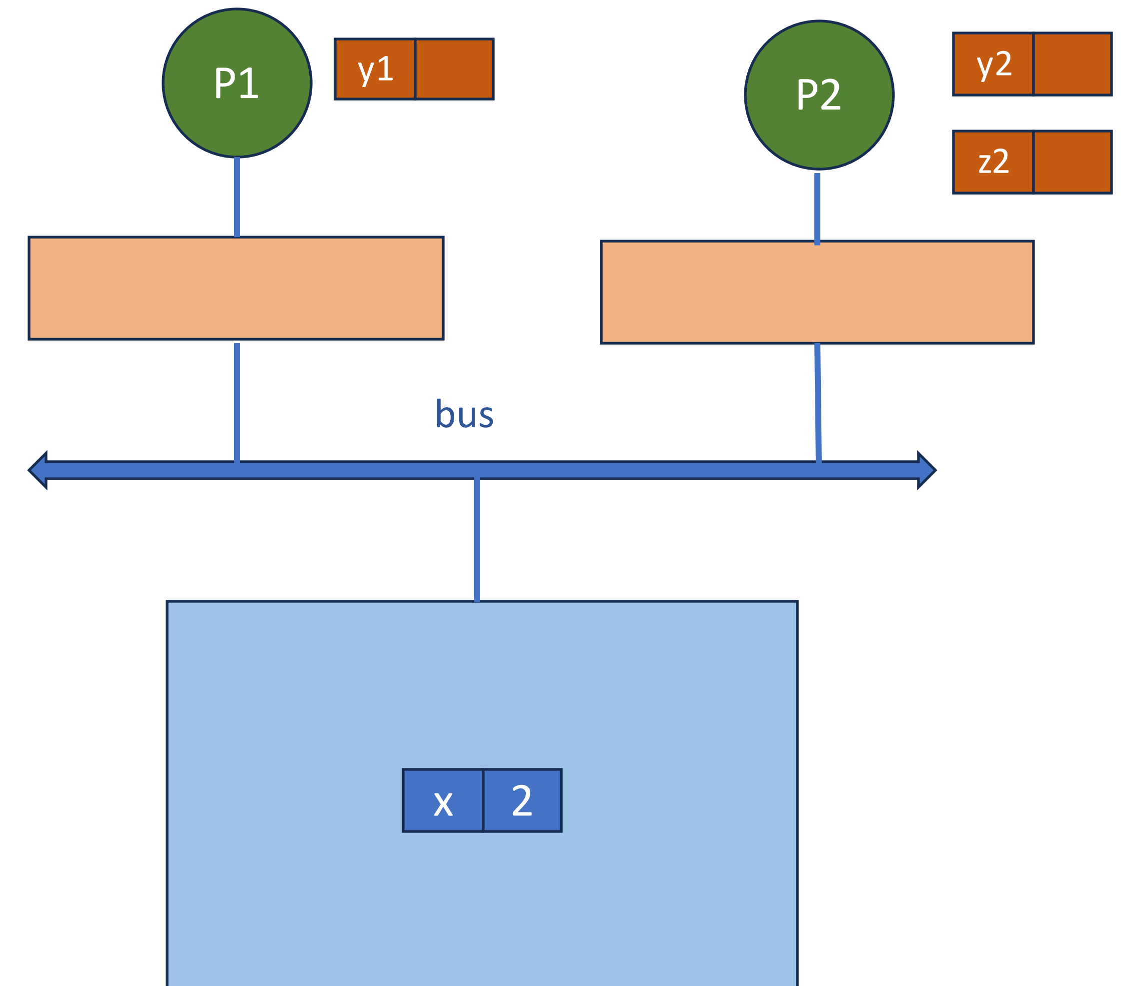
Similar to single core systems, to improve shared memory system's performance, different level of caches are used.



In shared memory systems:

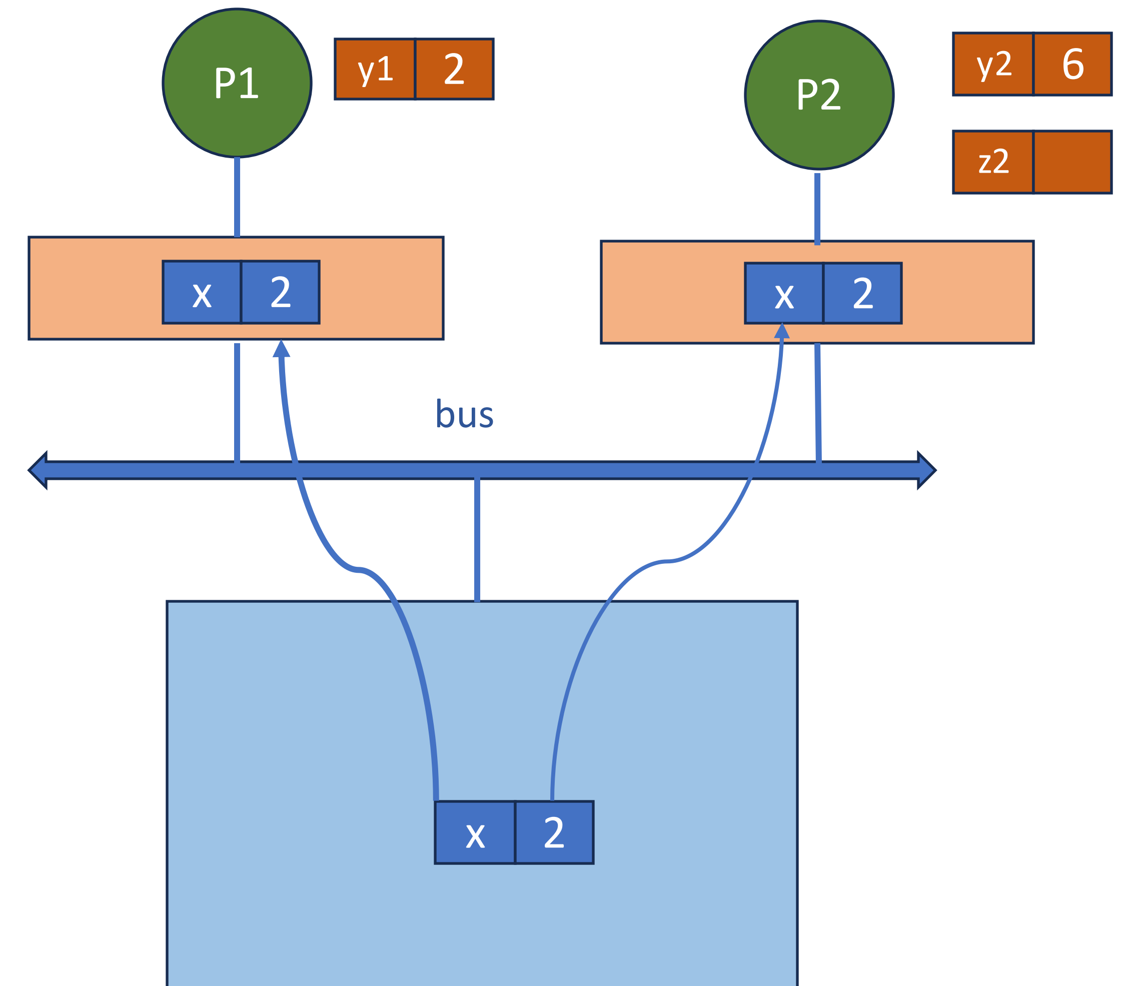
- **Each processor has its own cache (or multiple caches)**
- **Place data from memory into cache**
- **Writeback cache: don't send all writes over bus to memory**
- **Longer latencies.**
- **Problem - Cache Coherence!**

Cache Coherence



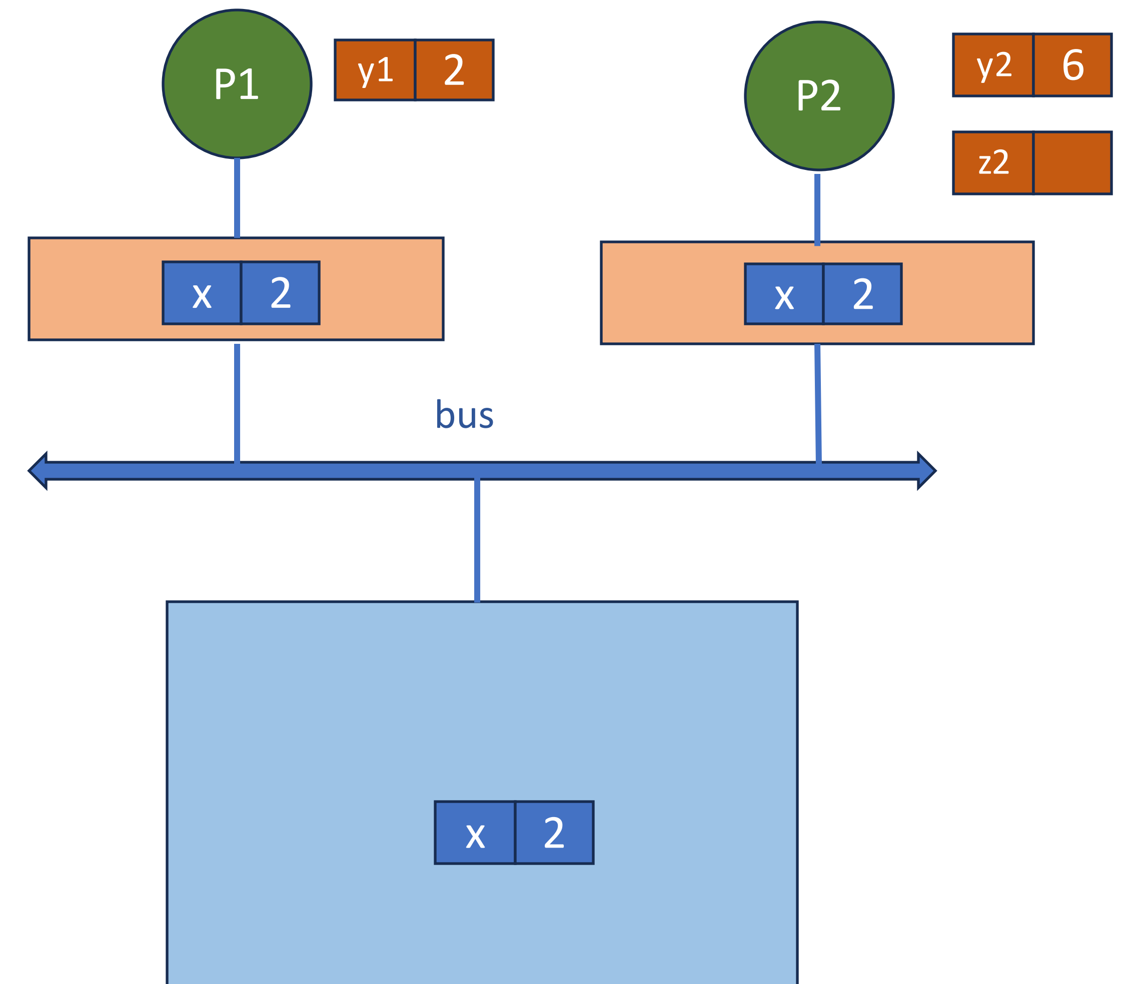
Cache Coherence

Time	Core 1	Core 2
0	`y1 = x`	`y2 = 3 * x`



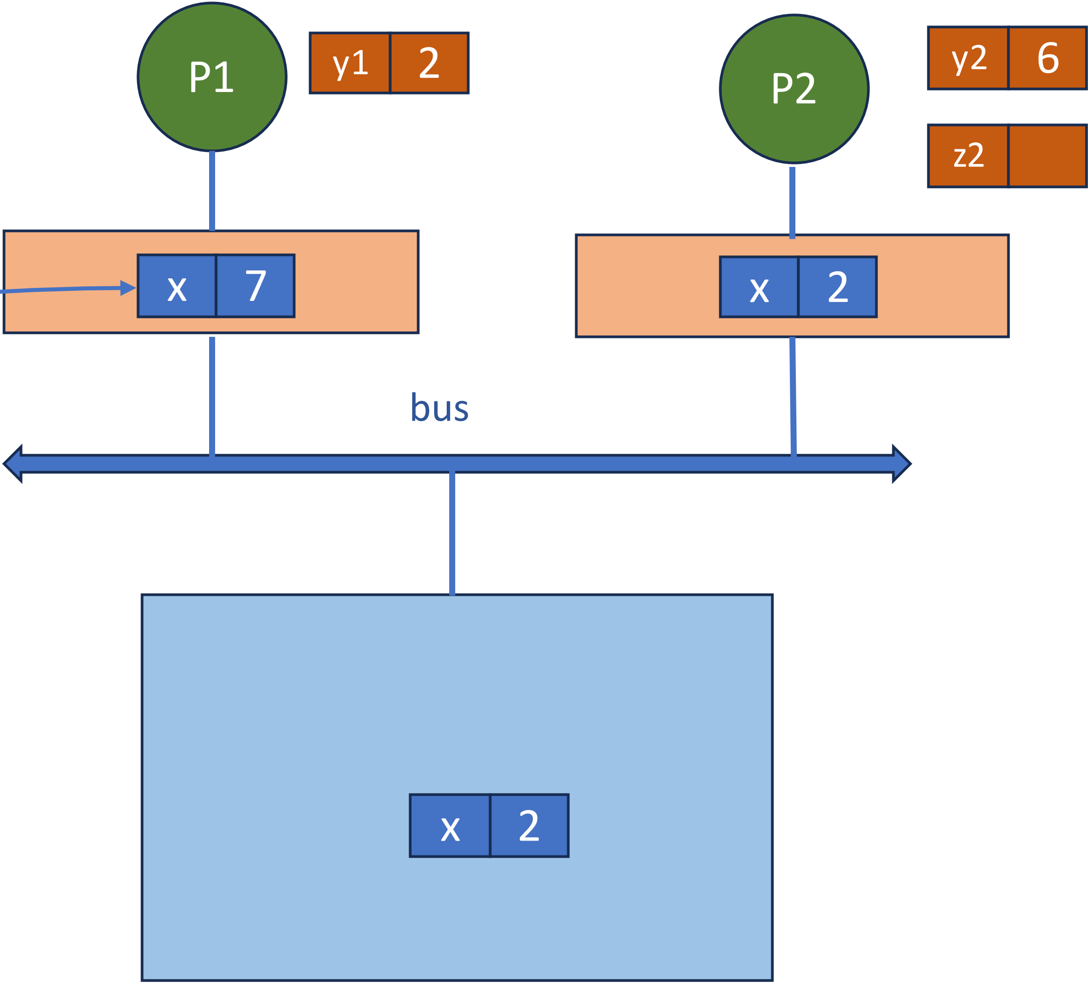
Cache Coherence

Time	P1	P2
0	`y1 = x`	`y2 = 3 * x`
1	`x = 7`	`--`



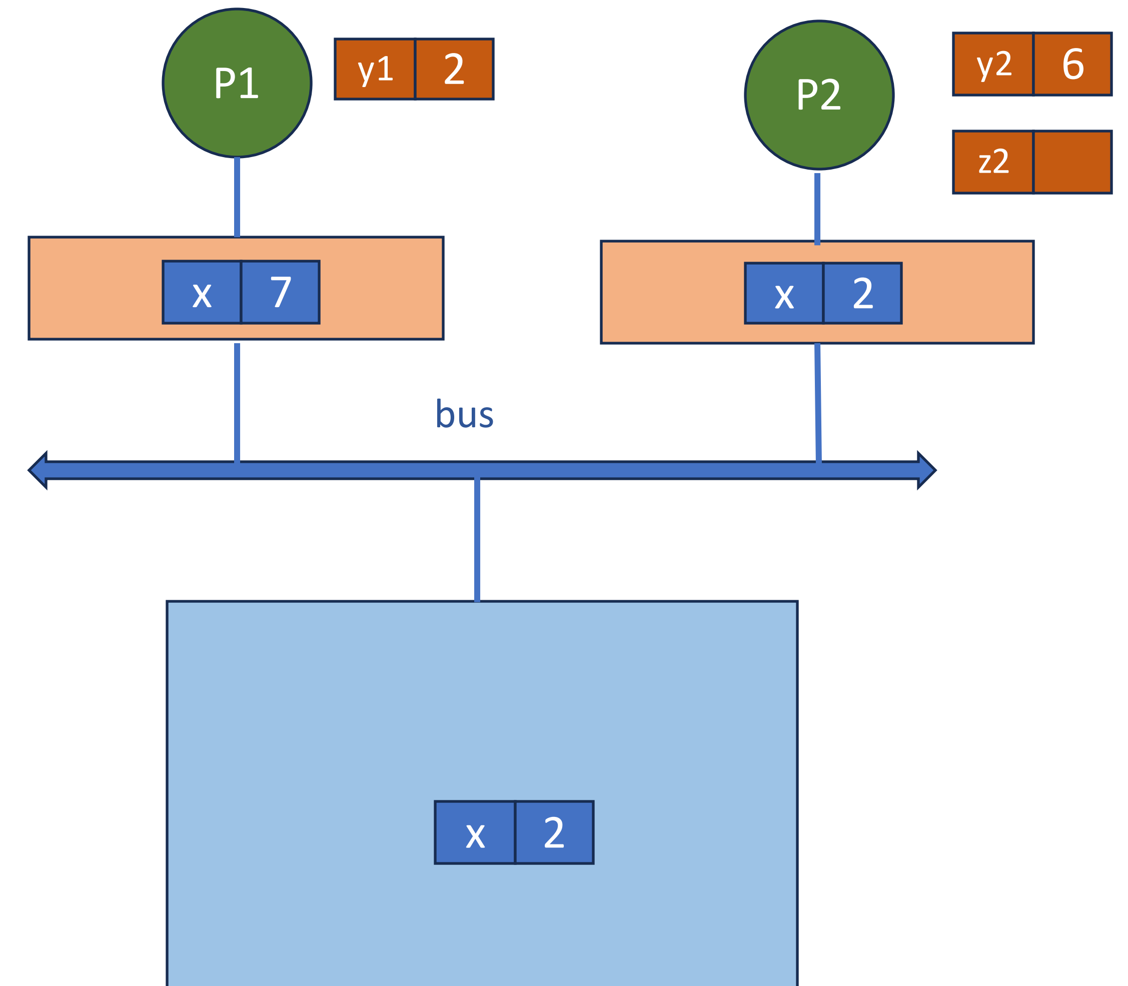
Cache Coherence

Time	P1	P2
0	y1 = x	y2 = 3 * x
1	x = 7	--



Cache Coherence

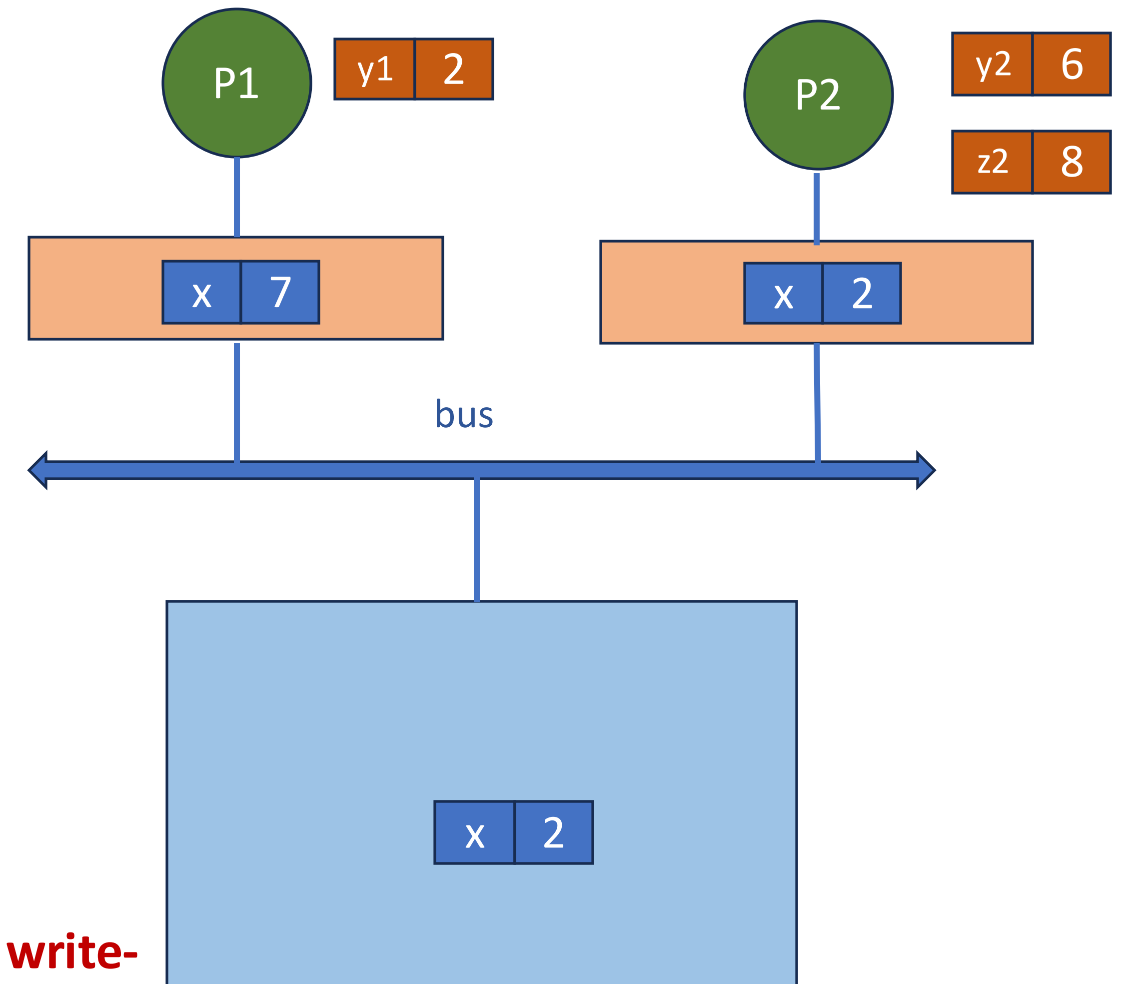
Time	Core 1	Core 2
0	y1 = x	y2 = 3 * x
1	x = 7	--
2	--	z2 = 4 * x



Cache Coherence

Time	Core 1	Core 2
0	y1 = x	y2 = 3 * x
1	x = 7	--
2	--	z2 = 4 * x

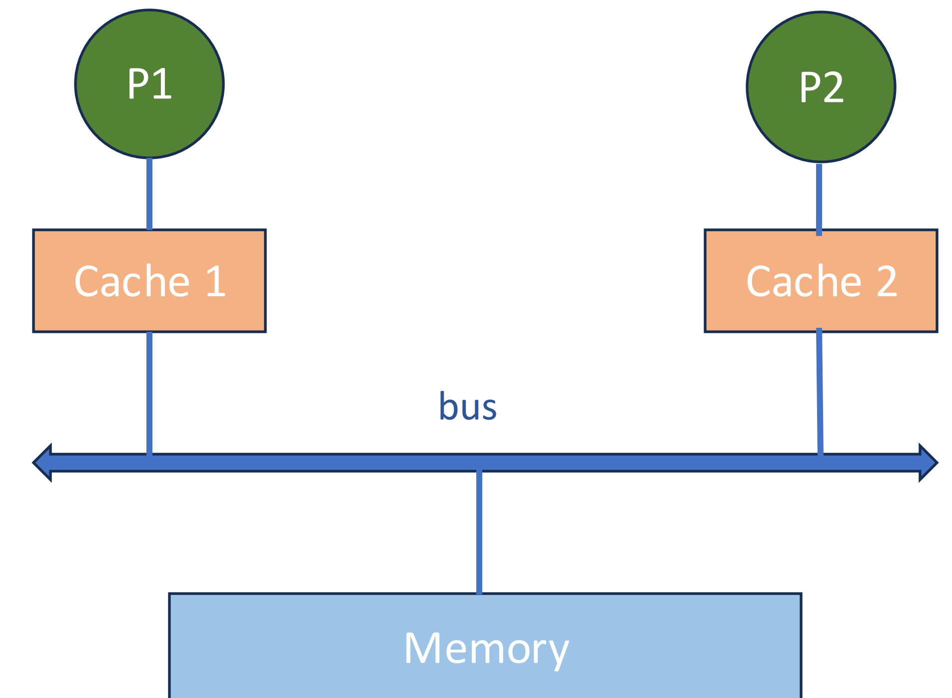
not 28, why?



This unpredictable behavior will occur regardless of using write-through or write-back policy. Why?

Cache Coherence

Cache Coherence Problem – in multi-core systems, inconsistencies arise when multiple caches hold copies of the same shared memory location, and one processor updates its cache without synchronizing the others.

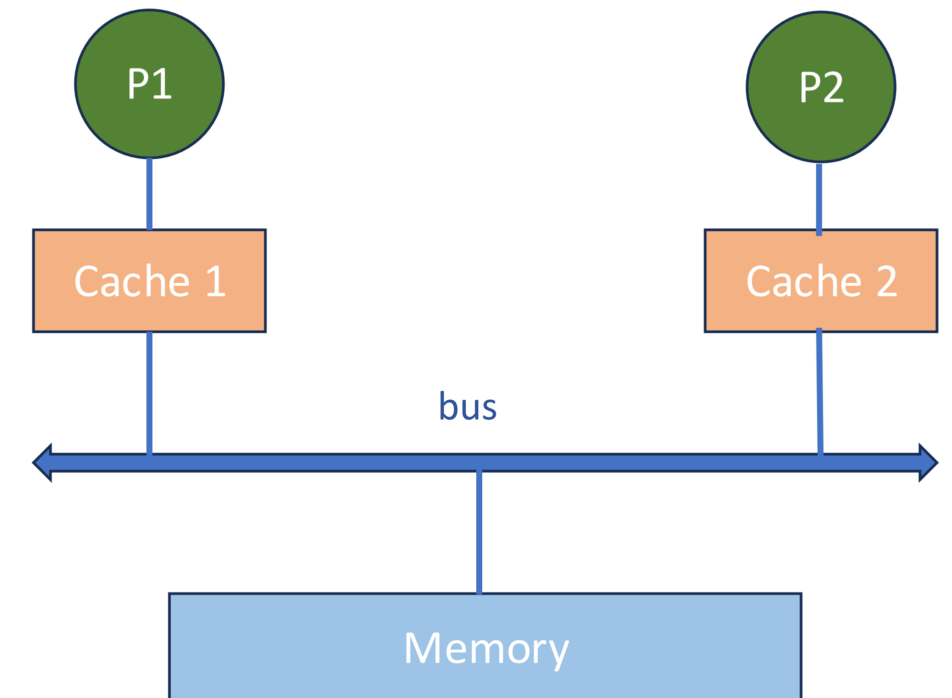


How to ensure cache coherence?

- **Snooping cache coherence**
- **Directory-based cache coherence**

Cache Coherence

Cache Coherence Problem – in multi-core systems, inconsistencies arise when multiple caches hold copies of the same shared memory location, and one processor updates its cache without synchronizing the others.



How to ensure cache coherence?

- **Snooping cache coherence**
- Directory-based cache coherence

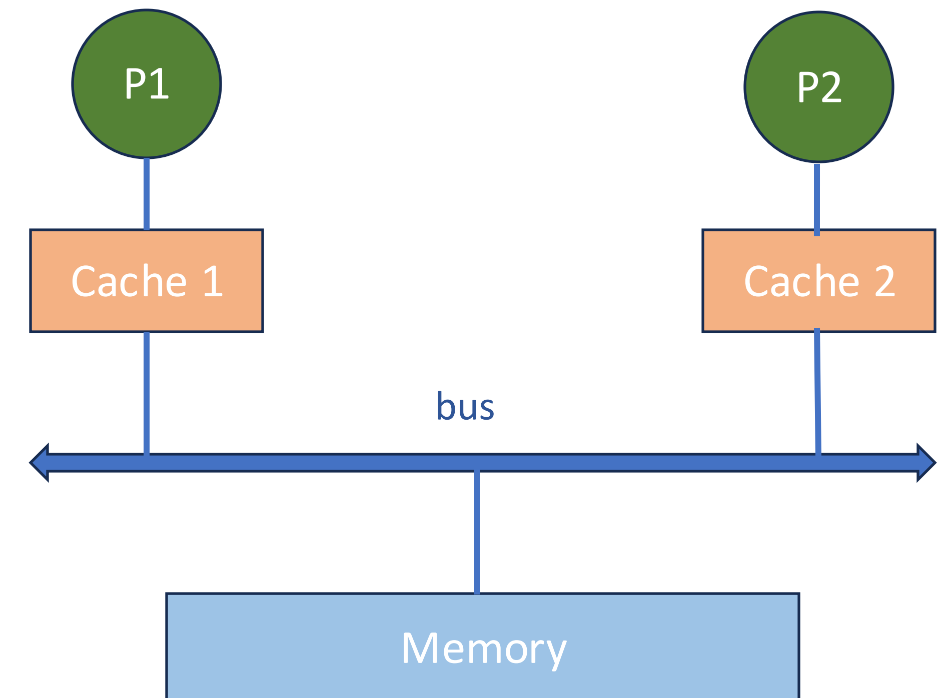
Snoopy Cache – Coherence Protocols

Memory bus is a broadcast medium

Any signal transmitted on the bus can be seen by all the cores connected to the bus.

When P1 updates x in its cache, it broadcasts update information (cache line containing x) across the bus.

P2 is snooping the bus and will see that x has been updated and it can mark its copy of x as invalid.



False Sharing

Remember that CPU caches operate on cache lines not individual variables.

This code can be parallelized by dividing the iterations in the outer loop among the cores.

If we have `p` cores, we can assign `m/p` iterations to each core.

```
int i, j, m, n;
double y[m];

for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
        y[i] += f(i, j);
```

False Sharing

Remember that CPU caches operate on cache lines not individual variables.

This code can be parallelized by dividing the iterations in the outer loop among the cores.

If we have `p` cores, we can assign `m/p` iterations to each core.

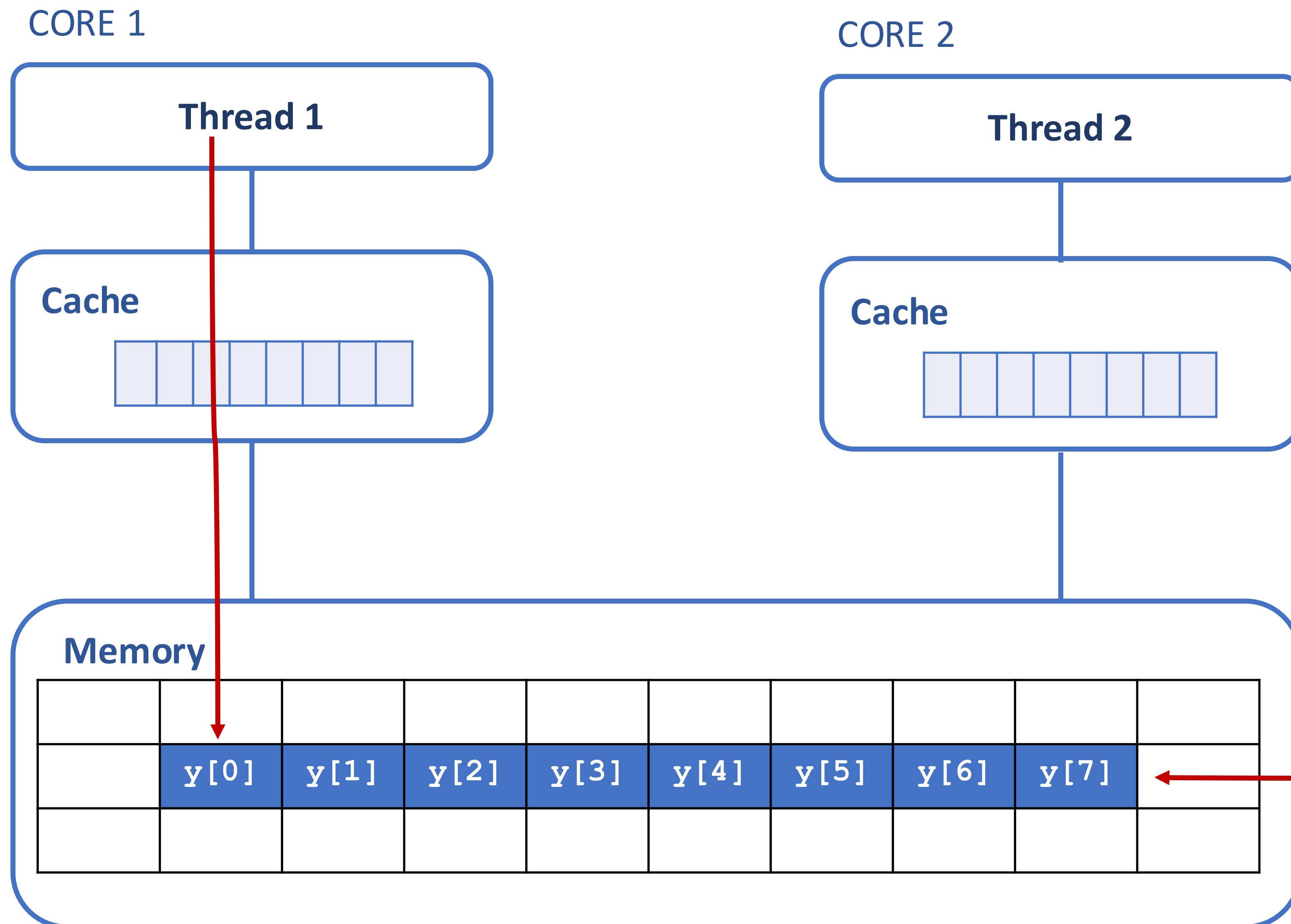
Let `m = 8`, `p = 2`, `cache_line_size = 64 bytes`

```
int i, j, iter_count;
int m, n, p;
double y[m];

iter_count = m / p;
```

```
for (i = 0; i < iter_count; i++)
    for (j = 0; j < n; j++)
        y[i] += f(i, j);
```

```
for (i = iter_count; i < 2 * iter_count; i++)
    for (j = 0; j < n; j++)
        y[i] += f(i, j);
```

False Sharing

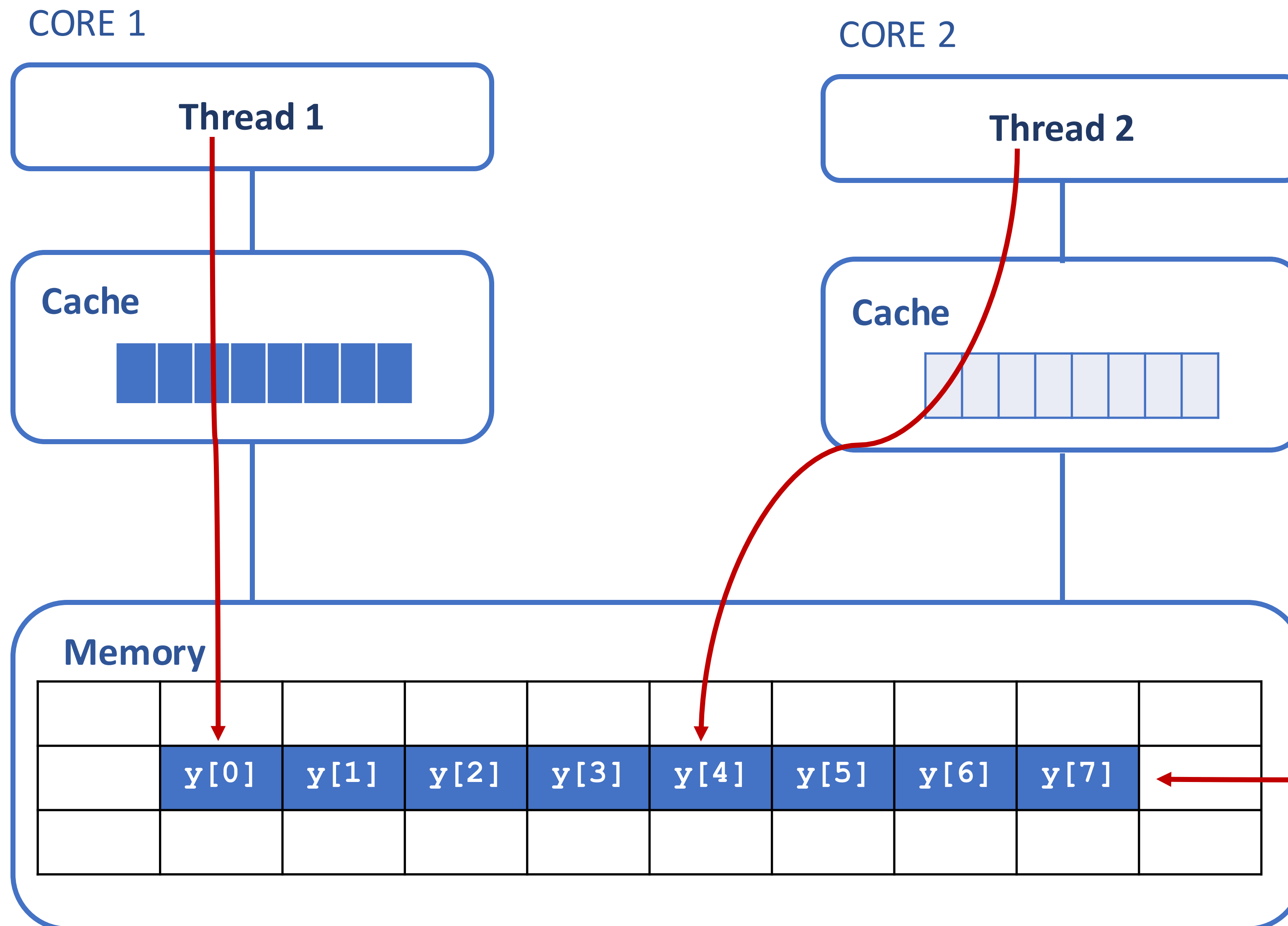



```
int i, j, iter_count;
int m, n, p;
double y[m];

iter_count = m / p;

for (i = 0; i < iter_count; i++)
    for (j = 0; j < n; j++)
        y[i] += f(i, j);

for (i = iter_count; i < 2 * iter_count; i++)
    for (j = 0; j < n; j++)
        y[i] += f(i, j);
```

CORE 1

CORE 2

False Sharing



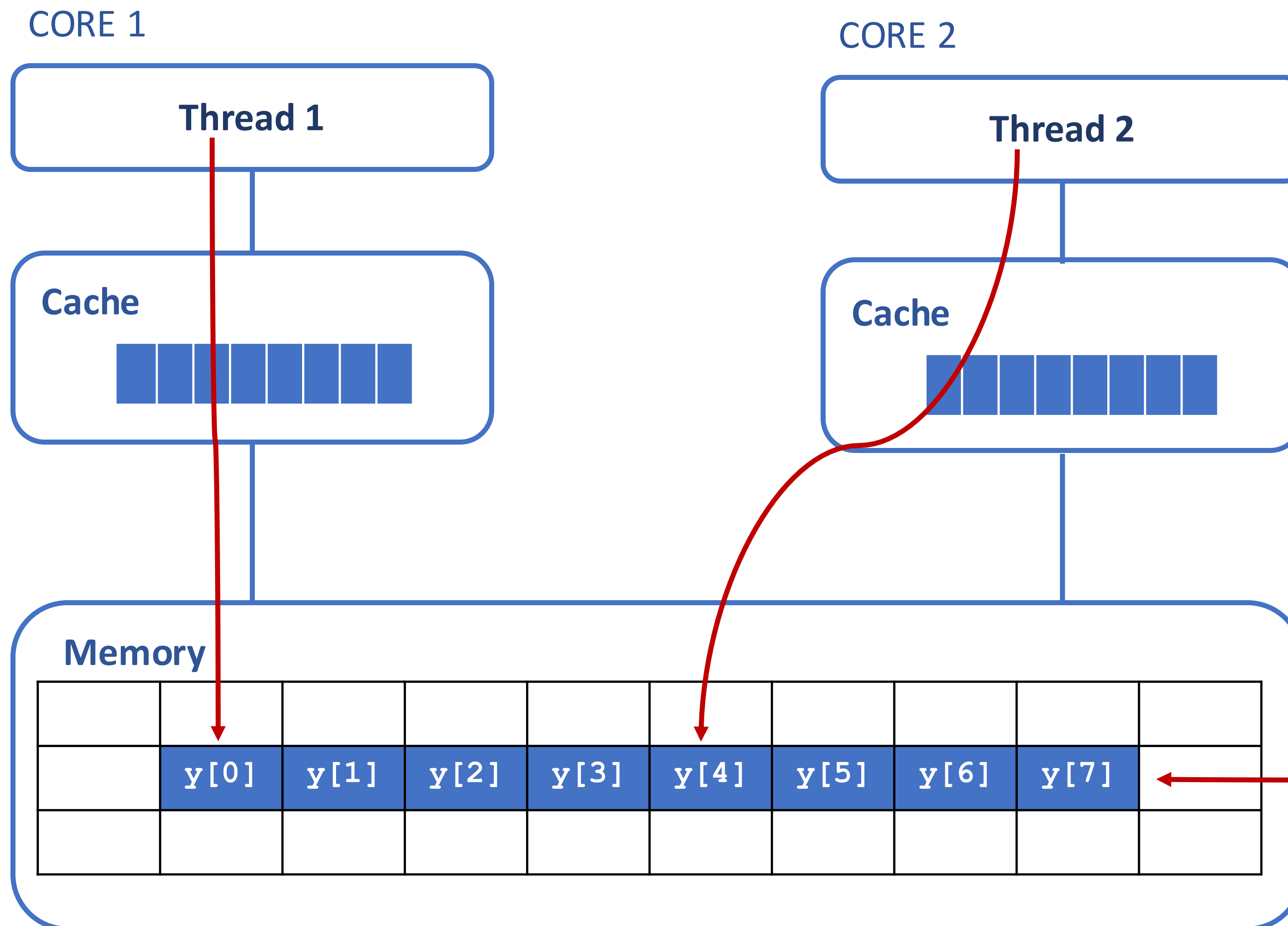

```
int i, j, iter_count;
int m, n, p;
double y[m];

iter_count = m / p;

for (i = 0; i < iter_count; i++)
    for (j = 0; j < n; j++)
        y[i] += f(i, j);

for (i = iter_count; i < 2 * iter_count; i++)
    for (j = 0; j < n; j++)
        y[i] += f(i, j);
```

CORE 1

CORE 2

The entire array y is in ONE cache line

False Sharing



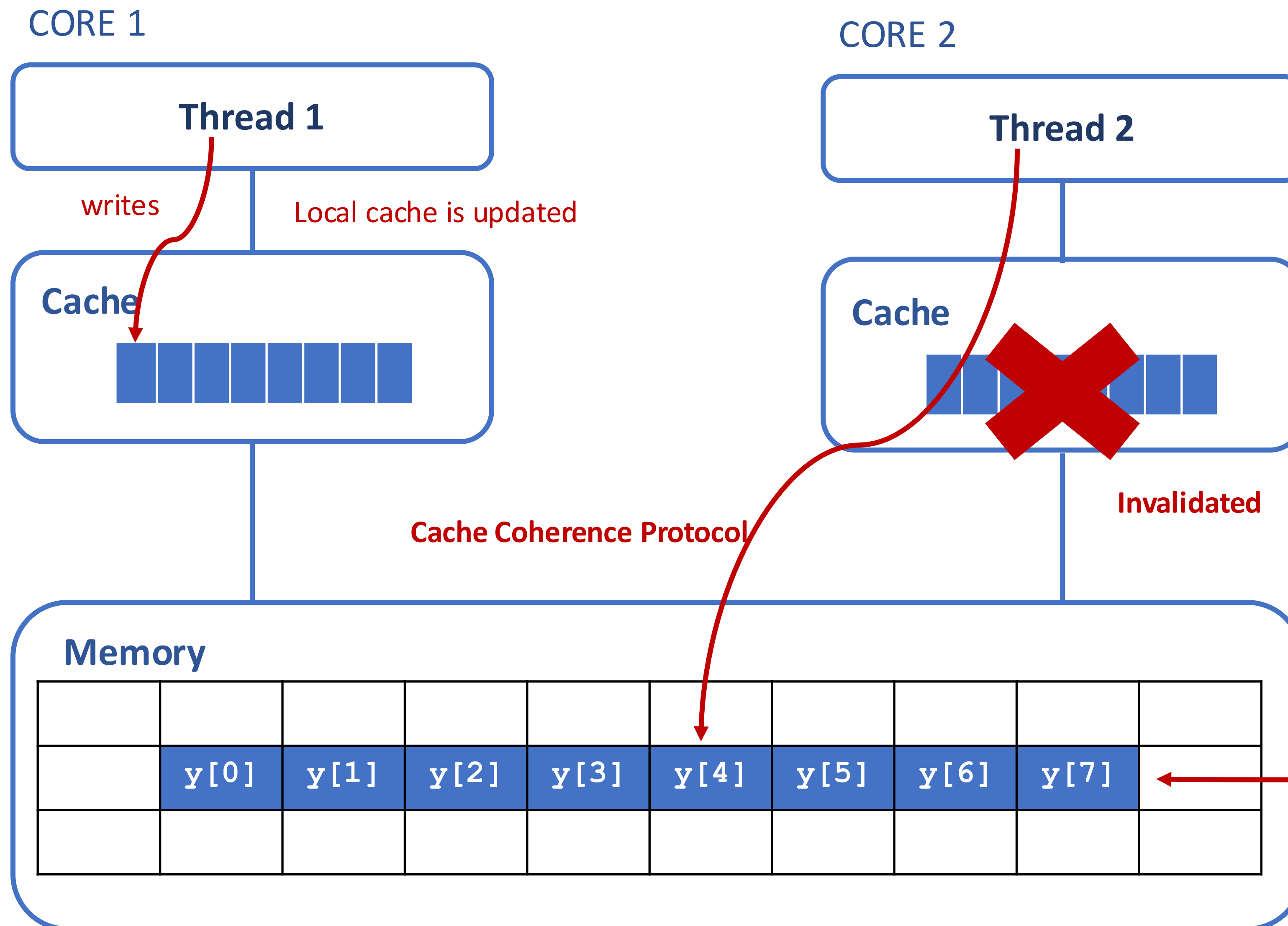

```c
int i, j, iter_count;
int m, n, p;
double y[m];

iter_count = m / p;

for (i = 0; i < iter_count; i++)
    for (j = 0; j < n; j++)
        y[i] += f(i, j);

for (i = iter_count; i < 2 * iter_count; i++)
    for (j = 0; j < n; j++)
        y[i] += f(i, j);
```

CORE 1

CORE 2

The entire array y is in ONE cache line

False Sharing




```
int i, j, iter_count;
int m, n, p;
double y[m];

iter_count = m / p;

for (i = 0; i < iter_count; i++)
    for (j = 0; j < n; j++)
        y[i] += f(i, j);                    CORE 1

for (i = iter_count; i < 2 * iter_count; i++)
    for (j = 0; j < n; j++)
        y[i] += f(i, j);                    CORE 2
```

The entire array y is in ONE cache line

The entire updated line from memory must be fetched

False Sharing

This problem is called false sharing, because the system is behaving as if the elements of y were being shared by the cores.

False sharing does not cause incorrect results, but it can ruin the performance by causing many more accesses to the main memory that needed.

How to handle false sharing problem?

- **Need to be sure that data modified by different threads live in different cache lines.**
- **Padding – insert extra unused bytes between elements**

```
#define CACHE_LINE_SIZE 64

typedef struct {
    double value;
    char padding[CACHE_LINE_SIZE - sizeof(double)];
} PaddedDouble;

PaddedDouble y[8];
```

Parallel PI Program

Parallel Pi Program

```
long num_steps = 1000000000;
double steps = 1.0 / num_steps;
double sum[NUM_THREADS] = {0.0};
int nthreads;
double pi = 0.0;

omp_set_num_threads(NUM_THREADS);

auto start = omp_get_wtime();

#pragma omp parallel
{
    int thread_id = omp_get_thread_num();
    int num_threads = omp_get_num_threads();
    double x = 0.0;

    if (thread_id == 0)
        nthreads = num_threads;

    for (int i = thread_id; i < num_steps; i += num_threads) {
        x = (i + 0.5) * steps;
        sum[thread_id] += fcn_eval(x);
    }
}

for (int i = 0; i < nthreads; i++)
    pi += sum[i] * steps;

auto end = omp_get_wtime();
```

Running PI program with 100 million steps

Threads	WCT [seconds]
1	2.239
2	2.224
4	1.552
8	2.805
12	3.251

Compiled with G++14 with -O0 and executed on Apple macOS X 15.3.2, 18 GB memory, and Apple M3 Pro chip with 12 HW Threads.

Parallel Pi Program (Padded)

```
#define NUM_THREADS 4
#define PAD_SIZE 16 // Assume 128 bytes cacheline size (16 * 8 bytes for double)

inline double fcn_eval(double x) {
    return 4.0 / (1.0 + x * x);
}

int main(int argc, char const *argv[]) {
    long num_steps = 1000000000;
    double steps = 1.0 / num_steps;
    double sum[NUM_THREADS][PAD_SIZE] = {0.0};
    int nthreads;
    double pi = 0.0;

    omp_set_num_threads(NUM_THREADS);

    auto start = omp_get_wtime();

#pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        int num_threads = omp_get_num_threads();
        double x = 0.0;

        if (thread_id == 0)
            nthreads = num_threads;

        for (int i = thread_id; i < num_steps; i += num_threads) {
            x = (i + 0.5) * steps;
            sum[thread_id][0] += fcn_eval(x);
        }
    }

    for (int i = 0; i < nthreads; i++) {
        pi += sum[i][0] * steps;
    }

    auto end = omp_get_wtime();
```

Running PI program with 100 million steps

Threads	WCT [seconds]	SpeedUp
1	2.213	-
2	1.129	1.96
4	0.575	3.848
8	0.415	5.332
12	0.375	5.901

$$Speedup = \frac{T_{serial}}{T_{parallel}}$$

Compiled with G++14 with -O0 and executed on Apple macOS X 15.3.2, 18 GB memory, and Apple M3 Pro chip with 12 HW Threads.

Synchronization in OpenMP

High level synchronization included in OpenMP

- **`critical`**
- **`barrier`**
- many more

Synchronization is used to impose order constraints and to protect access to shared data.

Synchronization in OpenMP

critical – only one thread at a time can enter the critical region

```
float res = 0.0f;

#pragma omp parallel
{
    float B;
    int i, id, nthrds;

    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();

    for (i = id; i < niters; i += nthrds) {
        B = big_job(i);

#pragma omp critical
        res += consume(B);
    }
}
```

barrier – a point in a program all threads must reach before any threads are allowed to proceed.

```
double Arr[8], Brr[8];
int numthrds;

omp_set_num_threads(8);

#pragma omp parallel
{
    int id = omp_get_thread_num();
    int nthrds = omp_get_num_threads();

    if (id == 0)
        numthrds = nthrds;

    Arr[id] = big_ugly_calc(id, nthrds);

#pragma omp barrier
    Brr[id] = really_big_and_ugly(id, nthrds, Arr)
}
```

Example: Synchronization to avoid false sharing

Create a scalar local to each thread to accumulate partial sums.

No array, no false sharing

```
#pragma omp parallel
    {
        double sum = 0.0;
        int thread_id = omp_get_thread_num();
        int num_threads = omp_get_num_threads();
        double x = 0.0;

        if (thread_id == 0)
            nthreads = num_threads;

        for (int i = thread_id; i < num_steps; i += num_threads) {
            x = (i + 0.5) * steps;
            sum += fcn_eval(x);
        }

#pragma omp critical
        {
            pi += sum * steps;
        }
    }

    auto end = omp_get_wtime();
```

Loop Parallelization

OpenMP can split up loop iterations among the threads in a team

Implicit `barrier` after the loop

Loop control variable `i` is local to each thread

```
#pragma omp parallel
{
#pragma omp for
    for (int i = 0; i < n; i++) {
        neat_stuff(i);
    }
}
```

Loop Parallelization

sequential code

```
for (int i = 0; i < N; i++) {
    a[i] = a[i] + b[i];
}
```

`#pragma omp parallel`

```
#pragma omp parallel
{
    int id, i, nthrds, istart, iend;

    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();

    istart = id * N / nthrds;
    iend = (id + 1) * N / nthrds;

    if (id == nthrds - 1)
        iend = N;

    for (i = istart; i < iend; i++) {
        a[i] = a[i] + b[i];
    }
}
```

`#pragma omp parallel`

`#pragma omp for`

```
#pragma omp parallel
{
    #pragma omp for
    for (int i = 0; i < N; i++) {
        a[i] = a[i] + b[i];
    }
}
```

`#pragma omp parallel for`

```
#pragma omp parallel for
for (int i = 0; i < N; i++) {
    a[i] = a[i] + b[i];
}
```

Loop Parallelization

Nested Loops

```
#pragma omp parallel for
for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
        C[i][j] = 0;
        for (int k = 0; k < K; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
        C[i][j] = 0;
        for (int k = 0; k < K; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Reduction

Hot do we handle this case?

We are combining values into a single accumulation variable ave. There is a true dependency between loop iterations that cannot trivially removed.

This is a very common situation which is called a reduction.

Reduction operations include `+`, `*`, `-`, `max`, `min`, etc..

Support for reduction operations is included in most parallel programming environments.

```
double ave = 0.0, A[MAX];
int i;

for (i = 0; i < MAX; i++) {
    ave += A[i];
}

ave = ave / MAX;
```

Reduction

OpenMP reduction clause:

`reduction (op: list)`

Inside a parallel construct:

- A local copy of each `list` variable is made and initialized depending on the `op`
- Updates occur on the local copy
- Local copies are reduced into a single value and combined with the original global value

The variables in `list` must be shared in the enclosing parallel region

```
double ave = 0.0, A[MAX];
int i;

#pragma omp parallel for reduction(+:ave)
for (i = 0; i < MAX; i++) {
    ave += A[i];
}

ave = ave / MAX;
```

Example: dot product

- Given two vectors a and b, the dot product is defined as

$$x = \sum_{i=0}^{N-1} a[i].\,b[i]$$

Dot product program in OpenMP

```
#pragma omp parallel for reduction(+:dot)
for (int i = 0; i < N; ++i) {
    dot += A[i] * B[i];
}
```

serial dot product

```
for (int i = 0; i < N; ++i) {
    dot += A[i] * B[i];
}
```

PI program with loop and reduction

```
    long num_steps = 1000000000;
    double steps = 1.0 / num_steps;

    double pi = 0.0;
    double sum = 0.0;

    omp_set_num_threads(NUM_THREADS);

    double start = omp_get_wtime();

#pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < num_steps; i++) {
        double x = (i + 0.5) * steps;
        sum += fcn_eval(x);
    }

    pi = sum * steps;
    double end = omp_get_wtime();
```

Exercise

Benchmark different parallel `OpenMP` implantations (manual, with padding, critical, and reduction) with different number of threads and compare the WCT.

Task – based parallelism in OpenMP

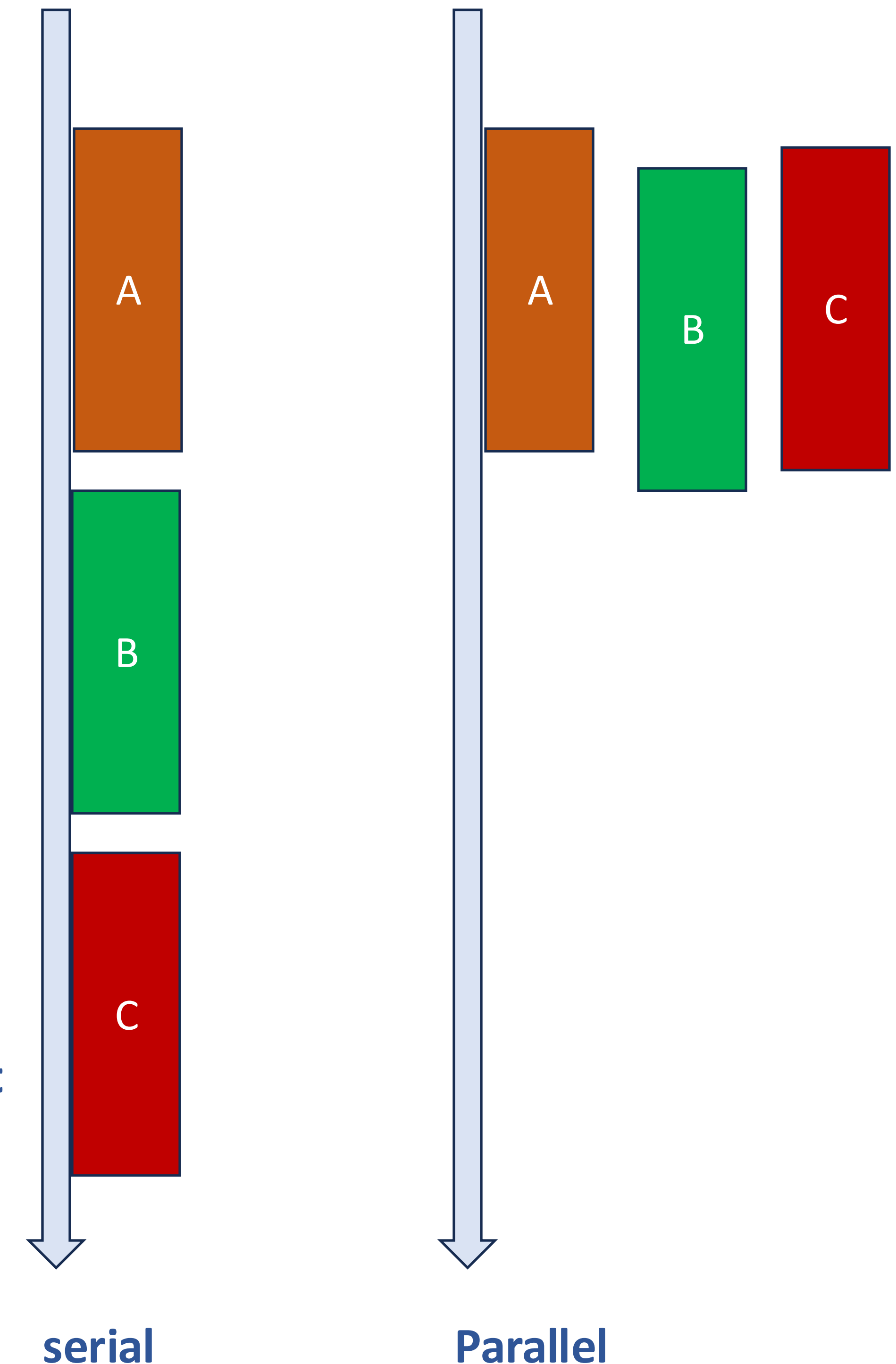
Tasks are independent units of work each composed of

- Code to Execute
- Data to compute with

Threads are assigned to perform the work of each task

- The thread that encounters the task construct may execute the task immediately
- The threads may defer execution until later

A common pattern is to have one thread create the tasks while other threads wait at a barrier and execute the tasks.



The `Single` Construct

The **`single`** construct denotes a **block of code that is executed by only one thread** (not necessarily the master thread)

A **`barrier`** is implied at the end of the single block.

```
#pragma omp parallel
{
    do_many_things();

    #pragma omp single
    {
        exchange_boundaries();
    }

    do_many_other_things();
}
```

Task Directive

```
#pragma omp task [clauses]
```

Defines a task that can be executed **asynchronously** by any thread in the team.

Instead of running code immediately, it **creates a task and adds it to a task queue**. Threads pull from this queue when they're free.

```
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        fred();

        #pragma omp task
        daisy();

        #pragma omp task
        billy();
    }
}
```

Create some threads → `#pragma omp parallel`

One thread creates tasks → `#pragma omp single`

Tasks executed by some thread in some order. → the three `#pragma omp task` blocks

All tasks complete before this barrier is released. → closing `}` of the single block

Task Directive

#pragma omp task [clauses]

Defines a task that can be executed **asynchronously** by any thread in the team.

Instead of running code immediately, it **creates a task and adds it to a task queue**. Threads pull from this queue when they're free.

Create some threads

One thread creates tasks

Tasks executed by some thread in some order.

All tasks complete before this barrier is released.

```
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        fred();

        #pragma omp task
        daisy();
        #pragma omp taskwait
        #pragma omp task
        billy();
    }
}
```

fred() and daisy() must complete before billy() starts

Resources

We just covered a summary of OpenMP specifications

https://github.com/UoB-HPC/openmp-tutorial/tree/master

Tim Mattson runs OpenMP tutorials worldwide.

Next Lecture: Parallel Program design and analysis