Optimizing Mulithreading Performance

Unveiling False Sharing and Harnessing Hardware Destructive Interference

Shivam Kunwar (shivam.kunwar@kdab.com)

KDAB

- → I work at KDAB as a Software Engineer.
- → Contributing to LLVM Compiler Infrastructure
- → Google Summer of Code Student
- → Student
- → Fascinated about particle physics and Cosmology

Outline of the talk:

- Intro to Mulithreading
- When and When not to use it?
- Factors affecting the performance of multithreaded code
- Dive in False Sharing

Multithreading

→ Let's have a fun analogy

KDAB

Multithreading

KDAB

The Busy Kitchen

- Imagine you're running a busy restaurant kitchen, and you have to prepare different dishes simultaneously to serve customers quickly.

Multithreading

KDAB

The Busy Kitchen

- Imagine you're running a busy restaurant kitchen, and you have to prepare different dishes simultaneously to serve customers quickly.
- You, as the chef, are like a computer's CPU, and the different dishes are tasks or processes that your computer needs to handle.

Multithreading

The Busy Kitchen

- Imagine you're running a busy restaurant kitchen, and you have to prepare different dishes simultaneously to serve customers quickly.
- You, as the chef, are like a computer's CPU, and the different dishes are tasks or processes that your computer needs to handle.
- Multithreading is like having multiple chefs (threads) in the kitchen, each working on a separate task at the same time.

KDAB
The Qt, OpenGL and C++ experts

Question

KDAB

→ What happen if your restaurant does not have enough worker?

Customers running out of a restaurant due to slow service!

So what is Multithreading, formally?

So what is Multithreading, formally?

It’s the ability of the operating system and software application to take advantage of the additional CPU cores available in the system, by splitting up the workload in several independent parts and performing calculations separately on each core.

Example

KDAB

- Let’s consider you have to calculate the sum of an array which have **10 million** elements.

Example

KDAB

→ Let’s consider you have to calculate the sum of an array which have **10 million** elements.

→ So you would write a pr

```
long long sum_array(int input[], int n) {
    long long total_sum = 0;
    for (int i = 0; i < n; i++) {
        total_sum += input[i];
    }
    return total_sum;
}
```

Example

ΔKDAB

- → Let’s consider you have to calculate the sum of an array which have **10 million** elements.
- → So you would write a pro

```
long long sum_array(int input[], int n) {
    long long total_sum = 0;
    for (int i = 0; i < n; i++) {
        total_sum += input[i];
    }
    return total_sum;
}
```

```
long long total = sum_array(input_array, 10 * M);
```

→ Alright, but the good thing about the previous problem is, it is trivial to parallelize!

- → Alright, but the good thing about the previous problem is, it is trivial to parallelize!
- → Which means, we can divide the array into two halves and calculate the sum for each halves separately on different cores and then merge the result.

→ Alright, but the good thing about the previous problem is, it is trivial to parallelize!

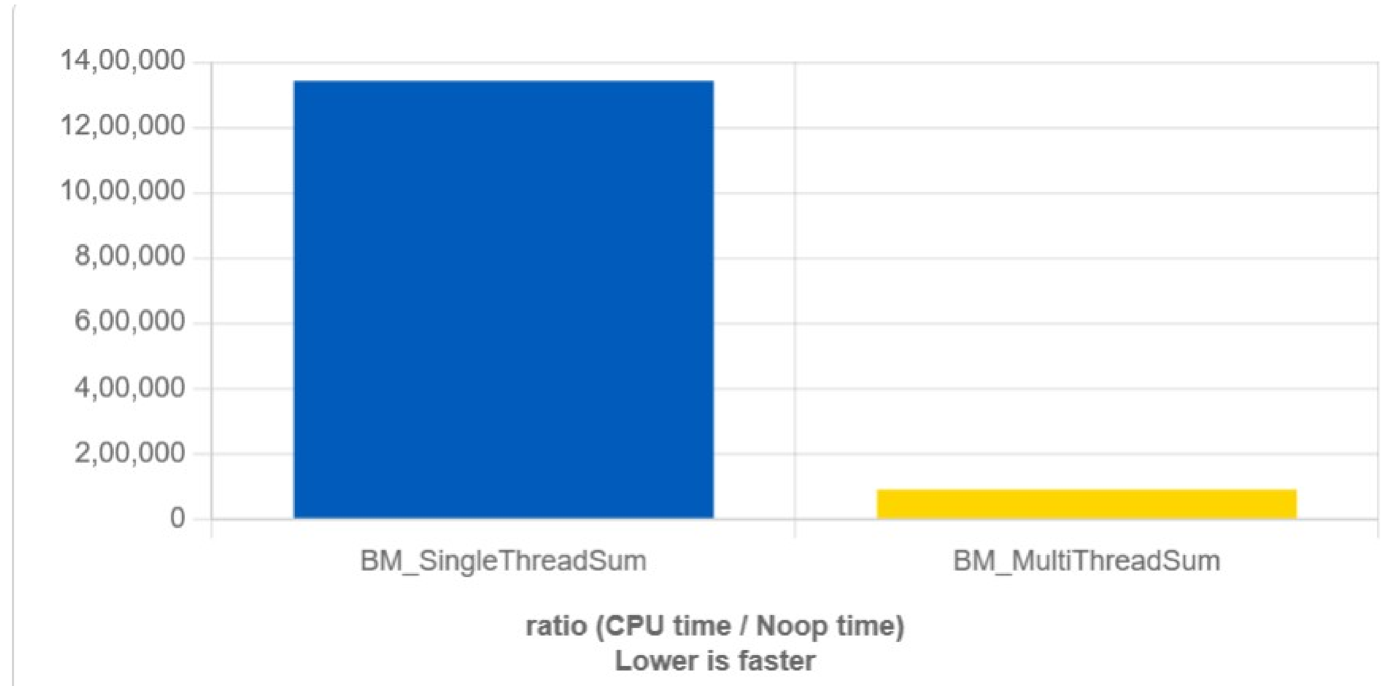
→ Which means, we can divide the array into two halves and calculate the sum for each halves separately on different cores and then merge the resu

```
thread_t left_sum_thread = spawn_thread(sum_array(input_array, 5 * M));
thread_t right_sum_thread = spawn_thread(sum_array(input_array + 5 * M, 5 * M));

// Both threads now execute in parallel
left_sum_thread.wait_to_complete();
right_sum_thread.wait_to_complete();

long long total_sum = left_sum_thread.result + right_sum_thread.result;
```

Quick benchmark for the summing the total number of elements in the vector

[Benchmark summing the vector elements in single thread and multithread]

- → BM_SingleThreadSum measuring the performance of summing numbers in a single-threaded fashion.
- → BM_MultiThreadSum measuring the performance of summing numbers using two threads.

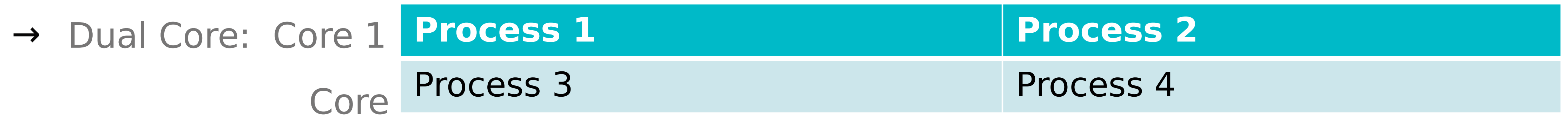


Multithreading

→ Single Core: Core 1

→ Dual Core: Core 1

Core

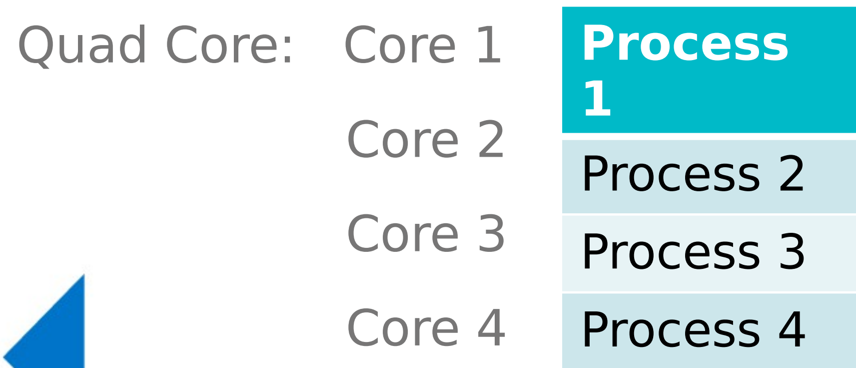


Quad Core: Core 1

Core 2

Core 3

Core 4



Generic way to use multithreading for performance improvement

KDAB

Generic way to use multithreading for performance improvement

KDAB

→ Divide the workload, based on the number of available hardware threads(N), partition the tasks into N smaller and independent sections.

Generic way to use multithreading for performance improvement

KDAB

→ Divide the workload, based on the number of available hardware threads(N), partition the tasks into N smaller and independent sections.

→ Launch N software threads, with each handling specific task.

Generic way to use multithreading for performance improvement

KDAB

- → Divide the workload, based on the number of available hardware threads(N), partition the tasks into N smaller and independent sections.
- → Launch N software threads, with each handling specific task.
- → Synchronize completion, wait for all of the thread to finish their task.

Generic way to use multithreading for performance improvement

KDAB

- → Divide the workload, based on the number of available hardware threads(N), partition the tasks into N smaller and independent sections.
- → Launch N software threads, with each handling specific task.
- → Synchronize completion, wait for all of the thread to finish their task.
- → If needed, combine the different computations to get the final result.

KDAB

And yes it isn't simple as it seems!

Concerns to care about

KDAB

→ Spawning number of threads is not cheap, it’s resource intensive task. Therefore, multithreading is more suitable for tasks that are sufficiently large to offset this overhead.

The Qt, OpenGL and C++ experts

Concerns to care about

KDAB

- → Spawning number of threads is not cheap, it’s resource intensive task. Therefore, multithreading is more suitable for tasks that are sufficiently large to offset this overhead.
- → Imbalance in handing the workload, so sometimes one or more threads may take longer to complete tasks and delaying the overall result time. So what is the solution for this?

Concerns to care about

KDAB

→ Spawning number of threads is not cheap, it’s resource intensive task. Therefore, multithreading is more suitable for tasks that are sufficiently large to offset this overhead.

→ Imbalance in handing the workload, so sometimes one or more threads may take longer to complete tasks and delaying the overall result time. So what is the solution for this?

→ A probable solution for this, that split the workload into more pieces than there are threads. So as soon as threads complete it’s task, it can fetch other tasks to execute from the queue.

Concerns to care about

KDAB

- → Spawning number of threads is not cheap, it’s resource intensive task. Therefore, multithreading is more suitable for tasks that are sufficiently large to offset this overhead.
- → Imbalance in handing the workload, so sometimes one or more threads may take longer to complete tasks and delaying the overall result time. So what is the solution for this?
- → A probable solution for this, that split the workload into more pieces than there are threads. So as soon as threads complete it’s task, it can fetch other tasks to execute from the queue.
- → Synchronization, when there are shared resources.

Concerns to care about

KDAB

- → Spawning number of threads is not cheap, it’s resource intensive task. Therefore, multithreading is more suitable for tasks that are sufficiently large to offset this overhead.
- → Imbalance in handing the workload, so sometimes one or more threads may take longer to complete tasks and delaying the overall result time. So what is the solution for this?
- → A probable solution for this, that split the workload into more pieces than there are threads. So as soon as threads complete it’s task, it can fetch other tasks to execute from the queue.
- → Synchronization, when there are shared resources.
- → Complexity in dividing the workloads dynamically during the computation

The Qt, OpenGL and C++ experts

Use Cases of Multithreading

KDAB

1. Web Servers

Use Cases of Multithreading

KDAB

1. Web Servers
2. Database systems

Use Cases of Multithreading

1. Web Servers
2. Database systems
3. Gaming

Use Cases of Multithreading

1. Web Servers
2. Database systems
3. Gaming
4. Scientific Computing

Use Cases of Multithreading

KDAB

1. Web Servers
2. Database systems
3. Gaming
4. Scientific Computing
5. Financial trading

Advantages of Multithreading

KDAB

→ Enhanced performance

→ Improve GUI responsiveness

→ Simultaneous and parallelized occurrence of tasks

→ Better use of cache storage by utilization of resources

→ Better use of CPU resource

Disadvantages of Multithreading

KDAB

- → Complex debugging and testing processes
- → Overhead switching of context
- → Increased potential for deadlock occurrence
- → Increased difficulty level in writing a program
- → Unpredictable results

When to use Multithreading?

KDAB

When to use Multithreading?

KDAB

→ 1. Separation of Concerns:

When to use Multithreading?

KDAB

- → 1. Separation of Concerns:
- → **Example**: Consider a music streaming application for a smartphone or computer. This application fundamentally has two primary responsibilities:

When to use Multithreading?

KDAB

- 1. Separation of Concerns:
- **Example**: Consider a music streaming application for a smartphone or computer. This application fundamentally has two primary responsibilities:
  - It must stream music data from the server, decode the audio files, and play them without any interruptions.

When to use Multithreading?

KDAB

- → 1. Separation of Concerns:
- → **Example**: Consider a music streaming application for a smartphone or computer. This application fundamentally has two primary responsibilities:
  - → It must stream music data from the server, decode the audio files, and play them without any interruptions.
  - → Simultaneously, it should respond to user inputs such as Play, Pause, Next, Previous, or adjusting the volume.

When to use Multithreading?

```
MusicStreamingApp app;

std::thread streaming_thread(&MusicStreamingApp::stream_music_data, &app);
std::thread ui_thread(&MusicStreamingApp::user_interface, &app);

streaming_thread.join();
ui_thread.join();
```

- 1. Separation of Concerns:
- **Example**: Consider a music streaming application for a smartphone or computer. This application fundamentally has two primary responsibilities:
  - It must stream music data from the server, decode the audio files, and play them without any interruptions.
  - Simultaneously, it should respond to user inputs such as Play, Pause, Next, Previous, or adjusting the volume.

When to use Multithreading?

KDAB

→ 2. Improved Performance - Task and Data Parallelism:

When to use Multithreading?

KDAB

→ 2. Improved Performance - Task and Data Parallelism:

Task Parallelism: Divide a single task into sub-tasks and execute them in parallel.

When to use Multithreading?

KDAB

→ 2. Improved Performance - Task and Data Parallelism:

Task Parallelism: Divide a single task into sub-tasks and execute them in parallel.

Data Parallelism: Each thread performs the same operation on different parts of the data.

When to use Multithreading?

KDAB

→ 2. Improved Performance - Task and Data Parallelism:

Task Parallelism: Divide a single task into sub-tasks and execute them in parallel.

Data Parallelism: Each thread performs the same operation on different parts of the data.

→ **Exampl**

→ array in parallel.

```
void process(int data) {
    // some processing on data
}
std::vector<int> dataArray = {1, 2, 3, 4};
std::for_each(std::execution::par, dataArray.begin(), dataArray.end(), process);
```

KDAB

→ std::execution::par is one of the execution policy from C++17.

- → std::execution::par is one of the execution policy from C++17.
- → It implies parallel execution for the standard library algorithms. And that means all the execution of different processes will happen in parallel “Safely”.

When not to use Multithreading?

KDAB

When not to use Multithreading?

→ 1. **Complexity vs. Benefit**:

When not to use Multithreading?

KDAB

→ 1. **Complexity vs. Benefit**:

Concurrency adds complexity to code, making it harder to understand and maintain.

When not to use Multithreading?

KDAB

→ 1. **Complexity vs. Benefit**:

Concurrency adds complexity to code, making it harder to understand and maintain.

If the expected performance gain isn't worth this increased complexity, it's better to avoid concurrency.

When not to use Multithreading?

→ 1. **Complexity vs. Benefit**:

Concurrency adds complexity to code, making it harder to understand and maintain.

If the expected performance gain isn't worth this increased complexity, it's better to avoid concurrency.

2. **Performance Overhead**:

When not to use Multithreading?

KDAB

→ 1. **Complexity vs. Benefit**:

Concurrency adds complexity to code, making it harder to understand and maintain.

If the expected performance gain isn't worth this increased complexity, it's better to avoid concurrency.

2. **Performance Overhead**: Starting a thread has its inherent overhead. If tasks complete quickly, the overhead of launching the thread might outweigh the benefits.

When not to use Multithreading?

3. **Limited Resources**:

When not to use Multithreading?

KDAB

3. **Limited Resources**: Threads consume system resources. Too many threads can slow down the system, exhaust memory, and even lead to resource contention.

When not to use Multithreading?

KDAB

3. **Limited Resources**: Threads consume system resources. Too many threads can slow down the system, exhaust memory, and even lead to resource contention.

4. **Potential for More Bugs**:

When not to use Multithreading?

KDAB

3. **Limited Resources**: Threads consume system resources. Too many threads can slow down the system, exhaust memory, and even lead to resource contention.

4. **Potential for More Bugs**: Multithreaded code can lead to race conditions, deadlocks, and other tricky bugs that might not appear in single-threaded applications.

Factors Affecting Multithreaded Performance

KDAB

Wait, Let's have a look at something basic.

Caches

But why?

Caches

KDAB

- Synchronization mechanisms, such as those involved in locking, are closely tied to cache coherence and the consistency of memory.

Caches

KDAB

- Synchronization mechanisms, such as those involved in locking, are closely tied to cache coherence and the consistency of memory.
- Many optimization techniques manipulate cache coherence protocols to enhance performance.

Caches

KDAB

- → Synchronization mechanisms, such as those involved in locking, are closely tied to cache coherence and the consistency of memory.
- → Many optimization techniques manipulate cache coherence protocols to enhance performance.
- → For instance, a locking mechanism may initially perform a read operation on a lock before attempting to change it atomically.

Caches

- Synchronization mechanisms, such as those involved in locking, are closely tied to cache coherence and the consistency of memory.
- Many optimization techniques manipulate cache coherence protocols to enhance performance.
- For instance, a locking mechanism may initially perform a read operation on a lock before attempting to change it atomically.
- This initial read operation does not provoke cache invalidations across other cores, which helps to conserve latency and on-chip bandwidth.

Storage Level Characteristics

	L1	L2	L3	Memory	Disk
Type of Storage	On-chip	On-chip	On-chip	Off-chip	Disk
Typical Size	100 KB	8 MB	32 MB	32 GB	Many GBs
Typical Access Time (ns)	.25	.50	10.8	50	5,000,000
Scaled Access Time	1 second	2 seconds	43 seconds	3.3 minutes	231 days
Managed by	Hardware	Hardware	Hardware	OS	OS

Taken from: John Hennessy and David Patterson, Computer Architecture: A
Quantitative Approach, Morgan-Kaufmann, 2007. (4th Edition)

Caches

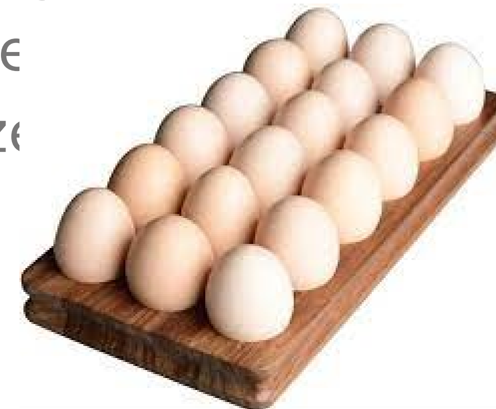
→ When the CPU retrieves a requested memory value that is already stored in the cache, it can access it rapidly. This occurrence is known as a **cache hit**.

Caches

- → When the CPU retrieves a requested memory value that is already stored in the cache, it can access it rapidly. This occurrence is known as a **cache hit**.
- → When the CPU requests a value that is not present in the cache, it necessitates retrieving the data from outside the processor chip. This event is referred to as a **cache miss**.

Caches

- When the CPU retrieves a requested memory value that is already stored in the cache, it can access it rapidly. This occurrence is known as a **cache hit**.
- When the CPU requests a value that is not present in the cache, it necessitates retrieving the data from outside the processor chip. This event is referred to as a **cache miss**.
- Even though the overall capacity of a cache can range from several kilobytes to megabytes, the data is not transferred all at once [...] aller segments known as cache lines. Commonly, the size [...] e line is around 64 bytes.



Generating Cache hits

Two Assumptions to be true

KDAB

Generating Cache hits

KDAB

Two Assumptions to be true

→ **Temporal Locality:** When a program accesses a memory location, it is likely to access that same location again in the near future.

Generating Cache hits

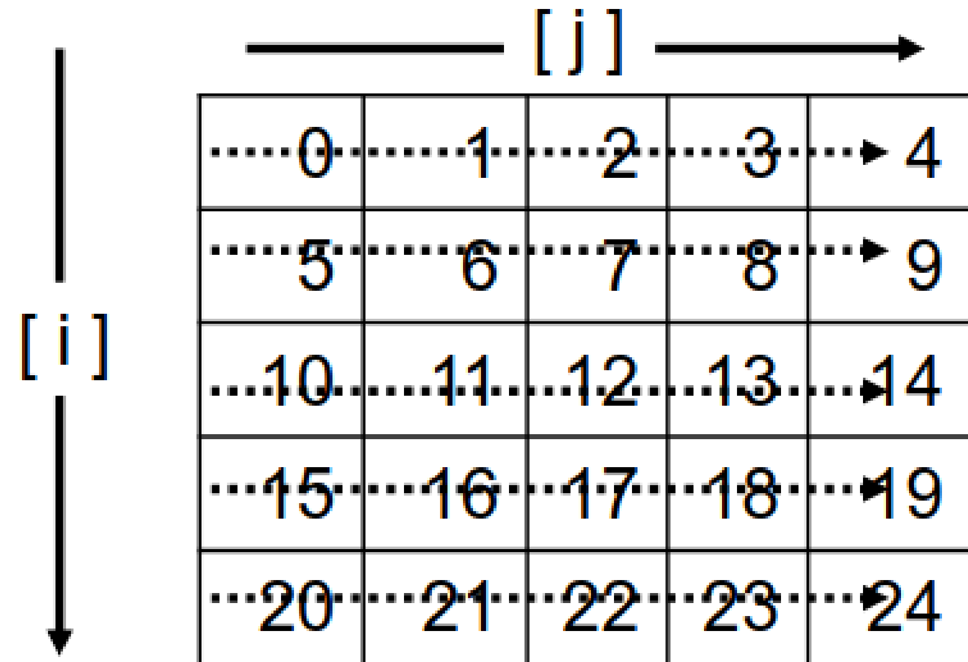
KDAB

- Two Assumptions to be true
- **Temporal Locality:** When a program accesses a memory location, it is likely to access that same location again in the near future.
- **Spatial Locality:** This concept predicts that if a particular storage location is accessed, locations whose addresses are close by are likely to be accessed soon.

And if these assumptions are not true then you will generate a lot of cache misses and re- loading the caches a lot.

How bad is Cache miss?

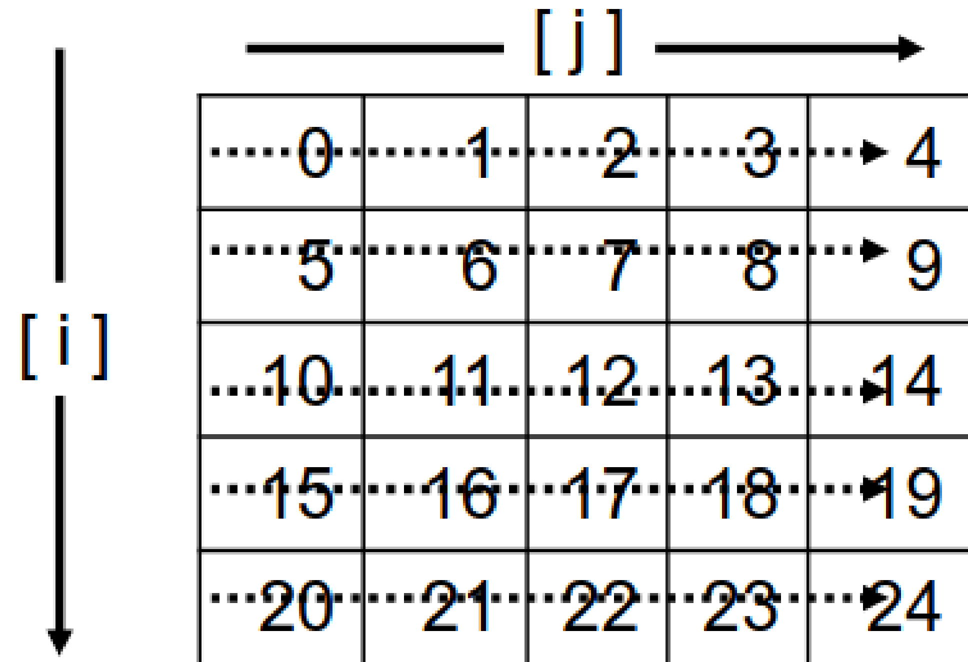
→ In C, C++, 2D arrays get stored in memory by row at a time, means A[i][j].



How bad is Cache miss?

→ In C, C++, 2D arrays get stored in memory by row at a time, means A[i][j].

→ For large arrays, would it be better to add the elements by row, or by column? Which will avoid the most cache misses?



How bad is Cache miss?

KDAB

```
const int ROWS = 10000;
const int COLS = 10000;
int array[ROWS][COLS];
```

How bad is Cache miss?

KDAB

```
const int ROWS = 10000;
const int COLS = 10000;
int array[ROWS][COLS];
```

```
void initializeArray() {
    for (int i = 0; i < ROWS; ++i) {
        for (int j = 0; j < COLS; ++j) {
            array[i][j] = 1; // Initialize all elements to 1 for simplicity
        }
    }
}
```

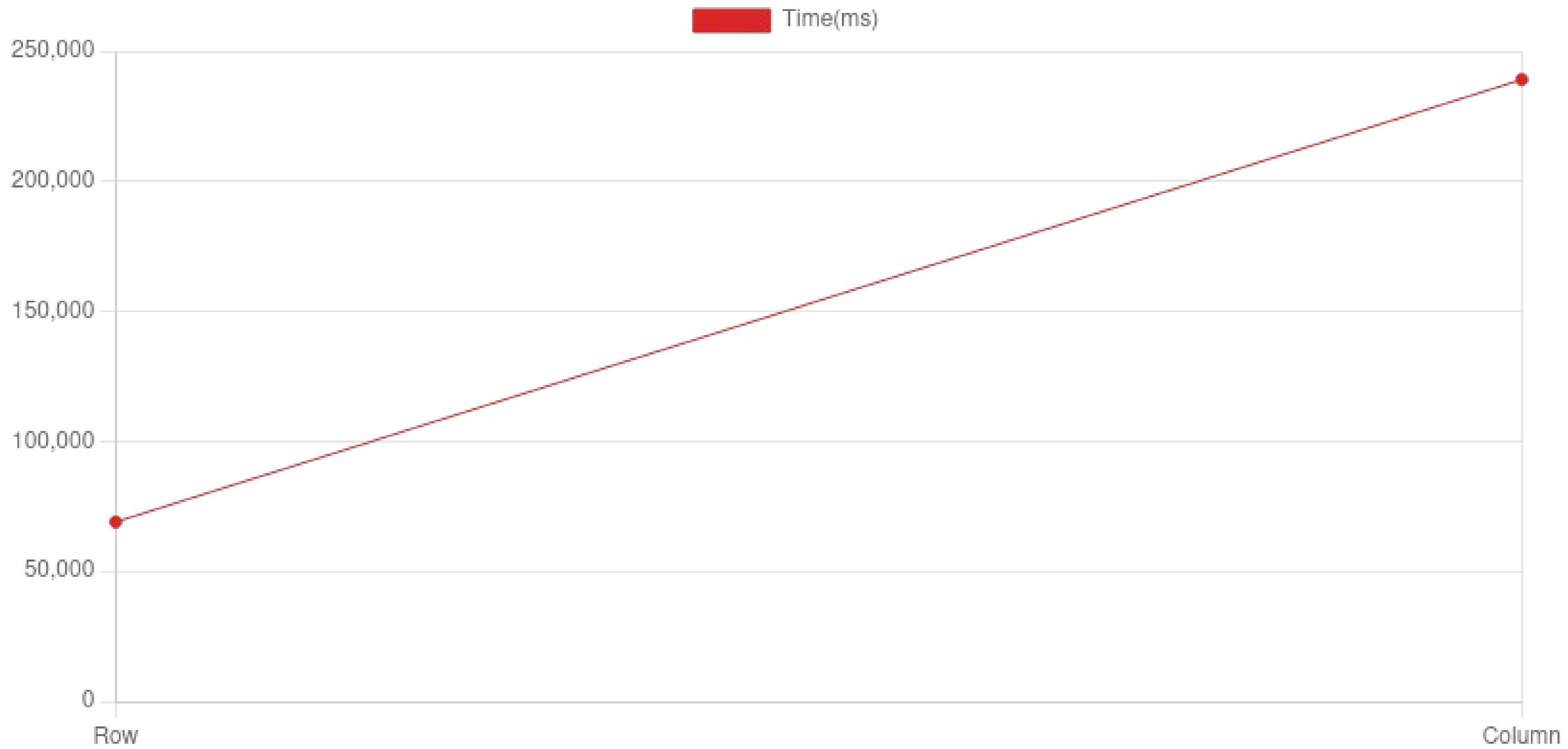
How bad is Cache miss?

```
void sumByRows() {
  volatile int sum = 0;
    for (int i = 0; i < ROWS; ++i) {
        for (int j = 0; j < COLS; ++j) {
            sum += array[i][j]; // access across row
        }
    }
}
```

```
void sumByColumns() {
  volatile int sum = 0;
    for (int j = 0; j < COLS; ++j) {
      for (int i = 0; i < ROWS; ++i) {
        sum += array[i][j]; // acess down column
      }
    }
}
```

How bad is Cache miss?

KDAB



How bad is Cache miss?

KDAB

→ So basically design your data structure so it can take benefit of sequential memory order to avoid cache misses

Sequential memory order -> Array[i][j]

Jump-around-in-memory order-> Array[j][i]

How different cores keep track of the cache line?

How different cores keep track of the cache line modifications?

→ We need some sort of organisations around cores so each aware of the modifications of the cache line.

How different cores keep track of the cache line modifications?

KDAB

→ We need some sort of organisations around cores so each aware of the modifications of the cache line.

→ Something called MESI came here in this scenario.

1. Modified
2. Exclusive
3. Shared
4. Invalid

A Simplified View of How MESI Works

Step	Cache Line A	Cache Line B
1	Exclusive	
2	Shared	Shared
3	Invalid	Modified
4	Shared	Shared

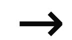
→ Imagine we have two cores A and B.

1. Core A fetches data into its cache. This data is now exclusively in Core A's cache, and the cache line status is marked as "Exclusive."
2. Subsequently, Core B accesses data from the identical memory region. This data is loaded into Core B's cache. Consequently, both cache lines (in Core A and Core B) transition to a "Shared" state to reflect that they hold identical data.
3. If Core B then modifies this shared data, its cache line status updates to "Modified." Concurrently, Core A's cache line that contains this data becomes "Invalid" to prevent outdated data usage.
4. When Core A needs to read the same data it originally fetched, it finds the data in its cache is invalidated due to Core B's modification. Core B must write its modified data back to the main memory, and Core A must reload the updated data from the main memory. Post this update, both cache lines are again in a "Shared" state, indicating they have the latest data from memory.

KDAB

- The last thing you just see, is "false sharing", It's called **false** because the cores aren't actually needing to share this data; they just happen to be located close enough in memory to end up on the same cache line and which have a huge performance hit.
- Thread A Cache Line   Thread B Cache Line

→ To be noted that false sharing does not create incorrect result but impact performance. (So it means that it prevents you from the incorrect results)



Factors affecting Multithreadead Performance

Factors affecting the performance of Multithreaded code

KDAB

→ **Number of Processors**: The performance of multithreaded applications can significantly differ based on whether the hardware has a single multicore processor or multiple processors with fewer cores. A program must spawn a number of threads that align with the available cores to fully utilize the hardware without leaving processing power unused.

Factors affecting the performance of Multithreaded code

→ **Number of Processors**: The performance of multithreaded applications can significantly differ based on whether the hardware has a single multicore processor or multiple processors with fewer cores. A program must spawn a number of threads that align with the available cores to fully utilize the hardware without leaving processing power unused.

Example: A server with 8 cores can optimally run 8 threads simultaneously. If your application creates 16 threads, it may not gain any additional performance benefit due to context switching overhead and could actually perform worse due to thread management overhead.

KDAB

The Qt, OpenGL and C++ experts

Factors affecting the performance of Multithreaded code

KDAB

→ **Number of Processors**: The performance of multithreaded applications can significantly differ based on whether the hardware has a single multicore processor or multiple processors with fewer cores. A program must spawn a number of threads that align with the available cores to fully utilize the hardware without leaving processing power unused.

Example: A server with 8 cores can optimally run 8 threads simultaneously. If your application creates 16 threads, it may not gain any additional performance benefit due to context switching overhead and could actually perform worse due to thread management overhead.

And you may want to use **std::thread::hardware_concurrency** in such cases to correctly identify the number of available concurrent threads, somethi

```
unsigned int n = std::thread::hardware_concurrency();
std::vector<std::thread> threads(n);

for (auto& t : threads) {
    t = std::thread(doWork);
}

for (auto& t : threads) {
    t.join();
}
```

KDAB

→ But using **std::thread::hardware_concurrency()** requires caution as it simply returns the number of hardware threads available on the system. It doesn't consider other running threads or applications, potentially leading to thread oversubscription and performance degradation.

- → But using **std::thread::hardware_concurrency()** requires caution as it simply returns the number of hardware threads available on the system. It doesn't consider other running threads or applications, potentially leading to thread oversubscription and performance degradation.
- → **std::async()** avoids the issue of oversubscription by being aware of all asynchronous calls within the application. It can schedule tasks more effectively without creating more threads than the system can handle efficiently.

Factors affecting the performance of Multithreaded code

→ **Data Contention and Cache Coherence Traffic**: When multiple threads try to read and write to the same data, they can interfere with each other, causing delays. Cache coherence mechanisms ensure that a change in one cache is reflected across all caches, which can cause traffic and slow down performance.

Factors affecting the performance of Multithreaded code

→ **Data Contention and Cache Coherence Traffic**: When multiple threads try to read and write to the same data, they can interfere with each other, causing delays. Cache coherence mechanisms ensure that a change in one cache is reflected across all caches, which can cause traffic and slow down performance.

Example:

```
std::atomic<int> sharedCounter{0};

void incrementCounter() {
    for (int i = 0; i < 1000000; ++i) {
        sharedCounter.fetch_add(1, std::memory_order_relaxed);
    }
}

int main() {
    std::thread t1(incrementCounter);
    std::thread t2(incrementCounter);

    t1.join();
    t2.join();

    return 0;
}
```

Factors affecting the performance of Multithreaded code

→ **Locality of Data**: Data that is accessed frequently should be kept close in memory to take advantage of cache locality. If data is spread out, more time is spent retrieving it, which can slow down the program.

Factors affecting the performance of Multithreaded code

→ **Locality of Data**: Data that is accessed frequently should be kept close in memory to take advantage of cache locality. If data is spread out, more time is spent retrieving it,

Example:

```
constexpr int dataSize = 100000;
std::vector<int> data(dataSize);

void processData(int start, int end) {
    for (int i = start; i < end; ++i) {
        // Perform some computations
        data[i] = data[i] * data[i];
    }
}

int main() {
    int chunkSize = dataSize / 4; // Assuming a 4-core processor
    std::thread workers[4];

    for (int i = 0; i < 4; ++i) {
        workers[i] = std::thread(processData, i * chunkSize, (i + 1) * chunkSize);
    }

    for (auto& worker : workers) {
        worker.join();
    }

    return 0;
}
```

KDAB
The Qt, OpenGL and C++ experts

Factors affecting the performance of Multithreaded code

→ **Excessive Context Switching**: If there are too many threads in relation to the number of processors, the operating system may spend a significant amount of time switching between threads (context switching), which can reduce the overall efficiency of the application.

Factors affecting the performance of Multithreaded code

→ **Excessive Context Switching**: If there are too many threads in relation to the number of processors, the operating system may spend a significant amount of time switching between threads (context switching), which can reduce the c

Example:

```
void performTask() {
    // Simulate a CPU-bound task
    for (int i = 0; i < 1000000; ++i) {}
}

int main() {
    std::vector<std::thread> threads;

    // Intentionally creating more threads than necessary
    for (int i = 0; i < 100; ++i) {
        threads.emplace_back(performTask);
    }

    for (auto& t : threads) {
        t.join();
    }

    return 0;
}
```

KDAB

The Qt, OpenGL and C++ experts

Factors affecting the performance of Multithreaded code

→ **False Sharing:** This occurs when threads on different processors modify variables that, while independent, are located on the same cache line. This can cause unnecessary invalidation and synchronization of caches, leading to performance degradation.

Factors affecting the performance of Multithreaded code

→ **False Sharing:** This occurs when threads on different processors modify variables that, while independent, are located on the same cache line. This can cause unnecessary invalidation and synchronization of caches, leading to performance degrad

Consider this example in

```
int sum1;
int sum2;

void thread1(int v[], int v_count) {
    sum1 = 0;
    for (int i = 0; i < v_count; i++)
        sum1 += v[i];
}

void thread2(int v[], int v_count) {
    sum2 = 0;
    for (int i = 0; i < v_count; i++)
        sum2 += v[i];
}
```

False Sharing

→ The functions **thread1** and **thread2** sum the values in the arrays they get as arguments to the variables **sum1** and **sum2**. Since **sum1** and **sum2** are defined next to each other, the compiler is likely to allocate them next to e[illegible] the same cache line.

```
int sum1;
int sum2;

void thread1(int v[], int v_count) {
    sum1 = 0;
    for (int i = 0; i < v_count; i++)
        sum1 += v[i];
}

void thread2(int v[], int v_count) {
    sum2 = 0;
    for (int i = 0; i < v_count; i++)
        sum2 += v[i];
}
```

False Sharing

→ First, **thread1** reads **sum1** into its cache. Since the line is not present in any other cache thread1 gets it in **exclusive** state:

```
int sum1;
int sum2;

void thread1(int v[], int v_count) {
    sum1 = 0;
    for (int i = 0; i < v_count; i++)
        sum1 += v[i];
}

void thread2(int v[], int v_count) {
    sum2 = 0;
    for (int i = 0; i < v_count; i++)
        sum2 += v[i];
}
```

Step	Thread1	Thread2
1	Exclusive	

False Sharing

KDAB

→ **thread2** now reads **sum2**. Since thread1 already had the cache line in **exclusive** state, this causes a downgrade of the line in thread1's cache and the line is now in shared state in both caches:

Step	Thread1	Thread2
1	Exclusive	
2	Shared	Shared

```
int sum1;
int sum2;

void thread1(int v[], int v_count) {
    sum1 = 0;
    for (int i = 0; i < v_count; i++)
        sum1 += v[i];
}

void thread2(int v[], int v_count) {
    sum2 = 0;
    for (int i = 0; i < v_count; i++)
        sum2 += v[i];
}
```

False Sharing

→ thread1 now writes its updated sum to sum1. Since it only has the line in shared state, it must upgrade the line and **invalidate** the line in **thread2's** cache:

Step	Thread1	Thread2
1	Exclusive	
2	Shared	Shared
3	Modified	Invalid

```
int sum1;
int sum2;

void thread1(int v[], int v_count) {
    sum1 = 0;
    for (int i = 0; i < v_count; i++)
        sum1 += v[i];
}

void thread2(int v[], int v_count) {
    sum2 = 0;
    for (int i = 0; i < v_count; i++)
        sum2 += v[i];
}
```

False Sharing

→ thread2 now writes its updated sum to sum2. Since thread1 has invalidate the cache line in it's cache it gets a coherence miss, and must invalidate the line in thread1's cache forcing thread1 to do a coheren

Step	Thread1	Thread2
1	Exclusive	
2	Shared	Shared
3	Modified	Invalid
4	Invalid	Modified

```
int sum1;
int sum2;

void thread1(int v[], int v_count) {
    sum1 = 0;
    for (int i = 0; i < v_count; i++)
        sum1 += v[i];
}

void thread2(int v[], int v_count) {
    sum2 = 0;
    for (int i = 0; i < v_count; i++)
        sum2 += v[i];
}
```

False Sharing

→ The next iteration of the loops now starts, and thread1 again reads sum1. Since thread2 just invalidated the cache line in thread1's cache, it gets a coherence miss. It must also downgrade the line in thread2's cache, forcing thread2 to do a coherence write-back:

Step	Thread1	Thread2
1	Exclusive	
2	Shared	Shared
3	Modified	Invalid
4	Invalid	Modified
5	Shared	Shared

```
int sum1;
int sum2;

void thread1(int v[], int v_count) {
    sum1 = 0;
    for (int i = 0; i < v_count; i++)
        sum1 += v[i];
}

void thread2(int v[], int v_count) {
    sum2 = 0;
    for (int i = 0; i < v_count; i++)
        sum2 += v[i];
}
```

False Sharing

KDAB

→ thread2 finally reads sum2. Since it has the cache line in shared state, it can read it without and coherence activity, and we are back in the same situation as after step 2:

Step	Thread1	Thread2
1	Exclusive	
2	Shared	Shared
3	Modified	Invalid
4	Invalid	Modified
5	Shared	Shared

```
int sum1;
int sum2;

void thread1(int v[], int v_count) {
    sum1 = 0;
    for (int i = 0; i < v_count; i++)
        sum1 += v[i];
}

void thread2(int v[], int v_count) {
    sum2 = 0;
    for (int i = 0; i < v_count; i++)
        sum2 += v[i];
}
```

False Sharing

KDAB

To be noted

False Sharing

KDAB

To be noted

→ Memory accesses may not interleave as described in earlier scenarios.

False Sharing

KDAB

- To be noted
- Memory accesses may not interleave as described in earlier scenarios.
- The same updates, coherence misses, and coherence write-backs would occur despite different interleaving.

False Sharing

KDAB

- To be noted
- Memory accesses may not interleave as described in earlier scenarios.
- The same updates, coherence misses, and coherence write-backs would occur despite different interleaving.
- In simple examples, the compiler might allocate sum1 and sum2 to registers, avoiding memory access and false sharing issues.

False Sharing

- To be noted
- Memory accesses may not interleave as described in earlier scenarios.
- The same updates, coherence misses, and coherence write-backs would occur despite different interleaving.
- In simple examples, the compiler might allocate sum1 and sum2 to registers, avoiding memory access and false sharing issues.
- For more complex programs, the compiler may not be able to keep sum1 and sum2 in registers, leading to potential false sharing.

False Sharing

KDAB

→ To fix a false sharing problem we need to make sure that the data accessed by the different threads is allocated to different cache lines.

False Sharing

KDAB

- To fix a false sharing problem we need to make sure that the data accessed by the different threads is allocated to different cache lines.
- So we can update our sum1 and sum2 variable like this in C++ .

```
alignas(64) int sum1_aligned; // Align sum1 to 64-byte boundary
alignas(64) int sum2_aligned; // Align sum2 to 64-byte boundary
```

False Sharing

KDAB

→ Or you can use standard way to align using **std::harware_destructive_interference_size**

False Sharing

KDAB

- Or you can use standard way to align using **std::harware_destructive_interference_size**
- It is typically provide the minimum offset between two objects to avoid **false sharing**.

False Sharing

KDAB

- → Or you can use standard way to align using **std::harware_destructive_interference_size**
- → It's value is typically 64 bytes (which is constant)
- → Example:

```
struct SharedData {
    alignas(hardware_destructive_interference_size) int data1;
    alignas(hardware_destructive_interference_size) int data2;
};
```

Causes of False Sharing

KDAB

Causes of False Sharing

KDAB

→ **Per-Thread Data Arrays**: Allocating an array where each element is used by a different thread, such as per-thread counters, can lead to false sharing due to proximity in memory.

Causes of False Sharing

KDAB

→ **Per-Thread Data Arrays**: Allocating an array where each element is used by a different thread, such as per-thread counters, can lead to false sharing due to proximity in memory.

Example: A typical programming pattern, for example consider this example

```
int sums[NUM_THREADS];

void threaded_sum(int thread_num, int v[], int v_count) {
    sum[thread_num] = 0;
    for (int i = 0; i < v_count; i++)
        sum[thread_num] += v[i];
}
```

Causes of False Sharing

→ **Matrix Parallelization Patterns**: Another common cause of false sharing is parallelizations of algorithms that work on matrices or multi-dimensional arrays.

Causes of False Sharing

- **Matrix Parallelization Patterns**: Another common cause of false sharing is parallelizations of algorithms that work on matrices or multi-dimensional arrays.
- Fine-grained division of matrices for multi-threading can lead to threads working on adjacent elements within the same cache line, increasing the chance of false sharing.

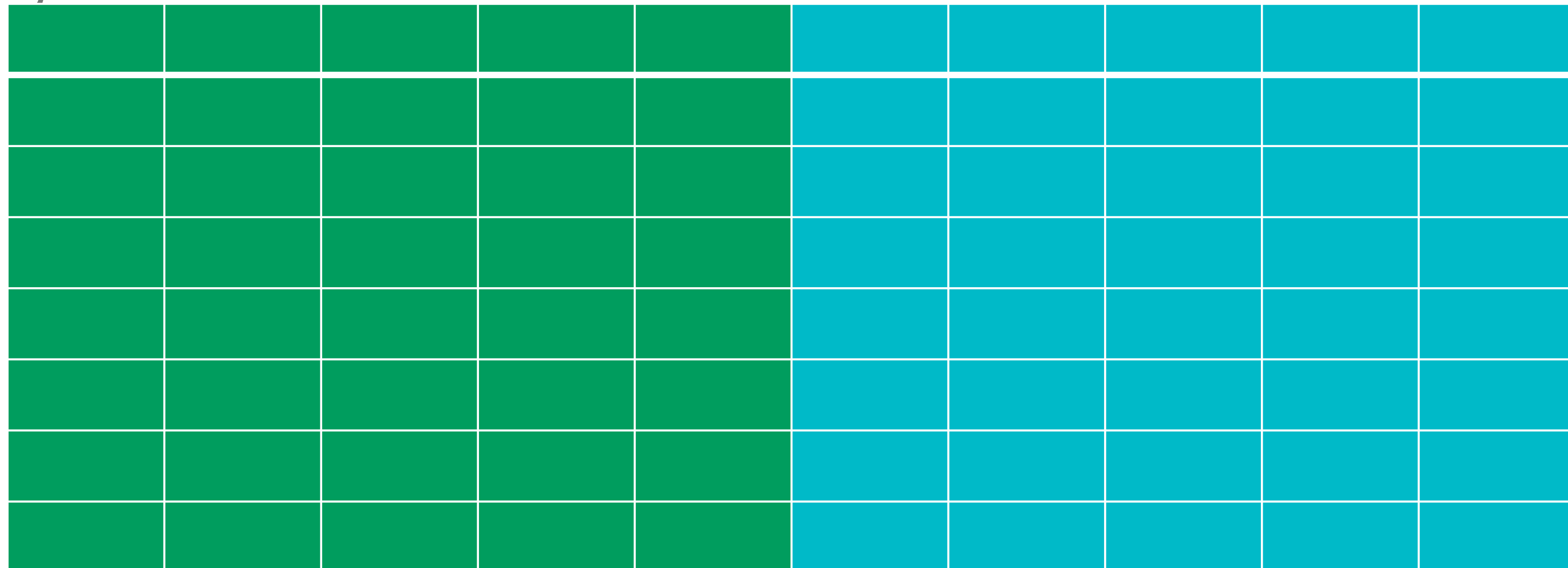
Causes of False Sharing

- **Matrix Parallelization Patterns**: Another common cause of false sharing is parallelizations of algorithms that work on matrices or multi-dimensional arrays.
- Fine-grained division of matrices for multi-threading can lead to threads working on adjacent elements within the same cache line, increasing the chance of false sharing.
- For example, assume the elements marked in green and blue in this example are written by two different threads:

Causes of False Sharing

KDAB

→ A more coarse grained division of the matrix between the threads will allow the threads to work on different cache lines to a greater degree, avoiding false sharing:



Causes of False Sharing

KDAB

→ **Struct Field Accesses**: Accesses to different fields in a structure from different threads.

Example:

```
struct SharedStruct {
    int fieldA;
    int fieldB;
    // Padding can be added here to prevent false sharing
    // char padding[60];
};
```

Causes of False Sharing

KDAB

→ **Dynamic Memory Allocation and False Sharing**: When a program dynamically allocates memory, particularly for small objects, it risks placing data used by concurrent threads within the same cache line.

Causes of False Sharing

→ **Dynamic Memory Allocation and False Sharing**: When a program dynamically allocates memory, particularly for small objects, it risks placing data used by concurrent threads within the same cache line.

→ One strategy to mitigate this is to allocate larger memory blocks for a thread's exclusive use. Instead of allocating each small object separately, which might scatter them across the same cache lines, allocating a single, larger array can localize a thread's data, reducing the chances of cache line collision.

Causes of False Sharing

KDAB

- **Dynamic Memory Allocation and False Sharing:** When a program dynamically allocates memory, particularly for small objects, it risks placing data used by concurrent threads within the same cache line.
- One strategy to mitigate this is to allocate larger memory blocks for a thread's exclusive use. Instead of allocating each small object separately, which might scatter them across the same cache lines, allocating a single, larger array can localize a thread's data, reducing the chances of cache line collision.
- And else, if it is possible, you can align the data objects causing false sharing.

Multithreading enabled standard library

KDAB

→ Many algorithms in standard library have there parallel versions.

Multithreading enabled standard library

KDAB

- → Many algorithms in standard library have there parallel versions.
- → Including **std::sort, std::find, std::replace, std:;count_if, std::for_each**

Multithreading enabled standard library

KDAB

- Many algorithms in standard library have there parallel versions.
- Including **std::sort, std::find, std::replace, std:;count_if, std::for_each, etc.**
- To use them you will call it in the same way, except for a new parameter **execution_policy.**

Multithreading enabled standard library

KDAB

- Many algorithms in standard library have there parallel versions.
- Including **std::sort, std::find, std::replace, std:;count_if, std::for_each, etc.**
- To use them you will call it in the same way, except for a new parameter **execution_policy.**
- For example, parallel sorting can be enabled something like this

```
std::sort(std::execution::par, v.begin(), v.end());
```

Final Summary

KDAB

- → Multithreading is indeed not easy.
- → Spawning thread is not cheap, and so synchronization.
- → The best synchronization is no synchronization. – Not my words
- → Use standard library that are trivial to parallelize for example **std::transform**, **std::reduce**.
- → Use tools like **C2C Linux Perf** to detect false sharing.
- → Believe in compiler.

→ Thanks for listening! Any Questions?

→ Keep in touch on twitter @phyBrackets.

Disclaimer: It’s much about science.