False Sharing and Contention in Parallel Codes

Paul Preney, OCT, M.Sc., B.Ed., B.Sc.
preney@sharcnet.ca / preney@uwindsor.ca

School of Computer Science / Office of Research and Innovation Services (ORIS)
University of Windsor, Windsor, Ontario, Canada

Jan. 17, 2024

Table of Contents

Resource Contention

Resource contention occurs when there is a need to access a **shared resource** involving multiple entities running concurrently.

Resource Contention (con't)

Restated, **resource contention** occurs when multiple processes/threads of execution attempt to use the **same shared resource**.

Resource Contention (con't)

If the same shared resource only has *one value* which exists and is only *read from* by all processes/threads of execution, then there won't be an issue.

Why? All processes/threads of execution will read the same value.

Resource Contention (con't)

Contention **problems** occur when the same shared resource's value is (or may be) updated during the time period processes / threads of execution may access that resource.

Which value will be read? Which value will get updated / written? etc. This is a big problem.

Resource Contention

Computing hardware (CPUs, GPUs, etc.), operating systems, programming languages and libraries all have constructs to deal with resource contention issues.

Resource contention is handled in programming languages and libraries using constructs such as: mutexes, locks/semaphores, atomics, fences/barriers, etc.

Doing *nothing* about resource contention is *not a solution* as code that performs reads and writes concurrently without properly dealing with contention is *incorrect* code and may/will produce incorrect values.

Resource Contention

- Suppose you are in a room with 3 other persons.
- Each person has a pencil and an eraser they can write and erase with.
- There is only one shared piece of paper capable of showing only one value written to it at a time.
  - If someone wants to update the value on the paper, one must first erase it and write a new value.
- There is no problem if everyone only needs to read what is on the shared piece of paper.
- But if *anyone* needs to update the value, then there is a problem.

Resource Contention (con't)

When a person needs to update the value:

- The person needs to acquire the shared piece of paper.
- Then erase the value written on the paper.
- Then write the updated value to the paper.
- Then release the paper back so everyone can read/write to it.

Ideally nobody can read/write to the paper while a person is updating it…hopefully!!

Resource Contention (con't)

Don't hope: we can and need to do much better!

Resource Contention (con't)

Similarly to this example, computer hardware, operating systems, and programming languages/libraries that deal with concurrency have constructs that address resource contention.

Modern computers have multiple "persons" (CPUs,GPUs) that can and do have shared resource contentions (e.g., updating the "shared piece of paper").

Table of Contents

False Sharing

But there is a kind of shared resource contention that is “hidden” from programmer’s: **false sharing**.

False Sharing (con't)

Imagine:

- you are in a room with 3 other persons
- each person (including yourself) has a pencil and an eraser
- there is one shared piece of paper in the room which you all use
- the shared piece of paper can have exactly 4 values on it
- each person updates only their own value on the paper and does not read/update any other person's value

Specifically notice this: No person is reading or updating any other person's value as each person is only reading/updating their own value.

False Sharing (con't)

But there is a problem:

- No person can read/update a value WHILE someone else is updating a their own value!

i.e., so everyone else **must wait** while someone is updating a value.

This is significant loss of concurrency as **everyone** else must wait to read/update their own value!!

This problem occurs with modern computing hardware that has caches and is called **false sharing**.

Table of Contents

Dealing With False Sharing

Dealing with false sharing in program code involves:

- recognizing when multiple values are close enough to one another in memory, and,
  - i.e., these values are effectively “on the same shared sheet of paper”
- locating those values further apart in memory.
  - i.e., using another sheet of paper so their is no sharing the same piece of paper to hold those values

Really? Yes (sorry!).

Dealing With False Sharing (con't)

To know whether or not values are effectively "on the same shared piece of paper":

- examine your code for values located close to one other in memory (that are being read and updated at the same time)

How close? One cache line (typically approx. 64 bytes of memory).

Cache line? What is a cache line?

- Computers read and write data to/from memory in terms of cache lines.
- A 32-bit computer will typically have a cache line size of 32-bits (4 bytes).
- A 64-bit computer will typically have a cache line size of 64-bits (8 bytes).
- etc.

Dealing With False Sharing (con't)

My computer has L1, L2, and L3 caches which are larger than 64 bytes. Does this false sharing problem matter, in practice, with those as well?

- In practice no! :-)

Dealing With False Sharing (con't)

What can I do in my C++11 or newer code to address such?

- use `std::hardware_destructive_interference_size` or `std::hardware_constructive_interference_size`
  - These values are effectively the cache line size.
  - In the worst case, if these are unavailable with your compiler than hard-code your cache line size (or use 64) when using **`alignas`** in your code.
- use **`alignas`** when declaring variables that need to be spaced further apart.

Dealing With False Sharing (con't)

Examples and questions.