

# CS610 Semester 2024–2025-I

## Assignment-2

Sandeep Nitharwal  
(210921)

September 7, 2024

### Problem.1

You are given a multithreaded program with  $N$  threads. The program reads  $N$  files, and should report the total number of words and lines processed by all the threads.

We provide a driver source code for the problem. Your task is to analyze the source code, and identify and report the performance bugs present in the source code, if any. If a performance bug is identified, provide a manually fixed version. Describe your modifications to the source code and report the performance gain.

Use the following commands to compile the attached driver and run the sample test case.

```
make
./problem1.out 4 ./test1/input
```

You can use the `perf c2c` tool to identify some forms of performance bugs. You can use the following links to learn more about using `perf`.

#### Solution

After analysing the code we can observe that it has both true and false sharing in the code.

#### False sharing

While incrementing the `tracker.word_count[threadsid]` we are hitting the same cache line for the threads. For resolving this we can introduce distance between thread ids, like I have made this array of size 40 (assuming each cache block of 64 byte and each word of 8 byte) so that each thread id line on different cache line.

This can also be observed by using `perf`. The below data is averaged over 10 reports. I have also modified the 5 input files each having approx 10 lakhs lines.

Type	HITM local	HITM Remote
Unmodified	9108	0
Modified	0	0

I have also tried by just removing the false sharing for `tracker.total_words_processed` but it gave around 3000 local HITM.

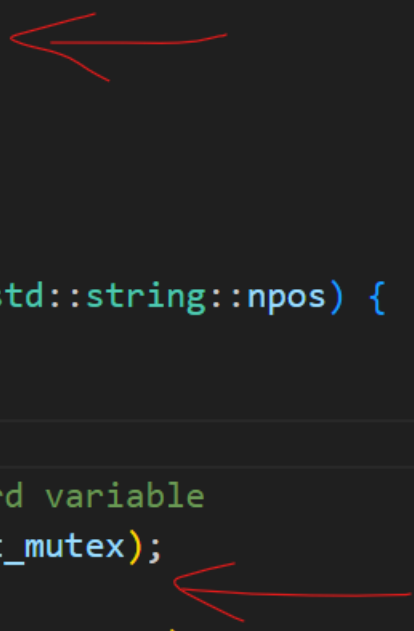
#### True sharing

In the below code, we can observe that two mutex lock have been taken again and again result in threads contenting for lock for the same location.

```

while (getline(input_file, line)) {
    pthread_mutex_lock(&line_count_mutex);
    tracker.total_lines_processed++;
    pthread_mutex_unlock(&line_count_mutex);
    std::string delimiter = " ";
    size_t pos = 0;
    std::string token;
    while ((pos = line.find(delimiter)) != std::string::npos) {
        token = line.substr(0, pos);
        // false sharing: on word count
        tracker.word_count[thread_id]++;
        // true sharing: on lock and total word variable
        pthread_mutex_lock(&tracker.word_count_mutex);
        tracker.total_words_processed++;
        pthread_mutex_unlock(&tracker.word_count_mutex);
        line.erase(0, pos + delimiter.length());
    }
}

```



This problem can be solved by introducing two local variable which can be incremented without taking lock as they are different for different threads, and later *tracker.total\_words\_processed* and *tracker.total\_lines\_processed* can be modified by using lock.

#### Improvement

Since the given test case contains very few lines, therefore I have modified the 5 input files each having approx 10 lakhs lines and also taking Maxthreads = 5.

This is done so that result are not biased.

The below data is averaged over 10 reports.

Type	Time(in s)
Unmodified	1.92
Semimodified	0.79
Modified	0.12

Through above data for file size (10 lakhs of line) we can observe that we get an approx 16 times speedup.

## Problem.2

Write a C++ program that takes five arguments from the command line: a string that represents the path to an input file to be read (say *R*), an integer representing the number of producer threads (say *T*), an integer representing the number of lines each thread should read from the file (say *L*), an integer representing the size of a shared buffer (say *M*), and a string that represents the path to an output file (say *W*).

### Solution

To run the code,

compile with **g++ problem2.cpp -o problem2.out**

the input format for argument follows below specification

`./problem2.out -inp=< input_path > -thr=T -lns=N -buf=M -out=< output_path`

where buffersize is  $> 0$ .

## Problem.3

Consider the following loop nest.

```
1  for i = 1, N-2
2  for j = i+1, N
3      A(i, j-i) = A(i, j-i-1) + A(i+1, j-i) + A(i-1, i+j-1)
```

List all flow, anti, and output dependences, if any, using the Delta test. Show your computation. Assume all array subscript references of array *A* are valid.

### Solution

#### **FLOW DEPENDENCE**

$$(1) A(i, j-i) = A(i, j-i-1)$$

$$i = i + \Delta i \Rightarrow \Delta i = 0$$

$$j-i = (j + \Delta j) - (i + \Delta i) - 1 \Rightarrow \Delta j - \Delta i = 1$$

From above equation we get,  $\Delta i = 0$ ,  $\Delta j = 1$ . Therefore the distance vector is (0,1) which is positive, Therefore it has **flow** dependency.

$$(2) A(i, j-i) = A(i+1, j-i)$$

$$i = i + \Delta i + 1 \Rightarrow \Delta i = -1$$

$$j-i = (j + \Delta j) - (i + \Delta i) \Rightarrow \Delta j - \Delta i = 0$$

From above equation we get,  $\Delta i = -1$ ,  $\Delta j = -1$ . Therefore the distance vector is (-1,-1) which is negative, Therefore it does **not has flow** dependency.

$$(3) A(i, j-i) = A(i-1, i+j-1)$$

$$i = i + \Delta i - 1 \Rightarrow \Delta i = 1$$

$$j-i = (j + \Delta j) + (i + \Delta i) - 1 \Rightarrow 2i + \Delta j + \Delta i = 1$$

From above equation we get,  $\Delta i = 1$ ,  $\Delta j = -2i$ . Therefore the distance vector is (1,-2i) which is positive, Therefore it has **flow** dependency.

**ANTI DEPENDENCE** For Anti dependency each of the distance vector are negative of flow dependency.

$$(1) A(i, j-i-1) = A(i, j-i)$$

The distance vector is (0,-1) which is negative, Therefore it does **not has anti** dependency.

$$(2) \ A(i+1, j-i) = A(i, j-i)$$

The distance vector is (1,1) which is positive, Therefore it has **anti** dependency.

$$(3) \ A(i-1, i+j-1) = A(i, j-i)$$

The distance vector is (-1,2i) which is negative, Therefore it does **not has anti** dependency.

### **OUTPUT DEPENDENCE**

There is **no** output dependence present in the above case as for it we require to write instructions writing at same location at same time but in our case only one instruction is present.