Program 1 - Chapter 1 Example Implementation

Author:

Name: Abdur Razzak
Aggie ID: 800810091
Email: arazzak@nmsu.edu

Hardware Information:

Linux
CPU

- Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz
- CPU Mhz: 3600
- Cache size: 56320 KB
- Cache alignment: 64

Memory:

- MemTotal: 65591100 kB
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 56320K

Task 1: Implement the algorithms in Figure 1.7

4 threads to count 3 on the shared variables named “count”.

Code:

```
#include <iostream>
#include <vector>
#include <thread>
#include <fstream>
using namespace std;

vector<int> array_3;
int t, length, count;

void count3s_thread(int thread_number) {
      int length_per_thread=length/t;
      int start=thread_number*length_per_thread;
      for(int i=start;i<start+length_per_thread; i++) {
          if(array_3[i]==3) {
```

```
                count++;
            }
        }
}
int main() {
        vector<thread> threads;
        array_3 = {2,3,0,2,3,3,1,0,0,1,3,2,2,3,1,0};
        t = 4;
        length = array_3.size();
        count = 0;
        cout<<"Length of Array: "<<length<<endl;
        for (int thread_number = 0; thread_number<t; thread_number++) {
            threads.emplace_back(thread(count3s_thread, thread_number));
        }
        for (thread& thread : threads) {
            thread.join();
        }

        cout<<"Total Count of 3: "<<count<<endl;
        return 0;
}
```

Output of the above Program:

```
(base) [arazzak@pearl04 ParallelProgramming]$ g++ -std=c++11 -pthread count3_1.cpp -o count3
(base) [arazzak@pearl04 ParallelProgramming]$ ./count3
Length of Array: 16
Total Count of 3: 5
(base) [arazzak@pearl04 ParallelProgramming]$ ./count3
Length of Array: 16
Total Count of 3: 5
(base) [arazzak@pearl04 ParallelProgramming]$ ./count3
Length of Array: 16
Total Count of 3: 5
```

Analysis:

As there are only 16 inputs on the array, each thread will get 4 subpart of the array to count 3. For example, thread1 will get first 4, thread 2 will get the next 4, and so on. However, the taskload for each processor is too small.

Incorrectness on 1.7 Algorithm:

As each thread will read the shared variable “count” and write after counting the 3’s. There will be a Race Condition occur as some thread will read before the previous thread write. As there will be an inconsistency in the counting.

However, as the array number is too small, we can not see the race condition here. To produce that, we need to use a big array.

Producing Race Condition:

To produce a big array, another cpp file named “random_number_generator.cpp” is used which will create a 16777216 (2^24) random numbers from 0 to 5 range and will write the output on a file so that we can use the same numbers in next runs to match the count number. The output numbers will be written on a file named “”

Code:

```cpp
#include <iostream>
#include <random>
#include <fstream>
using namespace std;

void generate_random_numbers(int sz, int lowest, int highest) {
	random_device rd;
	mt19937 gen(rd());
	uniform_int_distribution<int> distribution(lowest, highest);

	ofstream outputFile("integers.txt");
	for (int i = 0; i < sz; i++) {
	    outputFile << distribution(gen) << std::endl;
	}
	outputFile.close();
}

int main() {
	generate_random_numbers(16777216, 0, 5);
	return 0;
}
```

Running the file:

```
(base) [arazzak@pearl04 ParallelProgramming]$ g++ random_number_generator.cpp -o generate3
(base) [arazzak@pearl04 ParallelProgramming]$ ./generate3
(base) [arazzak@pearl04 ParallelProgramming]$
```

Change in the original Code:

In the main code, we will read the integers.txt file to load the array.

Updating in main method

```
read_generated_random_numbers();
// array_3 = {2,3,0,2,3,3,1,0,0,1,3,2,2,3,1,0};
```

And a new method will be introduced.

```
void read_generated_random_numbers() {
	ifstream inputFile("integers.txt");
	int number;
	while (inputFile >> number) {
	    array_3.push_back(number);
	}
	inputFile.close();
}
```

FINAL CODE:

```
#include <iostream>
#include <vector>
#include <thread>
#include <random>
#include <fstream>
#include <chrono>
#include <mutex>
using namespace std;

// struct padded_int {
// int value;
// char padding[60];
// } private_count[8];

vector<int> array_3;
int t, length, count;

// int private_count[8];
mutex m;

void count3s_thread(int thread_number) {
	auto currentTime_start = chrono::system_clock::now();
```

```
	auto millis_start =
chrono::duration_cast<chrono::milliseconds>(currentTime_start.time_since_epoch()).coun
t();

	int length_per_thread=length/t;
	int start=thread_number*length_per_thread;

	for(int i=start;i<start+length_per_thread; i++) {
	    if(array_3[i]==3) {
	    // m.lock();
	    count++;
	    // m.unlock();
	    // private_count[thread_number].value++;
	    }
	}
	// m.lock();
	// count+=private_count[thread_number].value;
	// m.unlock();

	auto currentTime_end = chrono::system_clock::now();
	auto millis_end =
chrono::duration_cast<chrono::milliseconds>(currentTime_end.time_since_epoch()).count(
);
	auto execTime = millis_end-millis_start;
	cout<<"Thread: "<<thread_number<<", execution time:"<<execTime<<endl;
	}
void read_generated_random_numbers() {
	ifstream inputFile("integers.txt");
	int number;
	while (inputFile >> number) {
	    array_3.push_back(number);
	}
	inputFile.close();
}

int main() {
	vector<thread> threads;
	read_generated_random_numbers();
	// array_3 = {2,3,0,2,3,3,1,0,0,1,3,2,2,3,1,0};

	t = 4;
	length = array_3.size();
```

```
        count = 0;
        cout<<"Length of Array: "<<length<<endl;
        for (int thread_number = 0; thread_number<t; thread_number++) {
            threads.emplace_back(thread(count3s_thread, thread_number));
        }

        for (thread& thread : threads) {
            thread.join();
        }
        cout<<"Total Count of 3: "<<count<<endl;
        return 0;
}
```

{Note: Please run the array generator file to create input.txt file to create numbers first and then run the main counting 3 file}

Output:

Now, if we run the main count 3 file again, the race condition can be found where count number is changed in each run.

```
(base) [arazzak@pearl04 ParallelProgramming]$ g++ -std=c++11 -pthread count3_1.cpp -o t4count3
(base) [arazzak@pearl04 ParallelProgramming]$ ./t4count3
Length of Array: 16777216
Thread: 0, execution time:138
Thread: 1, execution time:138
Thread: 2, execution time:139
Thread: 3, execution time:139
Total Count of 3: 932276
(base) [arazzak@pearl04 ParallelProgramming]$ ./t4count3
Length of Array: 16777216
Thread: 3, execution time:137
Thread: 2, execution time:139
Thread: 0, execution time:139
Thread: 1, execution time:140
Total Count of 3: 943296
(base) [arazzak@pearl04 ParallelProgramming]$
```

As we can see, different runs of the t4count3 output file produeces different count numbers which is an inconsistent result.

Task 2: Implement mutex in Figure 1.9

Process:

Took a shared variable.

```
mutex m;
```

And in the count3s_thread, added mutex before accessing shared variable count.

```
if(array_3[i]==3) {
        m.lock();
        count++;
        m.unlock();
}
```

Output:

After implementing the mutex, the total count of 3 is now consistent in different runs of the program which is 2796832.

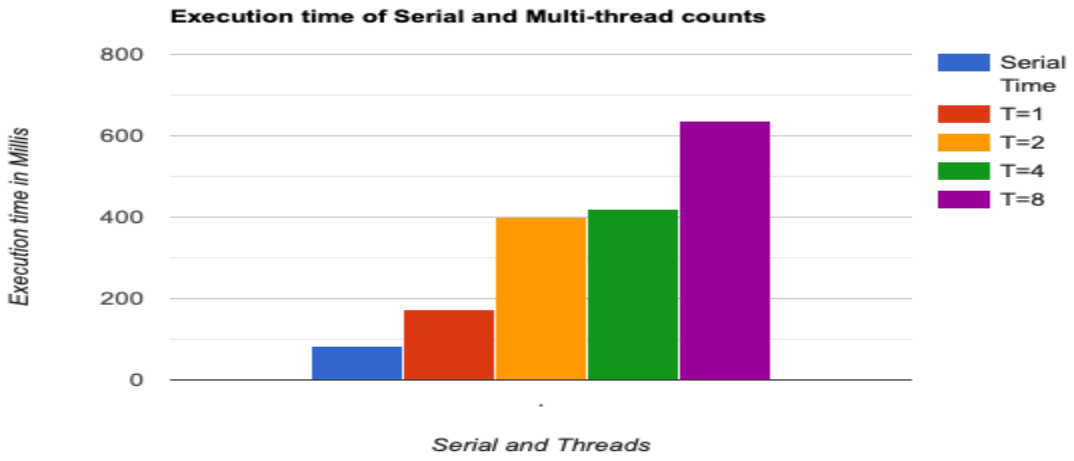
```
(base) [arazzak@pearl04 ParallelProgramming]$ g++ -std=c++11 -pthread count3_1.cpp -o t1count3
(base) [arazzak@pearl04 ParallelProgramming]$ ./t1count3
Length of Array: 16777216
Thread: 0, execution time:172
Total Count of 3: 2796832
(base) [arazzak@pearl04 ParallelProgramming]$ g++ -std=c++11 -pthread count3_1.cpp -o t2count3
(base) [arazzak@pearl04 ParallelProgramming]$ ./t2count3
Length of Array: 16777216
Thread: 1, execution time:395
Thread: 0, execution time:406
Total Count of 3: 2796832
(base) [arazzak@pearl04 ParallelProgramming]$ g++ -std=c++11 -pthread count3_1.cpp -o t4count3
(base) [arazzak@pearl04 ParallelProgramming]$ ./t4count3
Length of Array: 16777216
Thread: 1, execution time:412
Thread: 3, execution time:420
Thread: 2, execution time:424
Thread: 0, execution time:425
Total Count of 3: 2796832
(base) [arazzak@pearl04 ParallelProgramming]$ g++ -std=c++11 -pthread count3_1.cpp -o t8count3
(base) [arazzak@pearl04 ParallelProgramming]$ ./t8count3
Length of Array: 16777216
Thread: 7, execution time:629
Thread: 1, execution time:635
Thread: 0, execution time:635
Thread: 2, execution time:639
Thread: 3, execution time:639
Thread: 6, execution time:638
Thread: 5, execution time:640
Thread: 4, execution time:641
Total Count of 3: 2796832
```

Graphical View (like 1.10):

For Serial run, here is the output. And the multithread, the output is above (From the 3 runs, we will take the middle one to draw the comparison with Serial execution time)

```
(base) [arazzak@pearl04 ParallelProgramming]$ g++ serialCount.cpp -o serialCount
(base) [arazzak@pearl04 ParallelProgramming]$ ./serialCount
Length of Array: 16777216
Serial Calculation took: 84 milliseconds
Total Count of 3: 2796832
(base) [arazzak@pearl04 ParallelProgramming]$
```

Serial Execution: 84
T1 avg: 172, T2 avg: 400, T4 avg: 420, T8 avg: 637



Analysis:
As each thread is accessing the shared variable count each time of having a 3 and updating the count value is being synchronized, there is a lot of thread overhead time. As thread numbers increases, the mutex synchonized overhead is also increased.

Task 3: Implementing thread local count and add to shared count in the end Such as Figure 1.11

Process:
An array to keep the local calculated count.

```
int private_count[4];
mutex m;
```

Updating the shared variable “count” at the end.

```
for(int i=start;i<start+length_per_thread; i++) {
      if(array_3[i]==3) {
          // count++;
          private_count[thread_number]++;
      }
```

```
}
m.lock();
count+=private_count[thread_number];
m.unlock();
```

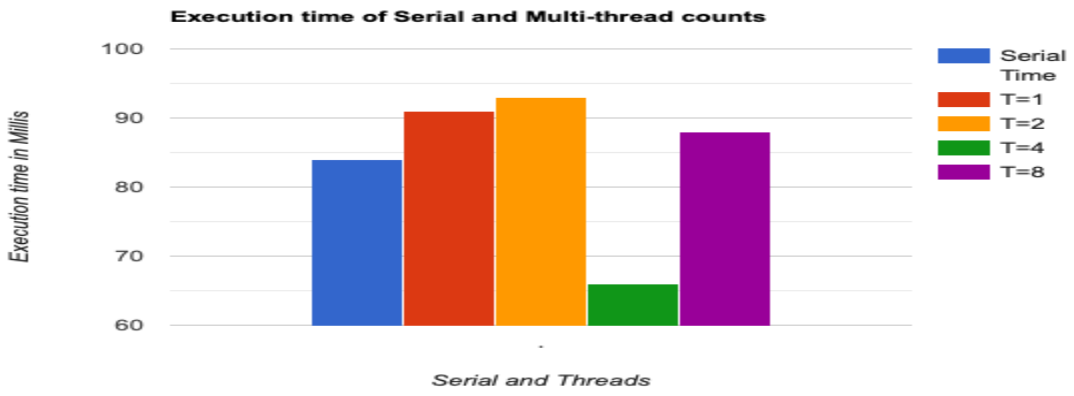
Output:

```
(base) [arazzak@pearl04 ParallelProgramming]$ g++ -std=c++11 -pthread count3_1.cpp -o t1count3
(base) [arazzak@pearl04 ParallelProgramming]$ ./t1count3
Length of Array: 16777216
Thread: 0, execution time:91
Total Count of 3: 2796832
(base) [arazzak@pearl04 ParallelProgramming]$ g++ -std=c++11 -pthread count3_1.cpp -o t2count3
(base) [arazzak@pearl04 ParallelProgramming]$ ./t2count3
Length of Array: 16777216
Thread: 1, execution time:93
Thread: 0, execution time:93
Total Count of 3: 2796832
(base) [arazzak@pearl04 ParallelProgramming]$ g++ -std=c++11 -pthread count3_1.cpp -o t4count3
(base) [arazzak@pearl04 ParallelProgramming]$ ./t4count3
Length of Array: 16777216
Thread: 3, execution time:64
Thread: 2, execution time:66
Thread: 0, execution time:67
Thread: 1, execution time:67
Total Count of 3: 2796832
(base) [arazzak@pearl04 ParallelProgramming]$ g++ -std=c++11 -pthread count3_1.cpp -o t8count3
(base) [arazzak@pearl04 ParallelProgramming]$ ./t8count3
Length of Array: 16777216
Thread: 1, execution time:84
Thread: 7, execution time:83
Thread: 2, execution time:84
Thread: 6, execution time:84
Thread: 4, execution time:84
Thread: 3, execution time:84
Thread: 0, execution time:84
Thread: 5, execution time:85
Total Count of 3: 2796832
(base) [arazzak@pearl04 ParallelProgramming]$
```

Graphical View (like 1.12):

Serial time: 84

T1 avg: 91, T2 avg: 93, T3 avg: 66, T4 avg: 84



Analysis:

As each thread is accessing the shared variable “count” just once at the end, the thread execution time is reduced now to nearly 90 on average. However, the threads are still taking more time than the serial execution. It is because of cache line granularity as each time 2 onchip processor continuously communicating each other and discard values in L1 onchip cache.

Task 4: Add padding to avoid cache line granularity and reduce onchip communication for L1 discarding Such as Figure 1.14

Process:

Added padding after each integer

```
struct padded_int {
        int value;
        char padding[60];
} private_count[4];
```

```
m.lock();
count+=private_count[thread_number].value;
m.unlock();
```

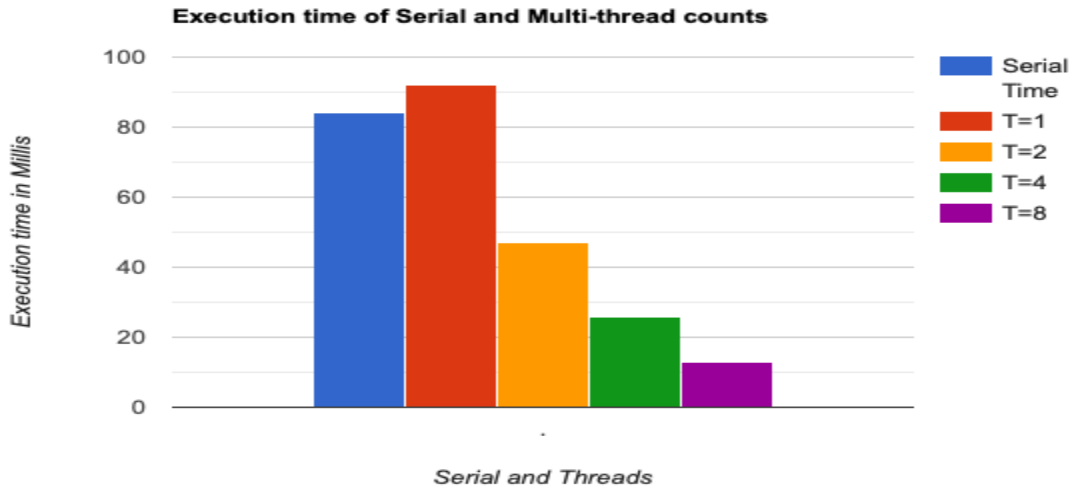
Output:

```
(base) [arazzak@pearl04 ParallelProgramming]$ g++ -std=c++11 -pthread count3_1.cpp -o t1count3
(base) [arazzak@pearl04 ParallelProgramming]$ ./t1count3
Length of Array: 16777216
Thread: 0, execution time:92
Total Count of 3: 2796832
(base) [arazzak@pearl04 ParallelProgramming]$ g++ -std=c++11 -pthread count3_1.cpp -o t2count3
(base) [arazzak@pearl04 ParallelProgramming]$ ./t2count3
Length of Array: 16777216
Thread: 1, execution time:47
Thread: 0, execution time:47
Total Count of 3: 2796832
(base) [arazzak@pearl04 ParallelProgramming]$ g++ -std=c++11 -pthread count3_1.cpp -o t4count3
(base) [arazzak@pearl04 ParallelProgramming]$ ./t4count3
Length of Array: 16777216
Thread: 3, execution time:26
Thread: 0, execution time:26
Thread: 1, execution time:26
Thread: 2, execution time:26
Total Count of 3: 2796832
(base) [arazzak@pearl04 ParallelProgramming]$ g++ -std=c++11 -pthread count3_1.cpp -o t8count3
(base) [arazzak@pearl04 ParallelProgramming]$ ./t8count3
Length of Array: 16777216
Thread: 7, execution time:13
Thread: 5, execution time:13
Thread: 4, execution time:13
Thread: 6, execution time:13
Thread: 1, execution time:13
Thread: 0, execution time:13
Thread: 2, execution time:13
Thread: 3, execution time:13
Total Count of 3: 2796832
(base) [arazzak@pearl04 ParallelProgramming]$
```

Graphical View (like 1.15):

Serial Execution: 84

T1 avg: 92, T2 avg: 47, T3 avg: 26, T4 avg: 13



Analysis:

As on-chip communication is reduced for L1 cache updating, a massive improvement can be found when the thread is increased.

Task 5: Hardware Memory Constrain Check like Figure 1.16

Running all the threads and serial code with the same size of array but this time there is no 3 in it.

Output:

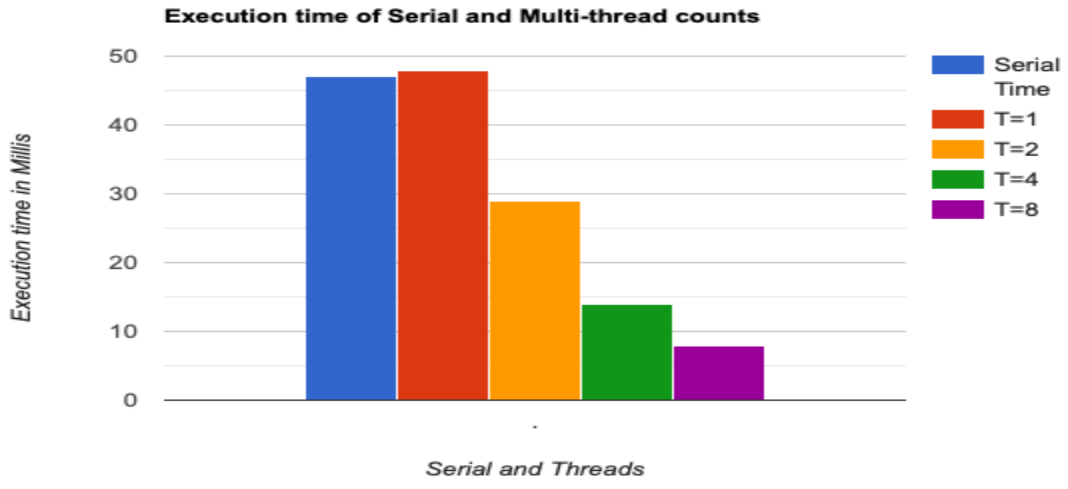
Serial code

```
(base) [arazzak@pearl04 ParallelProgramming]$ g++ serialCount.cpp -o serial_NO3s
(base) [arazzak@pearl04 ParallelProgramming]$ ./serial_NO3s
Length of Array: 16777216
Serial Calculation took: 47 milliseconds
Total Count of 3: 0
(base) [arazzak@pearl04 ParallelProgramming]$
```

Multi threaded run:

```
(base) [arazzak@pearl04 ParallelProgramming]$ g++ -std=c++11 -pthread count3_1.cpp -o t1_N03s_count3
(base) [arazzak@pearl04 ParallelProgramming]$ ./t1_N03s_count3
Length of Array: 16777216
Thread: 0, execution time:48
Total Count of 3: 0
(base) [arazzak@pearl04 ParallelProgramming]$ g++ -std=c++11 -pthread count3_1.cpp -o t2_N03s_count3
(base) [arazzak@pearl04 ParallelProgramming]$ ./t2_N03s_count3
Length of Array: 16777216
Thread: 1, execution time:29
Thread: 0, execution time:29
Total Count of 3: 0
(base) [arazzak@pearl04 ParallelProgramming]$ g++ -std=c++11 -pthread count3_1.cpp -o t4_N03s_count3
(base) [arazzak@pearl04 ParallelProgramming]$ ./t4_N03s_count3
Length of Array: 16777216
Thread: 3, execution time:14
Thread: 2, execution time:14
Thread: 1, execution time:15
Thread: 0, execution time:15
Total Count of 3: 0
(base) [arazzak@pearl04 ParallelProgramming]$ g++ -std=c++11 -pthread count3_1.cpp -o t8_N03s_count3
(base) [arazzak@pearl04 ParallelProgramming]$ ./t8_N03s_count3
Length of Array: 16777216
Thread: 7, execution time:7
Thread: 6, execution time:8
Thread: 5, execution time:7
Thread: 4, execution time:8
Thread: 0, execution time:8
Thread: 1, execution time:8
Thread: 3, execution time:8
Thread: 2, execution time:8
Total Count of 3: 0
(base) [arazzak@pearl04 ParallelProgramming]$
```

Serial time: 47
T1 avg: 48, T2 avg: 29, T4 avg: 14, T8 avg: 8



Aalysis:

Memory bandwidth limitations are preventing performance gains for eight processors.

Methodology:

This experiment was done on 1,2,4,8 processors of Intel(R) Xeon(R) CPU E5-2699 v4 which is running at 2.20GHz. The system has 3 caches.

L1: 8-way
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K, 8-way
L3 cache: 56320K or 55MB

The program operates on a big array where the size of the array is 16,777,216 where nerearly 16.67% of the data was 3's. To produce the array, a random number generator is used that all the values are from 0-9. Reported is the average program runs of twice. Command are used as
g++ -std=c++11 -pthread count3_1.cpp -o program
g++ serialCount.cpp -o serial_count

Conclusion

Producing correct and efficient parallel programs is often more challenging than creating correct and efficient serial programs. Mutexes highlight the importance of managing interactions among processors carefully, private counters emphasize the need to consider the granularity of parallelism, and padding underscores the significance of understanding machine details for optimizing performance. These factors collectively contribute to the complexity of parallel performance tuning.