Home Assignment 4

Yuxuan Jing

March 2020

Question 4.1

In the textbook, if m is evenly divisible by t, where m is the number of rows of the matrix, t is the total number of threads. Using p as the thread rank value, the formulas are:

$$\text{first component} = q \times \frac{m}{t}, \tag{1}$$

$$\text{last component} = (q+1) \times \frac{m}{t} - 1, \tag{2}$$

For this question, if m is not evenly divisible by t, the formulas are:

$$\text{block} = \frac{m}{t} \tag{3}$$

$$\text{remainder} = m\%t \tag{4}$$

```
if ( q < remainder ):
    first_component = q * (block + 1)
    last_component = ( q + 1 ) * (block + 1) - 1
else:
    first_component = q * block + remainder
    last_component = ( q + 1 ) * block + remainder - 1
```

Question 4.3

The original code:

```
int flag;
double sum;
```

Modified code:

```
int volatile flag;
double volatile sum;
```

Setting $thread\_count = 4$ and $n = 1.0e8$, the output is showed below.

- Without using volatile variables, with option -O2, I did not find any error.
- After declare $flag$ and sum as volatile variables, the performance with or without the optimization is almost the same as declare $flag$ and sum as non-volatile variables
- The -O2 optimization is more significant in single-threaded execution, nearly twice the speed, while in multi-threaded execution only increase about 50% speed.

No volatile with -O0

```
With n = 100000000 terms,
   Multi-threaded estimate of pi = 3.141592643589817
   Elapsed time = 8.784199e-02 seconds
   Single-threaded estimate of pi = 3.141592643589326
   Elapsed time = 2.213199e-01 seconds
   Math library estimate of pi = 3.141592653589793
```

No volatile with -O2

```
With n = 100000000 terms,
   Multi-threaded estimate of pi = 3.141592643589817
   Elapsed time = 4.796696e-02 seconds
   Single-threaded estimate of pi = 3.141592643589326
   Elapsed time = 9.585810e-02 seconds
   Math library estimate of pi = 3.141592653589793
```

Volatile with -O0

```
With n = 100000000 terms,
   Multi-threaded estimate of pi = 3.141592643589817
   Elapsed time = 7.682800e-02 seconds
   Single-threaded estimate of pi = 3.141592643589326
   Elapsed time = 2.137191e-01 seconds
   Math library estimate of pi = 3.141592653589793
```

Volatile with -O2

```
With n = 100000000 terms,
   Multi-threaded estimate of pi = 3.141592643589817
```

```
Elapsed time = 5.872416e-02 seconds
Single-threaded estimate of pi = 3.141592643589326
Elapsed time = 1.001680e-01 seconds
Math library estimate of pi = 3.141592653589793
```

Question 4.8

4.8.a

If the sequence of events in the question occurs, the program will fall into a DEAD LOCK. Since After time 0, $mut0$ and $mut1$ are both locked. In time 1, thread 0 is waiting for $mut1$ to be unlocked and thread 1 is waiting for $mut0$ to be unlocked. While there are no unlock of $mut0$ or $mut1$ between time 0 and time 1, the process will stuck here forever.

4.8.b

This will not happen on busy-waiting. There is no lock and unlock in busy-waiting. In busy-waiting, each thread executes in order, so that only after thread 0 finish all its work, thread 1 can start to read the memory.

4.8.c

Whether this will happen on semaphores depends. With a carefully management, dead lock can be avoid in semaphores. If we only have two semaphores on two memory address, $mut0$ and $mut1$, dead lock will happen. If we have two semaphores array on both memory address and thread numbers, dead lock can be avoided.

Question 4.11

4.11.a

Two deletes executed simultaneously

```
Thread 0 = delete A
Thread 1 = delete B
```

- If thread 0 and 1 are going to delete the same value and they are executed synchronous. When two thread reach the same node, after thread 0 free the node, thread 1 cannot free the node again.
- If thread 0 and 1 are going to delete different value ($B > A$). Thread 0 reach to its target node before thread 1, then thread 0 delete the node but have not link the previous node to the next node. At this time, if thread

1 reach to the previous node, thread 1 will believe it is the end node of this array.

4.11.b

An insert and a delete executed simultaneously

```
Thread 0 = insert
Thread 1 = delete
```

- If thread 0 and 1 are going to work on the same value. Thread 1 may pass the value before thread 0 insert it.
- If thread 0 and 1 are going to work on different value ($B > A$). Thread 0 reach to its target node before thread 1, then thread 0 insert the node but have not link the previous node to the next node. At this time, if thread 1 reach to this node and delete it. Thread 0 will not find the next_pointer and thread 1 will treat this node as the end node of the array.
- If thread 0 and 1 are going to work on different value ($B > A$). Thread 0 and 1 reach to its target at the same time, then thread 1 delete the node but have not link the previous node to the next node. At this time, thread 0 will not find the next_pointer of this node and return an crash.

4.11.c

A member and a delete executed simultaneously

```
Thread 0 = member
Thread 1 = delete
```

- Thread 1 delete the member before thread 0 find the member.
- Thread 0 and 1 reach to the same node, then thread 1 free the node. Thread 0 will not find the object for further operations.

4.11.d

Two inserts executed simultaneously

```
Thread 0 = insert A
Thread 1 = insert B
```

- If thread 0 and 1 are going to insert the same value and they are executed synchronous. Two new node may created. Problem will happen if new_node_1 connect to the curr_node and new_node_2 connect to the next_node.

- If thread 0 and 1 are going to insert different value ($B > A$). Thread 0 and 1 reach to the same node, then thread 0 insert a node but have not link to the next node. Thread 1 will jump to the inserted node and assume it is the end node of this array.

4.11.e

An insert and a member executed simultaneously

```
Thread 0 = insert
Thread 1 = member
```

- If the value to insert and member is the same, thread 1 may miss the member while thread 0 insert it later.
- Thread 0 and 1 reach to the same node, then thread 0 insert a node but have not link to the next node. Thread 1 will jump to the inserted node and assume it is the end node of this array.

Question 4.12

For the first phase (reading phase), read-lock is unnecessary and safe. As no writing will happen in the first phase, it is safe to lock the list using read-locks for the first phase.

For the second phase (writing phase), write-lock is necessary and safe. As multiple writing will happen in the second phase, it is safe to lock the list using write-locks for the first phase.

As both *insert* and *write* will cause a problem, without write-locks, that:

```
Thread 0 = insert
Thread 1 = delete
```

- If thread 0 and 1 are going to work on different value ($B > A$). Thread 0 reach to its target node before thread 1, then thread 0 insert the node but have not link the previous node to the next node. At this time, if thread 1 reach to this node and delete it. Thread 0 will not find the next_pointer and thread 1 will treat this node as the end node of the array.
- If thread 0 and 1 are going to work on different value ($B > A$). Thread 0 and 1 reach to its target at the same time, then thread 1 delete the node but have not link the previous node to the next node. At this time, thread 0 will not find the next_pointer of this node and return an crash.

Question 4.13

In this question, I test on both the file pth_ll_one_mut.c and pth_ll_mult_mut.c, with 4 threads. For both code I have the following parameters:

```
Initial keys = 100000
Number of searches = 4000
Number of insert or delete = 1000
\* the reason to set Initial keys much larger than the number of
    operations is to minimize the influence of the array length
    changing in the whole process. *\
```

The conclusion is as following:

- There is a difference in the overall run-times, and insert is more expensive than delete
- The difference between *insert* and *delete* is more significant in pth_ll_one_mut.c than pth_ll_mult_mut.c.
- Some bias may happen in this question since all-delete keep shorten the array and all-insert keep elongate the array. But since I set the initial length much more larger than the number of operations, I believe this bias is trivial.

pth_ll_one_mut.c

```
How many keys should be inserted in the main thread?
100000
How many total ops should be executed?
5000
Percent of ops that should be searches? (between 0 and 1)
0.8
Percent of ops that should be inserts? (between 0 and 1)
0.2
Inserted 100000 keys in empty list
Elapsed time = 4.197669e+00 seconds
Total ops = 5000
member ops = 3995
insert ops = 1005
delete ops = 0
```

```
How many keys should be inserted in the main thread?
100000
How many total ops should be executed?
5000
Percent of ops that should be searches? (between 0 and 1)
0.8
```

```
Percent of ops that should be inserts? (between 0 and 1)
0
Inserted 100000 keys in empty list
Elapsed time = 3.110941e+00 seconds
Total ops = 5000
member ops = 3995
insert ops = 0
delete ops = 1005
```

pth_ll_mult_mut.c

```
How many keys should be inserted in the main thread?
100000
How many total ops should the threads execute?
5000
Percent of ops that should be searches? (between 0 and 1)
0.8
Percent of ops that should be inserts? (between 0 and 1)
0.2
Inserted 100000 keys in empty list
Elapsed time = 2.844780e+00 seconds
Total ops = 5000
member ops = 3995
insert ops = 1005
delete ops = 0
```

```
How many keys should be inserted in the main thread?
100000
How many total ops should the threads execute?
5000
Percent of ops that should be searches? (between 0 and 1)
0.8
Percent of ops that should be inserts? (between 0 and 1)
0
Inserted 100000 keys in empty list
Elapsed time = 2.785215e+00 seconds
Total ops = 5000
member ops = 3995
insert ops = 0
delete ops = 1005
```

Question 4.16

If we assign the threads exactly the same way as in the textbook. It is not possible for false sharing happen between thread 0 and 2 or thread 0 and 3.

Since the length of a cache line is 8 doubles, and the gap between thread 0 and thread 2 is 2000 elements of y. The gap between thread 0 and thread 3 is 4000 elements of y.

Question 4.17

4.17.a

The minimum number of cache lines that are needed to store the vector y is 1. (For the dual-core system is 2 cache lines, 1 cache line for each core).

4.17.b

The maximum number of cache lines that are needed to store the vector y is 2. (For the dual-core system is 4 cache lines, 2 cache lines for each core).

4.17.c

If the boundaries of cache lines always coincide with the boundaries of 8-byte doubles, there are 8 ways to assign y to cache lines.

```
|1|2|3|4|5|6|7|8|
|x|1|2|3|4|5|6|7| -- |8|x|x|x|x|x|x|x|
|x|x|1|2|3|4|5|6| -- |7|8|x|x|x|x|x|x|
|x|x|x|1|2|3|4|5| -- |6|7|8|x|x|x|x|x|
|x|x|x|x|1|2|3|4| -- |5|6|7|8|x|x|x|x|
|x|x|x|x|x|1|2|3| -- |4|5|6|7|8|x|x|x|
|x|x|x|x|x|x|1|2| -- |3|4|5|6|7|8|x|x|
|x|x|x|x|x|x|x|1| -- |2|3|4|5|6|7|8|x|
```

4.17.d

There are three ways:

```
|1|2| + |3|4|
|1|3| + |2|4|
|1|4| + |2|3|
```

4.17.e

Yes. assignments of components to cache lines:

```
|x|x|x|x|1|2|3|4| -- |5|6|7|8|x|x|x|x|
```

threads to processors:

```
Thread 0 --> processor 0 --> core 0
Thread 1 --> processor 1 --> core 0
Thread 2 --> processor 2 --> core 1
Thread 3 --> processor 3 --> core 1
```

Threads' job on y:

```
Thread 0 --> y[0] and y[1]
Thread 1 --> y[2] and y[3]
Thread 2 --> y[4] and y[5]
Thread 3 --> y[6] and y[7]
```

4.17.f

8 different components to cache lines. 3 different threads to processors.

4.17.g

Only one assignment (as the answer of 4.17.e) will result in no false sharing.