Performance profile of synchronization constructs

21.02.2019

—

Kuldeep Singh Bhandari
111601009

CSE, 3rd Year Btech

IIT PALAKKAD

Introduction

In shared-memory programming, the race-condition is a prominent issue. A **race condition** is an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence in order to be done correctly. In shared-memory programming, the race condition occurs when two or more threads try to access particular piece of code (known as **critical section**) which is shared among all threads. To handle this, we often use different tools like busy-waiting, mutexes, semaphores etc. But the question arises which method is better than others and when? To answer this, we need to analyze different scenarios and decide accordingly which method will be optimal and when.

Specification

All the experiments discussed in this report are done in the machine with following specifications :

- **Processor :** Intel® Core™ i5-7200U CPU @ 2.50GHz × 4 (Quad-core)
- **Graphics :** Intel® HD Graphics 620 (Kaby Lake GT2)
- **Memory :** 7.7 GiB
- **OS-type :** 64-bit

NOTE : Unless specified, the measurement is the average value which is obtained by performing the experiment five times.

Busy-waiting and Mutexes

Busy-waiting, **busy-looping** or **spinning** is a technique in which a process repeatedly checks to see if a condition is true. On the other hand, a **mutex** (mutual exclusion object) is a program object that is created so that multiple program thread can take turns sharing the same resource.

To analyze performance of busy-waiting and mutexes, let us take the case of **“finding sum of elements of a given array”**. The performance measures of a parallel program are speed-up, efficiency, etc. These measures need the time taken by serial program which is a standard for all the parallel programs implementing the same task as serial program.

Given an vector<long> arr with n = 1e8.

```
long sum = 0;
for(int i = 0; i < n; i++) {
       sum += arr[i];
}
```

Times for the serial program

	TIME (in s)
Iteration 1	0.315391
Iteration 2	0.278593
Iteration 3	0.278541
Iteration 4	0.312446
Iteration 5	0.279567
MAX	0.315391
MIN	0.278541
AVG	0.292908

$$T_{serial} = 0.292908s$$

Busy-waiting :

Consider the case of busy-waiting. First of all, we will be creating <p> threads which will be running <partial_sum> function. Each thread will have its own rank provided in the parameter of the function. Depending upon the rank, each thread will be computing sum of its own set of elements ranging from <start> to <end>. Each thread is given (n/p) elements to find sum of, where <n> is the number of elements in the array and <p> is the number of threads.

We are wishing to find the time that has elapsed from when the first process/thread began execution of the code to the time the last process/thread finished execution of the code. But doing this is a tough job since there may not be any correspondence between the clocks on one node and the clock on the other. So, we use **barrier** function to synchronize all the threads to reach at a single point of time. So, when all the threads are at a single point, we will measure the execution time of each thread and note the time which is taken by the slowest thread which will be our desired time. We need to take average of the time obtained by slowest threads in multiple executions by running the program 5-6 times to make sure the measured time is not influenced by operating system scheduling or some other reason.

```
void *partial_sum (void *rank) {

   long my_rank = (long) rank;
   long start = my_rank * RANGE;
   long end   = (my_rank + 1)*RANGE - 1;
   long my_sum = 0;
   Barrier();
   auto start_time = chrono :: system_clock :: now();
   for (int i= start; i <= end; i++) {
       my_sum += arr[i];
   }
   while (flag != my_rank);
   sum += my_sum;
   flag = (flag + 1) % p;
   auto end_time = chrono :: system_clock :: now();

   chrono::duration<double> time_taken = end_time - start_time;
   slowest_thread_time = max (slowest_thread_time, (double) time_taken.count());
   return NULL;
}
```

Performance Measure of Busy-wait

Times for the busy-loop program (n = 1e8, p = 1)

	TIME (in s)
Iteration 1	0.296443
Iteration 2	0.312927
Iteration 3	0.297946
Iteration 4	0.299138
Iteration 5	0.299902
MAX	0.312927
MIN	0.296443
AVG	0.301271

Times for the busy-loop program (n = 1e8, p = 2)

	TIME (in s)
Iteration 1	0.160594
Iteration 2	0.157044
Iteration 3	0.192596
Iteration 4	0.157886
Iteration 5	0.156882
MAX	0.192596
MIN	0.156882
AVG	0.165

Times for the busy-loop program (n = 1e8, p = 4)

	TIME (in s)
Iteration 1	0.145499
Iteration 2	0.147274
Iteration 3	0.152561
Iteration 4	0.178185
Iteration 5	0.170722
MAX	0.178185
MIN	0.145499
AVG	0.158848

Times for the busy-loop program (n = 1e8, p = 8)

	TIME (in s)
Iteration 1	0.151728
Iteration 2	0.193836
Iteration 3	0.21199
Iteration 4	0.196208
Iteration 5	0.230925
MAX	0.230925
MIN	0.151728
AVG	0.196937

Times for the busy-loop program (n = 1e8, p = 16)

	TIME (in s)
Iteration 1	0.2302
Iteration 2	0.218886
Iteration 3	0.220313
Iteration 4	0.225063
Iteration 5	0.286209
MAX	0.286209
MIN	0.218886
AVG	0.236134

Times for the busy-loop program (n = 1e8, p =32)

	TIME (in s)
Iteration 1	0.348108
Iteration 2	0.454301
Iteration 3	0.384624
Iteration 4	0.397766
Iteration 5	0.503412
MAX	0.503412
MIN	0.348108
AVG	0.417642

Speedups and Efficiencies using busy-loop

p	1	2	4	8	16	32
S	0.97	1.77	1.84	1.48	1.24	0.70
E=S/p	0.97	0.88	0.46	0.18	0.07	0.02

p = number of threads, S = speedups, E = efficiency

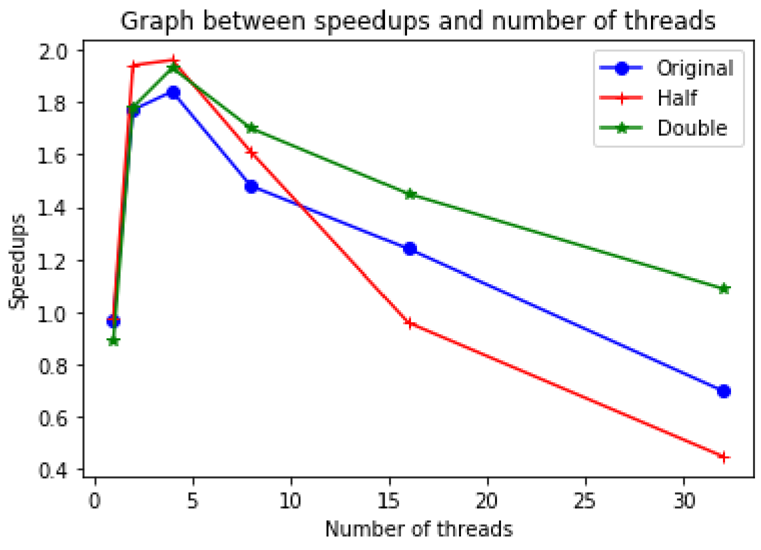
$$S = \frac{T_{serial}}{T_{parallel}} \qquad E = \frac{S}{p}$$

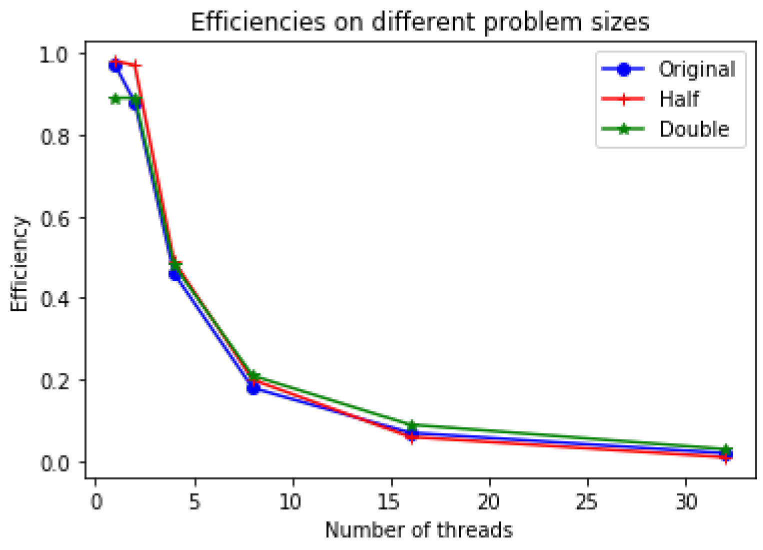
Here, $T_{parallel}$ is the average value of the iterations for each case.

Speedups and Efficiencies using mutexes for different problem sizes

We can see that performance of busy-loop is degrading with increase in number of threads and with increase in problem size also, no significant difference is observed.

	p	1	2	4	8	16	32
Half	S	0.98	1.94	1.96	1.61	0.96	0.45
	E	0.98	0.97	0.49	0.20	0.06	0.01
Original	S	0.97	1.77	1.84	1.48	1.24	0.70
	E	0.97	0.88	0.46	0.18	0.07	0.02
Double	S	0.89	1.78	1.93	1.70	1.45	1.09
	E	0.89	0.89	0.48	0.21	0.09	0.03





We can see that as number of threads are increasing, the speedup is initially increasing then keeps on decreasing after number of threads become high enough. Also, the efficiency is continuously decreasing as number of threads are increasing.

This is happening because busy-loop enforces ordering of execution of threads in critical section, i.e. the thread with rank 0 comes into critical section first, then the thread with rank 1, and it goes on until the last thread comes into the critical section. Thread 1 has to wait for thread 0 to finish even if thread 1 has finished the task before thread 0. Similarly, thread 2 has to wait for thread 1, and so on. The last thread has to wait for its previous thread to finish off. So, as we increase the number of threads, the waiting would increase hence $$T_{overhead}$$ will increase, which will subsequently increase $$T_{parallel}$$. Hence, speedup will increase initially and then decrease and efficiency keeps on decreasing continuously.

Mutex :

Now, let us see how we can use mutex instead of busy-wait. Pthread provides a library for using mutex. Mutexes don’t waste CPU cycle like busy-waiting, rather the thread is blocked until it is unlocked by the thread which has locked it. Also, the mutexes do not enforce ordering so the thread which has finished the execution first can lock the mutex irrespective of its rank which is a significant advantage of using mutexes. But locking and unlocking mutexes causes extra overhead which slows it down a little bit. There is only minute change to be made in the code of busy-loop for mutexes.

```
void *partial_sum (void *rank) {

   long my_rank = (long) rank;
   long start = my_rank * RANGE;
   long end   = (my_rank + 1)*RANGE - 1;

   long my_sum = 0;

   Barrier();
   auto start_time = chrono :: system_clock :: now();
   for (int i= start; i <= end; i++) {
       my_sum += arr[i];
   }
   pthread_mutex_lock (&sum_mutex);
   sum += my_sum;
   pthread_mutex_unlock (&sum_mutex);
   auto end_time = chrono :: system_clock :: now();

   chrono::duration<double> time_taken = end_time - start_time;
   slowest_thread_time = max (slowest_thread_time, (double) time_taken.count());

   return NULL;
}
```

Performance Measure of Mutexes

Times for the mutex program (n = 1e8, p = 1)

	TIME (in s)
Iteration 1	0.299654
Iteration 2	0.306255
Iteration 3	0.307017
Iteration 4	0.333684
Iteration 5	0.307172
MAX	0.333684
MIN	0.299654
AVG	0.310756

Times for the mutex program (n = 1e8, p = 2)

	TIME (in s)
Iteration 1	0.159438
Iteration 2	0.151491
Iteration 3	0.158869
Iteration 4	0.158816
Iteration 5	0.152748
MAX	0.159438
MIN	0.151491
AVG	0.156272

Times for the mutex program (n = 1e8, p = 4)

	TIME (in s)
Iteration 1	0.15013
Iteration 2	0.148954
Iteration 3	0.149251
Iteration 4	0.147987
Iteration 5	0.150422
MAX	0.150422
MIN	0.147987
AVG	0.149349

Times for the mutex program (n = 1e8, p = 8)

	TIME (in s)
Iteration 1	0.153047
Iteration 2	0.161201
Iteration 3	0.152544
Iteration 4	0.150654
Iteration 5	0.172293
MAX	0.172293
MIN	0.150654
AVG	0.157948

Times for the mutex program (n = 1e8, p = 16)

	TIME (in s)
Iteration 1	0.145205
Iteration 2	0.144739
Iteration 3	0.150458
Iteration 4	0.151347
Iteration 5	0.138757
MAX	0.151347
MIN	0.138757
AVG	0.146101

Times for the mutex program (n = 1e8, p =32)

	TIME (in s)
Iteration 1	0.136685
Iteration 2	0.145139
Iteration 3	0.13715
Iteration 4	0.145433
Iteration 5	0.113622
MAX	0.145433
MIN	0.113622
AVG	0.135606

Speedups and Efficiencies using mutexes

p	1	2	4	8	16	32
S	0.94	1.87	1.96	1.85	2.00	2.15
E=S/p	0.94	0.93	0.49	0.23	0.12	0.06

p = number of threads, S = speedups, E = efficiency

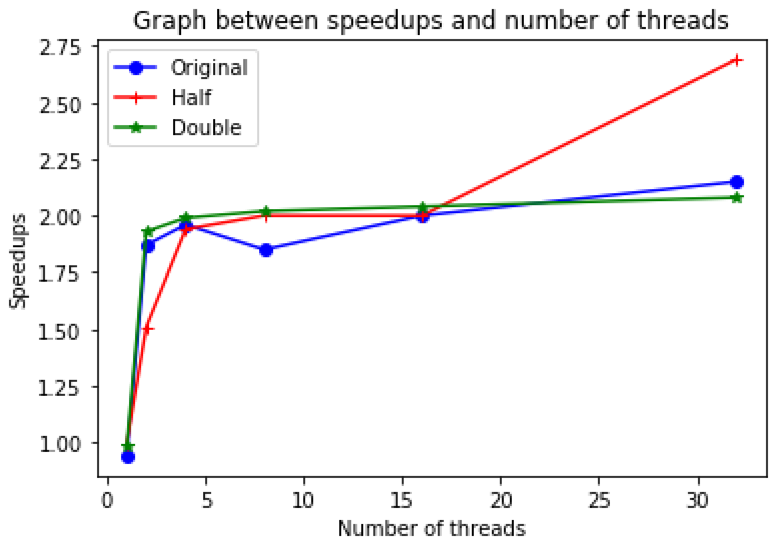
$$S = \frac{T_{serial}}{T_{parallel}} \qquad E = \frac{S}{p}$$

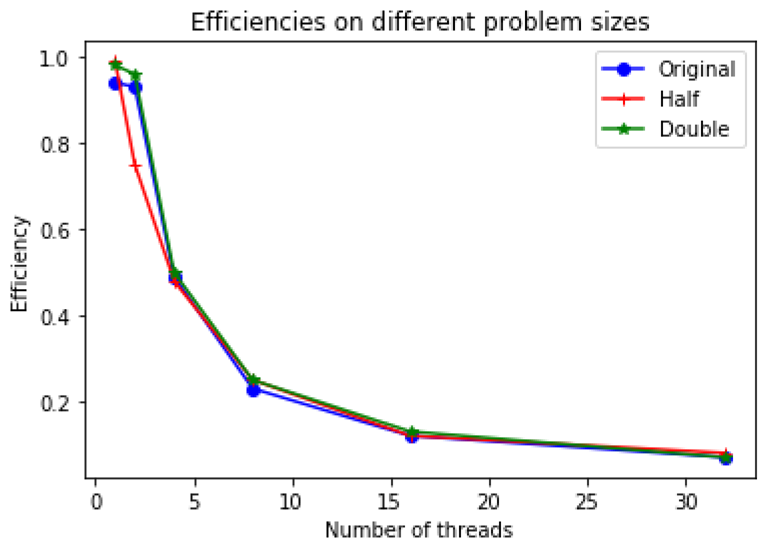
Here, $T_{parallel}$ is the average value of the iterations for each case.

Speedups and Efficiencies using mutexes for different problem sizes

Generally, efficiency increases with the increase in the problem sizes but in this case, not much change is observed.

	p	1	2	4	8	16	32
Half	S	0.99	1.51	1.94	2.00	2.00	2.69
	E	0.99	0.75	0.48	0.25	0.12	0.08
Original	S	0.94	1.87	1.96	1.85	2.00	2.15
	E	0.94	0.93	0.49	0.23	0.12	0.07
Double	S	0.98	1.93	1.99	2.02	2.04	2.08
	E	0.98	0.96	0.50	0.25	0.13	0.07





Comparison between mutex and busy-wait

p	Busy-wait (in s)	Mutex (in s)
1	0.301271	0.310756
2	0.165	0.156272
4	0.158848	0.149349
8	0.196937	0.157948
16	0.236134	0.146101
32	0.417642	0.135606

Here, p is the number of threads, Busy-wait is the average time taken by the slowest thread in busy-wait implementation and Mutex is the average time taken by the slowest thread in mutex implementation.

We can see that with small number of threads, busy-wait is working fine but with the increase in number of threads, the performance has been degraded because busy-wait enforces ordering and hence now more number of threads has to wait to its predecessor thread. On the other hand, with small threads mutexes may not work that efficiently because of the overhead in calling mutex_lock and mutex_unlock but as number of threads increases, the increase in this overhead is much slower than the decrease in the total time taken.

Barrier

Barrier is used for synchronizing the threads by making sure that they are all at the same point in a program. No thread can proceed beyond the barrier until all the threads have reached it. We have already used barrier during comparison of mutexes and busy-waiting. This is a very important application of barrier. We can put a barrier in a code so that all threads come to the same point then measure the time taken by each thread to compute some part of the code which we want to measure then find the thread which executes it the last which will be the total execution time.

To implement barrier, we will consider three techniques :

1. Busy-waiting and mutex :

```
long barrier_counter = 0;       // initialize to 0 by main
// barrier_mutex is also declared by the main
// p is the number of threads

void barrier_using_busy_wait_mutex () {

   // Barrier
   pthread_mutex_lock (&barrier_mutex);
   barrier_counter++;
   pthread_mutex_unlock (&barrier_mutex);

   // while counter is less than number of threads
   while (barrier_counter < p);

}
```

2. Semaphores :

```
// counting semaphore is initialized to 1 by main
// barrier semaphore is initialized to 0 by main
// barrier_counter is initialized to 0
// p is the number of threads

void barrier_using_semaphore () {

   // Barrier
   sem_wait (&count_sem);
   // if the last thread is executing this
   if (barrier_counter == p-1) {

       barrier_counter = 0;            // reset barrier_counter to 0
       sem_post (&count_sem);
       // for all other (p-1)-threads
       for (int thread = 0; thread < p-1; thread++) {
           sem_post (&barrier_sem);
       }

   } else {

       barrier_counter++;
       sem_post (&count_sem);
       // wait till the last thread signals it to go
       sem_wait (&barrier_sem);

   }

}
```

3. Condition Variables :

```
// barrier_mutex is declared by the main
// cond_var is declared by the main
// barrier_counter is initialized to 0
// p is the number of threads
void barrier_using_condition_variable () {

   pthread_mutex_lock (&barrier_mutex);
   barrier_counter++;

   if(barrier_counter == p) {
       barrier_counter = 0;
       pthread_cond_broadcast (&cond_var);
   } else {
       while ((pthread_cond_wait (&cond_var, &barrier_mutex)) != 0);
   }
   pthread_mutex_unlock (&barrier_mutex);

}
```

Comparison between busy-wait mutex, semaphores and condition variables

From the table, we can observe that busy-wait mutex works fine for small number of threads but with increase in number of threads, it's time taken increases in a much faster rate than other two techniques.

Both semaphore and condition variable techniques works in similar fashion. Both's time taken increases as number of threads increases but in most of the cases, semaphore is working slightly better than condition variable.

Conclusion : For higher number of threads, we should prefer semaphore or condition variable technique for implementing barrier.

→ All values are in seconds.

	p	1	2	4	8	16	32
Busy-wait and mutex	MAX	8.51e-07	8.24e-05	2.93e-05	0.026	0.047	0.135
	MIN	4.01e-07	5.53e-06	2.64e-06	0.005	0.035	0.081
	AVG	5.94e-07	4.59e-05	8.82e-06	0.017	0.043	0.104
Semaphores	MAX	1.17e-06	2.84e-05	7.89e-05	5.41e-05	0.00016	0.00023
	MIN	4.32e-07	2.26e-05	1.00e-05	3.38e-05	9.7717e-05	0.00016
	AVG	6.94e-07	2.61e-05	3.54e-05	4.29e-05	0.00012	0.00020
Condition variables	MAX	3.83e-07	5.34e-05	4.35e-05	6.76e-05	0.00015	0.00035
	MIN	1.90e-07	3.06e-05	1.31e-05	3.80e-05	0.00011	0.00021
	AVG	2.36e-07	3.78e-05	1.956e-05	5.33e-05	0.00013	0.00026

Read-write locks

Linked-list is a very famous data structure which is used in many places for various important tasks. How can we improve the performance of operations on a linked-list using threads?

The linked-list operated by multiple threads is called Multi-threaded linked list. Let's consider only three major operations on linked list : Member, Insert and Delete. We will consider two techniques to improve linked list operations using threads -

1. Mutex on entire list
2. Read-write lock

There is another technique "mutex per node" which is not a great technique as it involves a lot of mutexes and therefore it takes too much time.

Mutex on entire list :

Here, the idea is that we will put a mutex lock before any operation (member, insert, delete, etc) and unlock only after the operation is completed. This way we are putting a lock on the entire list, i.e. at a time only one thread can access the linked-list. We are basically serializing the list. This works fine when we have more **insert** and **delete** operations and less **member** operations as we'll need to serialize access to the list for most of the operations. If **member** operations are huge, then it will fail to exploit the opportunity of parallelism as in case of member we should be able to run multiple threads computing member as it only needs read access. To handle this issue, we introduce an advanced technique discussed below.

Read-write lock :

As we discussed in above section, we are not able to exploit the member operation. So to handle that, we can use read-write lock provided by Pthread library. It supports following operations :

```
pthread_rwlock_rdlock(&rwlock);
Member(value);
pthread_rwlock_unlock(&rwlock);
                . . .
pthread_rwlock_wrlock(&rwlock);
Insert(value);
pthread_rwlock_unlock(&rwlock);
                . . .
pthread_rwlock_wrlock(&rwlock);
Delete(value);
pthread_rwlock_unlock(&rwlock);
```

The functionality of read-write lock is as follows :

- When a thread puts read lock, then all the threads who wants to read can also lock but those threads who wants to write will have to wait for the lock to be unlocked.
- When a thread puts write lock, then all the threads who wants to read/write will have to wait for the lock to be unlocked.

Performance measure of entire-list mutex and read-write lock

Implementation	**Number of threads**					
	1	2	4	8	16	32
Entire-list mutex	0.009	0.027	0.020	0.015	0.018	0.019
Read-write lock	0.008	0.007	0.005	0.003	0.002	0.001

Linked List Times: 1000 Initial Keys, 100,000 operations, 99.9% Member, 0.05% Insert, 0.05% Delete

Implementation	**Number of threads**					
	1	2	4	8	16	32
Entire-list mutex	0.014	0.071	0.020	0.023	0.027	0.028
Read-write lock	0.020	0.020	0.021	0.024	0.027	0.028

Linked List Times: 1000 Initial Keys, 100,000 operations, 80% Member, 10% Insert, 10% Delete

We can see that when member operations were huge, Entire-list mutex performance was not good enough as compared to Read-write lock. But when member operations were reduced to 80%, we observed that performance of entire-list mutex was comparable to the Read-write lock but however not as consistent as Read-write lock but still somewhat comparable.

Conclusion : Read-write lock is the most preferable way for multi-threaded linked list among all the three techniques.

References :

An Introduction to Parallel Programming by Peter Pacheco.