International Journal of Computer and Communication Technology

Volume 8 | Issue 2 Article 11

April 2017

LOCK-FREE DINING PHILOSOPHER

VENKATAKASH RAJ RAOJILLELAMUDI
Department of Information Technology, Jadavpur University, Kolkata, India,
venkatakashrajraojillelamudi@gmail.com

SOURAV MUKHERJEE
Department of Information Technology, Jadavpur University, Kolkata, India, souravmukherjee@gmail.com

Ryan Saptarshi Ray
Department of Information Technology Jadavpur University, Kolkata, India, ryan.ray@rediffmail.com

Utpal Kumar Ray
Department of Information Technology, Jadavpur University, Kolkata, India, utpalkumarray@gmail.com

Follow this and additional works at: https://www.interscience.in/ijcct

Recommended Citation
RAOJILLELAMUDI, VENKATAKASH RAJ; MUKHERJEE, SOURAV; Ray, Ryan Saptarshi; and Ray, Utpal Kumar (2017) "LOCK-FREE DINING PHILOSOPHER," *International Journal of Computer and Communication Technology*: Vol. 8 : Iss. 2 , Article 11.
Available at: https://www.interscience.in/ijcct/vol8/iss2/11

This Article is brought to you for free and open access by Interscience Research Network. It has been accepted for inclusion in International Journal of Computer and Communication Technology by an authorized editor of Interscience Research Network. For more information, please contact sritampatnaik@gmail.com.

LOCK-FREE DINING PHILOSOPHER

VENKATAKASH RAJ RAOJILLELAMUDI[1], SOURAV MUKHERJEE[2], RYAN SAPTARSHI RAY[3], UTPAL KUMAR RAY[4]

Department of Information Technology, Jadavpur University, Kolkata, India
[1,2,]B.E.(I.T) Student 3rd Year, [3]PHD Scholar, [4]Associate Professor

Abstract- The past few years have marked the start of a historic transition from sequential to parallel computation.The necessity to write parallel programs is increasing as systems are getting more complex while processor speed increases are slowing down. Current parallel programming uses low-level programming constructs like threads and explicit synchronization using locks to coordinate thread execution. Parallel programs written with these constructs are difficult to design, program and debug. Also locks have many drawbacks which make them a suboptimal solution. Software Transactional Memory (STM) is a promising new approach to programming shared-memory parallel processors. It is a concurrency control mechanism that is widely considered to be easier to use by programmers than locking. It allows portions of a program to execute in isolation, without regard to other, concurrently executing tasks. A programmer can reason about the correctness of code within a transaction and need not worry about complex interactions with other, concurrently executing parts of the program. This paper shows the concept of writing code using Software Transactional Memory (STM) and the performance comparison of codes using locks with those using STM.

Keywords- *Parallel Programming; Multiprocessing; Locks; Transactions; Software Transactional Memory*

I. INTRODUCTION

Generally one has the idea that a program will run faster if one buys a next-generation processor. But currently that is not the case. While the next-generation chip will have more CPUs, each individual CPU will be no faster than the previous year's model. If one wants programs to run faster, one must learn to write parallel programs as currently multi-core processors are becoming more and more popular. The past few years have marked the start of a historic transition from sequential to parallel computation.The necessity to write parallel programs is increasing as systems are getting more complex while processor speed increases are slowing down. Parallel Programming means using multiple computing resources like processors for programming so that the time required to perform computations is reduced [1].

II. DINING PHILOSOPHER PROBLEM

In the Dining Philosopher problem, when there are 'n' numbers of philosophers then the number of chopsticks available is also 'n'. When a philosopher eats, he has to take two chopsticks at the same time. So, in this problem, multiple philosophers may access same chopsticks at the same time. The problem is to synchronize these accesses properly so that no philosopher ever faces starvation.

III. DINING PHILOSOPHER PROBLEM USING LOCKS

The hardest problem that should be overcome when writing parallel programs is that of synchronization. Multiple threads may need to access the same locations in memory and if careful measures are not taken the result can be disastrous. If two threads try to modify the same variable at the same time, the data can become corrupt. Currently locks are used to solve this problem. Locks ensure that a critical section, which is a block of code that contains variables that may be accessed by multiple threads, can only be accessed by one thread at a time. When a thread tries to enter a critical section, it must first acquire that section's lock. If another thread is already holding the lock, the former thread must wait until the lock-holding thread releases the lock, which it does when it leaves the critical section [2].

The following code shows the philosopher thread in Dining Philosopher problem using threads and locks:

```
void *philosopher(void *num_ptr)
{
   unsigned long byte_under_stm1,k,ki=0;
   unsigned char num, *number_ptr;
structtimevalini_tv;
number_ptr=num_ptr;
num=*number_ptr;
for((k=(((num*ARRAY_SIZE)/(NUM_THREADS))
));
k<(((num+1)*ARRAY_SIZE)/(NUM_THREADS)/2
);k++)
   {
pthread_mutex_lock(&mutex1);
pthread_mutex_lock(&mutex2);

      chopsticks[num]+=2;
printf("Philosopher %d is eating\n", num);
      chopsticks[num]-=2;

pthread_mutex_unlock(&mutex1);
pthread_mutex_unlock(&mutex2);
printf("Philosopher %d is thinking\n", num);
```

```
   }
pthread_exit(0);
}
```

In the above snippet "philosopher" is the thread,where multiple philosophers access the chopsticks. In the code, there is an array arr which is the buffer. A global array chopsticks[] store all the chopsticks.

In the thread "philosopher", chopsticks are accessed by the philosophers by the following statements.

```
for((k=(((num*ARRAY_SIZE)/(NUM_THREADS))
));
k<(((num+1)*ARRAY_SIZE)/(NUM_THREADS)/2
);k++)
   {
pthread_mutex_lock(&mutex1);
pthread_mutex_lock(&mutex2);

      chopsticks[num]+=2;
printf("Philosopher %d is eating\n", num);
      chopsticks[num]-=2;

pthread_mutex_unlock(&mutex1);
pthread_mutex_unlock(&mutex2);
printf("Philosopher %d is thinking\n", num);
   }
```

The following statement is used to record the time before the threads are created:

```
gettimeofday(&ini_tv,NULL);
```

The following statement is used to record the time when all threads have just finished their executions:

```
gettimeofday(&final_tv,NULL);
```

The total time taken is then calculated and printed using the following statement:

```
printf("Total Time Taken = %ld\n", final_tv.tv_sec -
ini_tv.tv_sec);
```

6 calls related with the mutex have been used in this program.

- pthread_mutex_init(&mutex1,NULL) and pthread_mutex_init(&mutex2,NULL) are used for lock initialization.
- pthread_mutex_lock(&mutex1) means that any thread must acquire the lock on mutex1 to execute the critical section following this function. pthread_mutex_unlock(&mutex1) is used for unlocking.
- pthread_mutex_lock(&mutex2) means that any thread must acquire the lock on mutex2 to execute the critical section following this function. pthread_mutex_unlock(&mutex2) is used for unlocking.

In this program the regions where more than one philosopher may access the global array chopsticks[] at the same time are the critical sections. Thus these regions are enclosed within locks. Hence , in this program, no philosopher ever faces starvation.

IV. EXPERIMENTAL RESULTS FOR DINING PHILOSOPHER PROBLEM USING LOCKS

The experimental data for the outputs shown in this paper have been recorded by running the codes on a machine which has 6 cores with hyper-threading. Thus, a maximum of 12 threads can run in parallel.
The experimental results for The Dining Philosopher Problem using locks are presented below:

Number of Threads	Time Taken (Seconds)	Speedup
1	132	1
2	67	1.970149254
3	48	2.75
4	29	4
5	22	5
6	19	6
7	17	7
8	13	8
9	12	9
10	11	10
11	10	11
12	7	12

Table 1:Experimental Results for Dining Philosophers Problem using Threads with Locks

The above experimental data has been represented graphically in Figure 1 and Figure 2 which show the variation of Time Taken for execution of the code, and Speedup respectively, with increase in the number of threads for the code of the Dining Philosophers Problem using threads with Locks.

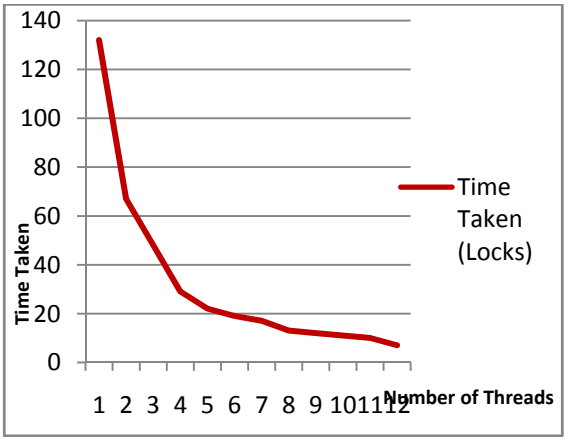

Figure 1: Graph showing the Time Taken vs. Number of Threads for Dining Philosopher Problem using Threads with Locks

It can be seen from the above graph that as the number of threads increases, the time taken for executing the code decreases.

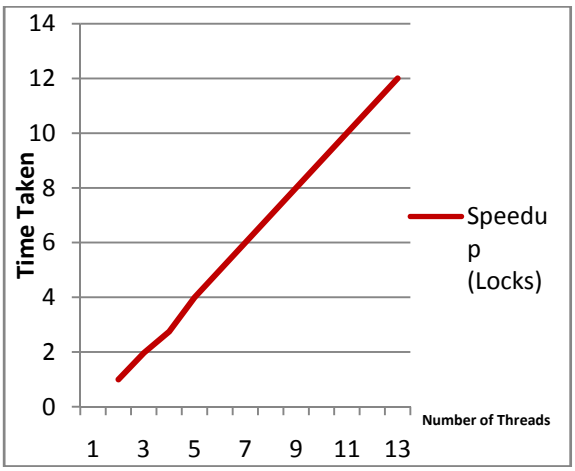

Figure 2: Graph showing the Speedup vs. Number of Threads for Dining Philosopher Problem using Threads with Locks

We can see that the speedup increases linearly with the number of threads.

V. DINING PHILOSPHERS PROBLEM USING STM

The synchronization problem can also be solved using STM. If STM is used in a program then we do not have to use locks in the program. Thus the problems which occur due to the presence of locks in a program do not occur in this type of code. The critical section of the program has to be enclosed within a transaction. Then STM by its internal constructs ensures synchronization in the program.
There are 8 categories of calls associated with STM which have been used in this program. They are as follows:

- stm_init : It is used to initialize the TinySTM library at the outset. It is called from the main thread before accessing any other functions of the TinySTM library.
- stm_init_thread : It is used to initializeeach thread that will perform transactions. It is called once from each thread that performs transactional operations before the thread calls any other functions of the TinySTM library. In the following code, it is called from the thread philosopher.
- stm_exit : Itis the corresponding shutdown function for stm_init. It cleans up the TinySTM library. It is called once from the main thread after all transactional threads have completed execution.
- stm_exit_thread : It is the corresponding shutdown function for stm_init_thread. It cleans up the transactional thread. It is called once from each thread that performs transactional operations upon exit. In the following code, it cleans up the thread philosopher.
- START(0,RW) : Itis used to start a transaction. In the following code, it is used in the thread philosopher.
- COMMIT: Itis used to close the transaction. In the following code, it is used in the thread philosopher.
- variable1=(unsigned char)LOAD(&variable2) : It stores the value of variable2 into variable1.
  - byte_under_stm1=(unsigned char)LOAD(&chopsticks[num]) : It stores the value of chopsticks[num] in byte_under_stm1. In the following code, it is used in the thread philosopher.
  - byte_under_stm2=(unsigned char)LOAD(&chopsticks[(num+1)%NUM_THREADS]) stores the value of chopsticks[(num+1)%NUM_THREADS] in byte_under_stm2. In the following code, it is used in the thread philosopher.
- STORE(&variable1], variable2) : It stores the value of variable2 into variable1.
  - STORE(&chopsticks[num], byte_under_stm1) stores the value of byte_under_stm1 in chopsticks[num]. In the following code, it is used in the thread philosopher.
  - STORE(&chopsticks[(num+1)%NUM_THREADS], byte_under_stm2) stores the value of byte_under_stm2 in chopsticks[(num+1)%NUM_THREADS]. In the following code, it is used in the thread philosopher.

The following code shows the philosopher thread using threads and STM which solves the Dining-Philosophers problem:

```
void *philosopher(void *num_ptr)
{
   unsigned long byte_under_stm1,byte_under_stm2, k=0;
   unsigned char num, *number_ptr;
number_ptr=num_ptr;
num=*number_ptr;

stm_init_thread();
for((k=(((num*ARRAY_SIZE)/NUM_THREADS)));
k<(((num+1)*ARRAY_SIZE)/NUM_THREADS)/2;
k++)
   {
      START(0,RW);
      byte_under_stm1=(unsigned char)
LOAD(&chopsticks
[num]);
      byte_under_stm2=(unsigned char)
LOAD(&chopsticks
```

```
[(num+1)%NUM_THREADS]);

        chopsticks[num]+=2;
printf("Philosopher %d is eating\n", num);
        chopsticks[num]-=2;

        STORE(&chopsticks[num],byte_under_stm1);

STORE(&chopsticks[(num+1)%NUM_THREADS],
byte_under_stm2);

printf("Philosopher %d is thinking\n", num);
        COMMIT;
    }
stm_exit_thread();
pthread_exit(0);
}
```

The program structure is same as that of the program for Dining-Philosopher problem using threads and locks. The only difference is that STM is being used in this program. In this program the regions where more than one philosopher may access the global array chopsticks[] at the same time are the critical sections. Thus these regions are enclosed within transactions using TinySTM which is a type of STM. Hence, in this program, no philosopher ever faces starvation.

VI. EXPERIMENTAL RESULTS FOR DINING PHILOSOPHER PROBLEM USING STM

The experimental results for The Dining Philosopher Problem using STM are presented below:

No. of Threads	Time Taken (Seconds)	Speedup
1	123	1
2	65	1.892307692
3	43	2.860465116
4	30	4
5	24	5
6	18	6
7	16	6
8	12	8
9	11	9
10	11	10
11	10	11
12	8	12

Table 2:Experimental Results for Dining Philosophers Problem using Threads with STM

The above data has been represented graphically in Figure 3 and Figure 4 which show the variation of Time Taken for execution of the code, and Speedup respectively, with increase in the number of threads for the code of the Dining Philosophers Problem using threads with STM.

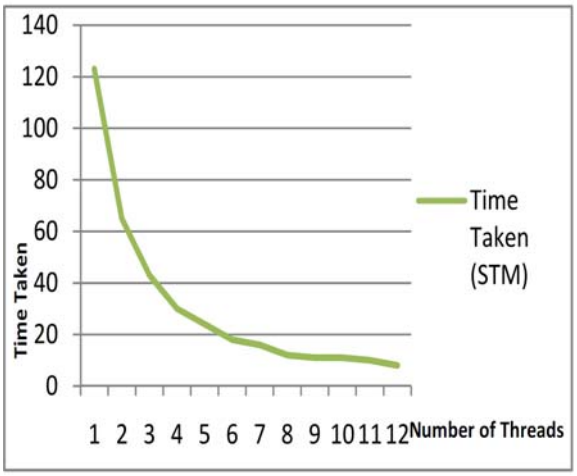

Figure3: Graph showing the Time Taken vs. Number of Threads for Dining Philosopher Problem using Threads with STM

It can be seen from the above graph that as the number of threads increases, the number of threads decreases.

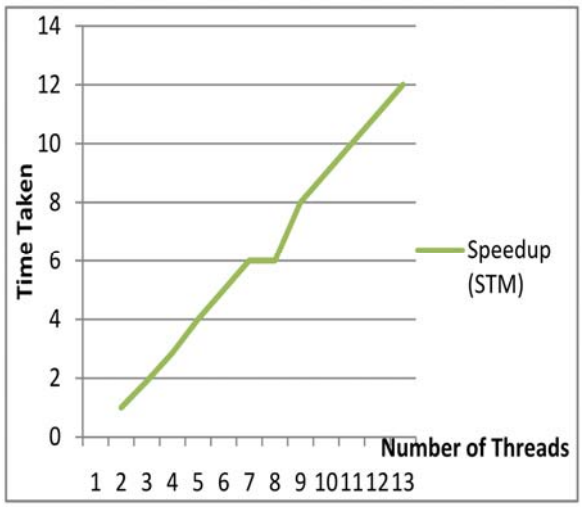

Figure 4: Graph showing the Speedup Taken vs. Number of Threads for Dining Philosopher Problem using Threads with STM

We can see from the above graph that the speedup increases almost linearly with the number of threads. Ideally the speedup should increase linearly with the number of threads, but practically, results differ.

VII. CONCLUSION

Figure 5 is a combination of Figures 1 and 3 as shown in previous sections. Similarly, Figure 6 is a combination of Figures 2 and 4 as shown in previous sections.

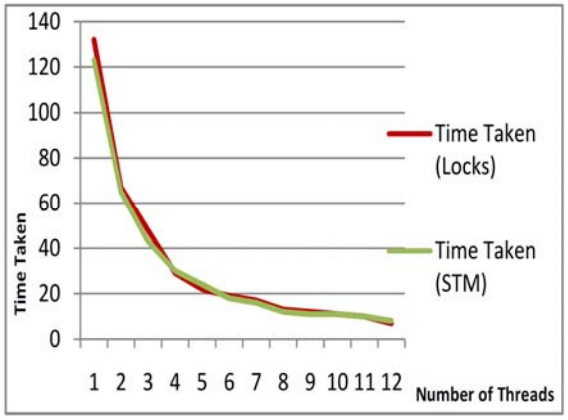

Figure 5: Graph showing the Time Taken vs. Number of Threads for Dining Philosopher Problem using both Threads with Locks and Threads with STM

We can see from the above graph that the time taken for executing the code using threads and STM is almost equal to the time taken for executing the same code using threads and locks. Thus, further research is being undertaken to improve the execution speed of STM.

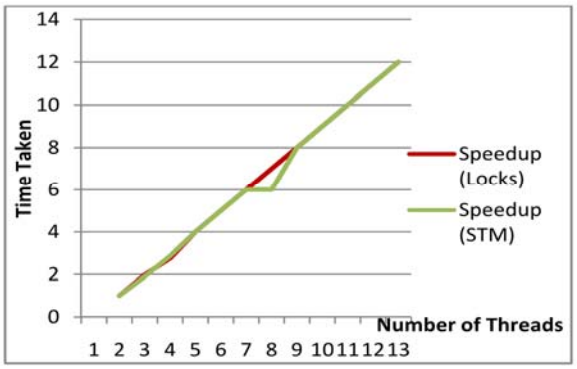

Figure 6: Graph showing the Speedup vs. Number of Threads for Dining Philosopher Problem using both Threads with Locks and Threads with STM

From the above graph we can see that the speedup for the codes using both locks and STM for this problem, are nearly the same. This is because the times taken for executing both the codes are also nearly equal.

STM has been shown in many ways to be a good alternative to using locks for writing parallel programs. STM provides a time-tested model for isolating concurrent computations from each other. This model raises the level of abstraction for reasoning about concurrent tasks and helps avoid many parallel programming errors.

This paper has discussed how STM can be used to solve the Dining Philosophers Problem of synchronization in parallel programs. STM has ensured that lock-free parallel programs can be written. This ensures that the problems which occur due to the presence of locks in a program do not occur in this type of code.

Many aspects of the semantics and implementation of STM are still the subject of active research. While it may still take some time to overcome the various drawbacks, the necessity for better parallel programming solutions will drive the eventual adoption of STM. Once the adoption of STM begins it will have the potential to pick up momentum and make a very large impact on software development in the long run. In the near future STM will become a central pillar of parallel programming.

REFERENCES

[1] Simon Peyton Jones, “Beautiful concurrency”.

[2] Elan Dubrofsky, “A Survey Paper on Transactional Memory”.

[3] Pascal Felber, Christof Fetzer, Torvald Riegel, “Dynamic Performance Tuning of Word-Based Software Transactional Memory”.

[4] http://en.wikipedia.org/wiki/Transactional_memory

[5] James Larus and Christos Kozyrakis. “Transactional Memory”

[6] Pascal Felber, Christof Fetzer, Patrick Marlier, Torvald Riegel, “Time-Based Software Transactional Memory”

[7] Tim Harris, James Larus, Ravi Rajwar, “Transactional Memory”

[8] Mathias Payer, Thomas R. Gross, “Performance Evaluation of Adaptivity in Software Transactional Memory”

[9] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, David A. Wood., “LogTM: Log-based Transactional Memory”

[10] Dave Dice , Ori Shalev , Nir Shavit., “Transactional Locking II”

[11] http://tmware.org

[12] Maurice Herlihy, J. Eliot B. Moss, “Transactional Memory: Architectural Support for Lock-Free Data Structures”.

[13] Martin Schindewolf, Albert Cohen, Wolfgang Karl, Andrea Marongiu, Luca Benini, “Towards Transactional Memory Support for GCC”.

[14] Virendra J. Marathe, Michael F. Spear, Christopher Heriot,Athul Acharya, David Eisenstat, William N. Scherer III, Michael L. Scott, “Lowering the Overhead of Nonblocking Software Transactional Memory”.

[15] Utku Aydonat, Tarek S. Abdelrahman,Edward S. Rogers Sr., “Serializability of Transactions in Software Transactional Memory”.

[16] Maurice Herlihy, Nir Shavit, “The Art of Multiprocessor Programming”.

[17] Brendan Linn, Chanseok Oh, “G22.2631 project report: software transactional memory”.

[18] Ryan Saptarshi Ray , “Writing Lock-Free Code using Software Transactional Memory”.

[19] http://en.wikipedia.org/wiki/Software_transactional_memory

[20] http://research.microsoft.com/~simonpj/papers/stm/

[21] http://www.haskell.org/haskellwiki/Software_transactional_memory.

◈◈◈