The Ministry of Education and Science of the Russian Federation

Lobachevsky State University of Nizhni Novgorod

Computing Mathematics and Cybernetics faculty

The competitiveness enhancement program

of the Lobachevsky State University of Nizhni Novgorod

among the world's research and education centers

Strategic initiative

“Achieving leading positions in the field of supercomputer technology

and high-performance computing”

TBB-Based Parallel Programming

Lecture 5. TBB Synchronization Primitives

Nizhni Novgorod

2014

Lecture 5. TBB Synchronization Primitives

Objectives

The purpose of this lecture is studying mutual exclusion problem and TBB synchronization primitives. The lecture also deals with threadsafe containers that may be useful for implementing a wide range of parallel algorithms.

Abstract

This lecture deals with TBB synchronization primitives. It describes various implementations of mutexes and readers/writers mutexes. It also describes the atomic template class whose methods are atomic. The lecture also deals with threadsafe containers that may be useful for implementing a wide range of parallel algorithms.

Guidelines

One of the main issues of parallel programming is the mutual exclusion problem. In case of multithread operation one thread may wait to receive data (computation results) from another one. This requires thread synchronization.

Let us consider data race, a typical situation requiring synchonization. Let data be the shared variable that is readable and writable for multiple threads. Each thread has to increment this variable (i. e. execute the data++ code). For this purpose, the CPU has to execute three operations: read the variable from RAM to the process register, increment the register and write the computed value into the variable (RAM).

Concurrent execution of such a code by two threads can be implemented in two ways (right down to thread permutation). The most likely application behaviour is shown in Fig. 1. First, thread 0 reads the variable, increments the register and writes its value to the variable, then thread 1 follows the same sequence. Thus, upon application termination the common variable will be equal to data+2.

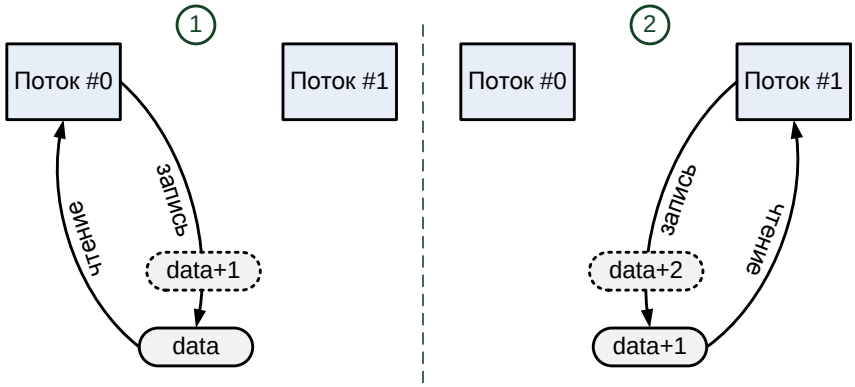

Fig. 1. Data race implementation variation

See other possible behaviour in Fig. 2. Thread 0 reads the variable to the register, increments the register while thread 1 reads the data variable. As in this case each thread has its own set of registers, thread 0 will continue execution and save data+1 value as the variable. Thread 1 will

also increment the register (the data value was read from RAM before data+1 value was saved by thread 0) and save the data+1 value as the common variable. Thus, upon application termination the common variable will be equal to data+1. Both implementations can be observed both for multicore (multiprocessor) and single-core (single-processor) systems.

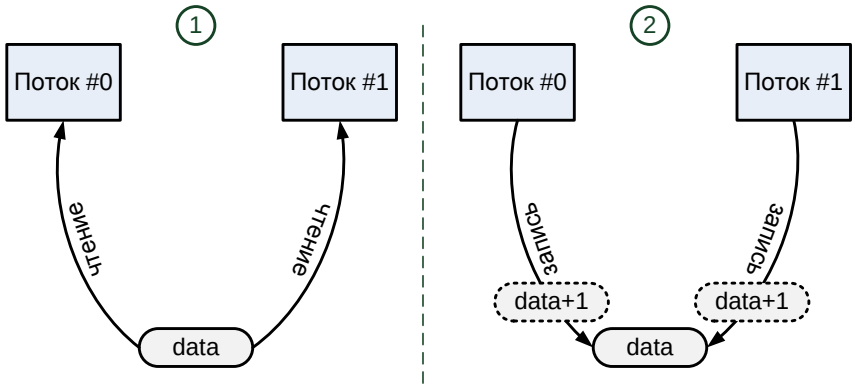

Fig. 2. Data race implementation variation

Thus, depending on the command queuing, the results may vary. This calls for a thread execution synchronization mechanism which could enable execution of a code part by no more than one thread at a time (mutual exclusion).

The primary way to solve the mutual exclusion problem is to use synchronization primitives. One of such primitives implemented in TBB is *mutex.*

The purpose of mutexes is to ensure execution of a critical code section (the one requiring synchronization) by one thread only. A mutex can be either free or acquired. Any thread can acquire a mutex thus making it pass from the “free” state to the “acquired” state. If any thread tries to acquire an already “acquired” mutex, the software code execution is suspended until the mutex passes to the “free” state.

Three type of mutexes are implemented in the TBB library

• mutex - the operation system mutex. This is a wrapper for operation system synchronization primitives (MS Windows OS are based on critical sections). As this type of mutex is implementing using the operation system objects, the thread attempting to acquire the mutex in the acquired state goes into a wait state while the operation system transfers control to other threads. When the mutex is released, the operation system makes the thread pass to the ready-to-run state. Therefore, the time between mutex release until the thread gains control, may be long enough;

• spin_mutex - a busy wait mutex. The thread attempting to acquire this type of mutex remains active. If a thread attempts to acquire an “acquired” mutex, it will continue its attempts until the mutex is free. Therefore, upon mutex release one of the waiting threads will start critical code execution. It is advisable to use this type of mutex when the critical code part execution time is short;

• queuing_mutex – this type of mutex performs busy wait for the mutex being acquired subject to thread priority, which means that threads are executed in the same order as they attempted to acquire the mutex. This is the slowest mutex type as its operation entails additional contingencies.

Apart from ordinary mutexes, TBB has implementations of reader/writer mutexes. These mutexes have an additional "writer" flag. Using this flag, one can indicate what type of mutex lock is to be acquired, the reader (writer=false) or the writer (writer=true) one. Several writers (the threads that acquired the reader lock) can execute the critical code simultaneously in absence of the writer lock. If the thread has acquired a writer mutex, all other threads will be blocked when they attempt to acquire a mutex.

The library has two types of reader/writer mutexes; these are spin_rw_mutex and queuing_rw_mutex. The parameters of these mutexes are similar with those of spin_mutex and queuing_mutex.

Many TBB library functions and classes ensure implicit synchronization (no mutexes are used explicitly). For example, the parallel_for function waits for completion of all threads involved in computation before giving up control to the thread that called for it.

Apart from mutexes, TBB contains the tbb::atomic template class that can also be used for synchronization purposes. Methods of this class are atomic, i. e. they cannot be executed by more than one thread at a time. If the threads attempt to execute the tbb::atomic methods simultaneously, one of such threads is blocked and waits until the other one completes method execution.

TBB contains a number of threadsafe containers similar to STL containers. However, contents and functionality of the TBB containers is different as they can be addressed by multiple threads.

Recommendations for Students

The information is mainly sourced from the official TBB web page https://www.threadingbuildingblocks.org/. The site features numerous documents and examples. A free library version for non-commercial use is also downloadable.

Andrews (2000) is a recommended introduction into parallel programming.

Quinn (2004) is also recommended as a description of typical problems of parallel programming.

Practice

1. Implement a program solving the producer-consumer problem using TBB synchronization primitives.

2. Implement a program solving the readers/writers problem using TBB synchronization primitives.

3. Implement a program solving the dining philosophers problem using TBB synchronization primitives.

Test questions

1. Will the count variable contain the number of operator() calls after the use of functor in parallel_for in the code below?

```
int Functor::count = 0;
mutex Functor::myMutex;

class Functor
{
private:
  static int count;
  static mutex myMutex;

public:
  // Functor methods
  //...
  void operator()(const blocked_range<int>& Range) const
  {
    mutex::scoped_lock lock;
    lock.acquire(myMutex);
    count++;
  }
};
```

   a. (+) Yes
   b. No, the program will get stuck.
   c. Yes, only if the program is run in one thread.

2. A mutex class is implemented in the TBB library. This is:
   a. A busy wait mutex.
   b. A mutex that preserves the order of threads that have acquired the mutex.
   c. (+) An operation system mutex.

3. A spin_mutex class is implemented in the TBB library. This is:
   a. (+) Busy wait
   b. A mutex that preserves the order of threads that have acquired the mutex.
   c. An operation system mutex.

4. A spin_mutex class is implemented in the TBB library. This is:
   a. A busy wait mutex.
   b. A mutex that preserves the order of threads that have acquired the mutex.
   c. An operation system mutex.

5. The following is typical for the operation system mutex:
    a. (+) The thread attempting to acquire the mutex in the acquired state goes into a wait state while the operation system transfers control to other threads.
    b. The thread attempting to acquire this type of mutex remains active.
6. The following is typical for the busy wait mutex:
    a. The thread attempting to acquire the mutex in the acquired state goes into a wait state while the operation system transfers control to other threads.
    b. (+) The thread attempting to acquire this type of mutex remains active
7. The scoped_lock class:
    a. (+) Is intended to acquire the mutex
    b. (+) Is intended to release the mutex
    c. Is intended to create mutexes
8. The scoped_lock::acquire() method:
    a. (+) Acquires the mutex.
    b. Performs nonblocking mutex lock.
    c. Releases the mutex.
9. The scoped_lock::try_acquire() method:
    a. Acquires the mutex.
    b. (+) Performs nonblocking mutex lock.
    c. Releases the mutex.
10. The scoped_lock::release():
    a. Acquires the mutex.
    b. Performs nonblocking mutex lock.
    c. (+) Releases the mutex.
11. Why the size method return a signed integer in the tbb::concurrent_queue class?
    a. (+) Negative values indicate that there have been more extraction operations in the queue than addition ones.
    b. Negative values indicate a mistake
    c. Negative values are not used

References

1. Intel® Threading Building Blocks Home Page: https://www.threadingbuildingblocks.org/
2. Intel® Threading Building Blocks Reference Manual: https://software.intel.com/en-us/node/506130
3. Intel® Threading Building Blocks User Guide: https://software.intel.com/en-us/node/506045
4. Andrews, G. R. (2000). Foundations of Multithreaded, Parallel, and Distributed Programming. – Reading, MA: Addison-Wesley.
5. Quinn, M. J. (2004). Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill.