Parallel Computing

Academic year – 2020/21, spring semester

Computer science

Lecture 7

Lecturer, instructor:

Balakshin Pavel Valerievich

(pvbalakshin@itmo.ru; pvbalakshin@hdu.edu.cn)

Assistant:

Liang Tingting

(liangtt@hdu.edu.cn)

Parallel computing paradigm

- **Explicit locking** for synchronization when addressing to the shared data.
- **Non-blocking algorithms** (lockless, lockfree algorithms) – use elementary atomic operations only.
- **Software Transactional Memory** – non-synchronous use of divisible data, followed by eliminating conflict.

Explicit locking

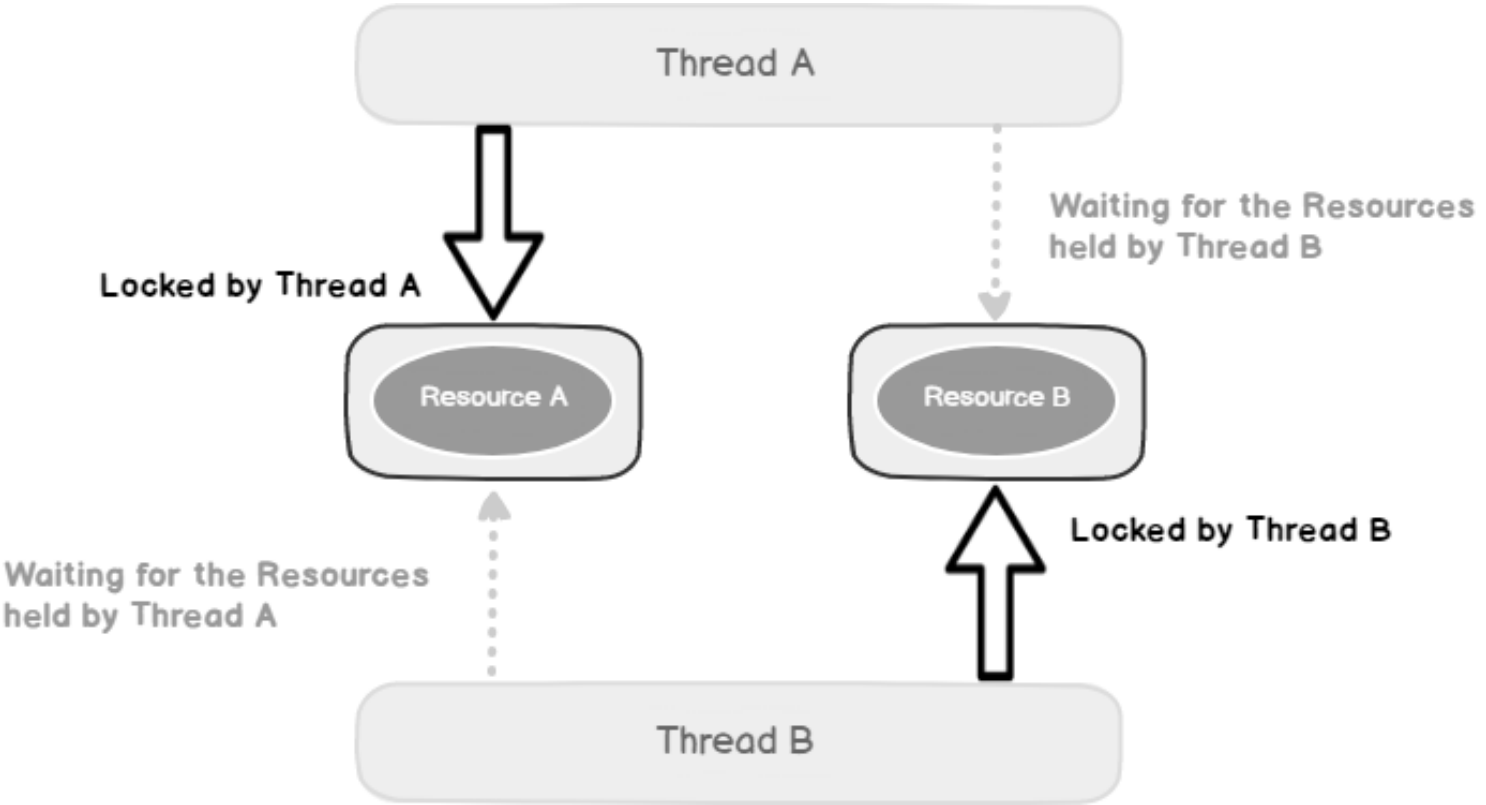
1. The most popular one:

- *Semaphore* – an object that limits the number of threads that can enter this area of code.
- *Reader/writer semaphore* gives the threads only read or write permissions, and while writing data to one thread, the rest of the threads do not have access to the resource.
- *Mutex* is a special case of a semaphore, in which only one thread can capture a given area of code.
- *Spinlock* is a lock in which a thread in a loop waits for a resource to free.
- *Seqlock* – synchronization mechanism designed to quickly record a variable in multiple threads.
- *Barrier* – a part of code in which the state of threads is synchronized.

Explicit locking

2. Unstable waiting time when locked without upper limit.
3. Deadlocks are possible.
4. Construction/removal overhead and waiting costs.
5. Difficulties in debugging.
6. Poor scalability.
7. Priority inversion is possible.

Deadlock



According to G. Gorelkin “Deadlocks, Livelocks и Starvation”

Deadlock (2)

```
omp_lock_t lock1, lock2;
omp_init_lock(&lock1); /* initialization of lock 1 */
omp_init_lock(&lock2); /* initialization of lock 2 */
#pragma omp parallel sections
 {
   #pragma omp section
   {
      omp_set_lock(&lock1);        omp_set_lock(&lock2);
      omp_unset_lock(&lock2);      omp_unset_lock(&lock1);
   }
   #pragma omp section
   {
      omp_set_lock(&lock2);        omp_set_lock(&lock1);
      omp_unset_lock(&lock1);      omp_unset_lock(&lock2);
   }
}
```

Non-blocking algorithms

```
1  /* Pseudocode of the instruction returning boolean value in C syntax */
2  int cas( int* addr, int old, int new )
3  {
4      if ( *addr != old )
5          return 0;
6
7      *addr = new;
8      return 1;
9  }
```

1. Block resource.
2. Read value of the variable.
3. Process some computations.
4. Write new value of the variable.
5. Release block of resource.

1. Read value of the variable.
2. Process some computations.
3. Process cmpxchg operation with new value of variable assuming that old value is still the same.
4. Repeat steps 2-3 if value of the variable was changed by another thread.

Non-blocking algorithms

- No standard limitations of blocking algorithms.
- Quite high input level.
- Program became bigger and more difficult to maintain and support.
- The limited number of atomic operations leads to a significant complication of the algorithm.

Transactional memory

- Easy of use – enclosing code sections in a transaction block.
- Absence of blocking.
- Working good and fast in case of small amount of access conflicts (can be roll back in case of failure).
- Ideally it requires hardware support (execution of transactions not by compiler, but processor with special internal registers to store old values), which is not yet implemented in all processors.
- Supported in GCC since version 4.7:

  _transaction atomic {…} – an indication that the code block is a transaction;

  _transaction_relaxed {…} – an indication that the unsafe code inside the block does not lead to side effects;

  _transaction cancel – explicit transaction cancellation.

Transactional memory (2)

```
int contains(int value)
{
  int result;
  node_t *prev, *next;

  __transaction_atomic {
    prev = set->head;
    next = prev->next;
    while (next->val < val) {
      prev = next;
      next = prev->next;
    }
    result = (next->val == val);
  }
  return result;
}
```

```
int add(int value)
{
  int result;
  node_t *prev, *next;

  __transaction_atomic {
    prev = set->head;
    next = prev->next;
    while (next->val < val) {
      prev = next;
      next = prev->next;
    }
    result = (next->val != val);
    if (result) {
      prev->next = new_node(val, next);
    }
  }
  return result;
}
```

Pthreads

Pthreads - the POSIX (IEEE) standard, which defines the API for thread management.

Key Pthreads tools provide:
- Create, delete, merge threads.
- Shared access to shared variables.
- Synchronization of threads.

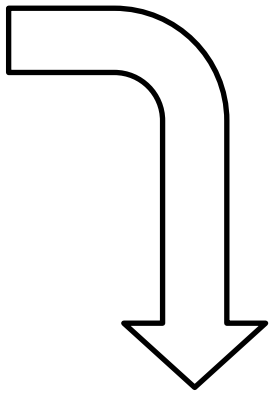
Compiling Pthreads-program

Intel GNU/Linux	`icc -pthread`
	`icpc -pthread`
PGI GNU/Linux	`pgcc -lpthread`
	`pgCC -lpthread`
GNU GCC GNU/Linux, Blue Gene	`gcc -pthread`
	`g++ -pthread`
IBM Blue Gene	`bgxlc_r / bgcc_r`
	`bgxlC_r, bgxlc++_r`

There are no "native" Microsoft tools for compiling for Windows, but you can use third-party libraries.

Creation and deletion of threads

```
int main(int argc, char *argv[])
{
   int a = 1, b = 2;
   #pragma omp parallel sections
   #pragma omp parallel section
   function_1(a);
   #pragma omp parallel section
   function_1(b);
}
```



```
int main(int argc, char *argv[])
{
   pthread_t t1, t2;
   int a = 1, b = 2;
   pthread_create(&t1, NULL, function_1, (void*)&a);
   pthread_create(&t2, NULL, function_1, (void*)&b);
   pthread_exit(NULL);
}
```

Creation and deletion of threads (2)

```c
#include <pthread.h>
#define NTHREADS 5

void *thread_fun(void *threadid) {
    long tid = (long)threadid;
    printf("Hello from %ld!\n", tid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    pthread_t threads[NTHREADS];
    int rc; long t;

    for (t = 0; t < NTHREADS; t++) {
        rc = pthread_create(&threads[t], NULL, thread_fun, (void *)t);
        if (rc) {
            printf("ERROR %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

Creation and deletion of threads (3)

```c
#include <pthread.h>
#define NTHREADS 5

void *thread_fun(void *threadid) {
    long tid = (long)threadid;
    printf("Hello from %ld!\n", tid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    pthread_t threads[NTHREADS];
    int rc; long t;

    for (t = 0; t < NTHREADS; t++) {
        rc = pthread_create(&threads[t], NULL, thread_fun, (void *)t);
        if (rc) {
            printf("ERROR %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

```c
int main(int argc, char *argv[])
{
   #pragma omp parallel num_threads(5)
   printf("Hello from %u\n!",
          omp_get_thread_num());
}
```

Creation and deletion of threads (4)

```
$ gcc -pthread -o prog ./prog.c
$ ./prog
Hello from 1!
Hello from 4!
Hello from 0!
Hello from 2!
Hello from 3!
```

Threadpool consolidation

```c
#include <pthread.h>
#define NTHREADS 5

// ...

int main(int argc, char *argv[]) {
    pthread_t threads[NTHREADS];
    int rc; long t;
    void *status;

    for (t = 0; t < NTHREADS; t++) {
        rc = pthread_create(&threads[t], NULL, thread_fun,
                            (void *)t);
    }

    for (t = 0; t < NTHREADS; t++) {
        rc = pthread_join(threads[t], &status);
    }
    pthread_exit(NULL);
}
```

Create threads and wait for all threads in the main function

```
intervals = atoi(argv[1]);
numThreads = atoi(argv[2]);
threads = malloc(numThreads * sizeof(pthread_t));
threadID = malloc(numThreads * sizeof(int));
pthread_mutex_init(&piLock, NULL);
for (i = 0; i < numThreads; i++) {
    threadID[i] = i;
    pthread_create(&threads[i], NULL, computePI, threadID + i);
}
for (i = 0; i < numThreads; i++)
    pthread_join(threads[i], &retval);

printf("Estimation of pi is %32.30Lf \n", pi);
```

Shared variable protection

```
pthread_mutex_t piLock;
long double intervals;
int numThreads;

void *computePI(void *id)
{
    long double x, width, localSum = 0;
    int i, threadID = *((int*)id);    width = 1.0 / intervals;
    for (i = threadID ; i < intervals; i += numThreads) {
        x = (i + 0.5) * width;
        localSum += 4.0 / (1.0 + x * x);
    }
    localSum *= width;
    pthread_mutex_lock(&piLock);
    pi += localSum;
    pthread_mutex_unlock(&piLock);
    return NULL;
}
```

Threads synchronization without their removal (stop)

```
void * entry_point(void *arg)
{
    int rank = (int)arg;
    for(int row = rank * ROWS / THREADS; row < (rank + 1) * THREADS; ++row)
        for(int col = 0; col < COLS; ++col)
            DotProduct(row, col, initial_matrix, final_matrix);

    // Synchronization point
    int rc = pthread_barrier_wait(&barr);
    if(rc != 0 && rc != PTHREAD_BARRIER_SERIAL_THREAD)
    {
        printf("Could not wait on barrier\n");
        exit(-1);
    }

    for(int row = rank * ROWS / THREADS; row < (rank + 1) * THREADS; ++row)
        for(int col = 0; col < COLS; ++col)
            DotProduct(row, col, final_matrix, initial_matrix);
}

int main(int argc, char **argv)
{
    pthread_t thr[THREADS];

    // Barrier initialization
    if(pthread_barrier_init(&barr, NULL, THREADS))
```