关于生产者消费者问题的OpenMP实现

1. 内容简介：

针对典型的生产者和消费者问题，使用OpenMP编程，实现生产者生成随机数，由消费者求和并打印的操作。

2. 问题分析：

- 数据竞争问题：当有多个生产者向同一个缓冲区写数据，以及有多个消费者从同一个缓冲区读取数据时，存在数据竞争的问题。

- 生产者消费者问题：生产者向缓冲区发送数据，消费者从缓冲区取出数据，但是当缓冲区满时，生产者不能发送数据，当缓冲区为空时，消费者不能从缓冲区取出数据。

3. 实验设计：

3.1 数据结构：

设计一个缓冲区的队列结构，生产者向队尾发送数据，消费者从队头取数据。数据结构中包含队头和队尾指针（用整数模拟，用数组模拟循环队列）。为了提高并行度，队列使用两个锁（队头锁和队尾锁）实现互斥，以解决数据竞争的问题，这样生产者和消费者就可以同时进行操作。如果只使用一个锁，那么对整个队列的操作都是互斥的，生产者和消费者的操作不能同时进行。

```
// messege_queue structure
struct messege_queue{
   int *msg;
   int front;
   int back;
   omp_lock_t front_mutex;
   omp_lock_t back_mutex;
};
```

3.2 操作函数：

1.初始化函数：

初始化锁，消息队列的分配，和相关的变量：

```
// initiatl every thread and its messege_queue
void initial(messege_queue* init){
   init->msg = (int*)malloc(sizeof(int)*MAX_MESSEGE_QUEUE_SIZE);
   init->front = 0;
   init->back = 0;
   omp_init_lock(&(init->front_mutex));
   omp_init_lock(&(init->back_mutex));
}
```

2.销毁函数：

销毁锁，释放队列的空间：

```
// destroy every thread and its messege_queue
void destroy(messege_queue* dstry){
   free(dstry->msg);
   omp_destroy_lock(&(dstry->front_mutex));
   omp_destroy_lock(&(dstry->back_mutex));
}
```

3.try_send函数：

向缓冲区发送数据之前，首先需要获得队尾操作的锁，然后判断这个队列是否已满，如果不满，则可以传送数据，更新队尾，并返回1；否则返回0.

```
// try to send messege
int try_send(messege_queue *pool, int messege){
   int flag;

   omp_set_lock( &(pool->back_mutex) );

   // if the pool is full, send messege will failed.
   if( pool->front % MAX_MESSEGE_QUEUE_SIZE == pool->back %
MAX_MESSEGE_QUEUE_SIZE && pool->front != pool->back) flag = 0;
   else {
      // send messege to target thread
      int t = pool->back;
      t = t % MAX_MESSEGE_QUEUE_SIZE;
      (*pool).msg[t] = messege;
      pool->back = pool->back + 1;

      // printf("thread %d get %d from send.\n", target, messege_);
```

```
        flag = 1;
    }
    omp_unset_lock( &((pool)->back_mutex));

    return flag;
};
```

4.try_receive函数：

从缓冲区获得数据之前，首先需要获得队头操作的锁，然后判断这个队列是否已经为空，如果非空，则可以获得消息，更新队头之后返回消息的值；否则返回-1。

```
// try to get messege
int try_recieve(messege_queue *pool){

    int messege;

    omp_set_lock( &(pool->front_mutex) );

    // if it is null, return -1
    if(pool->back == pool->front) messege = -1;

    else{
        // have messege to recieve
        int front = pool->front;
        front = (front)%MAX_MESSEGE_QUEUE_SIZE;
        messege = (*pool).msg[front];

        // set new front
        pool->front = pool->front + 1;

        // printf(" get %d from recieve.\n", messege);
    }

    omp_unset_lock( &(pool->front_mutex) );

    return messege;
};
```

5.生产者和消费者：

程序从用户的输入中获得并行的线程数，前一半作为生产者，后一半作为消费者。

首先需要在并行块之前声明好变量的属性：

```
#  pragma omp parallel num_threads(thread_count) shared(pool,
thread_count, send_finished, send_total, recieve_total)
```

生产者尝试一定数量的发送消息的操作，并记录本线程发送成功的消息值的和：

```
// act as producer
    if(my_id < thread_count/2){
      int my_messege;
      srand((unsigned int)time(NULL));
      my_messege = rand()%MAX_SEND_MESSEGE;

      for(int i = 0; i < OPERATION_NUM; i++){
        if(try_send(&pool, my_messege)){
          my_send_times ++;
          my_send_total += my_messege;

          #  ifdef OUTPUT
          printf("thread %d send messege %d succeed.\n", my_id, messege);
          #  endif
        }

        else{
          #  ifdef OUTPUT
            printf("thread %d send messege fail.\n", my_id);
          #  endif
          };

        srand((unsigned int)time(NULL));
        my_messege = rand()%MAX_SEND_MESSEGE;
      }

    }
```

消费者线程依据一定的判断条件不断尝试获取消息，并记录本线程获得消息的值的和：

```
// act as consumer
    else{
      int recieve;

      while(!Done(&pool, send_finished, thread_count)){
        // try to recieve messege
        recieve = try_recieve(&pool);
        //has messege
        if(recieve >= 0){
```

```
            my_recieve_times++;
            my_recieve_total += recieve;

            #   ifdef OUTPUT
                printf("thread %d get messege %d succeed.\n", my_id, recieve);
            #   endif
        }
        // don't have messege
        else{
            #   ifdef OUTPUT
                printf("thread %d try to get messege fail.\n", my_id);
            #   endif
        }
    }
}
```

当生产者和消费者的工作完成之后，需要更新已经完成的生产者的数量，要把本线程记录的值累加到全局变量：total_send和total_receive上，这里要注意数据竞争的问题（这里使用了atomic指令）：

```
// producer sum up
    if(my_id < thread_count/2){
        #       pragma omp atomic
        send_finished++;

        #       pragma omp atomic
        send_total += my_send_total;

        printf("thread %d send %d.\n",my_id, my_send_total);
    }

    // consumer sum up
    else{
        #   pragma omp atomic
        recieve_total += my_recieve_total;

        printf("thread %d get %d.\n",my_id, my_recieve_total);
    }
```

6.对于消费者来说，需要判断生产者是否已经发送完消息以及消息队列里是否还有消息（finished记录的是已经完成的生产者数量，只能由生产者更新），在没有完成之前，消费者会一直尝试获取消息：

```
int Done(messege_queue* pool, int finished, int thread_count){
   if(  finished == thread_count/2   && pool->front == pool->back)
      return 1;

   return 0;
}
```

4. 实验结果：

经过验证，生产者发送的消息的值得总和总是等于消费者获得的消息的值的总和相等，说明程序是正确的。

值得一提的是，生产者和消费者判断队列是否为空或是否已满时，存在数据竞争，但是这个不会影响程序的正确性。以消费者为例进行解释：一种可能的情况就是消费者判断队列是否为空，需要读取队头和队尾，但是生产者可能正在修改队尾值。如果消费者读取到的是更新之后的，则是我们想要的情况；如果读取到的是更新之前的值，则也不影响，因为消费者还会尝试读取队头和队尾，总会读取到更新之后的值。