Project 2 Report

Chai Yi 9008308232
chyi13@student.bth.se

1. Scalable bounded-buffer implementation

1.1 Scalable Parallel implementation

Implementation of a scalable bounded-buffer is a buffer which could be accessed by different threads simultaneously. So the same size of buffer is assigned to a threads pair which has a producer thread and a consumer thread. In this situation, the number of producers and the number of consumers are the same in my implementation. Every thread pair deals with a particular buffer as the example code bounded_buffer_simple. But my implementation has N buffers of the same size as the example that could be accessed parallelly.

1.1.1 Data structure

In my implementation, the buffer size of each threads pair is 10.

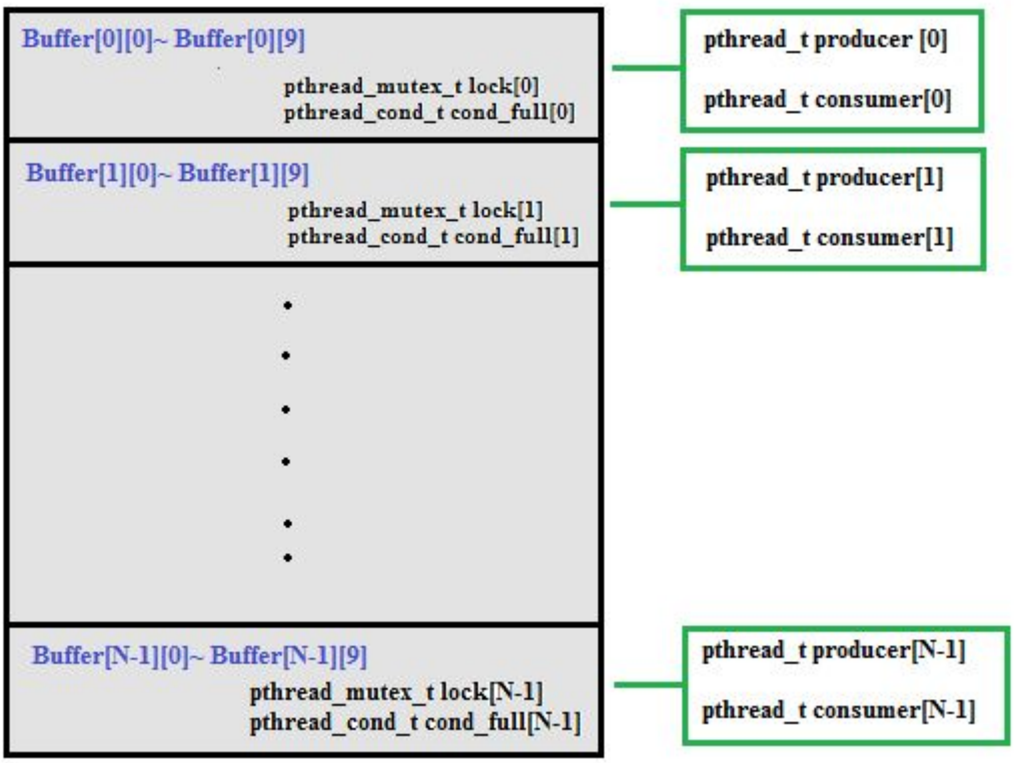

Figure 1

As in figure 1, every buffer could only be accessed by 1 thread pair. It is protected by pthread_mutex_t lock[n]. The total work is assigned to every producer and consumer equally as

ITEMS_PER_THREAD = ITEMS_TO_SEND/num_thread. Num_thread could be determined by user.

The buffer's structure is as follows:

typedef struct {

// total buffer = thread_num * SINGLE_BUFFER_SIZE

int buf[MAX_THREAD_NUM][SINGLE_BUFFER_SIZE];

int in[MAX_THREAD_NUM]; // producer item index

int out[MAX_THREAD_NUM]; // consumer item index

int no_elems[MAX_THREAD_NUM]; // current item number in the buffer

int no_items_sent[MAX_THREAD_NUM]; // items already sent per producer thread

int no_items_received[MAX_THREAD_NUM]; // items already received per consumer thread

pthread_mutex_t lock[MAX_THREAD_NUM]; // protects the buffer

pthread_cond_t cond_full[MAX_THREAD_NUM]; // condition full to inform consumer that the buffer is not empty

} buffer_t;

1.1.2 Procedure

In consumer[n], firstly it uses buffer.lock[n] to lock the element. Then it checks buffer[n][] whether it has items or not. If there is no items, it will use condition variable to wait the full signal from producer[n]. If buffer[n][] is not empty, it will get the item from buffer[n] and remove it from the buffer. When consumer[n] gets items_per_thread items, the thread will terminate.

In producer[n], the procedure is similar as consumer. It also checks buffer[n][] whether it has items or not. If buffer[n][] is not full, producer sets it to a new item index and sends a full signal to consumer[n]. When producer[n] sends items_per_thread items, producer[n] will terminate.

1.2 Experiment result

The total number of items is 8388608. (8 cores environment)

1.2.1 Compiling of scalable version

gcc -pthread -O -o scalable_bounded_buffer scalable_bounded_buffer.c

```
$ gcc -pthread -O -o scalable_bounded_buffer scalable_bounded_buffer_v4.c
scalable_bounded_buffer_v4.c: In function 'consumer':
scalable_bounded_buffer_v4.c:88:4: warning: format '%d' expects argument of type
 'int', but argument 2 has type 'long int' [-Wformat]
scalable_bounded_buffer_v4.c:100:3: warning: format '%d' expects argument of typ
e 'int', but argument 2 has type 'long int' [-Wformat]
scalable_bounded_buffer_v4.c: In function 'producer':
scalable_bounded_buffer_v4.c:110:5: warning: format '%d' expects argument of typ
e 'int', but argument 2 has type 'long int' [-Wformat]
scalable_bounded_buffer_v4.c:126:5: warning: format '%d' expects argument of typ
e 'int', but argument 2 has type 'long int' [-Wformat]
scalable_bounded_buffer_v4.c:139:3: warning: format '%d' expects argument of typ
e 'int', but argument 2 has type 'long int' [-Wformat]
```

1.2.2 bounded_buffer_simple

```
$ /usr/bin/time ./bounded_buffer_simple
Buffer size = 10, items to send = 8388608
11.15user 89.46system 0:12.58elapsed 799%CPU (0avgtext+0avgdata 4384maxresident)
k
0inputs+0outputs (0major+346minor)pagefaults 0swaps
```

Elapsed time is 12.58s

1.2.2 Scalable version of 1 thread

```
$ /usr/bin/time ./scalable_bounded_buffer -n 1 -p 0
Buffer size = 1, items to send = 8388608
20.69user 16.09system 0:23.80elapsed 154%CPU (0avgtext+0avgdata 2976maxresident)
k
0inputs+0outputs (0major+241minor)pagefaults 0swaps
```

Elapsed time is 23.80s

1.2.3 Scalable version of 8 threads

```
$ /usr/bin/time ./scalable_bounded_buffer -n 8 -p 0
Buffer size = 8, items to send = 8388608
52.17user 46.86system 0:12.78elapsed 774%CPU (0avgtext+0avgdata 3376maxresident)
k
0inputs+0outputs (0major+270minor)pagefaults 0swaps
```

Elapsed time is 12.78s

1.2.4 conclusion

The scalable version can perform better by introducing more threads. But the running environment is not always stable. The speedup between 8 threads and 1 thread below is 23.80/12.78 = 1.86.

2. Gaussian elimination

2.1 Work partition

The work is partitioned by thread indexes. Sequential Gaussian elimination is row-wise. So I distribute work by interlaced rows. If there is N threads, the 1st row, the 1+Nth row, 1+2*Nth row and other interlaced rows will be distributed to the first thread. If the row index is i, the thread which index is (i%N) will process this row.

2.2 Procedure

Every iteration of Gaussian elimination has two steps, the first step is division and the second is elimination. So my implementation has the same procedure. And extra pthread mutex and condition variables are introduced to synchronize the iteration.

2.2.1 pthread data structure

```
typedef struct
{
    pthread_mutex_t lock;    // lock
    pthread_cond_t cond;     // condition
    int count;               // thread count
} barrier_t;
```

I use this structure as barrier. The barrier is to synchronize the iteration. When the thread completes its substep, it will add 1 to count. And every thread will not continue until count = THREAD_NUM which means every thread has completed its work at current substep.

2.2.2 Procedure

The Gaussian elimination pseudocode is as follows:

```
int my_id = thr_id; // get thread id
for row_id←0 to N { // iterate all rows in matrix A
    // first step
    if (row_id % THREAD_NUM == my_id){    // find the specific row for current thread
        Division for this row;
    }
    Barrier();// wait for all threads to complete the first step

    // second step
    for subrow_id←row_id+1 to N{
        if (subrow_id%THREAD_NUM == my_id){//find the specific row for current thread
            Elimination for this row;
        }
    }
    Barrier(); // wait for all threads to complete the second step
}
```

2.3 Experiment result

Matrix A size is 4096*4096. The programs runs on a 8 cpus machine.

2.3.1 Sequential version result

```
$ /usr/bin/time ./gs -n 4096 -P 0

size      = 4096x4096
maxnum    = 15
Init      = rand
Initializing matrix...done

Command exited with non-zero status 255
132.73user 0.11system 2:12.85elapsed 99%CPU (0avgtext+0avgdata 526480maxresident
)k
0inputs+0outputs (0major+32944minor)pagefaults 0swaps
```

Elapsed time is 2 min 12.85s

2.3.2 Parallel version result (1 thread)

```
$ /usr/bin/time ./gp -n 4096 -P 0 -p 1

thread_num = 1
size       = 4096x4096
maxnum     = 15
Init       = rand
Initializing matrix...done

109.07user 0.14system 1:49.33elapsed 99%CPU (0avgtext+0avgdata 527024maxresident
)k
0inputs+0outputs (0major+32985minor)pagefaults 0swaps
```

Elapsed time is 1min 49.33s

2.3.3 Parallel version result (8 threads)

```
$ /usr/bin/time ./gp -n 4096 -P 0 -p 8

thread_num = 8
size       = 4096x4096
maxnum     = 15
Init       = rand
Initializing matrix...done

346.14user 0.75system 0:44.83elapsed 773%CPU (0avgtext+0avgdata 527232maxresiden
t)k
0inputs+0outputs (0major+33001minor)pagefaults 0swaps
```

Elapsed time is 44.83s

2.3.4 Conclusion

The parallel version could improve its result with more running threads. The speedup between 8 threads and 1 thread is 55.10/44.83 = 2.438.