Parallel BFS with Push/Pull operation using queue & bitmap

REPORT

Push operation is implemented for BFS using 2 approaches of queuing:

1. Local queue for each thread 2. Shared queue for all threads

Pull operations is implemented for BFS using bitmaps.

Push operation => Top-down approach

Pull operation => Bottom-up approach

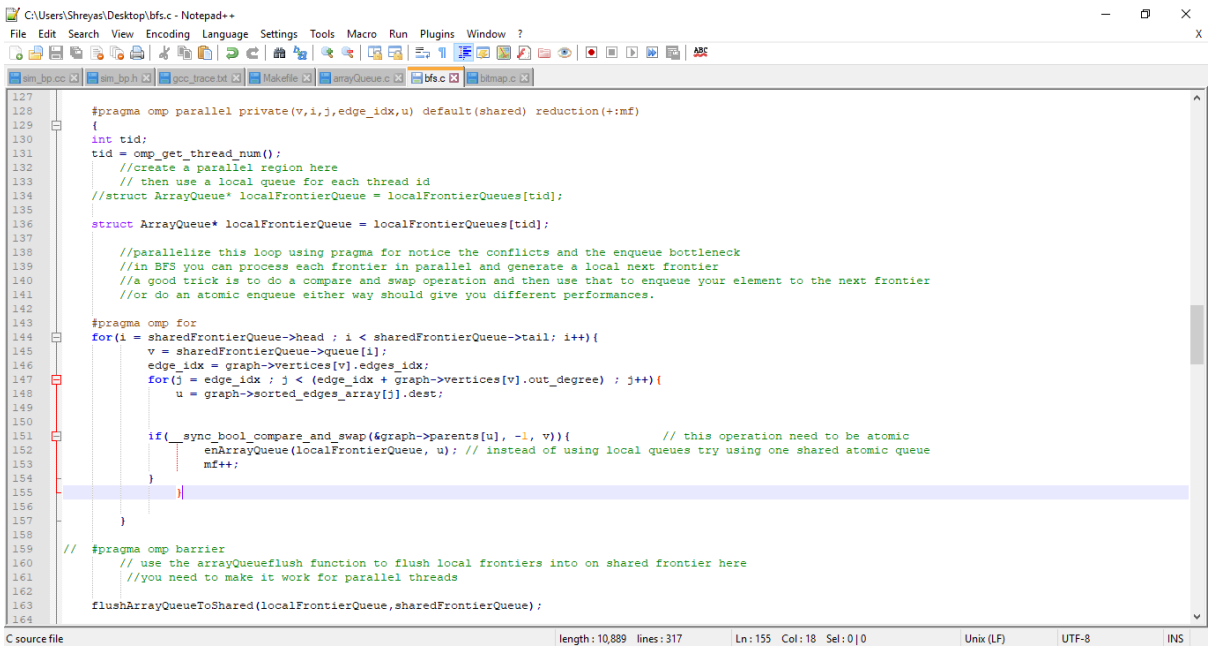
In each approach, 4 types of atomic approach are done:

OMP atomic, Locks, Critical section, GCC built-ins.

PUSH OPERATION – LOCAL QUEUES:

File: Bfs.c

Function: topDownStepGraphCSR



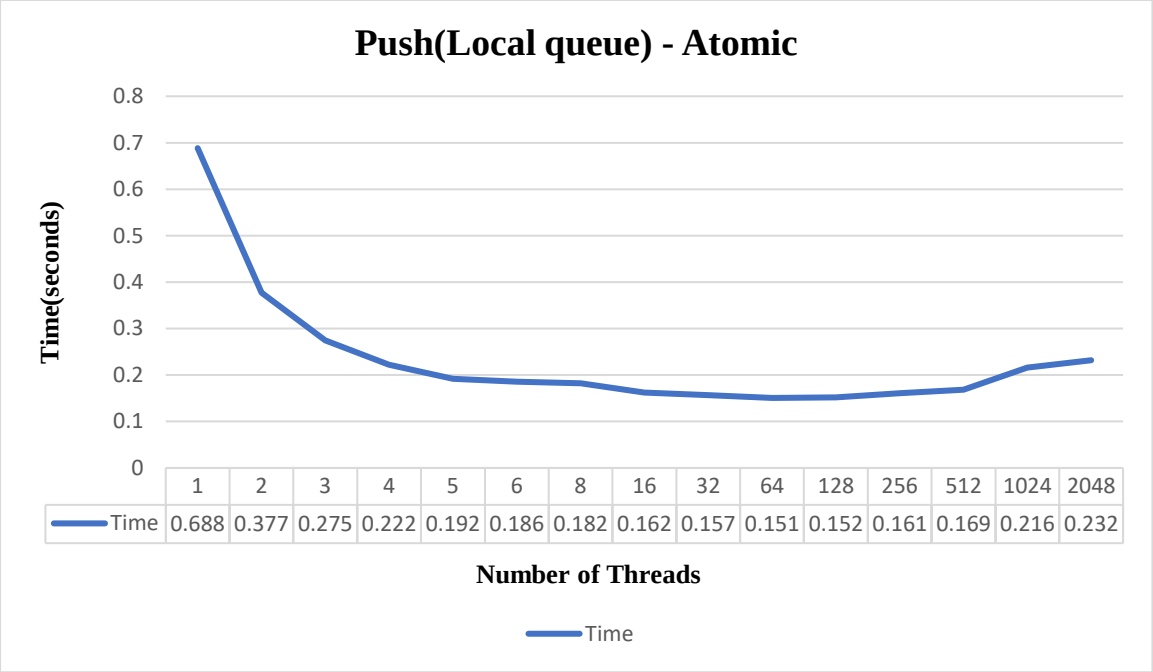
Atomic:

File: enArrayQueue.c

Function: flushArrayQueueToShared

```
void flushArrayQueueToShared(struct ArrayQueue *local_q, struct ArrayQueue *shared_q){


/*++++++++++++++++++++++++++++++++++ For OpenMp Atomic ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
__u32 shared_q_tail_next;
#pragma omp atomic capture
{
    shared_q_tail_next = shared_q->tail_next;
    shared_q->tail_next += local_q->tail;
}
```



Thus, using a local queue, Array enqueuing is performed by parallelizing the top-down approach in BFS push. As the number of threads is increased, the time taken for the process to complete decreases till a certain point beyond which the time again increases (due to higher context switching, cache pollution). Post computing the elements of each local queue, the flushing operation to the shared queue is performed atomic. In order to avoid race conditions, the instructions involving shared variables are performed atomically, that is, only one thread will execute that section at a time.

Time taken for 1 thread = 0.688s Time taken for 2 threads = 0.377s

Speed-up achieved = 1.82

The process is now repeated for other 3 approaches – Locks, critical, gcc built-in commands.

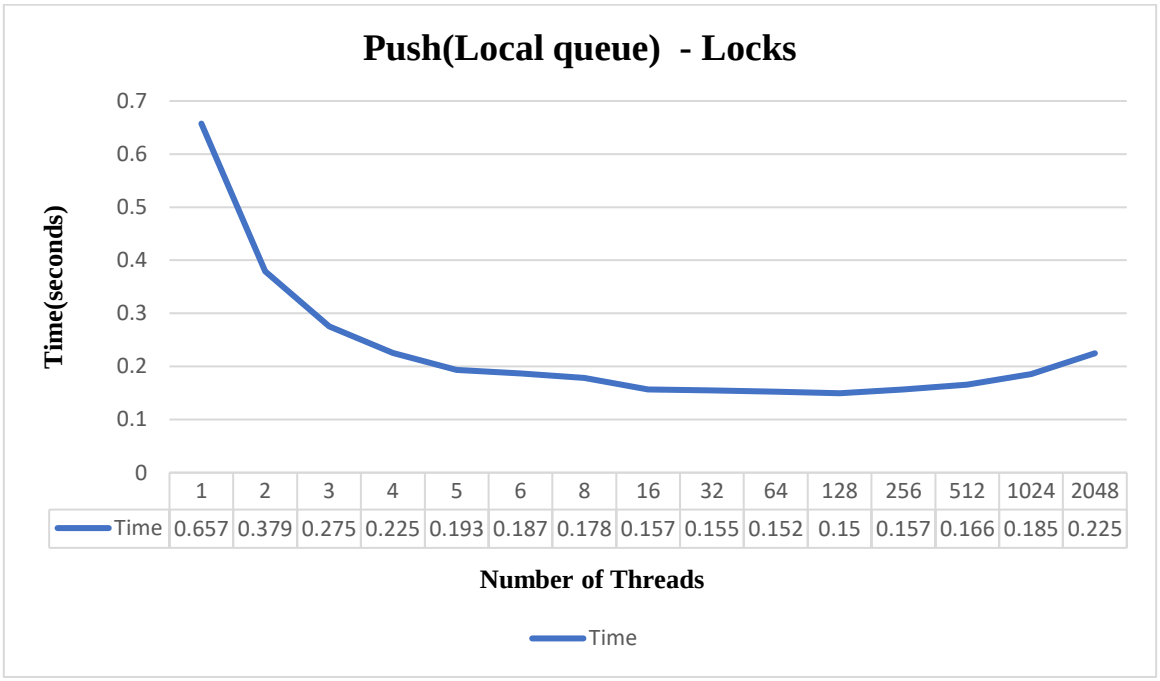
Locks:

File: enArrayQueue.c

Function: flushArrayQueueToShared

```
/*+++++++++++++++++++++++++++++++++++ For OpenMp locks ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
/* Defined the lock globally in arrayQueue.h & initialized the lock in bfs.c */

omp_set_lock(&writelock);
    __u32 shared_q_tail_next = shared_q->tail_next;
    shared_q->tail_next += local_q->tail;
omp_unset_lock(&writelock);
```



Time taken for 1 thread = 0.657s Time taken for 2 threads = 0.379s

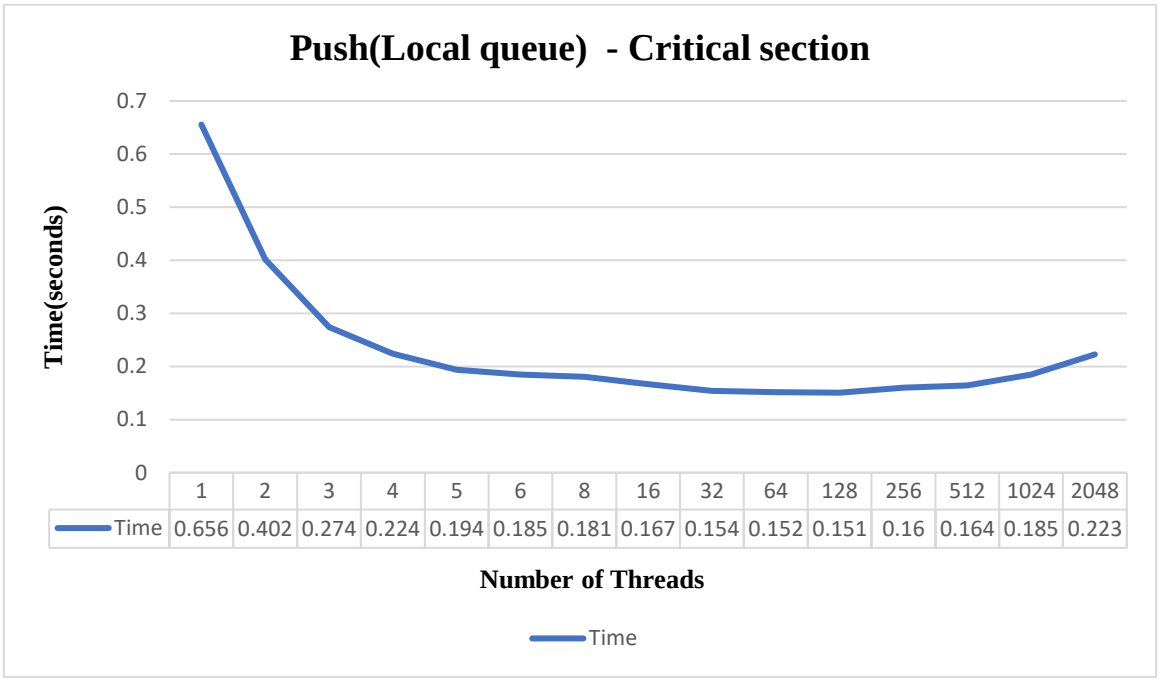
Speed-up achieved = 1.73

Critical section:

File: enArrayQueue.c

Function: flushArrayQueueToShared

```
/*+++++++++++++++++++++++++++++++++++ For critical ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
__u32 shared_q_tail_next;
#pragma omp critical
{
    shared_q_tail_next = shared_q->tail_next;
    shared_q->tail_next += local_q->tail;
}
```



Time taken for 1 thread = 0.656s Time taken for 2 threads = 0.402s

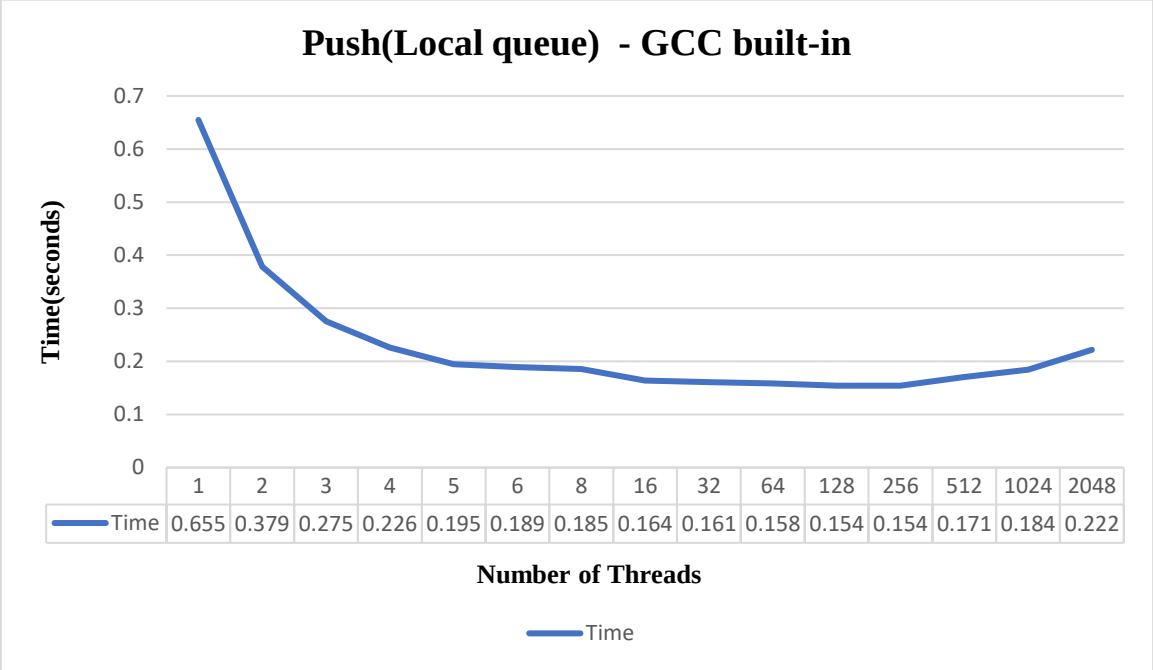
Speed-up achieved = 1.63

GCC Built-ins:

File: enArrayQueue.c

Function: flushArrayQueueToShared

```
/*+++++++++++++++++++++++++++++++++++ For gcc built ins +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
_u32 shared_q_tail_next = __sync_fetch_and_add(&shared_q->tail_next,local_q->tail);
```



Time taken for 1 thread = 0.655s Time taken for 2 threads = 0.379s

Speed-up achieved = 1.72

Thus, on observing the speed-ups achieved for various approaches, it is found that the OMP atomic performs the best and critical achieves the lowest speed-up.

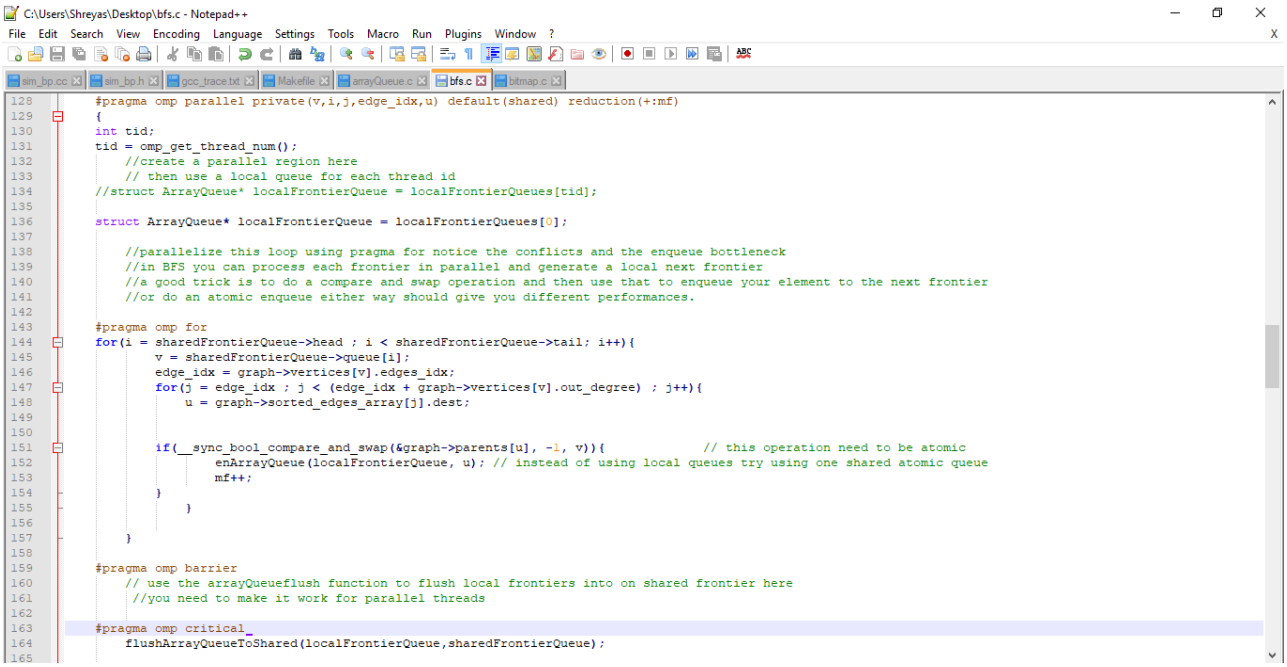
Critical has more overhead than atomic since on using a critical keyword, the entire section is locked for a single thread and allows concurrent threads to execute only post the completion of execution by previous thread. While in atomic, only the memory update in the next instruction is performed atomically. Race conditions are avoided through direct control of concurrent threads that might read or write to or from that particular memory location.

PUSH OPERATION – SHARED QUEUE:

Push operation in BFS is now done using a single shared queue instead of multiple local-queues for each thread.

File: Bfs.c

Function: topDownStepGraphCSR



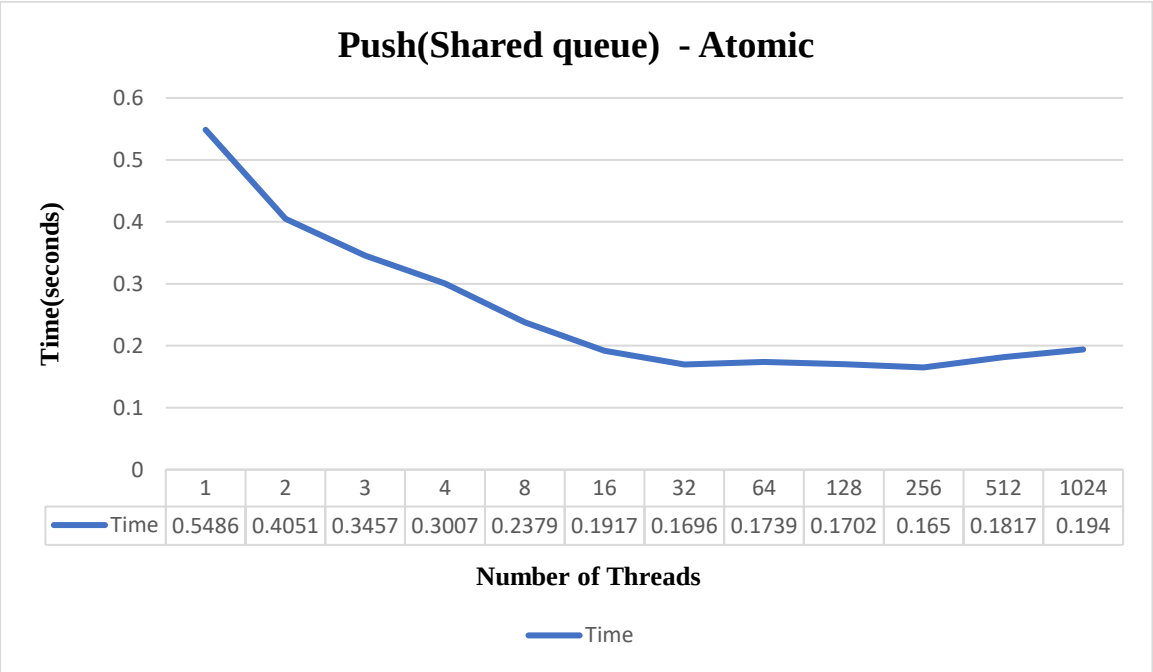
Atomic:

File: ArrayQueue.c

Function: enArrayQueueAtomic

```
void enArrayQueueAtomic (struct ArrayQueue *q, __u32 k){


/*+++++++++++++++++++++++++++++++++++ For OpenMp Atomic ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
    __u32 temp;
    #pragma omp atomic capture
        temp = q->tail++;
    q->queue[temp] = k;
```



Time taken for 1 thread = 0.5486s Time taken for 2 threads = 0.4051s

Speed-up achieved = 1.35

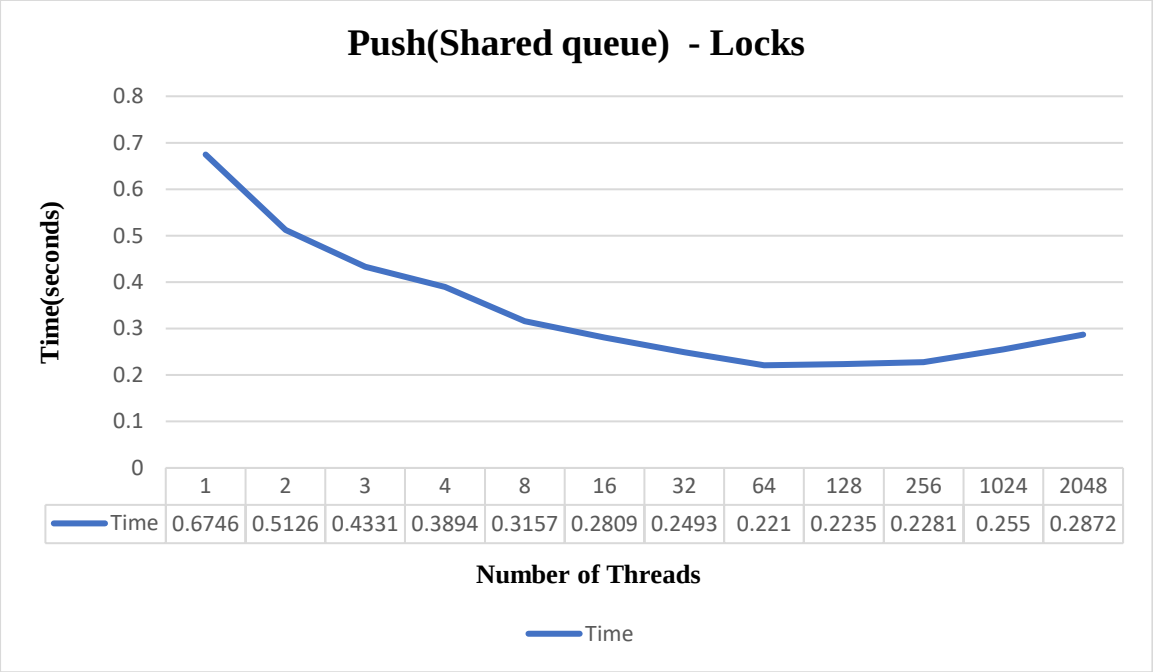
Here, we observe that the speed-up achieved is lower than that obtained using local-queues. This is because of using a single shared-queue for all threads and thus each thread needs to wait on updating for each frontier. Thus, the context switching is higher on providing access to the shared queue for multiple threads. Race conditions are avoided by ensuring only one thread access the shared queue at a time.

Locks:

File: ArrayQueue.c

Function: enArrayQueueAtomic

```
/*+++++++++++++++++++++++++++++++++++ For OpenMp locks ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
    omp_set_lock(&writelock);
        q->queue[q->tail] = k;
        q->tail = (q->tail+1)%q->size;
        q->tail_next = q->tail;
    omp_unset_lock(&writelock);*/
```



Time taken for 1 thread = 0.6746s Time taken for 2 threads = 0.5126s

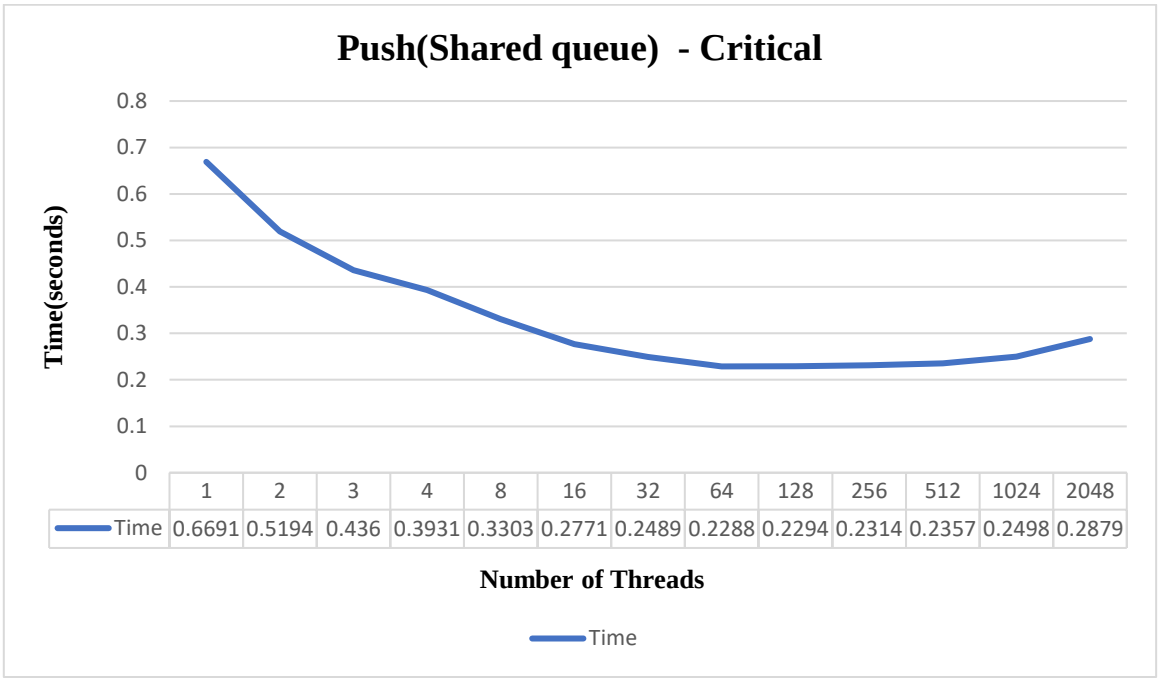
Speed-up achieved = 1.32

Critical Section:

File: ArrayQueue.c

Function: enArrayQueueAtomic

```
/*++++++++++++++++++++++++++++++++++ For critical ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
    #pragma omp critical
    {
        q->queue[q->tail] = k;
        q->tail = (q->tail+1)%q->size;
        q->tail_next = q->tail;
    }
```



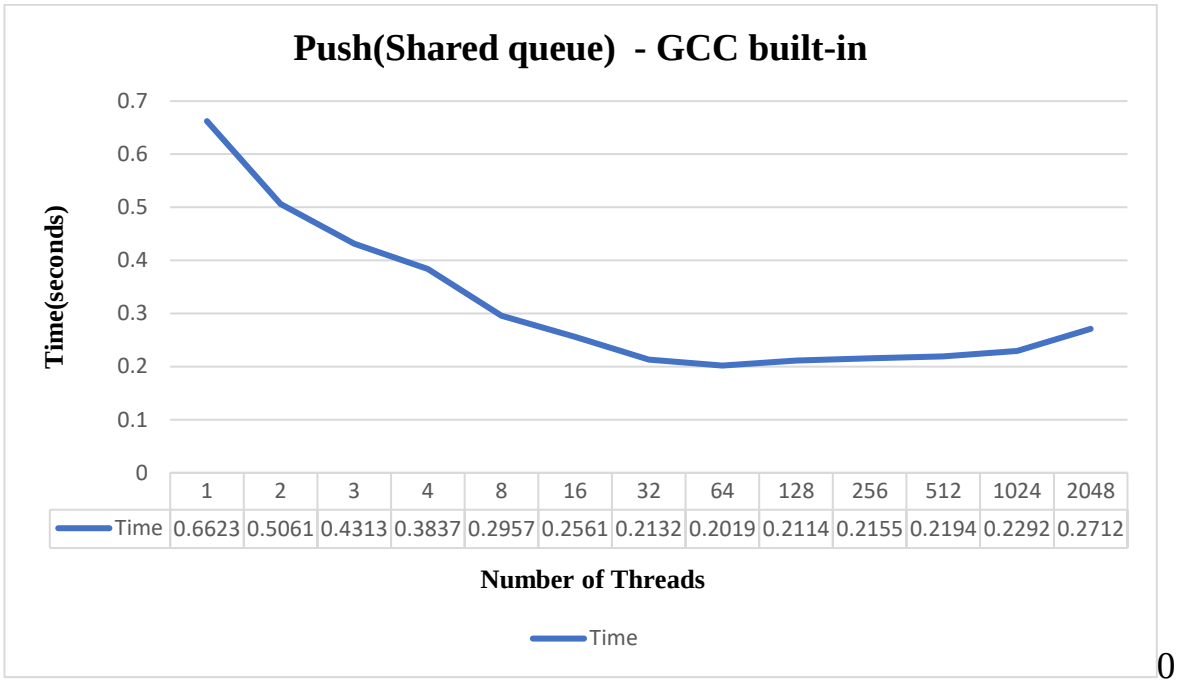
Time taken for 1 thread = 0.6691s Time taken for 2 threads = 0.5194s

Speed-up achieved = 1.28

GCC built-ins:

File: ArrayQueue.c

Function: enArrayQueueAtomic



0

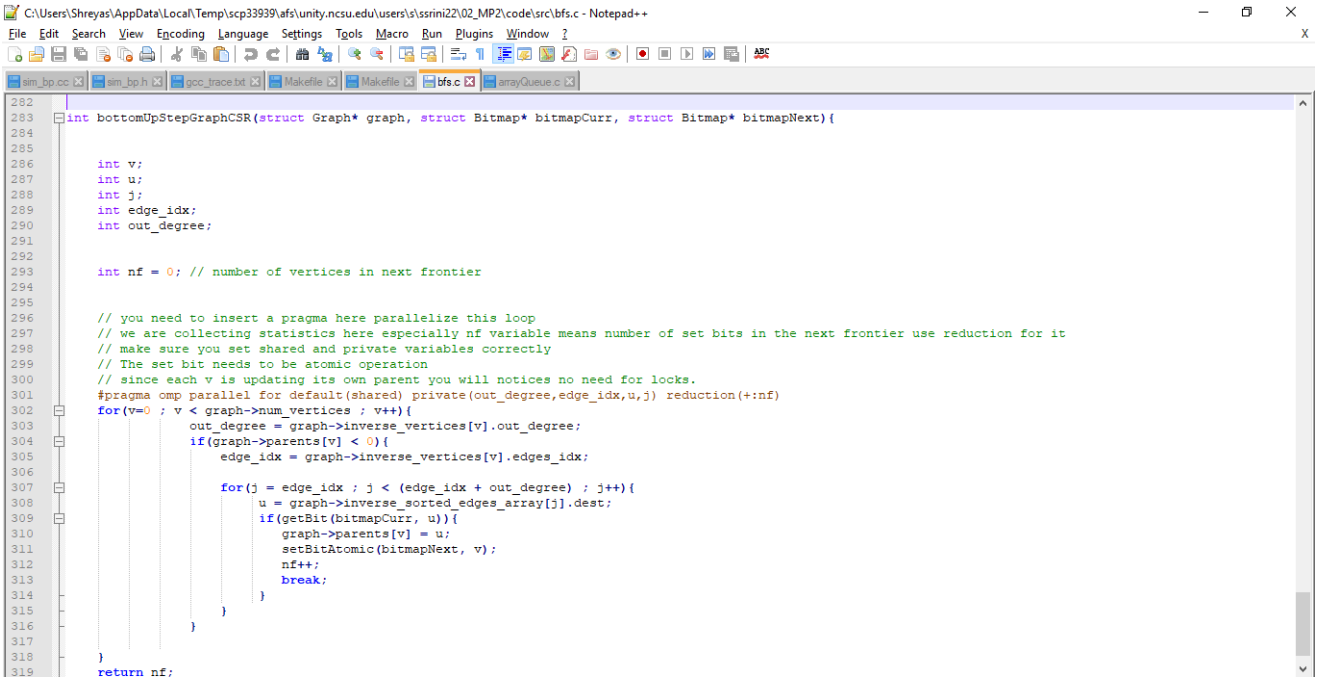
Time taken for 1 thread = 0.6623s Time taken for 2 threads = 0.5061s

Speed-up achieved = 1.31

Similar the Push-local queue approach, OMP atomic performs the best while critical achieves the lowest speed-up, though the achieved speed-up is lower than that using local queue for each approach.

PULL OPERATION:

File: bottomUpStepGraphCSR.c



Atomic:

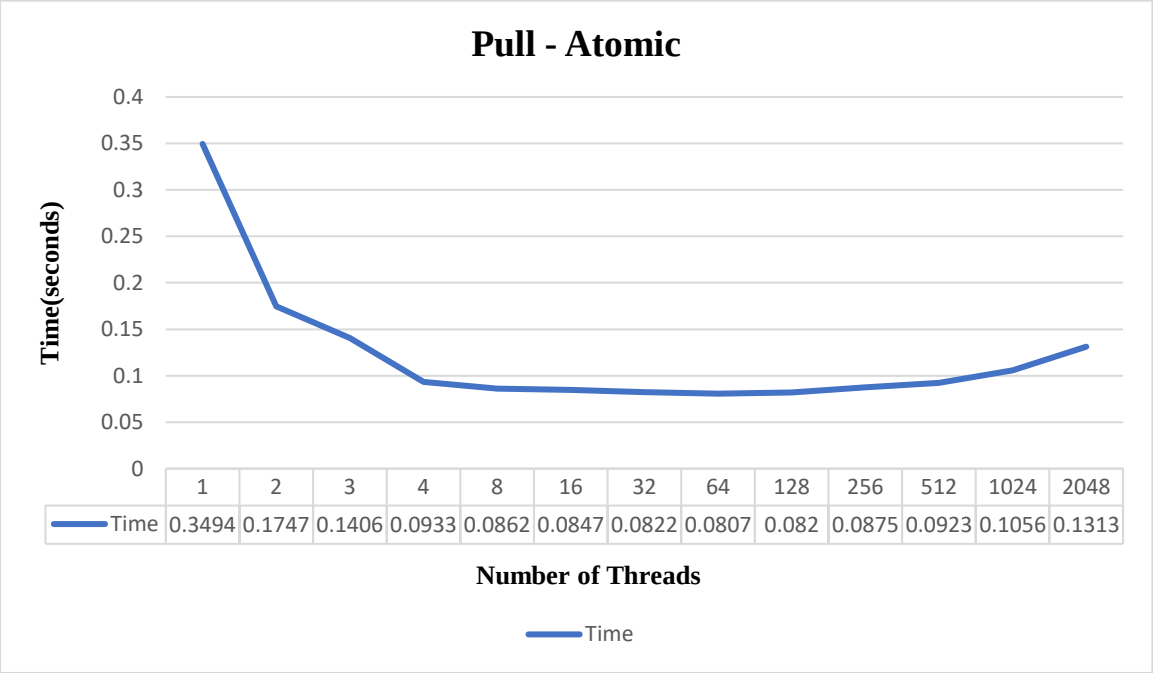
File: bitmap.c

Function: enArrayQueueAtomic

```
void setBitAtomic(struct Bitmap* bitmap, __u32 pos){

/*+++++++++++++++++++++++++++++++++++ For OpenMp Atomic ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/

#pragma omp atomic
    bitmap->bitarray[word_offset(pos)] |= (__u32) (1 << bit_offset(pos));
```



Time taken for 1 thread = 0.3494s Time taken for 2 threads = 0.1747s

Speed-up achieved = 2.0

Using bitmaps, bitset is done for corresponding frontier values and the process is continued in bottom-up traversal. As the number of threads is increased, the time taken for the process to complete decreases till a certain point beyond which the time again increases

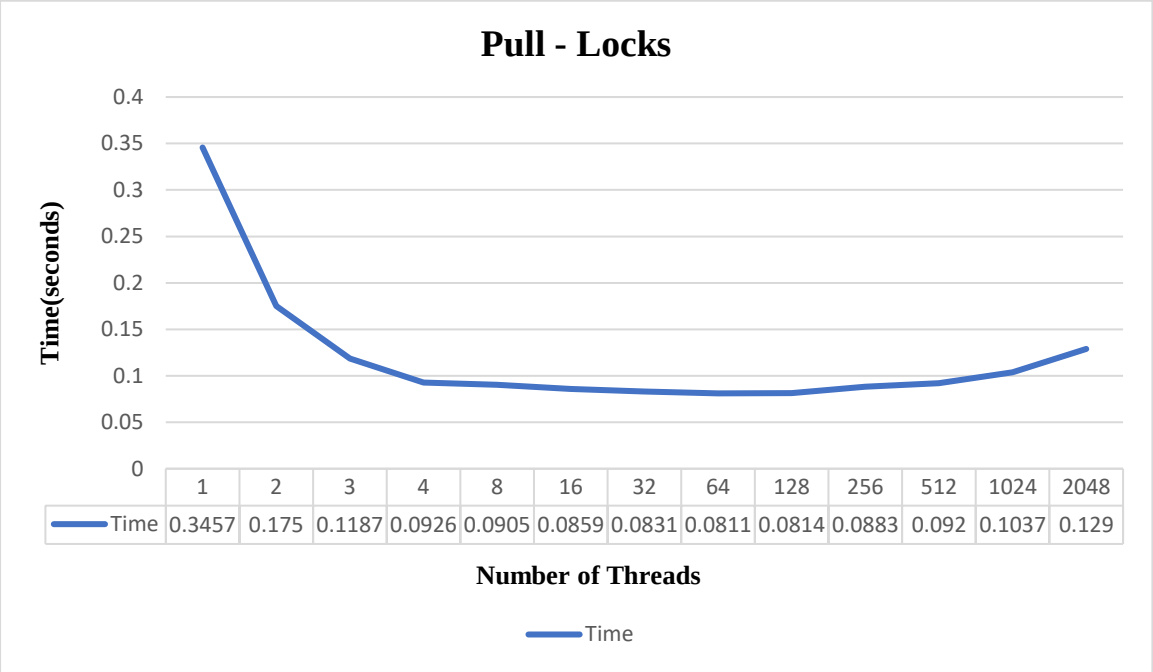
Locks:

File: bitmap.c

Function: enArrayQueueAtomic

```
omp_lock_t writelock;
    omp_init_lock(&writelock);

omp_set_lock(&writelock);
bitmap->bitarray[word_offset(pos)] |= (__u32) (1 << bit_offset(pos));
omp_unset_lock(&writelock);
```



Time taken for 1 thread = 0.3457s Time taken for 2 threads = 0.175s

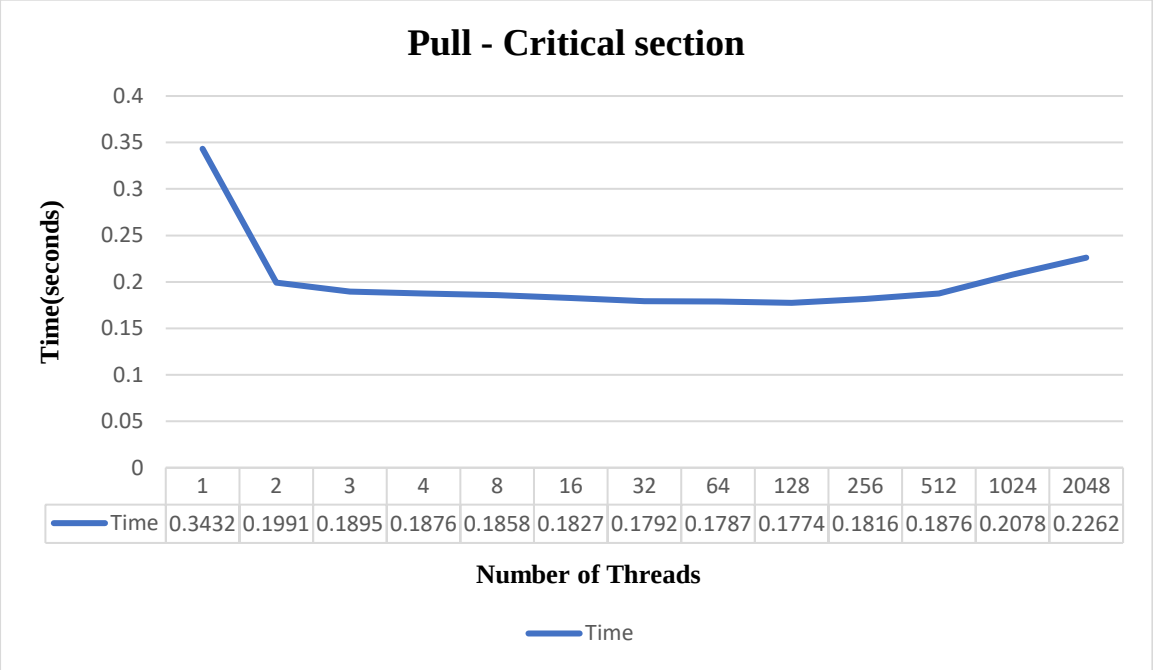
Speed-up achieved = 1.975

Critical section:

File: bitmap.c

Function: enArrayQueueAtomic

```
/*+++++++++++++++++++++++++++++++++++ For critical ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
#pragma omp critical
{
    bitmap->bitarray[word_offset(pos)] |= (__u32) (1 << bit_offset(pos));
}
```



Time taken for 1 thread = 0.3432s Time taken for 2 threads = 0.1991s

Speed-up achieved = 1.72

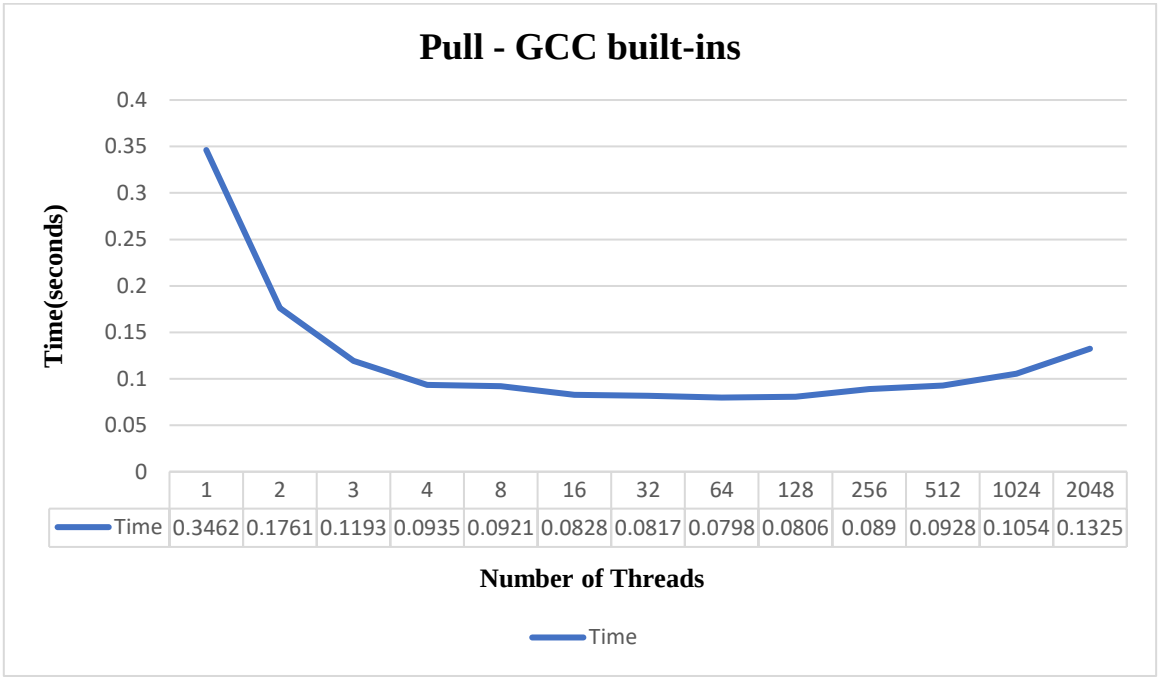
GCC built-ins:

File: bitmap.c

Function: enArrayQueueAtomic

```
/*++++++++++++++++++++++++++++++++++++ For gcc built ins ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
    __sync_or_and_fetch(&(bitmap->bitarray[word_offset(pos)]),(1 << bit_offset(pos)));

}
```



Time taken for 1 thread = 0.3462s Time taken for 2 threads = 0.1761s

Speed-up achieved = 1.965

Thus, the performance of each approach is measured and on calculating the speed-up, it is found that OMP atomic provides the best performance and critical provides the least. This is due to the higher overhead occurring during operation in critical section.