

# **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

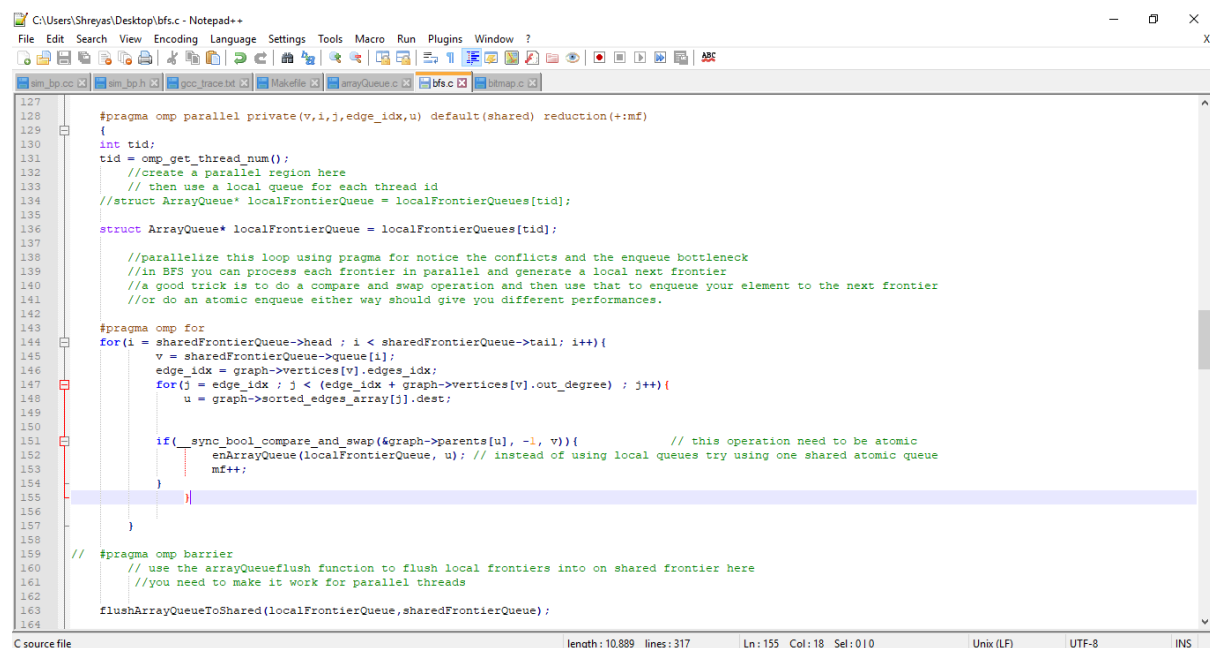
In each approach, 4 types of atomic approach are done:

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

## PUSH OPERATION – LOCAL QUEUES:

**File: Bfs.c**

**Function: topDownStepGraphCSR**



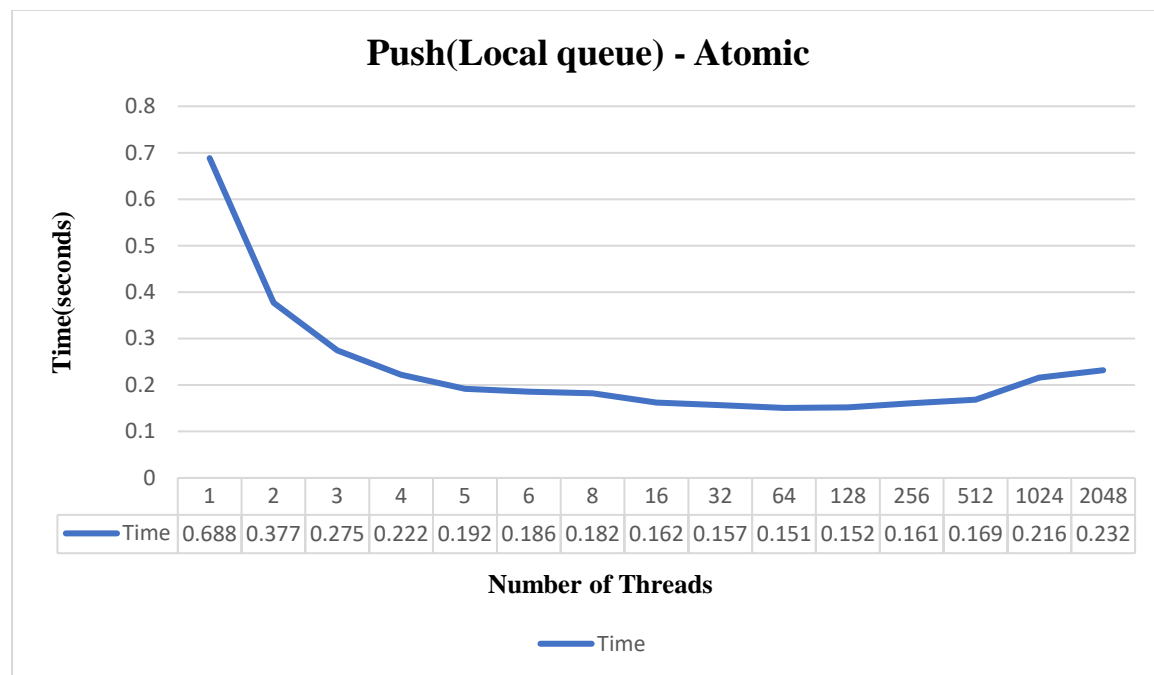
```
127 #pragma omp parallel private(v,i,j,edge_idx,u) default(shared) reduction(+:mf)
128 {
129     int tid;
130     tid = omp_get_thread_num();
131     //create a parallel region here
132     // then use a local queue for each thread id
133     //struct ArrayQueue* localFrontierQueue = localFrontierQueues[tid];
134     struct ArrayQueue* localFrontierQueue = localFrontierQueues[tid];
135
136     //parallelize this loop using pragma for notice the conflicts and the enqueue bottleneck
137     //in BFS you can process each frontier in parallel and generate a local next frontier
138     //a good trick is to do a compare and swap operation and then use that to enqueue your element to the next frontier
139     //or do an atomic enqueue either way should give you different performances.
140
141     #pragma omp for
142     for(i = sharedFrontierQueue->head; i < sharedFrontierQueue->tail; i++){
143         v = sharedFrontierQueue->queue[i];
144         edge_idx = graph->vertices[v].edges_idx;
145         for(j = edge_idx; j < (edge_idx + graph->vertices[v].out_degree); j++){
146             u = graph->sorted_edges_array[j].dest;
147
148             if(!_sync_bool_compare_and_swap(&graph->parents[u], -1, v)){ // this operation need to be atomic
149                 enqueueArrayQueue(localFrontierQueue, u); // instead of using local queues try using one shared atomic queue
150                 mf++;
151             }
152         }
153     }
154
155     // #pragma omp barrier
156     // use the arrayQueueflush function to flush local frontiers into on shared frontier here
157     //you need to make it work for parallel threads
158
159     flushArrayQueueToShared(localFrontierQueue, sharedFrontierQueue);
160
161
162
163
164
```

## 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 enqueueing 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

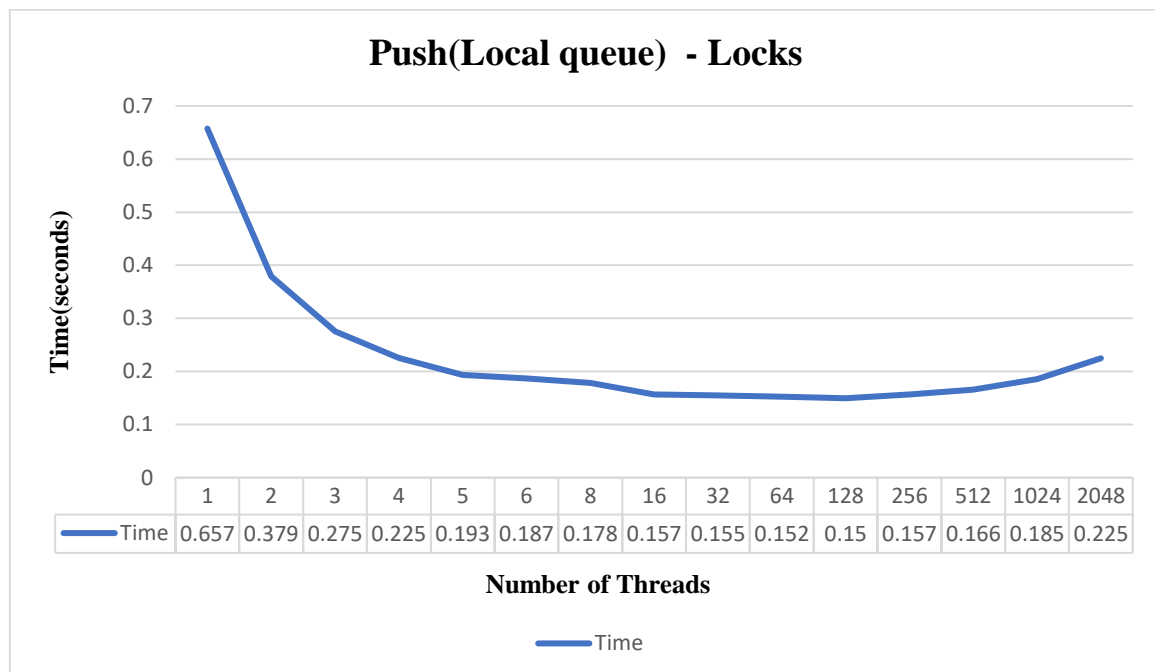
The process is now repeated for other 3 approaches – Locks, critical, gcc built-in commands.

## 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

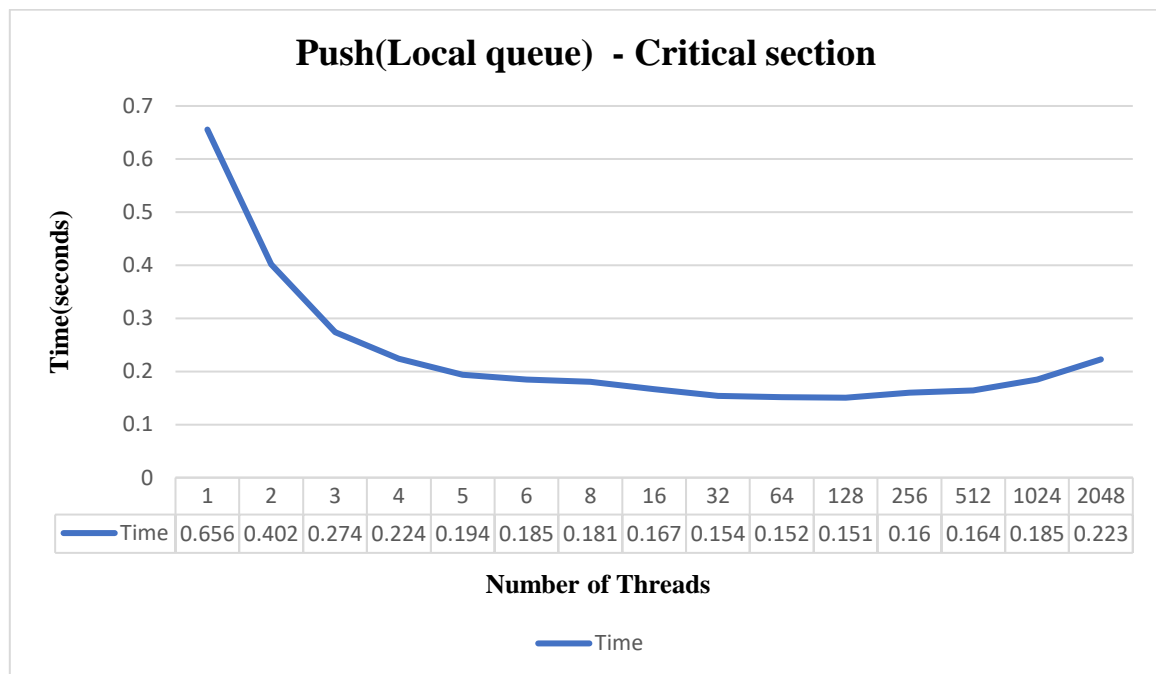
Speed-up achieved = 1.73

## 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

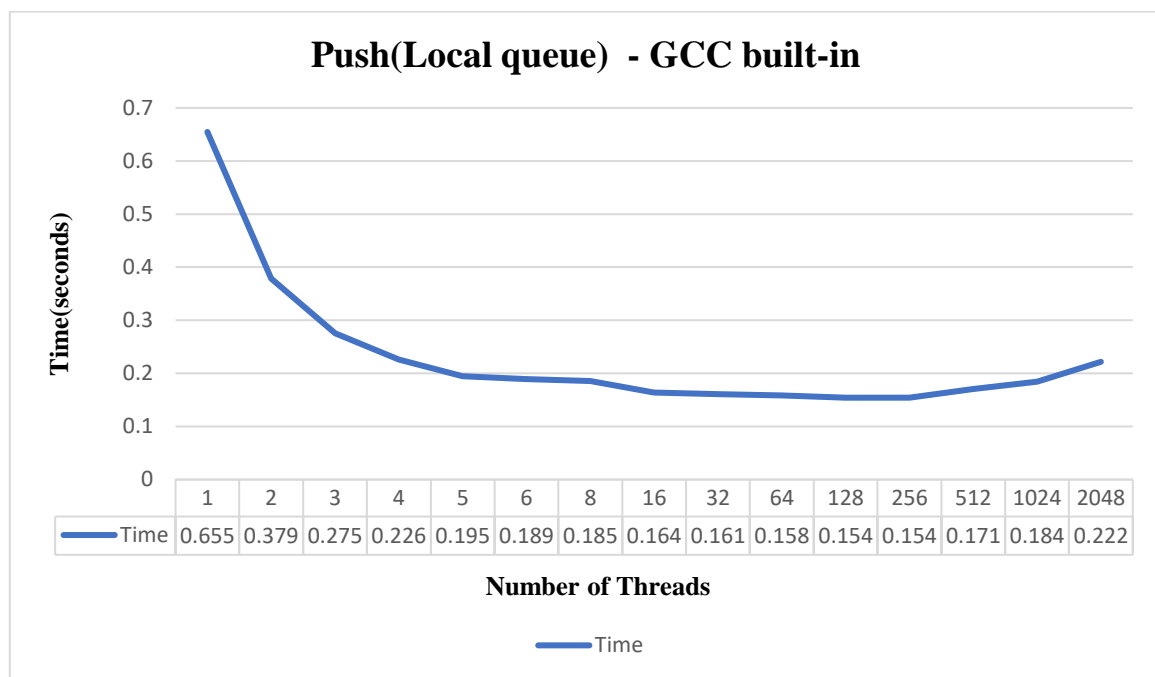
Speed-up achieved = 1.63

## 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.

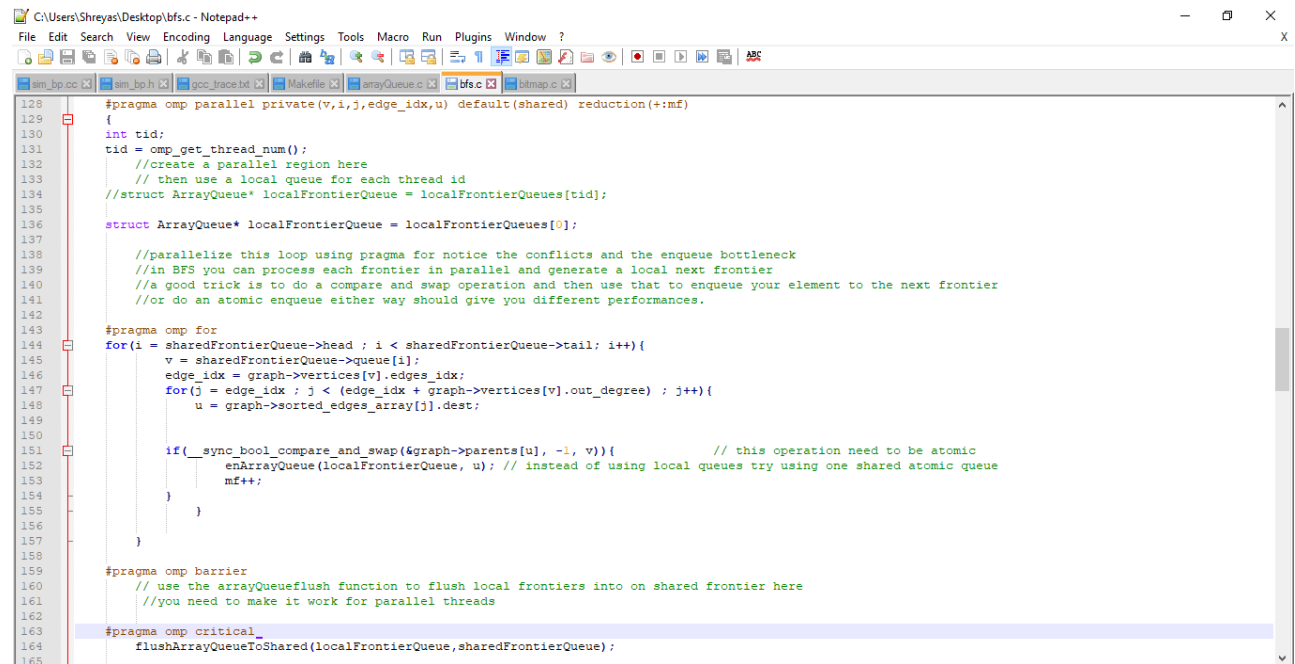
Critical has more overhead than atomic since on using a critical keyword, the entire section is locked for a single thread and allows concurrent threads to execute only post the completion of execution by previous thread. While in atomic, only the memory update in the next instruction is performed atomically. 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**



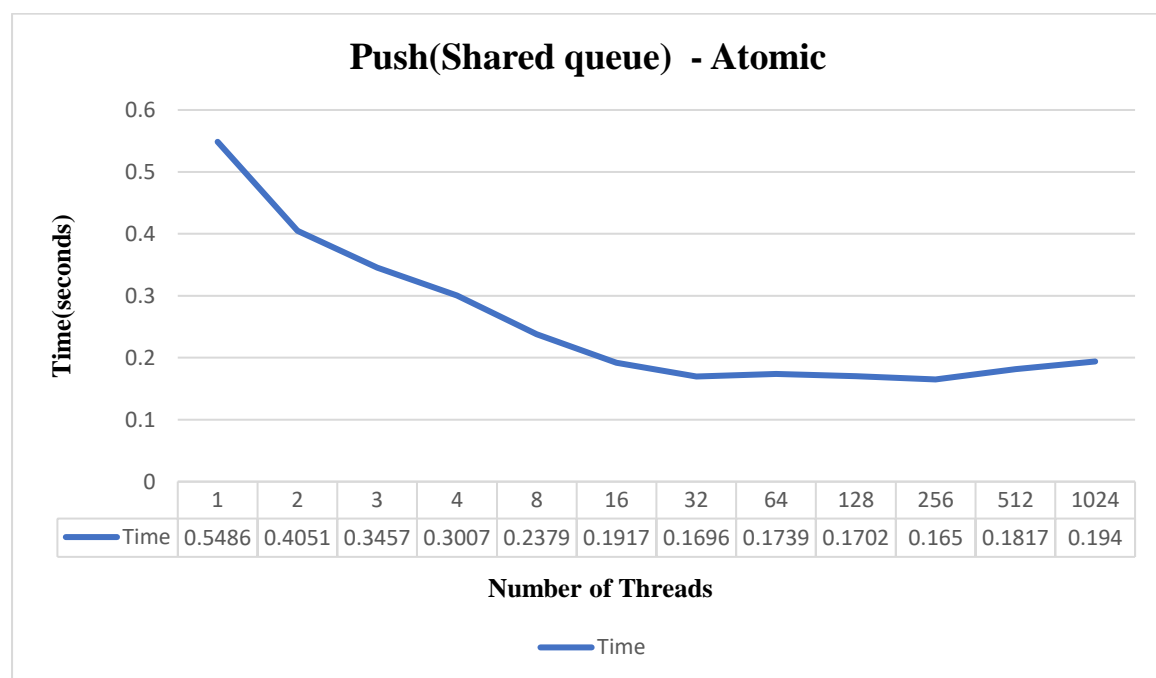
```
128 #pragma omp parallel private(v,i,j,edge_idx,u) default(shared) reduction(+:mf)
129 {
130     int tid;
131     tid = omp_get_thread_num();
132     //create a parallel region here
133     // then use a local queue for each thread id
134     struct ArrayQueue* localFrontierQueue = localFrontierQueues[tid];
135
136     struct ArrayQueue* localFrontierQueue = localFrontierQueues[0];
137
138     //parallelize this loop using pragma for notice the conflicts and the enqueue bottleneck
139     //in BFS you can process each frontier in parallel and generate a local next frontier
140     //a good trick is to do a compare and swap operation and then use that to enqueue your element to the next frontier
141     //or do an atomic enqueue either way should give you different performances.
142
143     #pragma omp for
144     for(i = sharedFrontierQueue->head; i < sharedFrontierQueue->tail; i++){
145         v = sharedFrontierQueue->queue[i];
146         edge_idx = graph->vertices[v].edges_idx;
147         for(j = edge_idx; j < (edge_idx + graph->vertices[v].out_degree); j++){
148             u = graph->sorted_edges_array[j].dest;
149
150             if(_sync_bool_compare_and_swap(&graph->parents[u], -1, v)){ // this operation need to be atomic
151                 enArrayQueue(localFrontierQueue, u); // instead of using local queues try using one shared atomic queue
152                 mf++;
153             }
154         }
155     }
156
157     #pragma omp barrier
158     // use the arrayQueueFlush function to flush local frontiers into on shared frontier here
159     //you need to make it work for parallel threads
160     #pragma omp critical_
161     flushArrayQueueToShared(localFrontierQueue, sharedFrontierQueue);
162
163     #pragma omp critical_
164     flushArrayQueueToShared(localFrontierQueue, sharedFrontierQueue);
165 }
```

## 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.

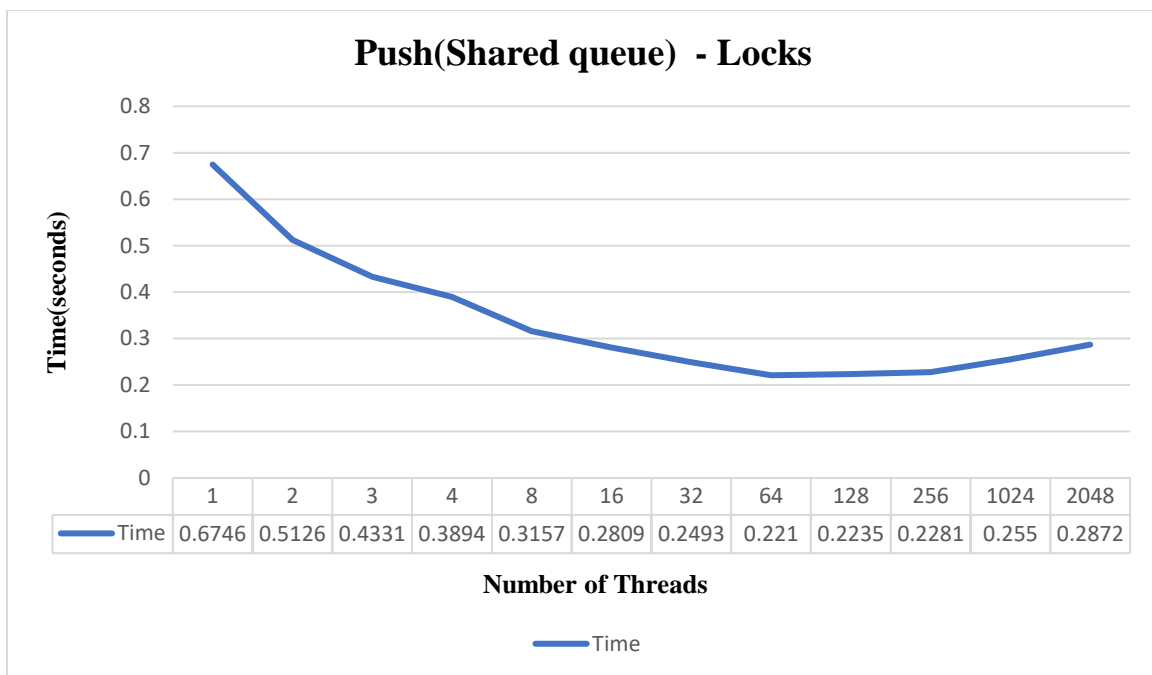


## Locks:

**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

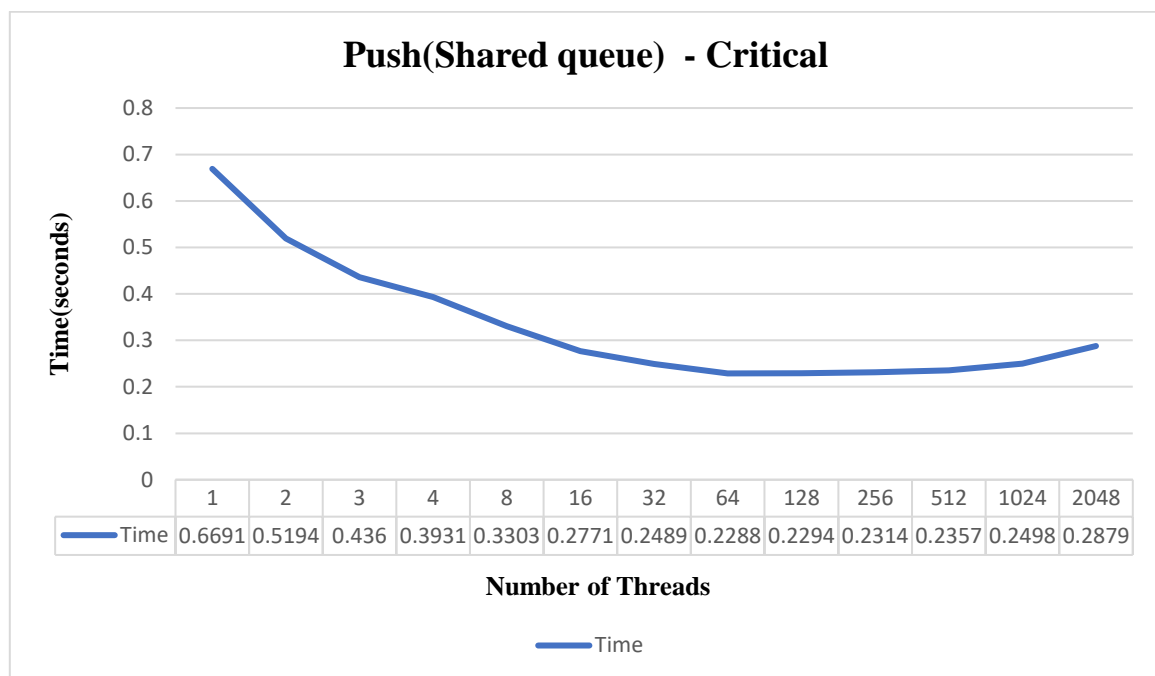
## Critical Section:

**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

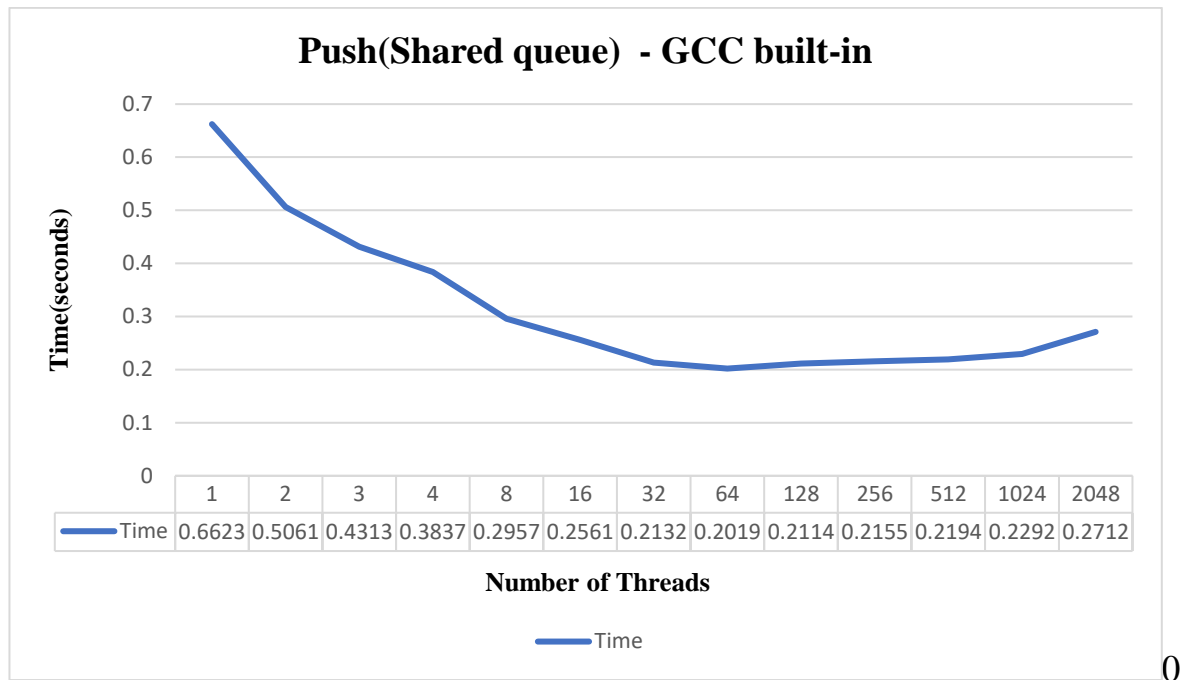
Time taken for 2 threads = 0.5194s

Speed-up achieved = 1.28

## GCC built-ins:

**File:** ArrayQueue.c

**Function:** enArrayQueueAtomic



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

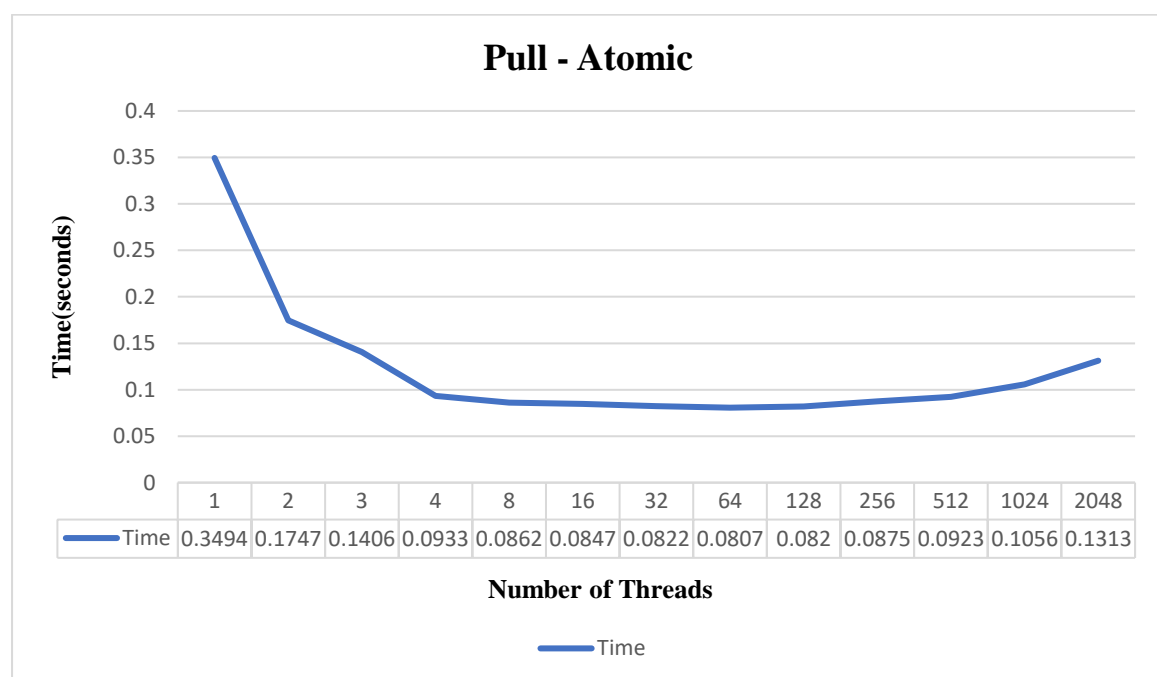
```
C:\Users\Shreyas\AppData\Local\Temp\scp33939\afs\unity.ncsu.edu\users\s\ssrini22\02_MP2\code\src\bfs.c - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
sim_bp.cc sim_bp.h gcc_trace.txt Makefile bfs.c arrayQueue.c
282 int bottomUpStepGraphCSR(struct Graph* graph, struct Bitmap* bitmapCurr, struct Bitmap* bitmapNext){
283
284
285
286     int v;
287     int u;
288     int j;
289     int edge_idx;
290     int out_degree;
291
292
293     int nf = 0; // number of vertices in next frontier
294
295
296     // you need to insert a pragma here parallelize this loop
297     // we are collecting statistics here especially nf variable means number of set bits in the next frontier use reduction for it
298     // make sure you set shared and private variables correctly
299     // The set bit needs to be atomic operation
300     // since each v is updating its own parent you will notices no need for locks.
301     #pragma omp parallel for default(shared) private(out_degree,edge_idx,u,j) reduction(+:nf)
302     for(v=0 ; v < graph->num_vertices ; v++){
303         out_degree = graph->inverse_vertices[v].out_degree;
304         if(graph->parents[v] < 0){
305             edge_idx = graph->inverse_vertices[v].edges_idx;
306
307             for(j = edge_idx ; j < (edge_idx + out_degree) ; j++){
308                 u = graph->inverse_sorted_edges_array[j].dest;
309                 if(getBit(bitmapCurr, u)){
310                     graph->parents[v] = u;
311                     setBitAtomic(bitmapNext, v);
312                     nf++;
313                     break;
314                 }
315             }
316         }
317     }
318     return nf;
319 }
```

## 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

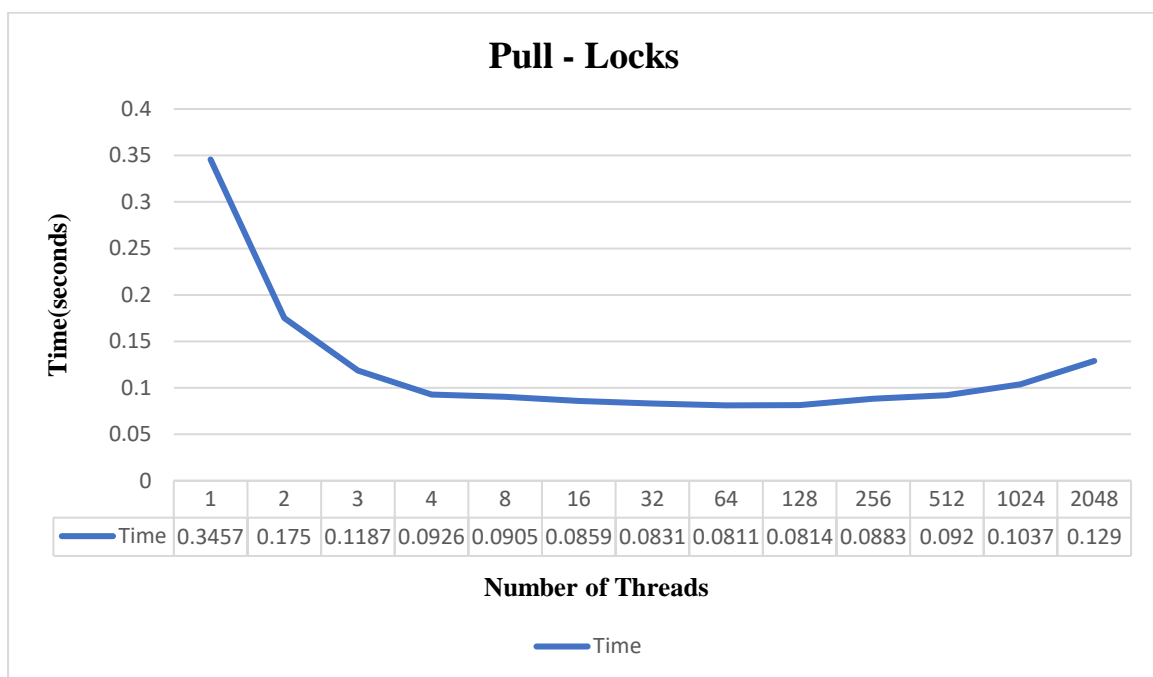
## Locks:

**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

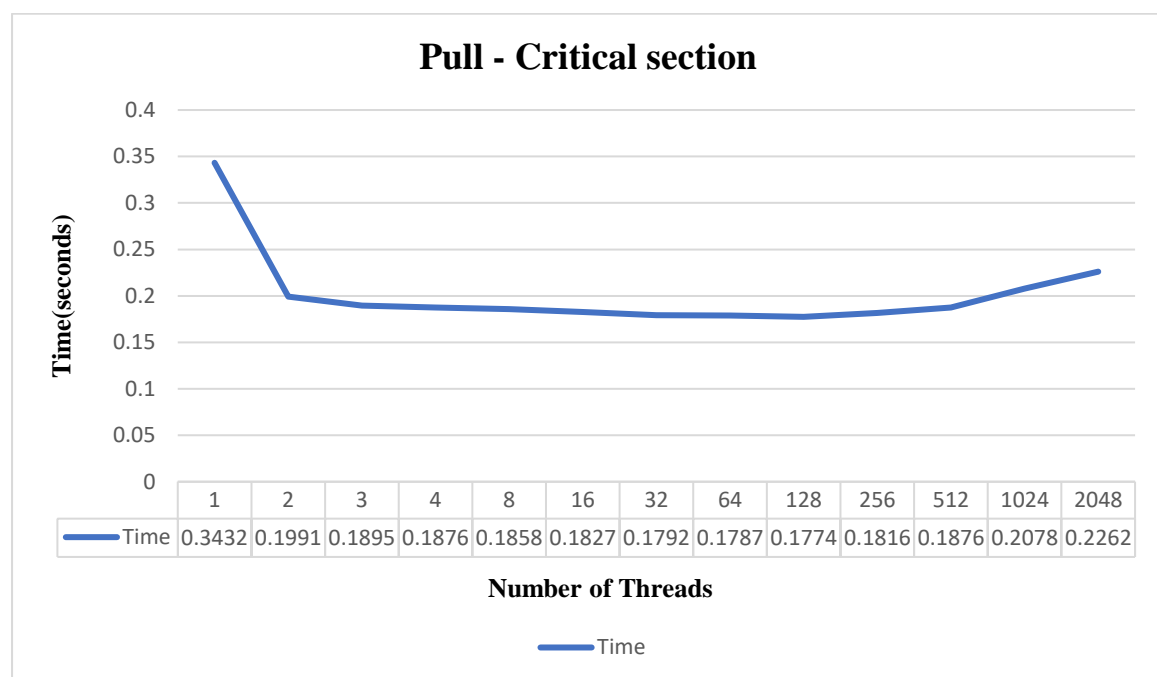
Speed-up achieved = 1.975

## 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

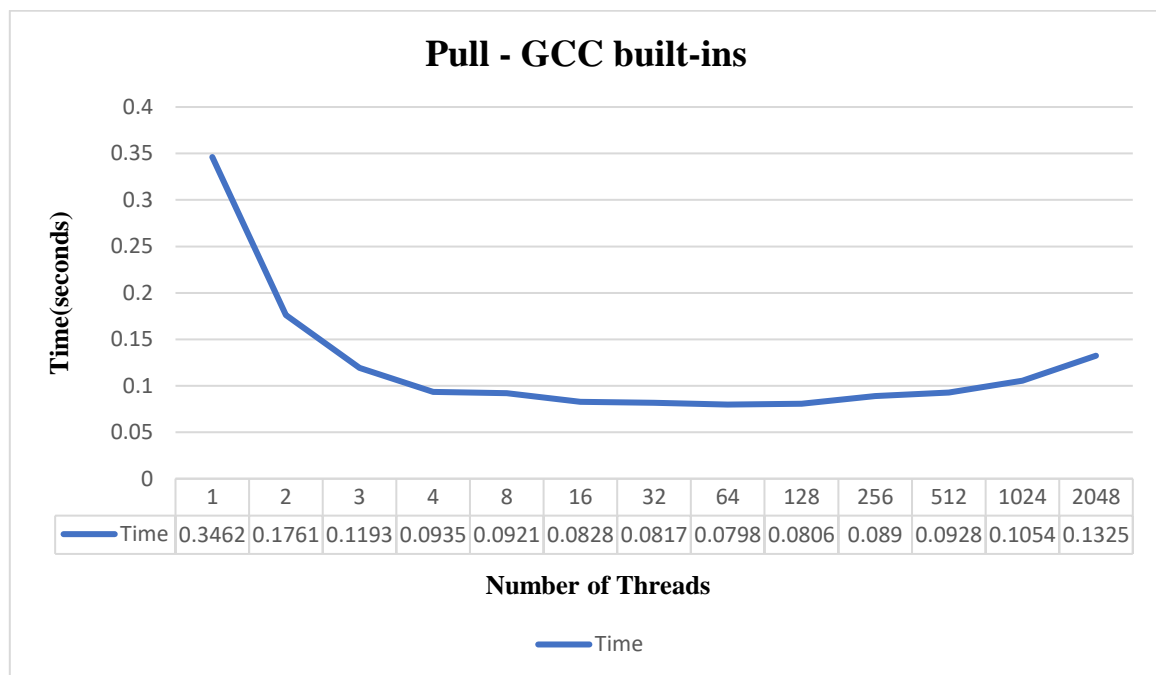
Speed-up achieved = 1.72

## 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.