PCP Project Report
Work Stealing Dequeues

By: Jatin Sachdeva - CS19B1013 & Abhijeet Wankhade - CS19B1028

Introduction

Work Stealing as its name goes is a method in which when a thread runs out of work tries to steal work from other threads. In this approach, each thread maintains a double ended queue (DEQueue) which keeps a pool of tasks stored inside it. This DEQueue provides 3 important methods i.e popTop(), popBottom(), and pushBottom().
When a thread is provided with some task it registers this task into its double ended queue by applying pushBottom() on the queue. When a thread decides to work on some task it calls popBottom() to remove that task from its DEQueue. Finally when a thread has no more tasks stored in its DEQueue, it becomes a thief and chooses some other thread to be a victim. This thief steals a task from the victim's DEQueue by calling popTop() on it.
Ideally the pop method is linearizable in nature whose pop methods always return a task if available. In practice it makes more sense to make the thief retry the popTop() operation on a different, randomly chosen DEQueue each time. To

support this idea popTop() may return null if it conflicts with concurrent popTop() calls.

Threads simply runs the task if tasks exist in its DEQueue. But if there are no more tasks than a thread becomes a thief and chooses a random victim:

```
while (task == nullptr && !hasPushedTask && count > 0)
{
    std::this_thread::yield();
    int victim = rand() % length;
    while (victim == me)
    {
        victim = rand() % length;
    }

    if (!queue[victim]->isEmpty())
    {
        task = queue[victim]->popTop();
        // count = length;
    }
    count--;
}
```

In the above code block we can see that the thief chooses its victim randomly and assigns itself a new task from victims DEQueue by popTop() method call. We should guarantee progress by ensuring that threads that have work to do are not unreasonably delayed by (thief) threads which are idle except for task-stealing. To prevent this situation we need to call Thread.yield() immediately before trying to steal a task.This call yields the thief ’s processor to another thread, allowing descheduled threads to regain a processor and make progress.

Bounded Dequeue

Bounded Dequeue consists of an array of tasks indexed by bottom and top fields that reference the bottom and top of the dequeue. It provides pushBottom() and popBottom() which uses reads-writes to manipulate its bottom reference.
The Bounded Dequeue class has 3 fields: tasks, bottom, and top. Tasks fields hold an array of runnable tasks in the double ended queue. Bottom is the index of the first empty slot in the tasks queue. And top field which is an stamped reference i.e it holds a reference that is an index to the first task in the queue and a stamp which is a counter that increments every time the reference changes.

```
class BoundedDequeue: public Dequeue {
    RunnableTask** tasks;
    volatile int bottom;
    std::atomic<StampedReference<int>>* top;
    public:
        BoundedDequeue();
        BoundedDequeue(int capacity);
        void pushBottom(RunnableTask* task);
        RunnableTask* popBottom();
        bool isEmpty();
        RunnableTask* popTop();
};
```

Bounded Dequeue provide mainly four methods (as can be seen from above code) :

1. pushBottom: This Method simply stores the new task at the bottom queue location and increments the bottom. It increments the size of the queue first and then inserts a new task at the bottom of the queue.

```
void pushBottom(RunnableTask *task)
{
    tasks[bottom] = task;
    bottom++;
}
```

2. popBottom: This method is more complex when the queue is empty, as it returns immediately, and otherwise, it decrements bottom, claiming a task. To ensure that thieves can recognize an empty Bounded DEQueue, the bottom field must be declared volatile. This Method reads the task and returns if compareAndSet() succeeds, otherwise returns the one it read.

```
RunnableTask popBottom()
{
    if(bottom == 0)
```

```
        return nullptr;
    bottom--;

    RunnableTask *r = tasks[bottom];

    StampedReference<int> oldTop = top->load();
    StampedReference<int>  newTop(0, oldTop.stamp+1);

    if (bottom > oldTop.value)
        return r;

    if(bottom == oldTop.value){
        bottom = 0;
        if (top->compare_exchange_strong(oldTop, newTop))
            return r;
    }
    top->store(newTop);
    return nullptr;
}
```

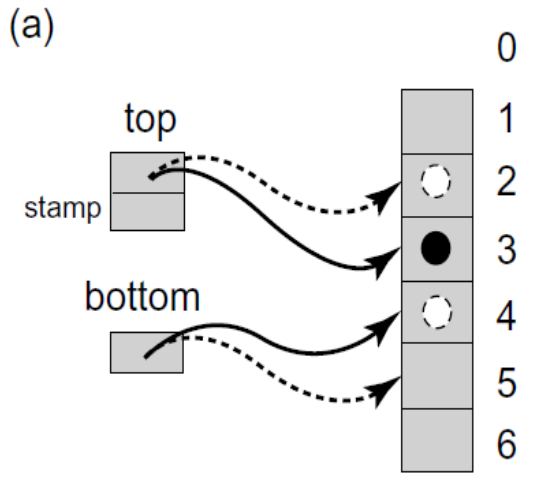
3. isEmpty: This method simply used to check whether dequeue is empty or not. To check whether the dequeue is empty or not it simply checks bottom <= top.

```
bool isEmpty()
{
    int localTop = top->load().value;
    int localBottom = bottom;
    return (localTop < localBottom);
}
```

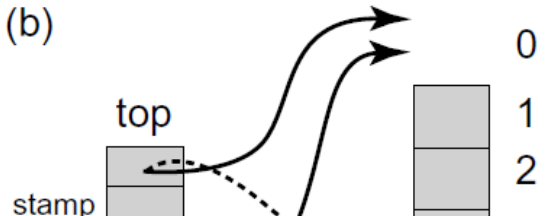
4. popTop: This Method checks whether the Bounded DEQueue is empty, and if not, tries to steal the top element by calling compareAndSet() to increment top. If compareAndSet() succeeds it means that the thief has successfully stolen a task and if it fails it returns null.

```
RunnableTask* popTop()
{
    StampedReference<int> oldTop = top->load();
    StampedReference<int>  newTop(oldTop.value+1,
oldTop.stamp+1);

    if(bottom <= oldTop.value)
        return nullptr;

    RunnableTask *r = tasks[oldTop.value];

    if (top->compare_exchange_strong(oldTop, newTop))
        return r;
    return nullptr;
}
```

One of the important cases that popTop may encounter is when there is only one task in the queue, then the caller might encounter interference from a thief trying to steal that task. So if the bottom is close to top, the calling thread switches to using compareAndSet() to pop tasks.



From this example we can see that popTop() and popBottom() are called concurrently while there is more than one task in the BoundedDEQueue. The popTop() method reads the element in entry 2 and calls compareAndSet() to redirect the top reference to entry 3. The popBottom() method redirects the bottom reference from 5 to 4 using a simple store and then, after checking that bottom is greater than top it removes the task in entry 4.



In this example there is only a single task. When popBottom() detects that after redirecting from 4 to 3 top and bottom are equal, it attempts to redirect top with a compareAndSet(). Before doing so it redirects bottom to 0 because this last task will be removed by one of the two popping methods. If popTop() detects that top and bottom are equal it gives up, and otherwise it tries to advance top using compareAndSet().

Unbounded Dequeue

Bounded Dequeue works perfectly in many cases. But for bounded dequeue, we should know the size of the queue before initializing the bounded dequeue. It is not always possible to predict the size beforehand. For example, we don't know how many tasks the CPU scheduler will push in the queues. It depends on CPU load. It will also waste a lot of storage if we randomly initialize the bounded dequeue of some big size.
To address this issue unbounded dequeue is introduced. Unbounded Dequeue dynamically resizes itself.
Unbounded Dequeue is implemented as a cyclic array with top and bottom field. bottom field = index of the circular array in which the next task will go.
The Unbounded Dequeue class also has a task field to store the reference of a circular array object.

Circular array can be implemented as:

```
class CircularArray {
private int logCapacity;
private Runnable[] currentTasks;
 CircularArray(int myLogCapacity) {
 logCapacity = myLogCapacity;
 currentTasks = new Runnable[1 << logCapacity];
 }

 int capacity() {
    return 1 << logCapacity;
```

```
 }

Runnable get(int i) {
 return currentTasks[i % capacity()];
}

 void put(int i, Runnable task) {
   currentTasks[i % capacity()] = task;
 }

 CircularArray resize(int bottom, int top) {
 CircularArray newTasks = new CircularArray(logCapacity+1);
 for (int i = top; i < bottom; i++) {
   newTasks.put(i, get(i));
 }
 return newTasks;
}
}
```

Circular Array provides get() and put() methods that add and remove tasks, and a resize() method that allocates a new circular array and copies the old array's contents into the new array.

Unbounded Deque like the Bounded Dequeue provide mainly four methods:

1. pushBottom: It simply pushes tasks in the circular array and increments the bottom field. It also resizes the circular array when it becomes full.
   We can linearize pushBottom() calls when bottom is incremented,

```
public void pushBottom(Runnable r) {
int oldBottom = bottom;
int oldTop = top.get();
CircularArray currentTasks = tasks;
int size = oldBottom - oldTop;
```

```
 if (size >= currentTasks.capacity()-1) {
 currentTasks = currentTasks.resize(oldBottom, oldTop);
tasks = currentTasks;
 }
tasks.put(oldBottom, r);
bottom = oldBottom + 1;
}
```

2. popBottom: It used to get recent tasks pushed in the queue. It first decrements the bottom and checks the size of the queue, if it is empty(i.e size < 0, queue is empty when top = bottom and it decremented bottom in the start) then it returns null and reset bottom = top. If size > 0 it simply returns the task at bottom. If size = 0, there is only one task in the circular array so conflict can occur as another queue may try to steal this task using popTop method.So the calling thread switches to using compareAndSet() on top field to pop tasks. If it succeeds, it increments the top atomically using compare and swap and returns the task. If it fails it simply returns null. Whether it succeeds or fails it sets bottom = top as the queue will become empty i.e either it will pop the task or task will be stolen by another thread.
   We lineraze it when the bottom is decremented.

```
public Runnable popBottom() {
CircularArray currentTasks = tasks;
bottom--;
 int oldTop = top.get();
int newTop = oldTop + 1;
int size = bottom - oldTop;
 if (size < 0) {
 bottom = oldTop;
 return null;
 }
 Runnable r = tasks.get(bottom);
if (size > 0)
 return r;
 if (!top.compareAndSet(oldTop, newTop))
```

```
r = null;
 bottom = oldTop + 1;
return r;
 }
```

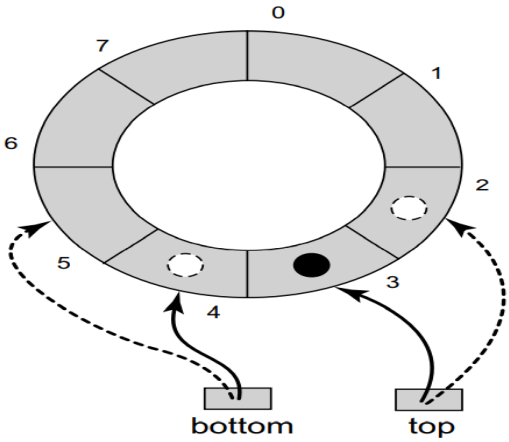
3. popTop : It is used by other threads to steal the task from the dequeue. It first checks the size of dequeue. If size <= 0, dequeue is empty and it simply returns null. size < 0 is only possible when the dequeue is empty and popBottom is running concurrently and has decremented the bottom field. We can also get size = 0, when there is only one task in the dequeue and popBottom is trying to get the task and has a decremented bottom. So popTop will simply return null in this case and will not try to steal the task. If size > 0, it tries to steal the task using the compare and swap on the top field as other threads also may be trying to steal the task. If it succeeds it will return the task else it will return null.
   We linearize each unsuccessful popTop() call at the point where it detects that the Dequeue is empty, or at a failed compareAndSet(). Successful popTop() calls are linearized at the point when a successful compareAndSet() takes place.

```
 public Runnable popTop() {
 int oldTop = top.get();
int newTop = oldTop + 1;
int oldBottom = bottom;
CircularArray currentTasks = tasks;
 int size = oldBottom - oldTop;
 if (size <= 0) return null;
 Runnable r = tasks.get(oldTop);
if (top.compareAndSet(oldTop, newTop))
    return r;
 return null;
 }
```

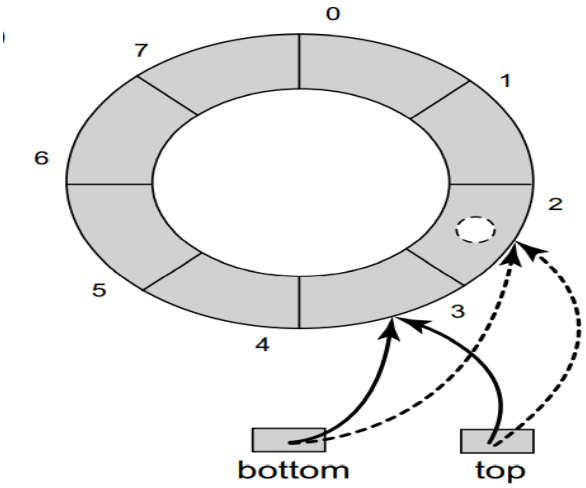
4. isEmpty: This method simply used to check whether dequeue is empty or not. To check whether the dequeue is empty or not it simply checks bottom <= top.

```
boolean isEmpty() {
int localTop = top.get();
int localBottom = bottom;
return (localBottom <= localTop);
 }
```

Ideally, a work-stealing algorithm should provide a linearizable implementation whose pop methods always return a task if one is available. In practice, however, we can settle for something weaker, allowing a popTop() call to return null if it conflicts with a concurrent popTop() call. Though we could have the unsuccessful thief simply try again, it makes more sense in this context to have a thread retry the popTop() operation on a different, randomly chosen DEQueue each time.



The above figure shows the running of popBottom and popTop methods concurrently when there is more than one task in the dequeue. The bottom field was pointing to index 5 before popping the task whereas the top is pointing to index 2.
popBottom decrements bottom to 4 and simply pop the task at the bottom.
popTop also pop the task at top and increment the top to 3.



The above figure shows the running of popBottom and popTop concurrently when there is only one task in the dequeue.
Initially bottom refers to index 3 and top to 2. The popBottom() method first decrements bottom from 3 to 2. Then, when popBottom() detects that the gap between the newly-set bottom and top is 0, it attempts to increment top by 1 using compare and swap on the top field. The popTop() method attempts to do the same. The top field is incremented by one of them, and the winner takes the last task. Finally, the popBottom() method sets bottom back to Entry 3, which is equal to top.

Application

Check All Prime Numbers In Range

For Comparing performance we have made an application to check prime numbers. Our Program takes a number input and checks all the prime numbers from 1 to that number. Numbers are equally divided among the threads to check prime numbers except the last thread which will take all the remaining numbers. Ex- If we have 10 numbers and 3 threads, the first two threads will take 3 numbers and the last thread will take 4 numbers. Also the threads in the end have larger numbers to check prime. We have done this to simulate the effect that some thread is overloaded. When the threads in the beginning finish their task early (as they have a

comparatively smaller number to check prime) they will try to steal tasks from the overloaded thread and reduce their burden.
Each thread given a range of numbers from start to end check prime for all the numbers in the range excluding start. They first push the task in dequeues and than call popBottom to get the task and than run the task. Other threads also try to steal the task using popTop method. We are trying to simulate CPU scheduler. Pushing the task in the dequeue is like assigning task to some thread. When CPU has to run the task it will call popBottom first to get the task and than run the task. We are also not pushing all the numbers assigned to thread in one go. We are pushing the task in power of 2. Like first we push 1 number, than thread will check prime for that and than push two numbers, than thread will check prime for the two tasks, than thread will push four numbers and so on….. We have done like this because we want to see the performance of Unbounded Dequeue. In this way Unbounded Dequeue size kept on changing as the number of tasks increases. Once threads finishes it's all the task and do not have any task to push it will try stealing task from other threads. It will try to steal number of threads - 1 times, if it fails the thread will terminate.
As a final result our programs give a count of prime numbers to in the range.

We have also made a Normal Dequeue to compare the performance. It is a Normal Dequeue Which do not try to steal tasks from other threads. It will terminate just by completing its own task. Its methods are also not concurrent.

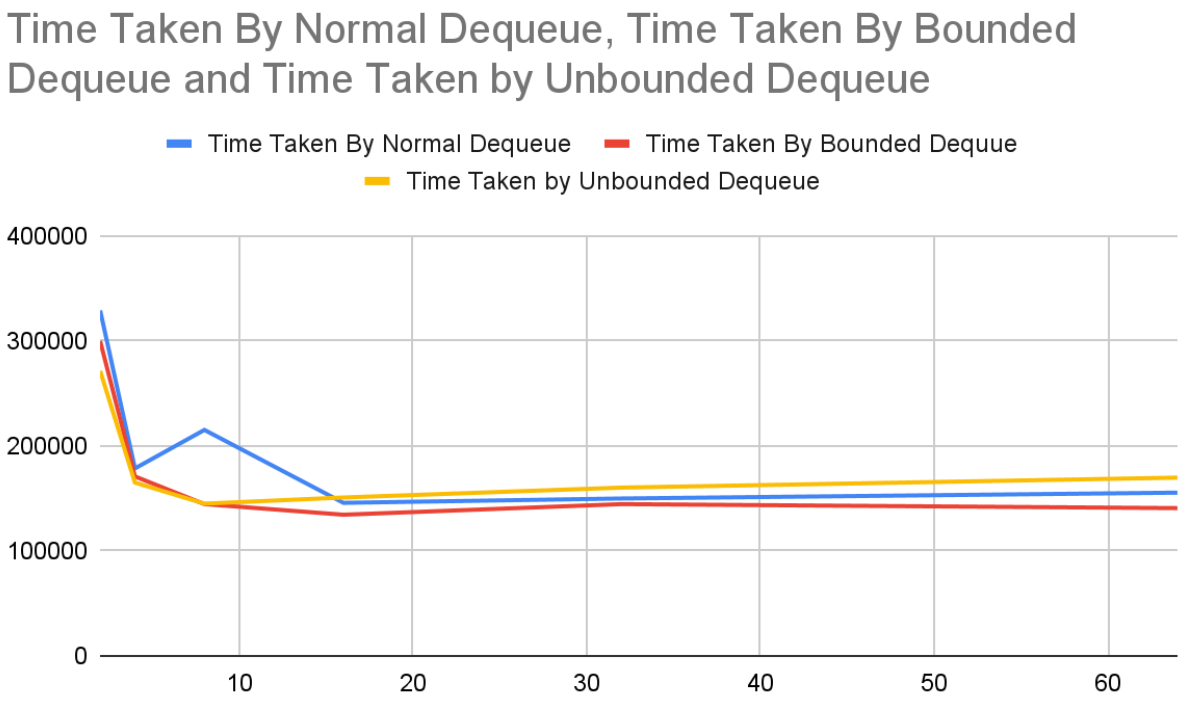
Matrix Multiplication

We have also made a second application in which we are computing matrix multiplication parallely using work-stealing dequeue. For the final result of multiplication size n * m, we have created the n * m task of multiplying the row of matrix 1 with the column of matrix two and then adding them. Like to multiply two 2 * 2 matrices there will be four tasks corresponding to each res[i][j]. Like to calculate res[0][0] we need to multiply row 0 of matrix 1 with column 0 of matrix 2 and then add them. We have then assigned this

task randomly to 4 threads by pushing in their dequeue. We have pushed randomly because we intentionally want load on threads to be different so that light loaded threads finish early and try to help other loaded threads by calling popTop method.
We have fixed the thread to 4 just for demonstration purposes.

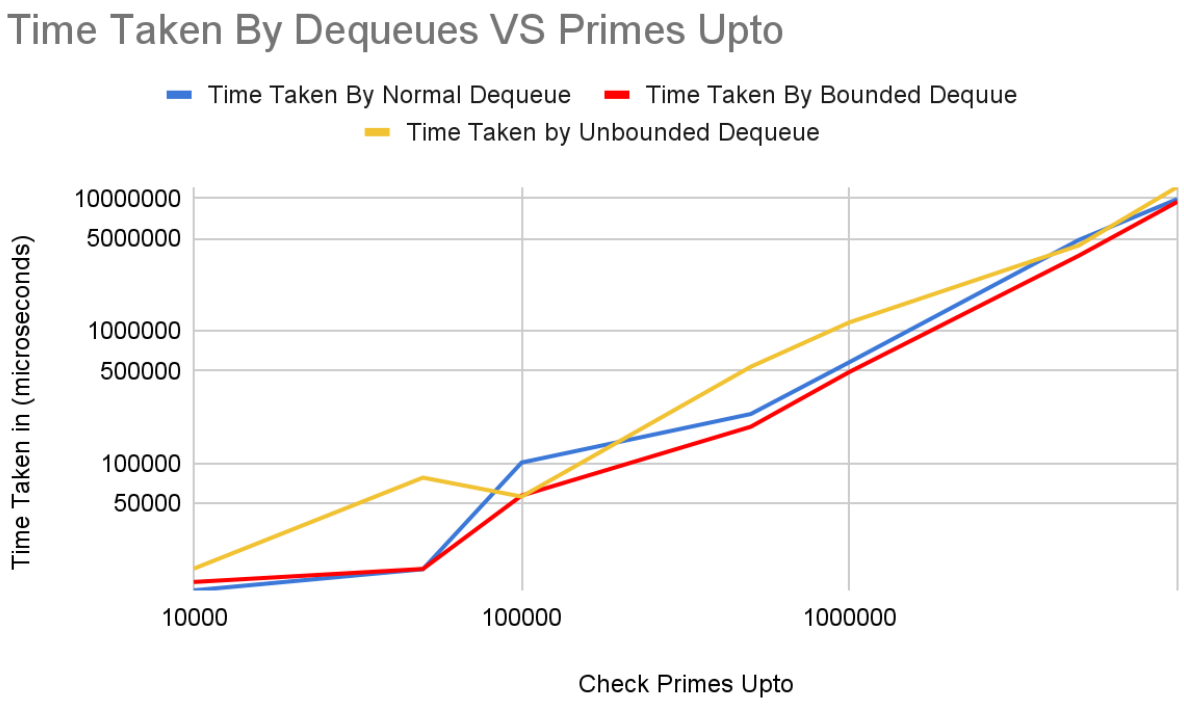
Performance Comparison

Time Taken in Microseconds by Number of Threads To Check Prime Upto 10^6

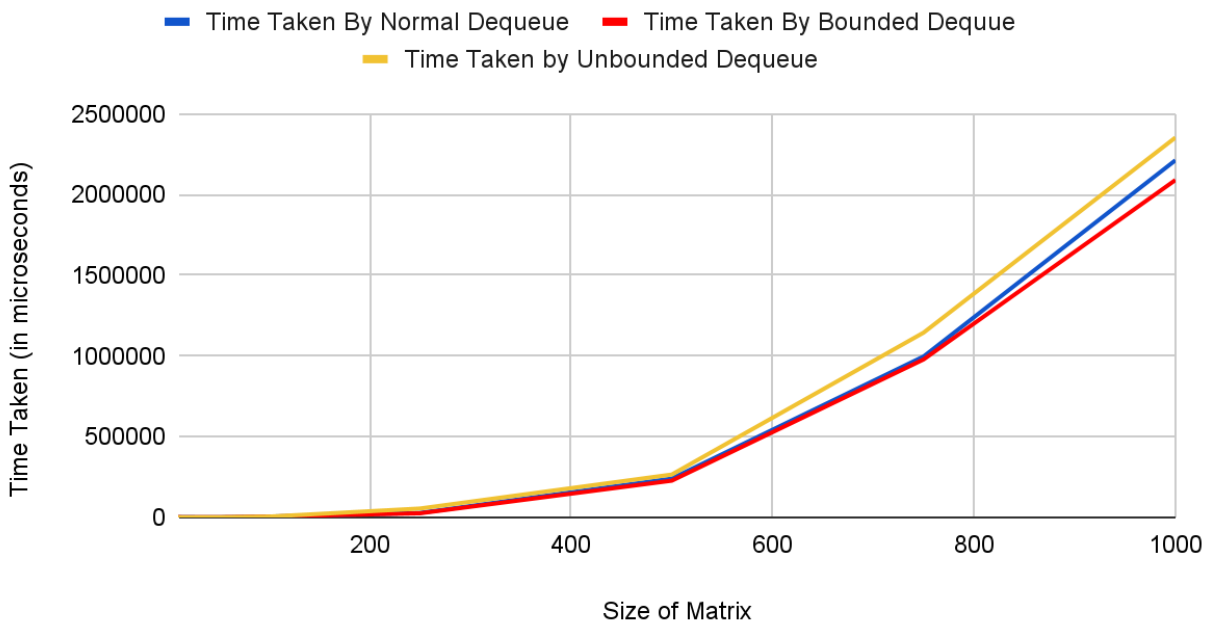


As we can see Bounded Dequeue is performing better than both normal Dequeue and Unbounded Dequeue. This is because in a bounded dequeue when some of the threads become free they try to help other threads by stealing their tasks and being able to achieve more parallelism.

Unbounded dequeue has not performed well may be because of the overhead of resizing the dequeue. Initially when threads were fewer and each thread had a lot of tasks to do overhead of resizing was not affecting the performance and unbounded dequeue was performing like the bounded dequeue. But as we keep increasing threads and each thread have a fewer tasks to do overhead of resizing a clearly visible and unbounded dequeue is taking even more time than the normal dequeue.



Here also as we can see Bounded Dequeue is performing better than both normal Dequeue and Unbounded Dequeue. This is because in a bounded dequeue when some of the threads become free they try to help other threads by stealing their tasks and are able to achieve more parallelism. Initially when tasks are less unbounded, dequeue is not performing well. But as tasks are increasing unbounded dequeue is performing similar to Bounded Dequeue. We can observe similar results for different tasks (Matrix Multiplication) in the below graph where number of threads are fixed to 4 and range of matrix values are set to 1000.



References

The Art of Multiprocessor Programming by Maurice Herlihy & Nir Shavit