Project Report

Work Stealing Dequeues

CS5300-Parallel and Concurrent Programming

Under Dr. Sathya Peri

Aditya Agrawal - CS19B1003 Kushagra Indurkhya -CS19B1017

Index

Introduction

As our CS5300 project, we have implemented, tested, and demonstrated implementation of linearizable data structures - work-stealing bounded and unbounded double-ended queues as described in the literature, The Art of Multiprocessor Programming by Maurice Herlihy, Nir Shavit: Chapter 16.

Need for Work Distribution

In modern multiprocessor systems, the key to achieving a good speedup is to keep user-level threads supplied with tasks, so that the resulting schedule is as greedy as possible, but multithreaded computations create and destroy tasks dynamically, sometimes in unpredictable ways. A work distribution algorithm is needed to assign ready tasks to idle threads as efficiently as possible. Two popular approaches to work distribution are

- **Work Dealing**: an overloaded task tries to offload tasks to other, less heavily loaded threads. The flaw in this approach is if most threads are overloaded, then they waste effort in a futile attempt to exchange tasks.

- **Work Stealing**: in which a thread that runs out of work tries to “steal” work from others. An advantage of work-stealing is that if all threads are already busy, then they do not waste time trying to offload work on one another.

We’ll here discuss the Work Stealing approach in this project. The core idea of this approach is that threads keep the tasks assigned to them in their respective dequeue(double-ended queue). The thread pushes to and pops the tasks from just the one end of their dequeue and the other end is popped from when a thread tries to steal from this thread.

The threads first complete the work they’re allotted and when they become empty they steal from other threads. Meanwhile, on a dynamic end, if the thread is allotted some task it’ll start executing that as soon as it completes the current task.

We discuss two implementations of this approach with their pros and cons while depicting the idea of work balancing through stealing.

Bounded and Unbounded Dequeues

Here, we'll be implementing two types of linearizable and non-blocking work-stealing dequeues, vis-a-vis., Bounded and Unbounded. The queues contain tasks for the threads to be executed. Some threads can steal others' tasks if the particular thread has completed its queue. These queues have the following operations:

- ○ pushBottom(): Insert into the bottom of the queue.
- ○ popBottom(): Pop from the bottom of the queue.
- ○ popTop(): Pop from the top of the queue.

● Bounded dequeues have a fixed size, whereas Unbounded dequeues dynamically resize themselves as needed.

These queues/algorithms/methods can be used to increase the overall efficiency of the system and increase throughput. The advantage of using these queues in a multiprocessing environment is that:

● These queues are non-blocking. While on dedicated processors, access to the queues can be synchronized using locks, this is not favorable for a multiprogramming environment since the operating system might preempt the worker thread holding the lock, blocking the progress of any other workers that try to access the same queue.

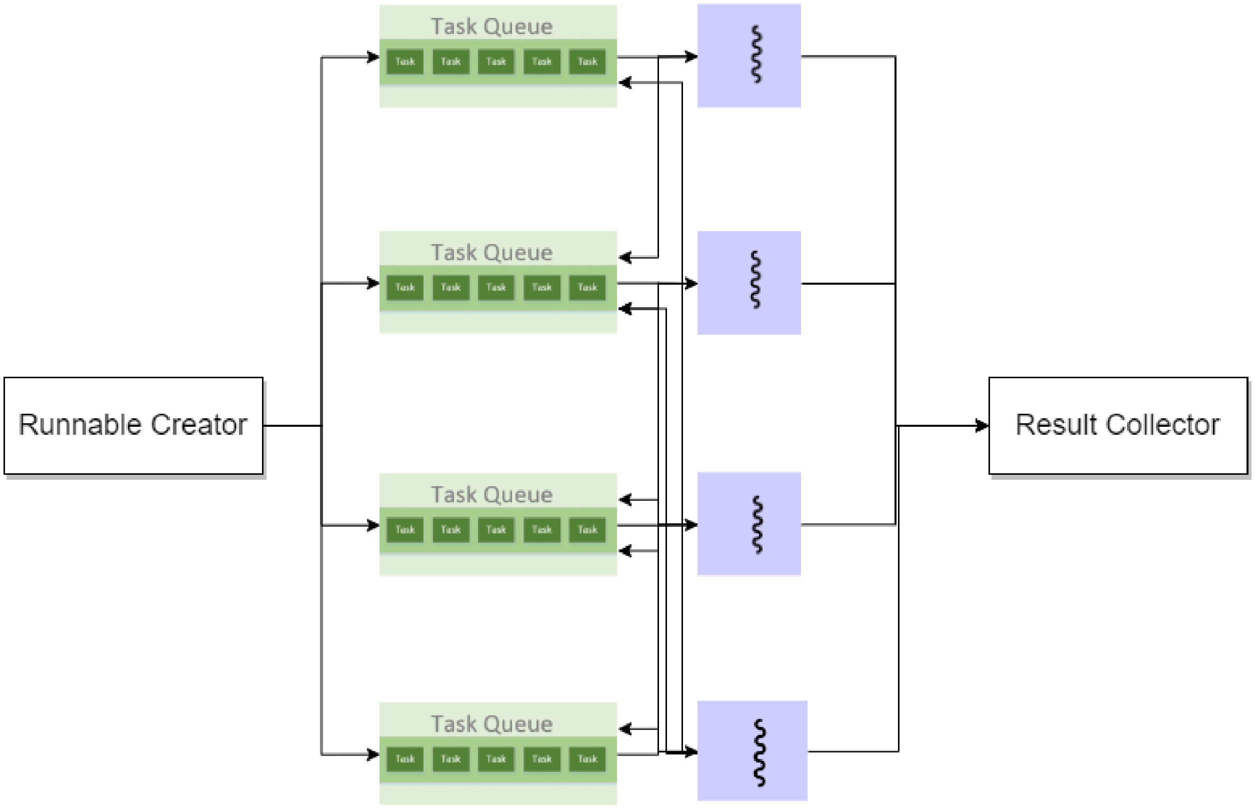
● Before each attempt to steal work, a worker thread calls a "yield" system call that yields the processor on which it is scheduled to the OS, to prevent starvation.

In this project, we have tried to implement an efficient scalable load balancing technique using work-stealing by implementing a nonblocking linearizable DEQueue class.

Our dequeues employ work-stealing i.e., as soon as a thread completes the tasks allotted to it, It starts stealing other threads' tasks. This helps reduce the load on the unevenly distributed workgroup both on a qualitative and a quantitative level.

All the threads are given a dequeue (double-ended queues - can be dequeued from both sides) of their own which contains tasks allotted to them. The thread starts by completing the tasks allotted to it, and as soon as it completes the tasks it checks whether queues of other threads are empty or not and tries to take a task out of those if possible and performs the task. In this way, if the work is distributed unevenly based on quantity (uneven number of tasks distributed: 100 1 10 6000...) with the same workload (amount of computation required), the thread with a lesser number of tasks will steal work from the loaded thread. If the work is distributed unevenly based on quality (workload is highly uneven: task A takes 10s in sequential execution, task B takes 10us), the thread which completes the work first will steal work from other threads.

Implementation Design



The high-level implementation looks something like the image shown. There's a task generator that inserts the tasks in the dequeues of the threads. This can be real-time (while the threads are executing these tasks) or batch mode. The threads can steal tasks from other threads if idle.

We'll go through each component required and describe it.

Utils Implementation

Our project requires some common utility elements which are used multiple times in our implementations, those are:

- **Runnable**: This class describes the task to be performed. Although in this project we show a limited number of implementations, there is great flexibility to perform any kind of task possible in the language thanks to this design. The implementation offers to pass a function and its arguments to be executed as a task.

```
class Runnable
{

public:
  long id;
  std::function<void()> task = NULL;
  template <typename func, typename... Args>
  Runnable(func f, Args... args);
  void run();
  Runnable() {}
};
```

Queue Implementation

- **DEQueue interface** is an abstract class that is implemented by the several implementations possible for these dequeues.

```
class DEQueue {
public:
  Runnable *tasks;
  volatile int bottom;
  AtomicStampedReference<int> *top;
  virtual void pushBottom(Runnable r) = 0;
  virtual bool isEmpty() = 0;
  virtual Runnable *popTop() = 0;
  virtual Runnable *popBottom() = 0;
};
```

BDEQueue

This is an implementation of the DEQueue class which implements a bounded dequeue with fixed number of tasks possible to be performed. An attractive aspect of this design is that an expensive compareAndSet() call is needed only rarely when the BoundedDEQueue is almost empty.

ABA Problem

- An index is read twice and have the same value for both reads, and when the value is same an assumption is made that nothing had changed in between.
- However, there may be a possibility that another thread has executed between the two reads and changed the value, do other work, then change the value back
- Thus our first thread thinks "nothing has changed" even though the other thread has executed tasks in between making the operations illegal.

Solution to ABA problem

- **AtomicStampedReference**
- This class provides an atomic timestamp-based value-keeping mechanism to implement lock-free data structures.
- This prevents the ABA problem as:
    - The stamp here uses a counter which is incremented each time the corresponding reference is changed.
    - Thus to prevent ABA problem we just need to perform a CAS on (reference,stamp)

```
template <typename T> class AtomicStampedReference {
public:
  std::atomic<AtomicStamp<T>> snap;
  AtomicStampedReference();
  AtomicStampedReference(T data, int stamp);
  bool compareAndSet(T oldRef, T newRef, int oldStamp, int
newStamp);
  T get(int *ptr);
  T getReference();
  int getStamp();
  void set(T data);
  void set(T data, int stamp);
};
```

Implementation Design

- Tasks are allocates in a simple array.
- *pushBottom()*
- we take a `Runnable r` and assign it to current bottom and increment the bottom of the queue

  ```
  void pushBottom(Runnable r) {
    this->tasks[bottom] = r;
    bottom++;
  }
  ```

- *popBottom()*
- Thread is working on its own tasks
  - If the queue is empty we return null, if it is not empty:
    - We decrement bottom
    - We assign current bottoms runnable to r to be returned later
    - We fetch the old stamp and assign a new stamp which is oldStamp+1
    - If bottom>oldTop
      - We have elements left between top and bottom so we can safely return the r we fetched earlier
    - Else if bottom==top:(Implies there's only one task left)
      - Bottom is reset to 0
      - We perform a CAS between (oldTop,oldStamp) and (newTop,newStamp) and return r to ensure that if some thread is also concurrently stealing from this thread, both of these threads don't run the same task.
      - Top is set to the (newTop,newStamp)

```
Runnable *popBottom() {
  if (bottom == 0)
    return nullptr;

  bottom--;
  Runnable *r = &tasks[bottom];
  int *stamp = new int[1];
```

```
    int oldTop = top->get(stamp), newTop = 0;
    int oldStamp = stamp[0], newStamp = oldStamp + 1;
    if (bottom > oldTop)
      return r;
    if (bottom == oldTop) {
      bottom = 0;
      if (top->compareAndSet(oldTop, newTop, oldStamp, newStamp))
        return r;
    }
    top->set(newTop, newStamp);
    return nullptr;
  }
```

- *popTop()*
- Letting other threads steal tasks off the deque
- Get current stamp and top in oldStamp and oldTop increment both to new values
- If bottom<oldTop(empty queue) return null
- Perform a CAS on (oldTop,oldStamp),(newTop,newStamp),if returns true we return R

```
Runnable *popTop()
   {
     int *stamp = new int;
     int oldTop = top->get(stamp), newTop = oldTop + 1;
     int oldStamp = stamp[0], newStamp = oldStamp + 1;
     if (bottom <= oldTop)
       return nullptr;
     Runnable *r = &tasks[oldTop];
     if (top->compareAndSet(oldTop, newTop, oldStamp, newStamp))
       return r;
     return nullptr;
   }
```

UnboundedDEQueue

This is an implementation of the DEQueue class which removes the limitations of bounded queues which was that the size of the queue is fixed.

- In real world scenario predicting the number of tasks for a thread is nearly impossible.
- Also having each queue of size max_process based on some heuristics is a waste of memory.
- Pros
    - In this implementation we implemented a queue which dynamically resizes itself when needed.

    - In the Circular array top is always increment never decrememted hence we dont need a stamped reference here as ABA problem doesnt occur in this implementation

- Cons

    - The resizing ability carries a significant overhead as every call reads the top to see if resizing is necessary

Implementation Design

Circular Array

For resizing capabilities we employ the use of circular array which provides, get(), put() and resize() methods.

The circular array is implemented as follows:

```
class CircularArray
{
  int logCapacity;
  Runnable *currentTasks;

public:
  CircularArray(int myLogCapacity)
  {
    this->logCapacity = myLogCapacity;
    this->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;
  }
};
```

PushBottom()

- This is similar to pushBottom of boundedDequeue
- Here we resize the array if current size of array is exceeded

```
void pushBottom(Runnable r)
{
  int oldBottom = this->bottom;
  int oldTop = top->getReference();
  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;
}
```

PopBottom()

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

- If the queue is empty we return null
- Else we:
    - Fetch current top
    - Set newTop=oldTop+1
    - fetch the current bottom
    - If the old top is still same as when we started then we , set newTop
        - set bottom as oldTop
        - Return r

PopTop()

```
Runnable *popTop()
{
  int oldTop = top->getReference();
  int newTop = oldTop + 1;
  int oldBottom = bottom;

  CircularArray *currentTasks = tasks;
  int size = oldBottom - oldTop;
  if (size <= 0)
    return nullptr;
  Runnable *r = tasks->get(oldTop);
  if (top->compareAndSet(oldTop, newTop, 0, 0))
    return r;
  return nullptr;
}
```

Work Stealing Threads

Constructor of WorkStealingThread looks like this:

```
WorkStealingThread(int me, DEQueue **queue, int n, bool enable_ws)
{
  this->me = me; // Thread Id
  this->queue = queue; //Task Queue
  this->totalQueues = n; //Total number of the queues
  this->enable = enable_ws; //Is work stealing enabled
}
```

When we run the WorkStealingThread

- First the thread runs all its own tasks popping from the bottom
- Then while the queue doesnt have task of its own it tries to steal other thread's task
- We use yield to yield processors control if needed by other threads
- We check if all the queues are empty if they are we break out of the loop
- If not we popTop from the first queue we found which was not empty and assign task = queue[victim]->popTop();
- Which is run in the next iteration of the forever while loop

```
void run()
{
  Runnable *task = queue[me]->popBottom();
  while (true)
  {
    while (task != nullptr)
    {
      task->run();
      task = queue[me]->popBottom();
    }
    if (!enable)
      break;
    while (task == nullptr)
    {
      std::this_thread::yield();
      int victim = 0;
      for (; victim < totalQueues; victim++)
      {
        if (victim == me || queue[victim]->isEmpty())
          continue;
        break;
      }
      if (victim + 1 >= totalQueues)
        return;
      if (!queue[victim]->isEmpty() && victim != me)
        task = queue[victim]->popTop();

      if (task != nullptr)
        printf("work stole %d --> %d\n", victim, me);
    }
  }
}
```

Experiment

Finding nth Fibonacci number:

In this example task we give a number ***k*** as input and the task is to find the ***k****th* Fibonacci number. The following struct contains the data for the task from input to the final result:

```
struct task_data
{
  int n;
  long long res;
  long long createdAt;
  long long completedAt;
};
```

We performed two types of test here (shown the corresponding runnable creator lambdas):

- Random Non-Skewed Distribution - All tasks given are to find a Fibonacci number for a given range

```
[=]()
{
  int a = 25 + rand() % 5;
  arr[count].n = a;
  arr[count].createdAt = getEpoch();
  return Runnable(fib_thread, &arr[count++], a);
};
```

- Skewed Distribution - Half of the tasks are to find a fibonacci number and half are sleep for 10us

```
[=]()
{
  int a = (int(totalCount / 2) > count) ? 25 : 1;
  arr[count].n = a;
  arr[count].createdAt = getEpoch();
  return Runnable(fib_thread, &arr[count++], a);
};
```

for all 4 combinations possible

- Bounded Queues - With Work Stealing
- Bounded Queues - Without Work Stealing
- Unbounded Queues - With Work Stealing

- Unbounded Queues - Without Work Stealing

Using the above described Fibonacci application we devised some experiments and metrics to compare the performance of our queues implementing work-stealing algorithms.

Note: The experiments performed here were batch executions. It should be noted that the implementation is not limited to this and can be very well extended to real-time systems too.

For our experiments we have measured the following metrics:

- **Number of work steals**
- **Total Time Taken**
- **Average Waiting Time**

Experiment 1: Random Non-Skewed Distribution

In this experiment the workload is distributed with equal probability with each task as finding the ***kth*** Fibonacci number where $25 <= k < 30$.

Here the idea is to see how the queues react in a randomly but nearly equally distributed workload systems to depict the usage in the real world. There's no drastic change expected here due to the even workload but some improvements can occur due to randomness in the execution of tasks and the exponential increase in work required in Fibonacci.

Experiment 2: Skewed Distribution

In this experiment, the workload is constant (the task is to find the $25th$ or $1st$ Fibonacci number) but the extent of disparity in workload was large.

If there are n threads then n/2 of those get to find the 1st load type (25th Fibonacci number) and others get the 2nd load type (1st Fibonacci number).

Here the idea is to show the stealing capabilities as the thread with the 1st load type will complete the tasks in its queue in the minimal time while the other threads that have tasks with high CPU execution time requirements will be having a queue with a high load. The threads which have completed their tasks will start stealing and the workload will get balanced. If we assume k tasks per thread, then with stealing the number of tasks to be executed per thread would be $k/2$ (assuming half stolen by 1st load type threads) and without stealing the per-thread task count will be k but effectively number of threads executing will reduce down to $n/2$.

Note: These experiments were performed on an ARM-based M1 chip MacBook Air.

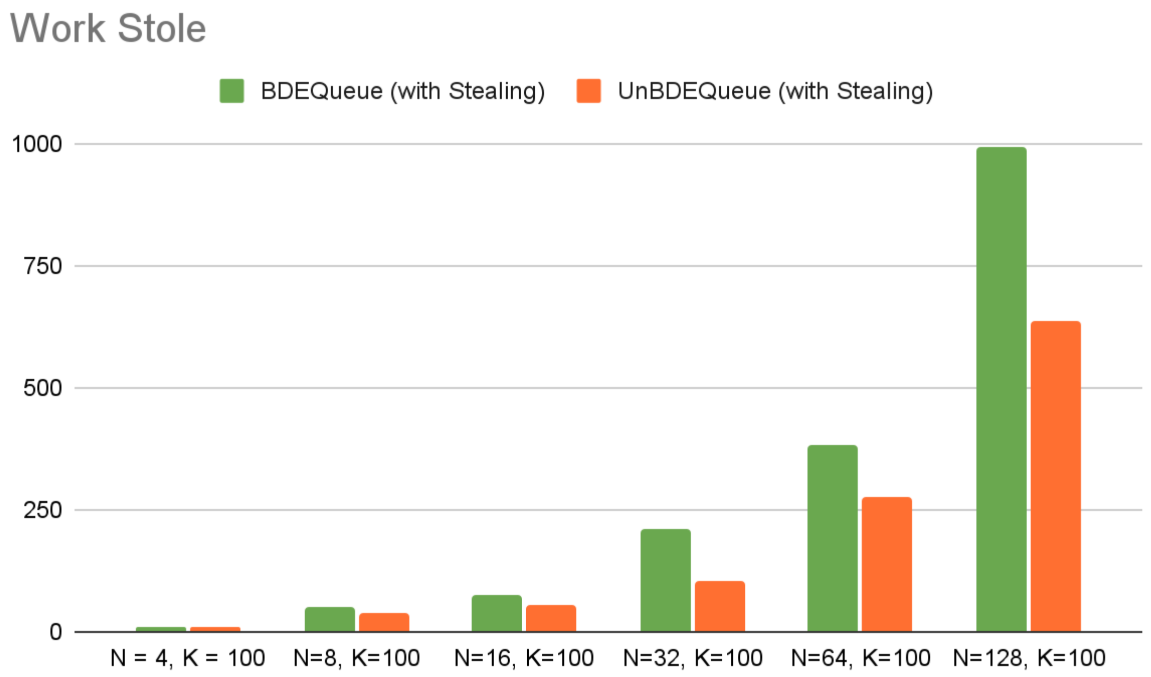
Other Experiments

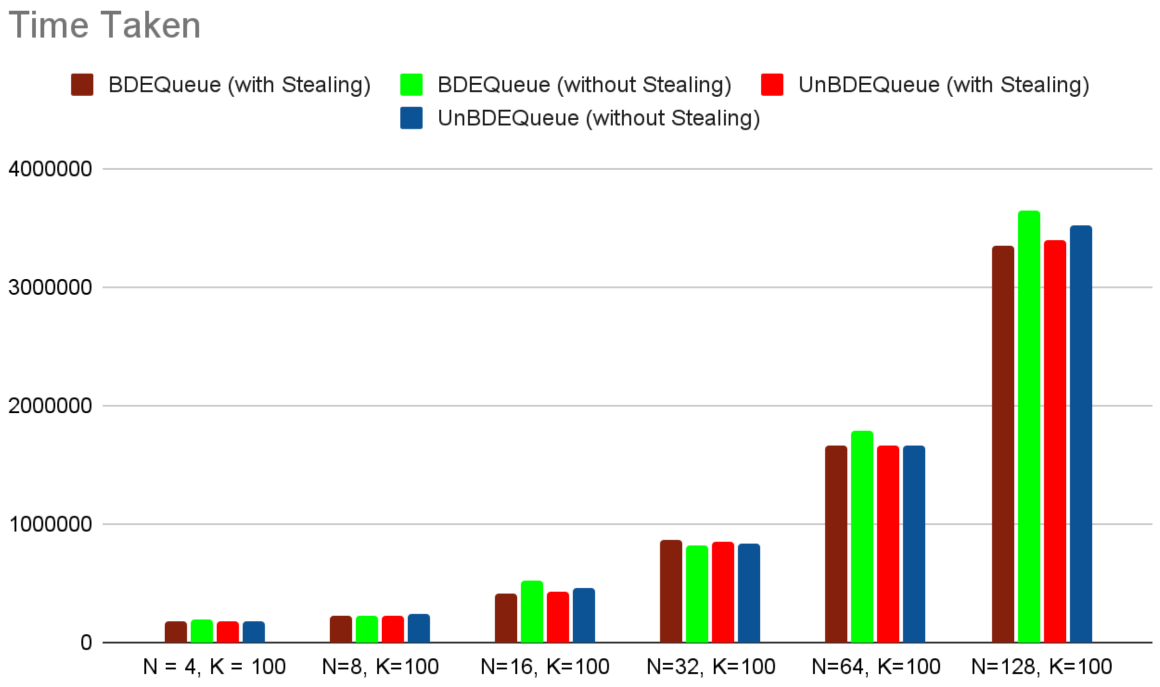
The tasks are not limited to any specific set of problems. Whatever can be implemented in a C++ function can be executed through this. Some example applications can be matrix multiplication, finding PI, and much more.

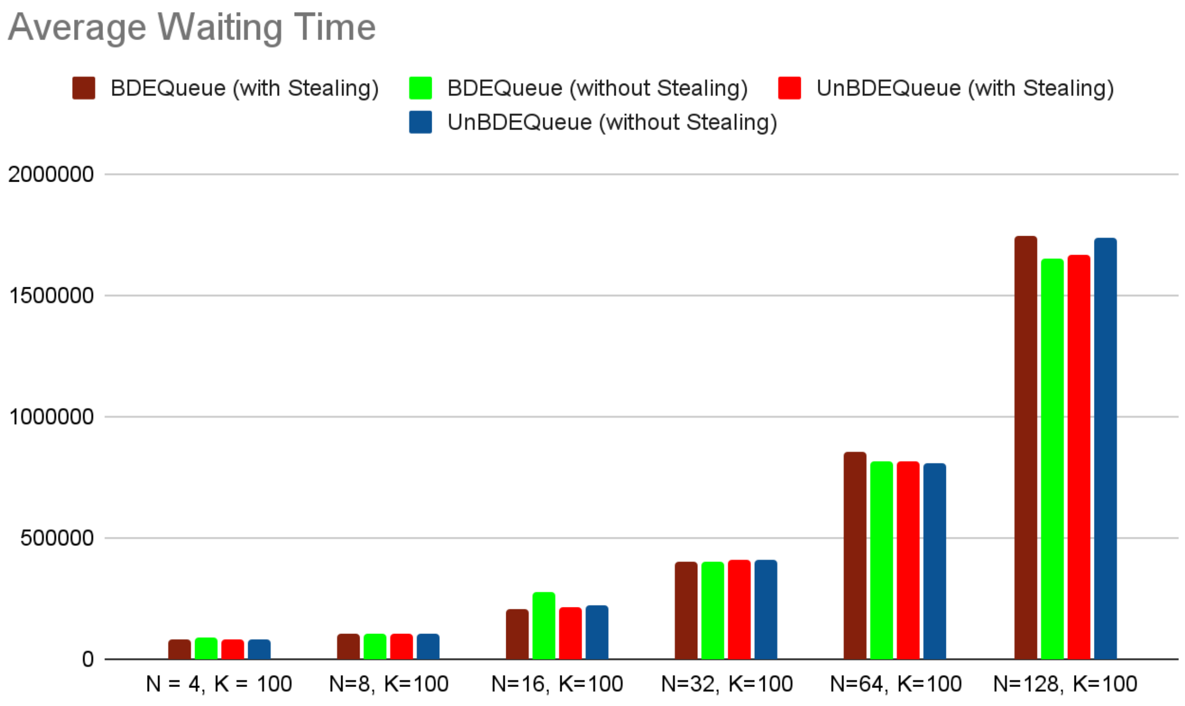
Also, these don’t have to be batch jobs (create tasks first then execute), in the internal capability tests, we’ve tried generating tasks on the go to simulate real-time systems, in this we made the main thread a generating thread to provide random queues tasks at random intervals.

Experiment Results

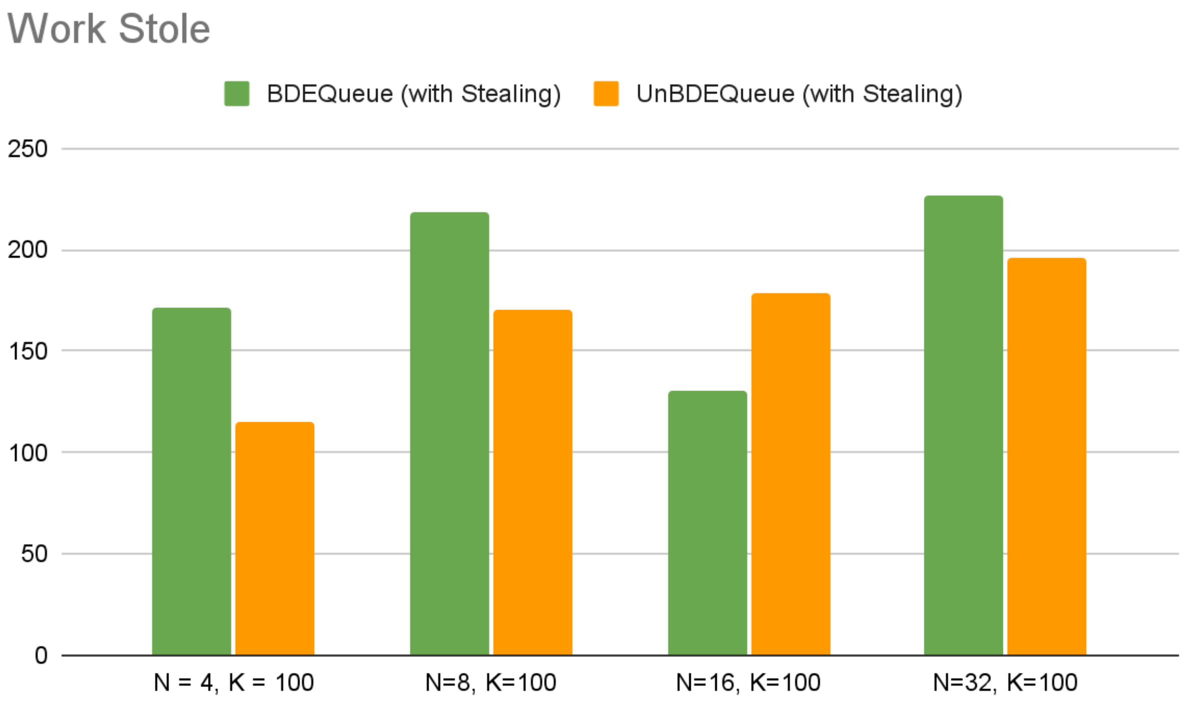
Parameters		Even Load (Experiment-1)			
		BDEQueue (with Stealing)	BDEQueue (without Stealing)	UnBDEQueue (with Stealing)	UnBDEQueue (without Stealing)
N = 4 K = 100	# work stole	9	-	8	-
	Total execution time	185044	192438	185653	188512
	Avg waiting time for a task	86754	88879	86950	86053
N=8 K=100	# work stole	53	-	38	-
	Total execution time	230564	221868	233830	240501
	Avg waiting time for a task	102779	102672	109730	104333
N=16 K=100	# work stole	77	-	56	-
	Total execution time	421460	519738	426879	456209
	Avg waiting time for a task	206259	275019	214059	223175
N=32 K=100	# work stole	209	-	104	-
	Total execution time	872304	825647	859300	832115
	Avg waiting time for a task	404076	399948	409622	409456
N=64 K=100	# work stole	381	-	278	-
	Total execution time	1792006	1658356	1665805	1658923
	Avg waiting time for a task	858033	814321	818989	812149
N=128 K=100	# work stole	992	-	637	-
	Total execution time	3651608	3345921	3401575	3527035
	Avg waiting time for a task	1744162	1650746	1669644	1739432


Work Stole
BDEQueue (with Stealing)
UnBDEQueue (with Stealing)
1000
750
500
250
0
N = 4, K = 100
N=8, K=100
N=16, K=100
N=32, K=100
N=64, K=100
N=128, K=100

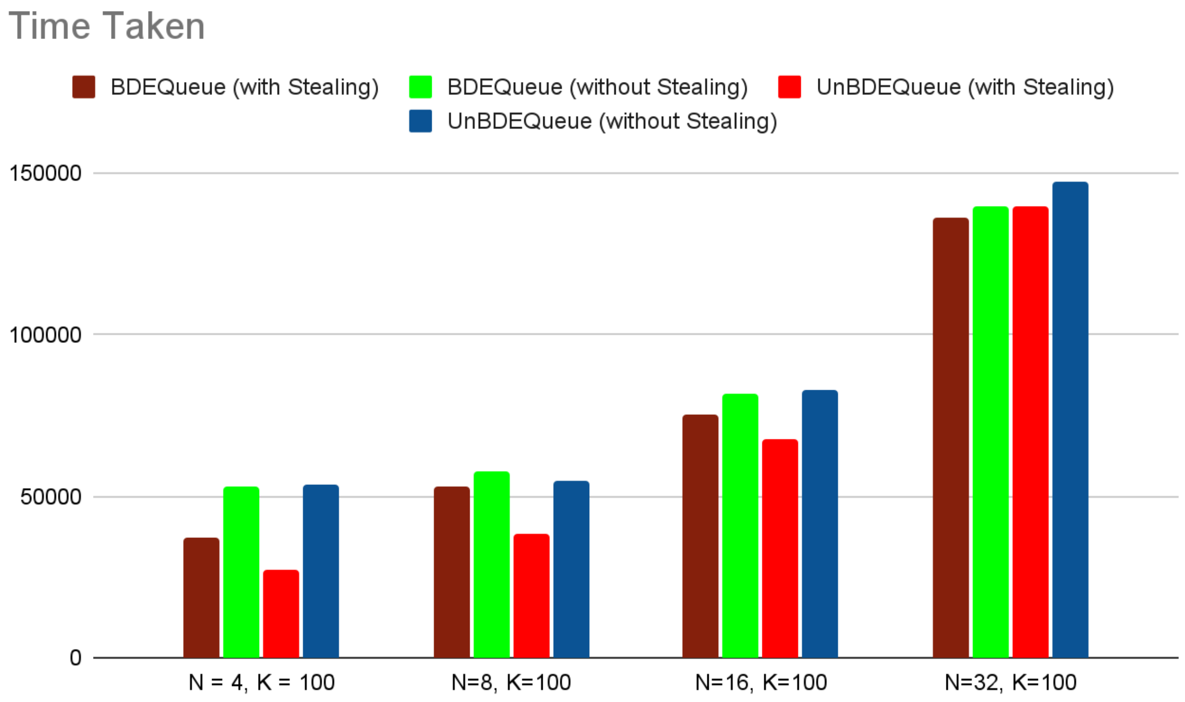

Time Taken
BDEQueue (with Stealing)
BDEQueue (without Stealing)
UnBDEQueue (with Stealing)
UnBDEQueue (without Stealing)
4000000
3000000
2000000
1000000
0
N = 4, K = 100
N=8, K=100
N=16, K=100
N=32, K=100
N=64, K=100
N=128, K=100


Average Waiting Time
BDEQueue (with Stealing)
BDEQueue (without Stealing)
UnBDEQueue (with Stealing)
UnBDEQueue (without Stealing)
2000000
1500000
1000000
500000
0
N = 4, K = 100
N=8, K=100
N=16, K=100
N=32, K=100
N=64, K=100
N=128, K=100

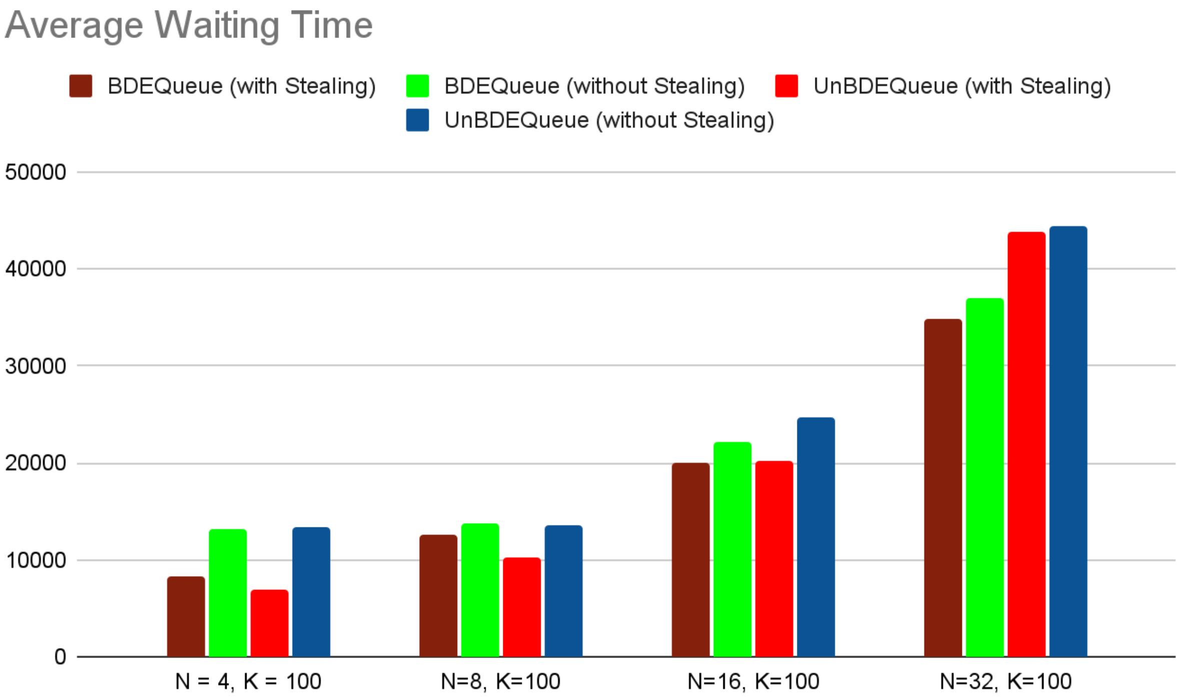
Parameters		Uneven Load (Experiment-2)			
		BDEQueue (with Stealing)	**BDEQueue (without Stealing)**	**UnBDEQueue (with Stealing)**	**UnBDEQueue (without Stealing)**
N = 4 K = 100	# work stole	179	-	115	-
	Total execution time	46987	52879	27717	53320
	Avg waiting time for a task	12170	13190	6934	13371
N=8 K=100	# work stole	308	-	170	-
	Total execution time	57994	53246	38596	54810
	Avg waiting time for a task	15596	13318	10455	13699
N=16 K=100	# work stole	131	-	179	-
	Total execution time	85145	78895	82906	82108
	Avg waiting time for a task	25051	22146	23419	23709
N=32 K=100	# work stole	227	-	196	-
	Total execution time	136189	139572	139487	147394
	Avg waiting time for a task	34863	37083	43881	44375



Work Stole



Time Taken



`fibo.o  16 100 bounded`

Results and Conclusion

From our implementations and experiments on using array-based bounded dequeues and circular array-based unbounded dequeues to balance the workload, We can conclude that:

- *Work Stealing works better in unevenly distributed systems*: As we can see from the graphs of both the experiments, in an evenly distributed system, there's less chance of work-stealing as all threads are almost equally loaded. Although in uneven distribution, the work stealing hugely benefits the overall system in terms of the total time taken and average waiting time for a task. This is because lightly loaded threads finish off their work very quickly compared to the highly loaded threads making them idle. And instead of exiting, they start stealing work which helps use the resources more efficiently.
- *Work Steal in Bounded queue implementation is almost always performing better than Unbounded queue implementation*: The advantage that bounded provides over unbounded is in terms of an added feature of providing support when number of tasks is not previously known, but in providing the resizing functionality unbounded queues suffer a overhead impacting their performance against bounded dequeues
- *Highly Applicable*: Although what we've discussed here was in terms of optimizing computing resources, the application can be extended to network, I/O, and other kinds of computing operations where we have multiple resources (hardware/software) on which the workload is distributed and can be balanced with these dequeues to achieve optimal performance. One example would be to use multiple networking adapters on a single machine and balance the work with these dequeues.

Challenges Faced

Making the executable task as flexible as possible

One of the tasks was to find a way for the task to be flexible which can run in the WorkStealingThread without hardcoding some parts again and again for different task implementations.

Although this task seems easy in modern languages like Python and JavaScript, we had to make some compromises to achieve this in C++ like nothing can be returned and allowing a callback pattern only.

Memory issues in atomic

As the atomic construct in C++ operates on an instruction level when using the standard compare_and_exchange method, if it is performed on a struct (or any other abstract data type), the comparison may take some garbage values (outside the memory allocated to the abstract data type) in consideration which will make the comparison fail and new values won't be written at all. To fix this issue, we created a specific struct considering its size, for the comparison, in mind.

Scopes

The use of these dequeues can be done on various levels:

- **Single Process Multiple Threads**: On a multicore machine, a process can spawn multiple threads for concurrent processing and these dequeues can be used to balance the work. An example of this application is the experiments that we performed and showed in this report.

- **Multiple Processes**: On a multicore machine, multiple processes (of the same or different types and purposes) can share a single dequeue or multiple dequeues to balance the work. An example of this is a process with multiple capabilities stealing work from other task focussed processes. Another example would be in terms of web browsers. Modern-day browsers work on a single process per tab model although mostly only one tab is in focus most of the time. Here the tasks can be stolen by the processes of other tabs to make use of the resources effectively.

- **Multiple Machines**: In a distributed system, for example, a Kubernetes cluster, multiple hosts can communicate with each other to balance work by keeping a dequeue per host. This will be an extended implementation and there is a need to build clients and servers for the implementation but the core idea of implementation remains the same. An example of this would be a cluster of an event processing node that acts on the event data passed (Highly applicable in media and content delivery and optimization). To be specific, imagine a cluster of image processing nodes that performs batch jobs. As these are image processing jobs these are asynchronous and event-based mostly. Here, if some node is sitting idle it can steal work from other nodes.

References

- The Art of Multiprocessor Programming - Maurice Herlihy, Nir Shavit
- Dynamic circular work-stealing deque | Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures
- LNCS 8806 - Lace: Non-blocking Split Deque for Work-Stealing (springer.com)
- Efficient Work-Stealing with Blocking Deques | IEEE Conference Publication | IEEE Xplore
- https://www.baeldung.com/java-work-stealing