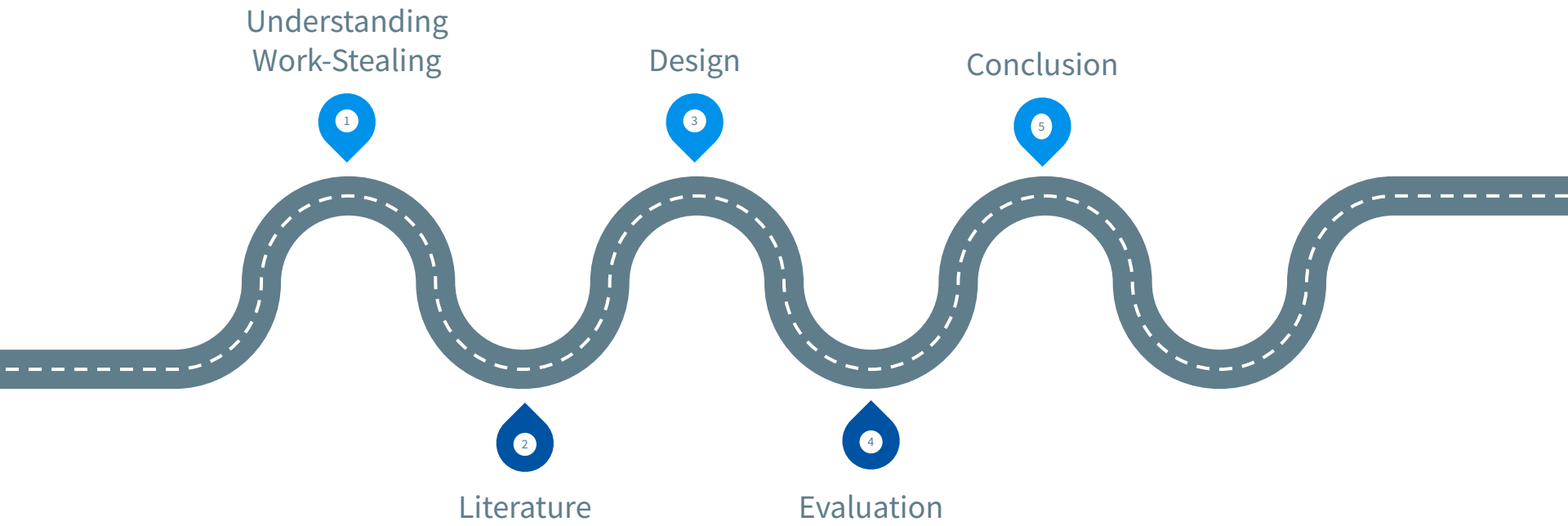
Optimizing Work-Stealing Algorithms

Raj Sahu

VIRGINIA TECH

12th December, 2022

Roadmap


Understanding Work-Stealing
1
Literature
2
Design
3
Evaluation
4
Conclusion
5

1.
Understanding Work Stealing

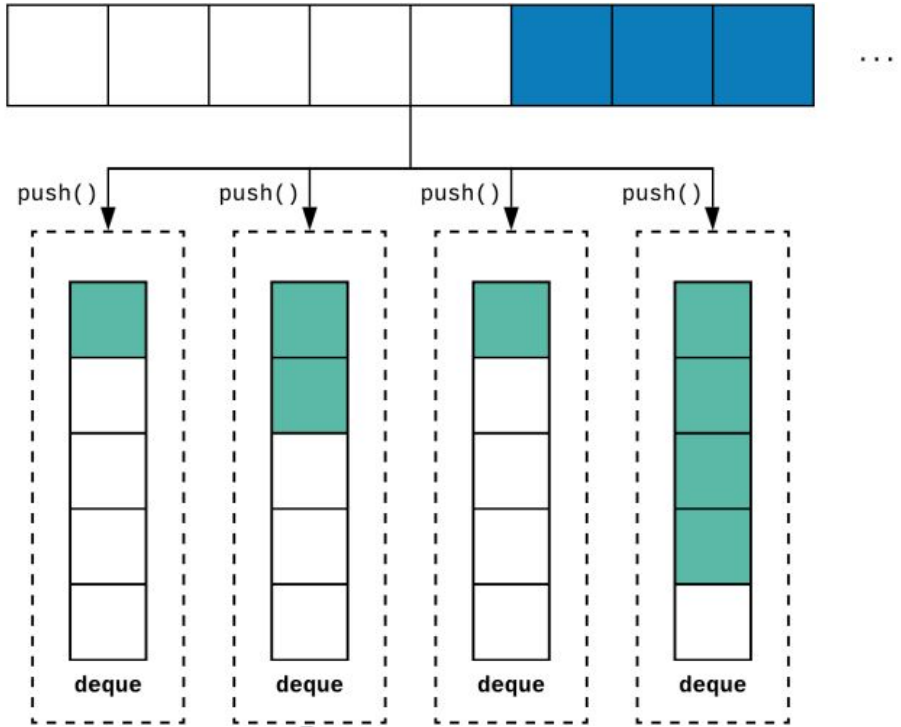
Brief Introduction

- Creating and throwing new threads
  - More time spent on managing them

- Creating and throwing new threads
  - More time spent on managing them

Thread Pool enters

Thread Pools

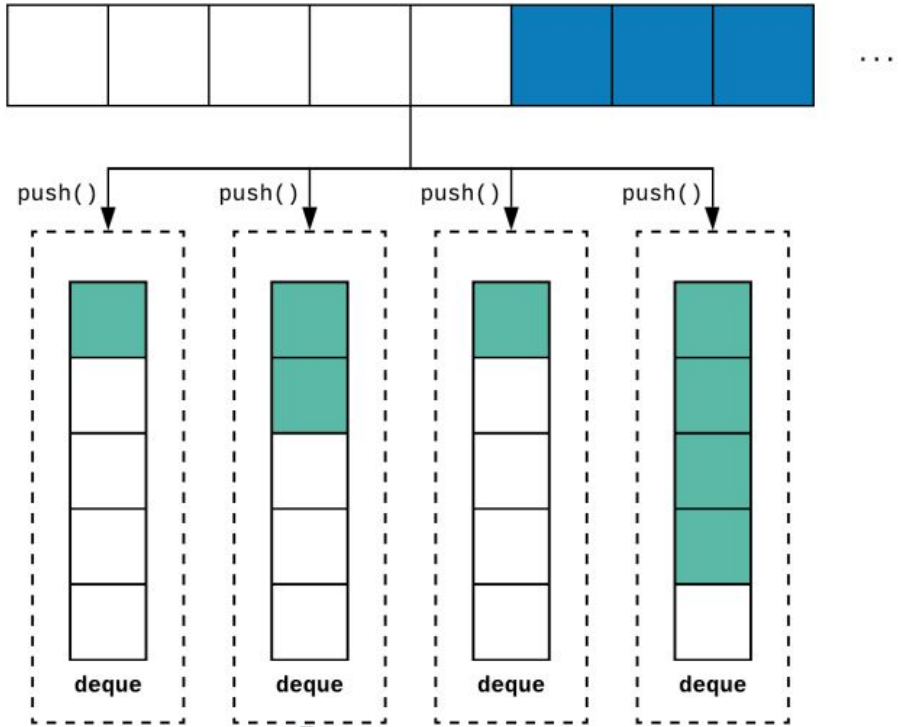


Ref : https://theboreddev.com/discover-java-forkjoinpool/

Thread Pools

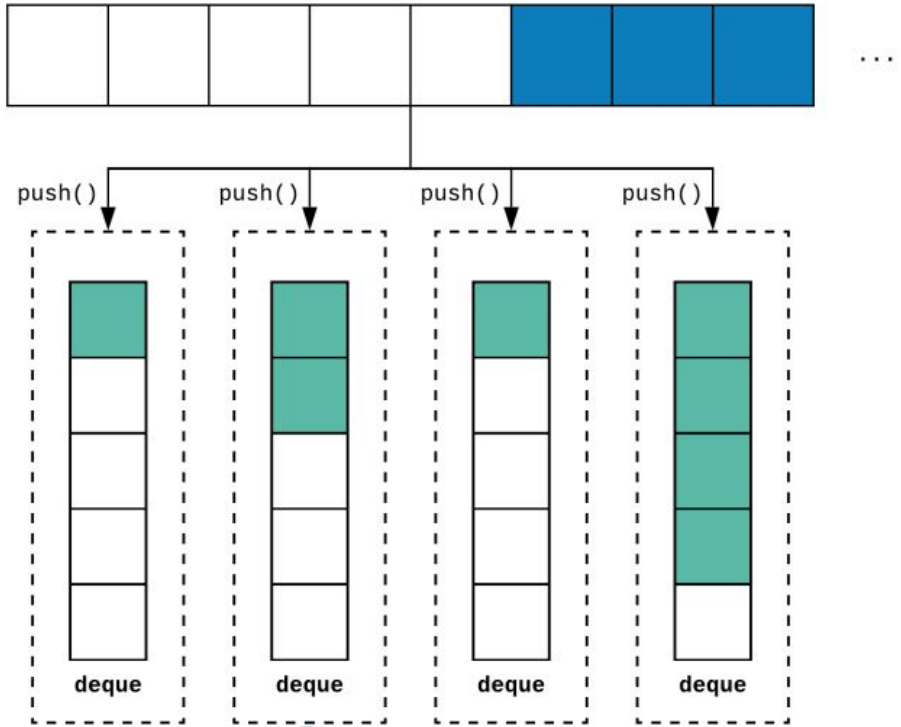
Create once

- Init fixed #threads in beginning (Or..)
- Receive all tasks to be processed.
- Divide tasks to use parallelism.



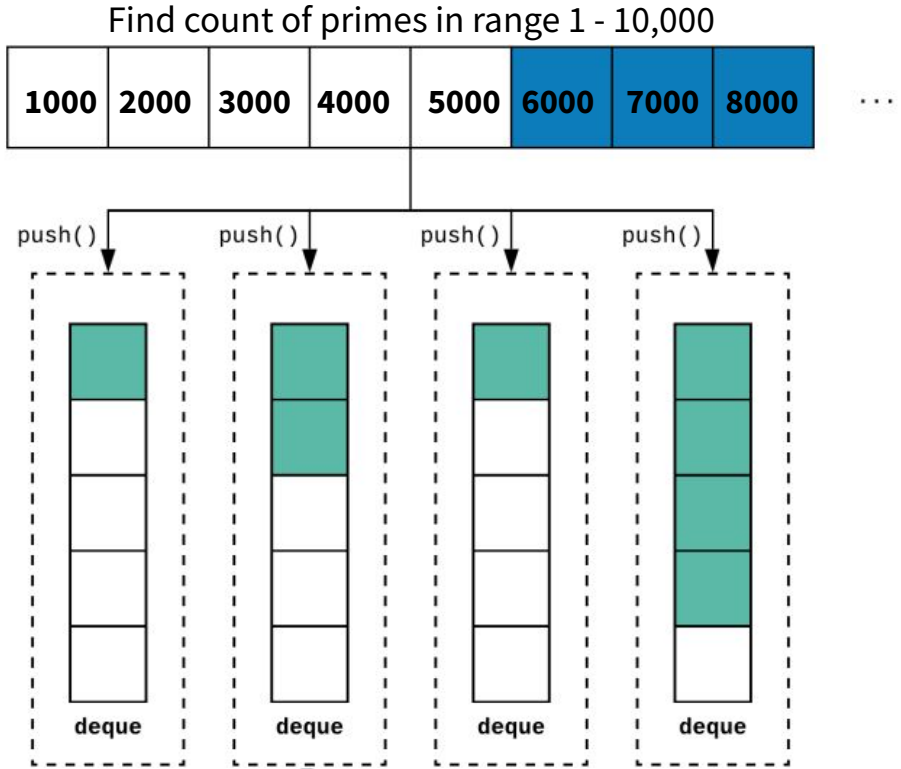
Ref : https://theboreddev.com/discover-java-forkjoinpool/

Thread Pools - Example



Ref : https://theboreddev.com/discover-java-forkjoinpool/

Thread Pools - Example

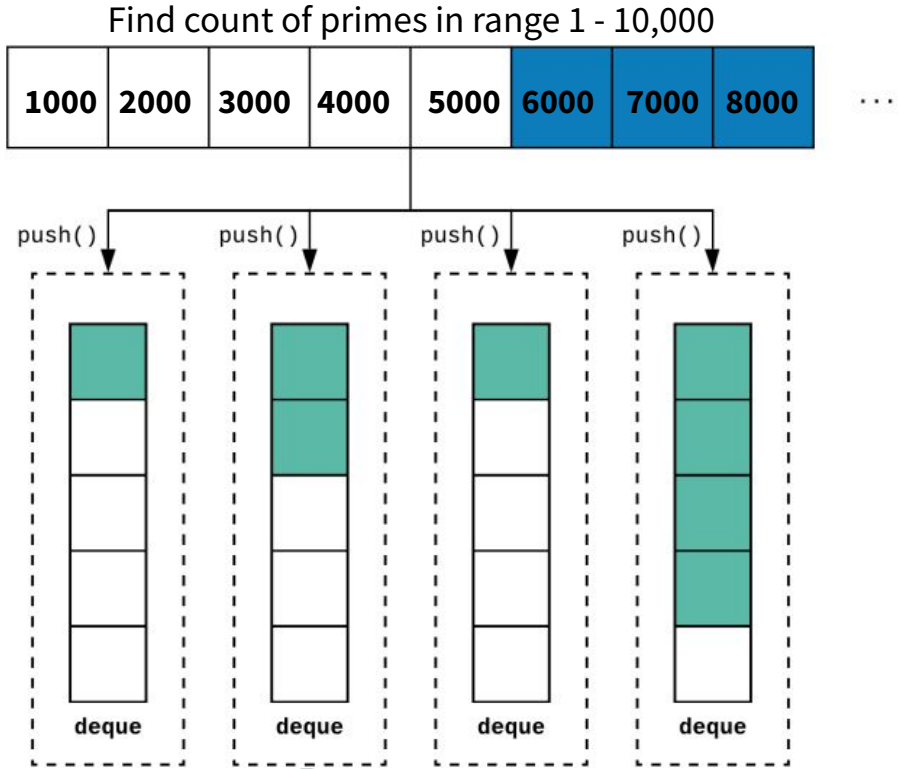


Ref : https://theboreddev.com/discover-java-forkjoinpool/

Thread Pools - Example

Counting Prime Numbers

- Push the task (1-10,000) to the pool.
- Pool decides to split task if larger than defined granularity.
- Each task is run parallelly and final answers combined before returning.

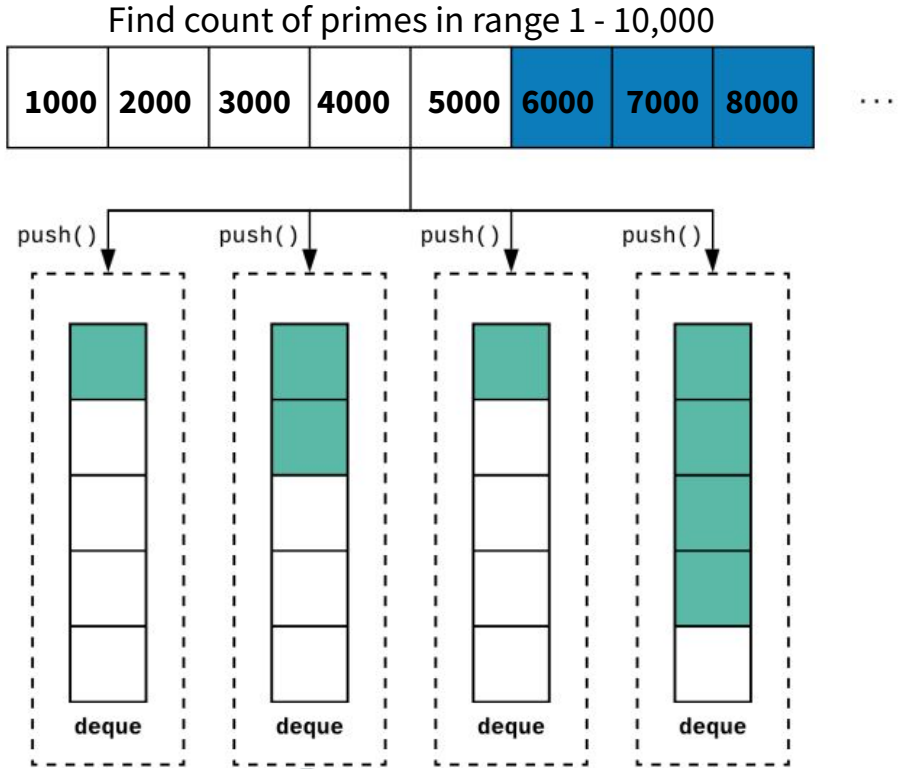


Ref : https://theboreddev.com/discover-java-forkjoinpool/

Thread Pools - ISSUES

Counting Prime Numbers

-

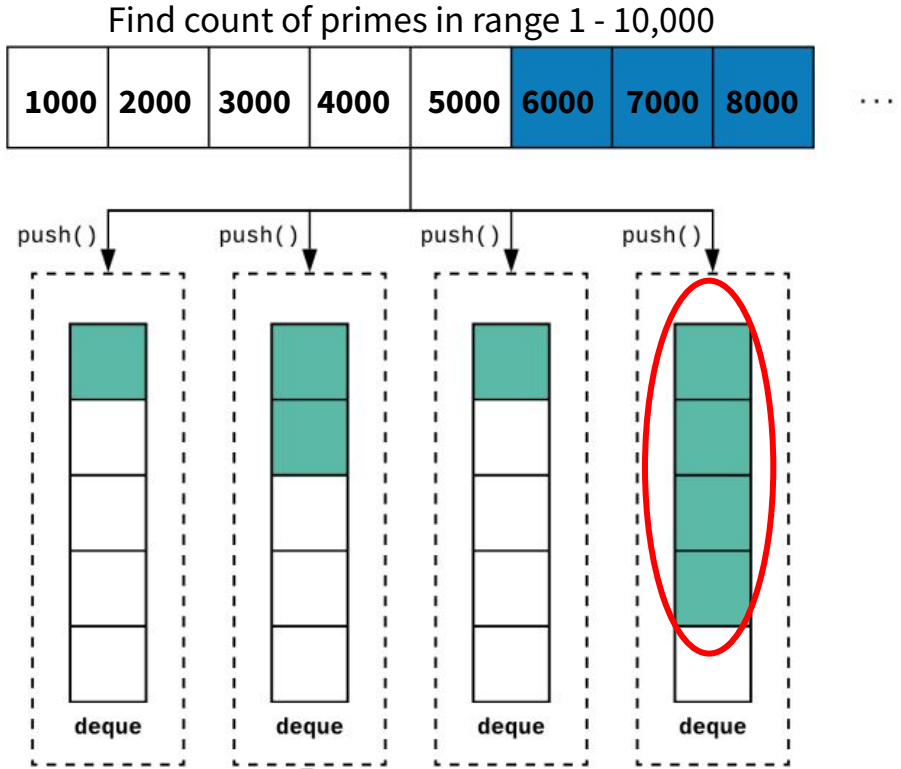


Ref : https://theboreddev.com/discover-java-forkjoinpool/

Thread Pools - ISSUES

Counting Prime Numbers

- **Skew**
- Some task can take more time :
  - Counting primes in range 8000-9000 takes longer than for 1000-2000.
- Some thread can get more runtime than others.

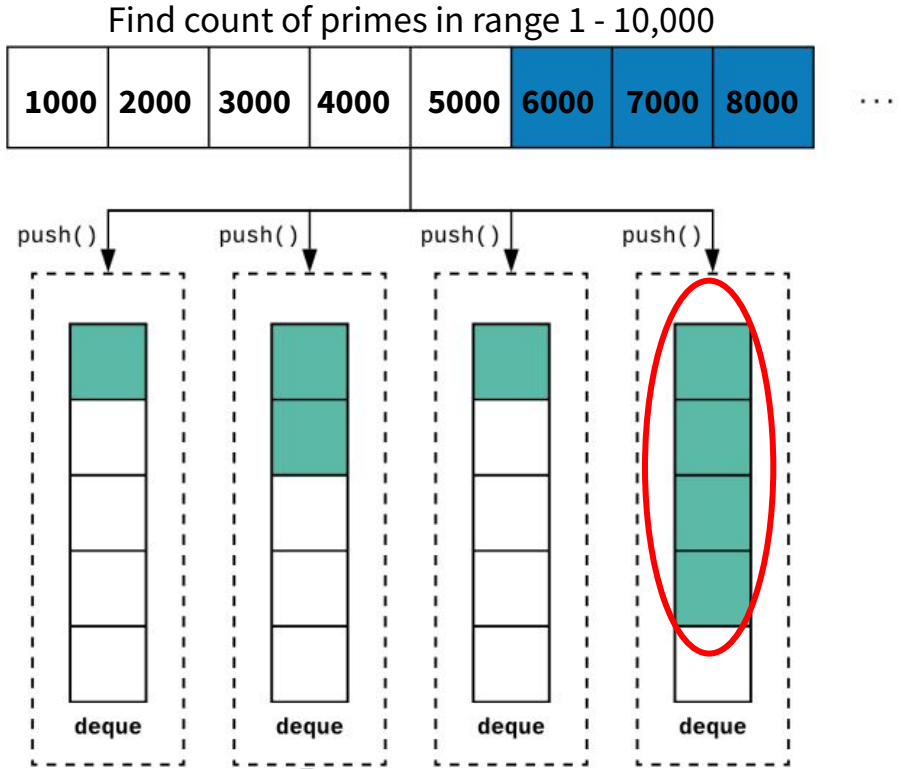


Ref : https://theboreddev.com/discover-java-forkjoinpool/

Thread Pools - ISSUES

Counting Prime Numbers

- **Skew**
- Some task can take more time :
  - Counting primes in range 8000-9000 takes longer than for 1000-2000.
- Some thread can get more runtime than others.



Work Stealing enters

Ref : https://theboreddev.com/discover-java-forkjoinpool/

Work Stealing

- Free threads become thieves.

2.
Literature

Studying existing optimizations

Three broad categories

Task Granularity

Local Sensitivity

Task Queue

Three broad categories

Task Granularity

Determining how small or heavy the final task should be.

Local Sensitivity

Task Queue

Three broad categories

Task Granularity

Determining how small or heavy the final task should be.

Too heavy : Skew!

Too small : Dominating overheads

Local Sensitivity

Task Queue

Three broad categories

Task Granularity

Determining how small or heavy the final task should be.

Too heavy : Skew!

Too small : Dominating overheads

Adjust granularity in runtime.

Local Sensitivity

Task Queue

Three broad categories

Task Granularity

Determining how small or heavy the final task should be.

Too heavy : Skew!

Too small : Dominating overheads

Adjust granularity in runtime.

Local Sensitivity

Considering architecture related overheads such as cache invalidation .

Task Queue

Three broad categories

Task Granularity

Determining how small or heavy the final task should be.

Too heavy : Skew!

Too small : Dominating overheads

Adjust granularity in runtime.

Local Sensitivity

Considering architecture related overheads such as cache invalidation.

Try to keep tasks within a processor by using common queue, mailbox, etc.

Task Queue

Three broad categories

Task Granularity

Determining how small or heavy the final task should be.

Too heavy : Skew!

Too small : Dominating overheads

Adjust granularity in runtime.

Local Sensitivity

Considering architecture related overheads such as cache invalidation.

Try to keep tasks within a processor by using common queue, mailbox, etc.

Task Queue

Reducing time spent by threads to just obtain a task from the queue.

Three broad categories

Task Granularity

Determining how small or heavy the final task should be.

Too heavy : Skew!

Too small : Dominating overheads

Adjust granularity in runtime.

Local Sensitivity

Considering architecture related overheads such as cache invalidation.

Try to keep tasks within a processor by using common queue, mailbox, etc.

Task Queue

Reducing time spent by threads to just obtain a task from the queue.

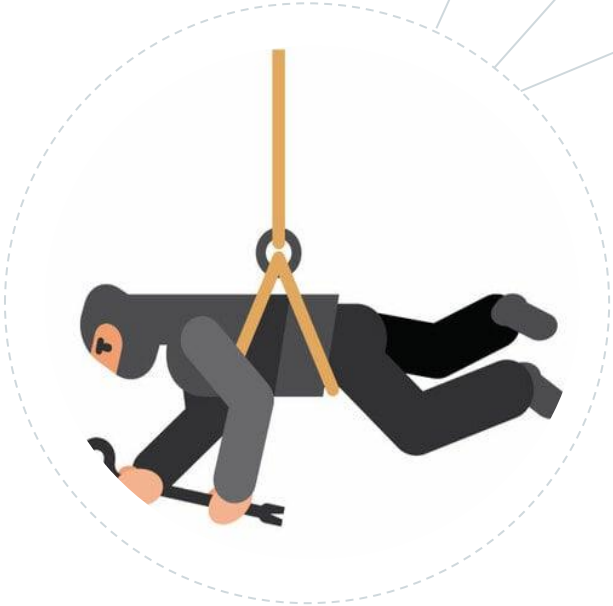
LockFree implementation,Hardware implementation, shared and private data section

3.
Design

Motivation and Implementation

Stealing consumes too much CPU

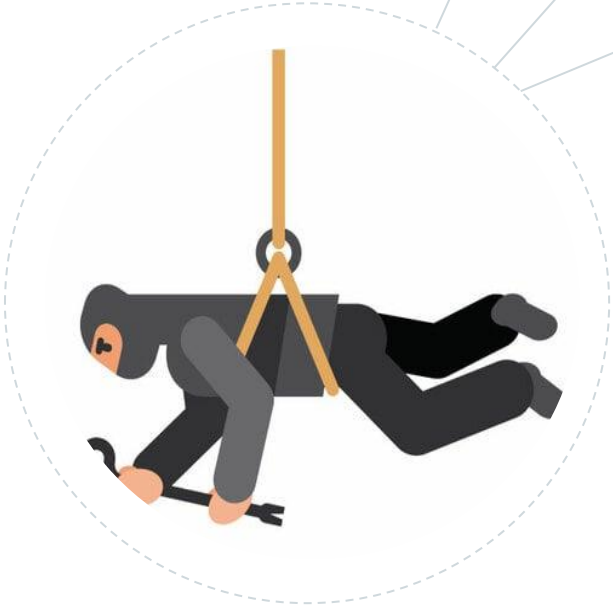
If not efficient enough, overall stealing time becomes significant portion of total runtime.



Stealing consumes too much CPU

If not efficient enough, overall stealing time becomes significant portion of total runtime.

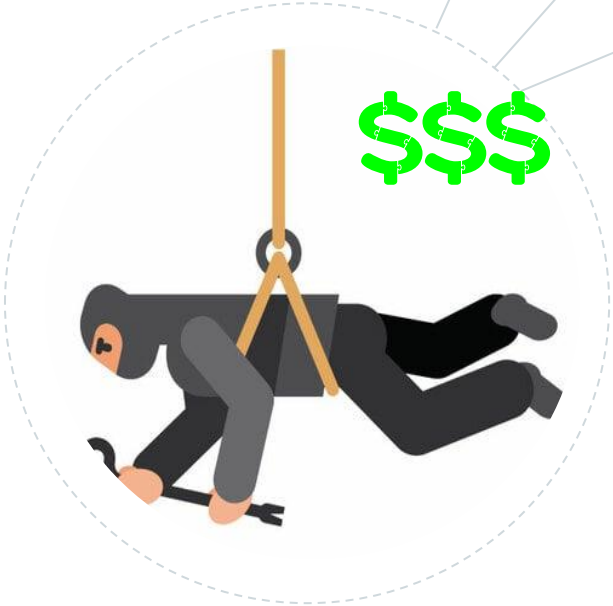
Failed steal attempts are frequent.



Stealing consumes too much CPU

If not efficient enough, overall stealing time becomes significant portion of total runtime.

Failed steal attempts are frequent.



Design

Thread Selection

Granularity Adjustment

Design

Thread Selection

memoStealing :

Instead of deciding whom to steal everytime (1000s / second), determine the busiest worker and steal its tasks repeatedly until empty.

Granularity Adjustment

Design

Thread Selection

memoStealing :

Instead of deciding whom to steal everytime (1000s / second), determine the busiest worker and steal its tasks repeatedly until empty.

Once empty, look for next busiest and continue.

Granularity Adjustment

Design

Thread Selection

memoStealing :

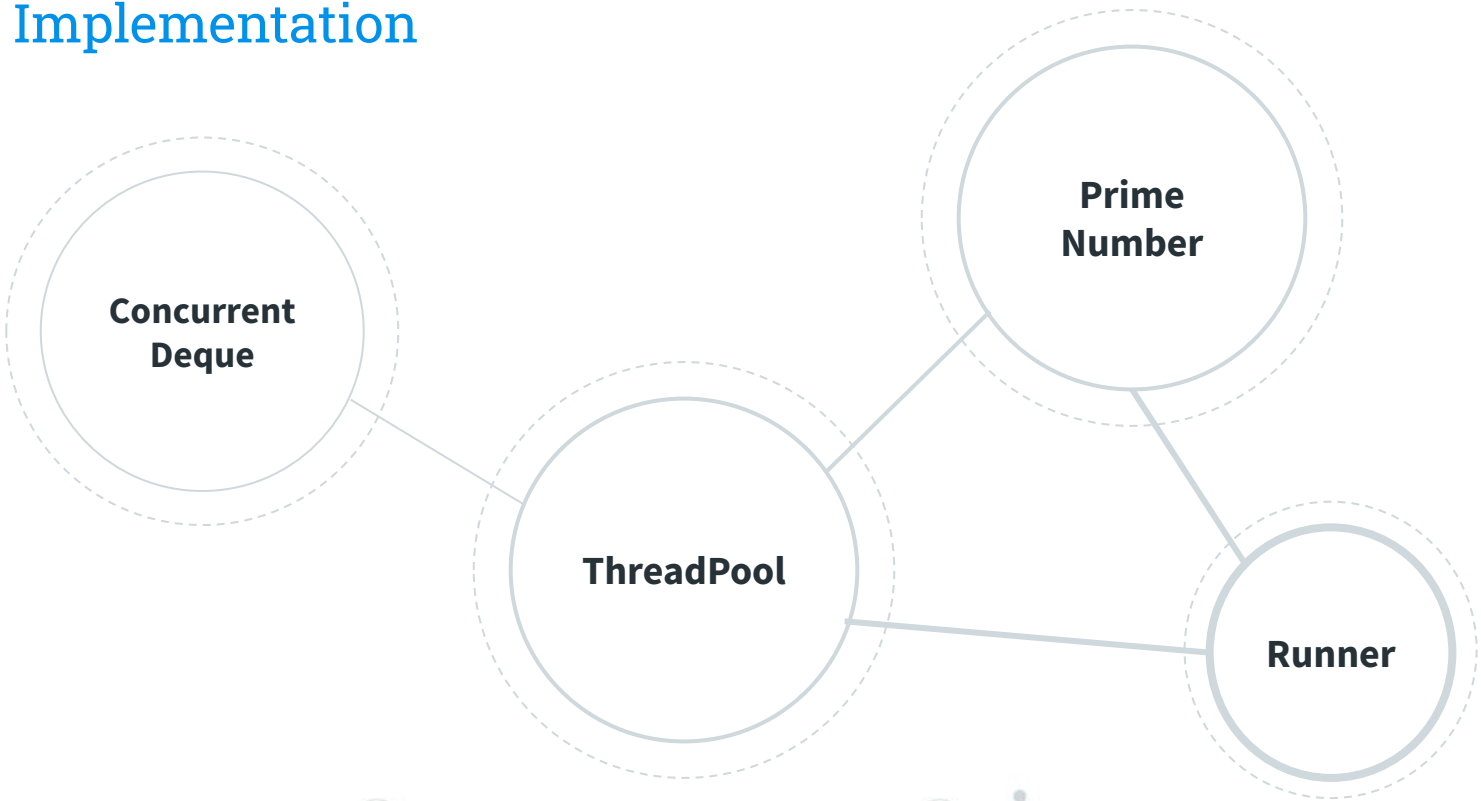
Instead of deciding whom to steal everytime (1000s / second), determine the busiest worker and steal its tasks repeatedly until empty.

Once empty, look for next busiest and continue.

Granularity Adjustment

Determine the irregularity of task loads of all threads. If more than a threshold, decrease the granularity by 10% for better distribution.

Implementation



Concurrent Deque

class ConcurrentDeque{

- void **pushBottom**(Runnable r)
- Runnable **popTop**()
- Runnable **popBottom**()
- int **size**()
- boolean **isEmpty**()

}

Concurrent Deque

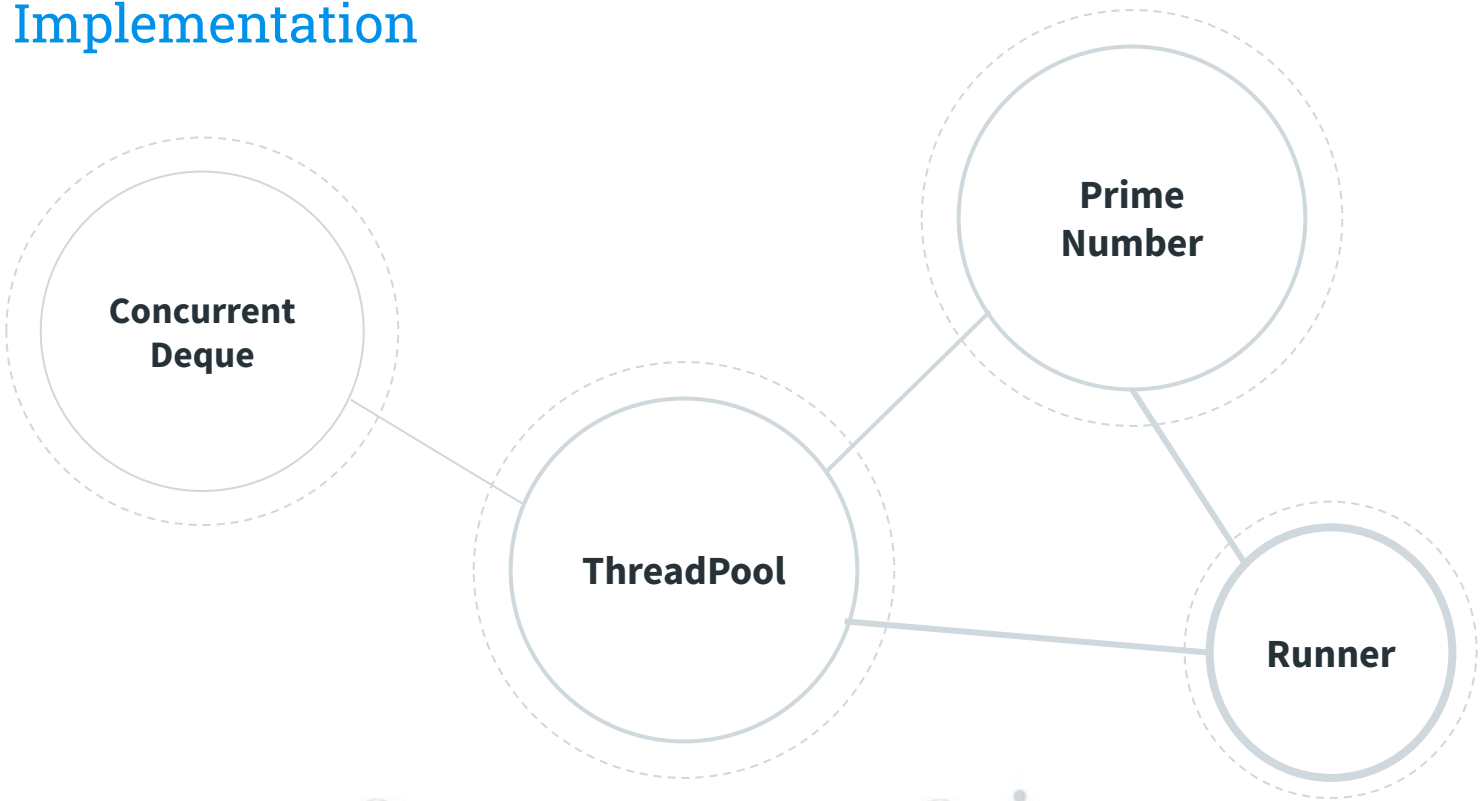
class ConcurrentDeque{

- void **pushBottom**(Runnable r)
- Runnable **popTop**()
- Runnable **popBottom**()
- int **size**()
- boolean **isEmpty**()

}

The concurrent Dequeue is lock-free and unbounded with the help of atomic CAS() operations and resizable Circular Array data structure.

Implementation



ThreadPool

class workStealing{

- void **submit**(Runnable task)
- Runnable **take**()
- Runnable **steal**()
- void **awaitTermination**()
- void **shutdown**()

}

ThreadPool

class workStealing{

- void **submit**(Runnable task)
- Runnable **take**()
- Runnable **steal**()
- void **awaitTermination**()
- void **shutdown**()

}

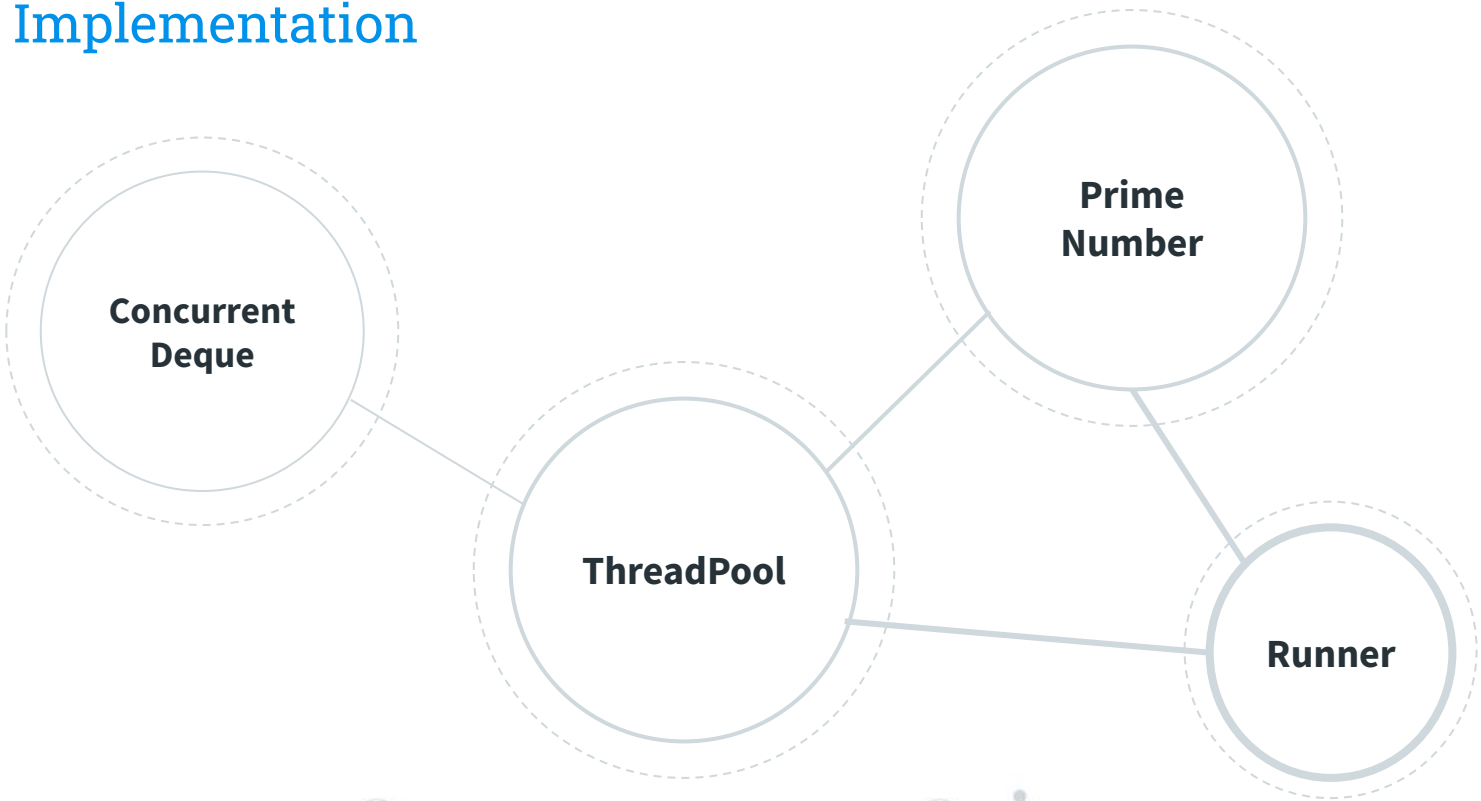
The threadPool constructor takes input the fixed number of threads to instantiate which immediately starts looking for tasks until explicitly interrupted.

The constructor also needs a stealing algorithm such as random stealing, bestOf2, etc.

submit() is used to input a Runnable task.

shutdown() is a blocking call to wait for all threads to finish and then terminate.

Implementation



PrimeNumbers

class PrimeNumbers{

- void **compute**()
- void **makeFine**()

}

PrimeNumbers

class PrimeNumbers{

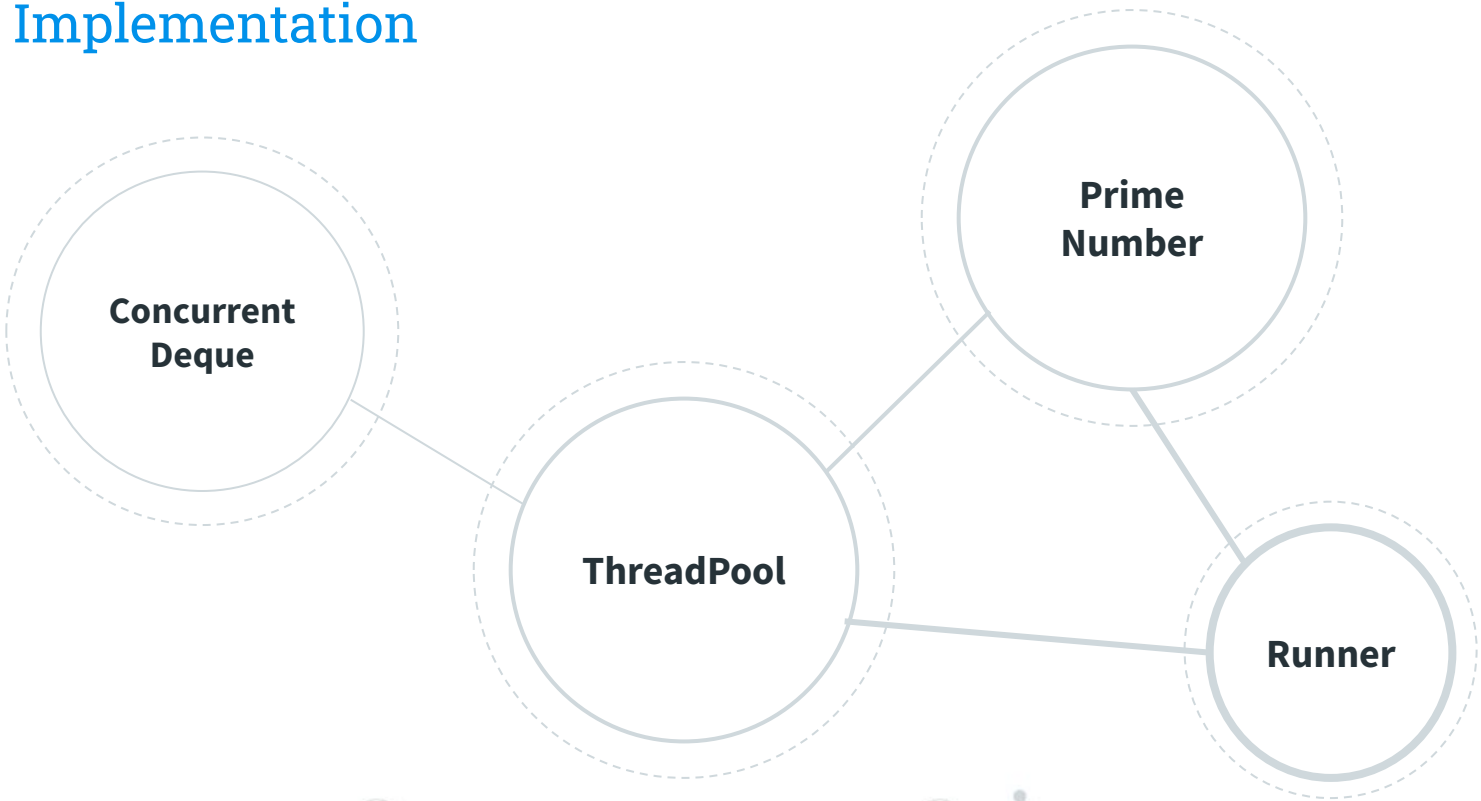
- void **compute**()
- void **makeFine**()

}

The compute() function checks whether the task is small enough. If not, it bifurcates the task and submits one-half to the pool while starts executing the remaining half.

Provides a makeFine() method to be used by the ThreadPool to decrease the granularity by 10% whenever needed.

Implementation



Runner

class Runner{

- void **main**()

}

Runner

class Runner{

- void **main**()

}

Create object of the threadpool in beginning.

Iteratively creates object of PrimeNumbers class and submits to the pool and waits for completion.

Calculated runtime and other metrics after Warmup is complete.

4.
Evaluation

Analysing metrics

Two-step Analysis

- Performance of Thread Selection approach.

Two-step Analysis

- Performance of Thread Selection approach.
- Performance of hybrid
  (Thread Selection + Granularity adjustment).

Different parameters studied

Different parameters studied

Runtime

Total time to execute task varying with number of threads in the Pool.

Different parameters studied

Runtime

Total time to execute task varying with number of threads in the Pool.

Throughput

Inverse of runtime. Helps understand performance more intuitively.

Different parameters studied

Runtime

Total time to execute task varying with number of threads in the Pool.

Throughput

Inverse of runtime. Helps understand performance more intuitively.

Time spent Stealing

For varying thread count, %age of runtime spend during an execution to steal tasks from other threads.

Different parameters studied

Runtime

Total time to execute task varying with number of threads in the Pool.

Throughput

Inverse of runtime. Helps understand performance more intuitively.

Time spent Stealing

For varying thread count, %age of runtime spend during an execution to steal tasks from other threads.

Steal success %

For total number of steal attempts, how many were actually successful.

Two-step Analysis

- Performance of Thread Selection approach.
- Performance of hybrid
  (Thread Selection + Granularity adjustment).

Performance of Thread Selection approach.

Performance of Thread Selection approach.

- Different implementations compared:
  - First available
  - Best of 2 random
  - Best of all
  - *memoStealing*

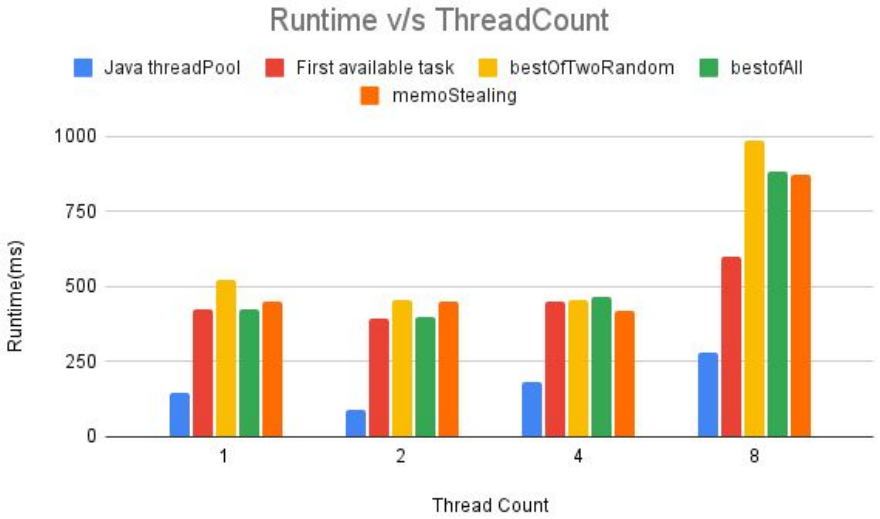
Performance of Thread Selection approach.

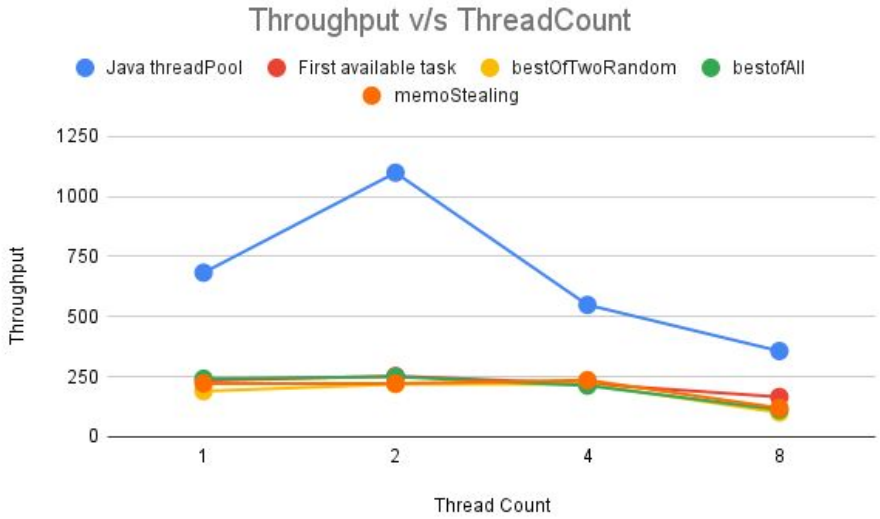
- Different implementations compared.
- Fixed granularity of 100 i.e. a task will be split until its range <= 100.

Performance of Thread Selection approach.

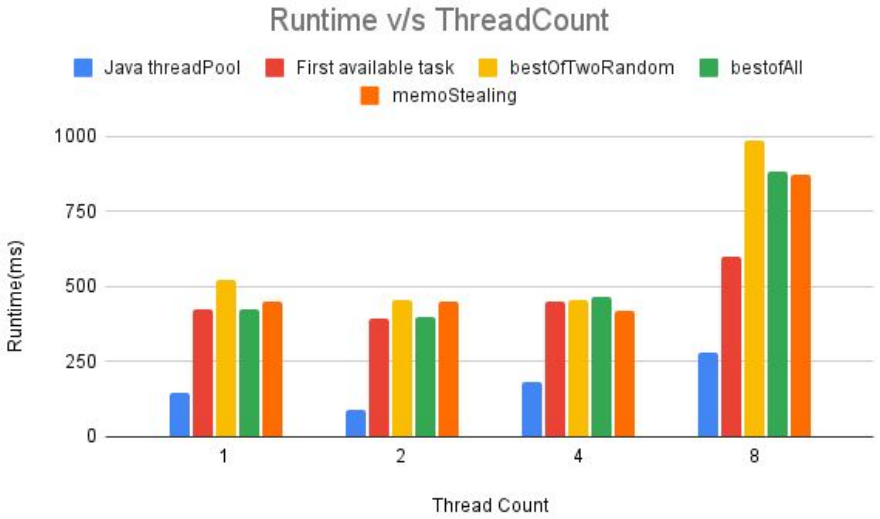
- Different implementations compared.
- Fixed granularity of 100 i.e. a task will be split until its range <= 100.
- Vary the thread count from 1 (no stealing) to 8

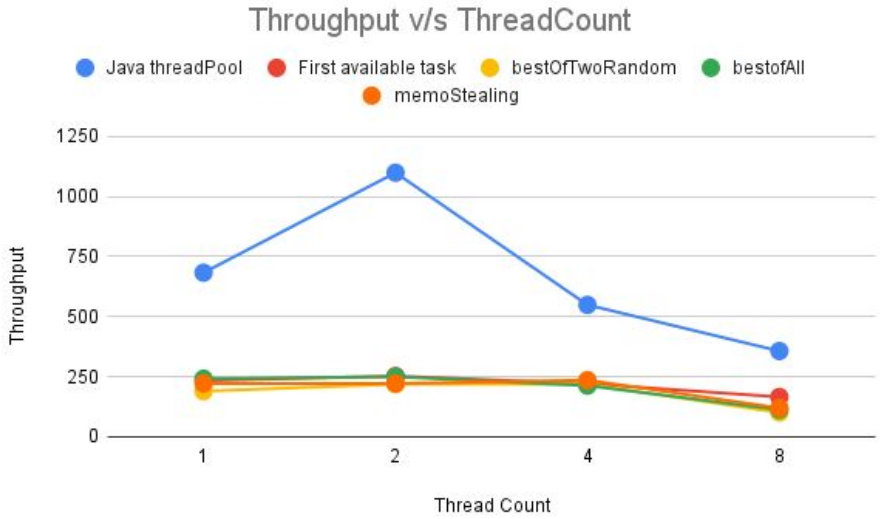
Performance of Thread Selection approach.





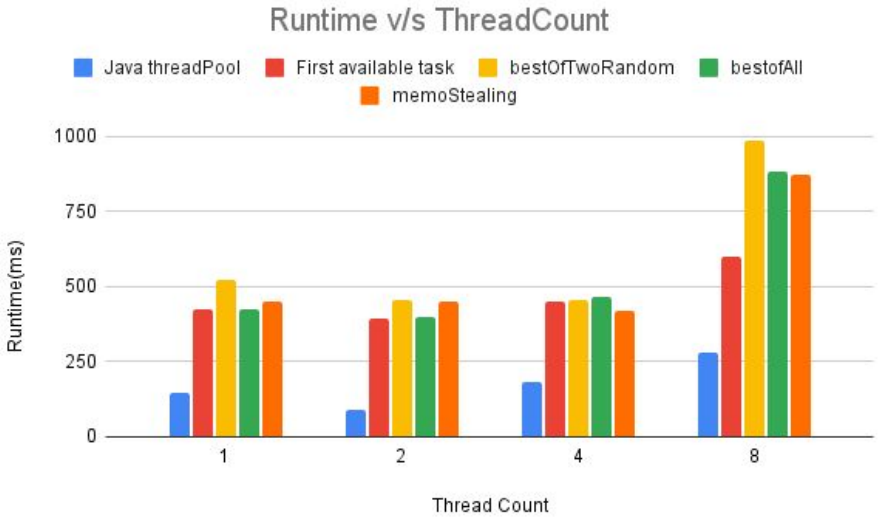
Performance of Thread Selection approach.

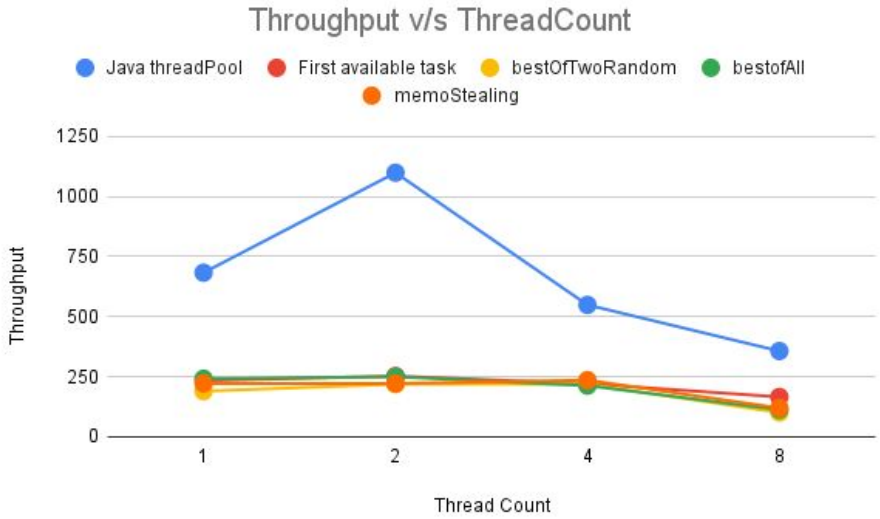




Our custom implementation has considerable overheads when compared to Java's inbuilt lib.

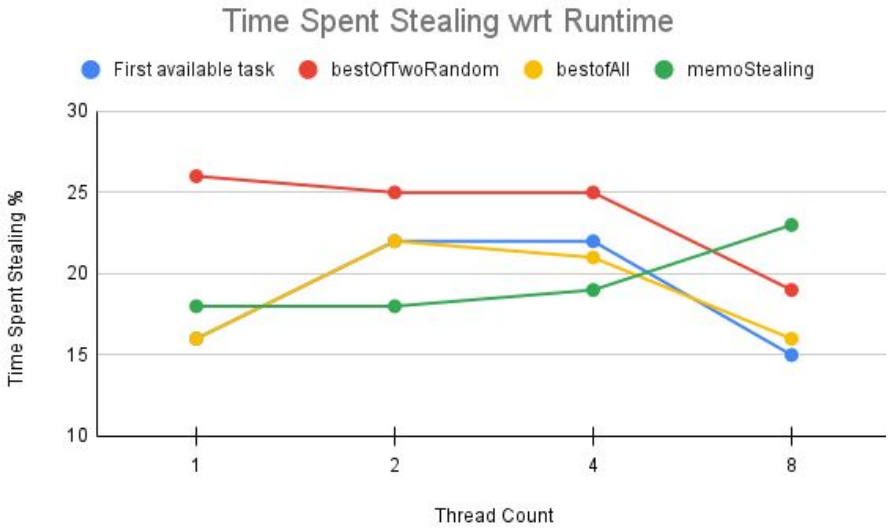
Performance of Thread Selection approach.

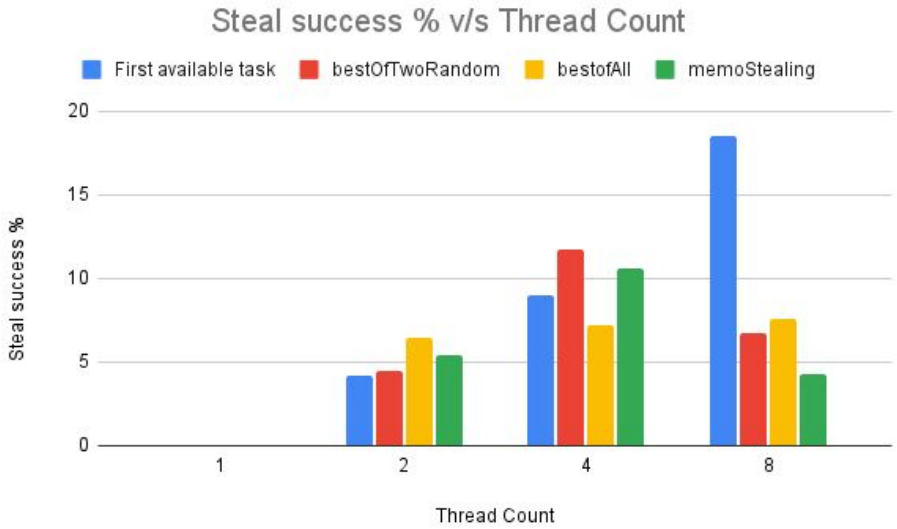




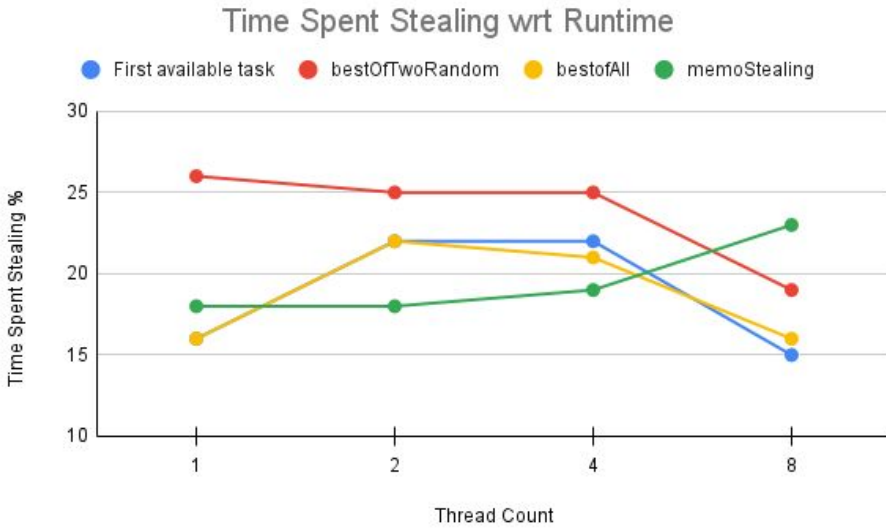
All the 4 thread selection algorithms perform roughly the same in terms of runtime.

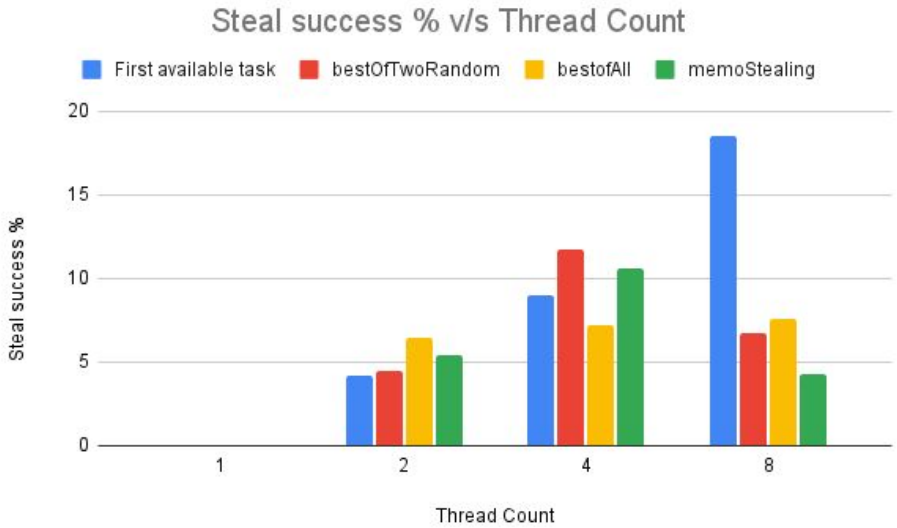
Performance of Thread Selection approach.





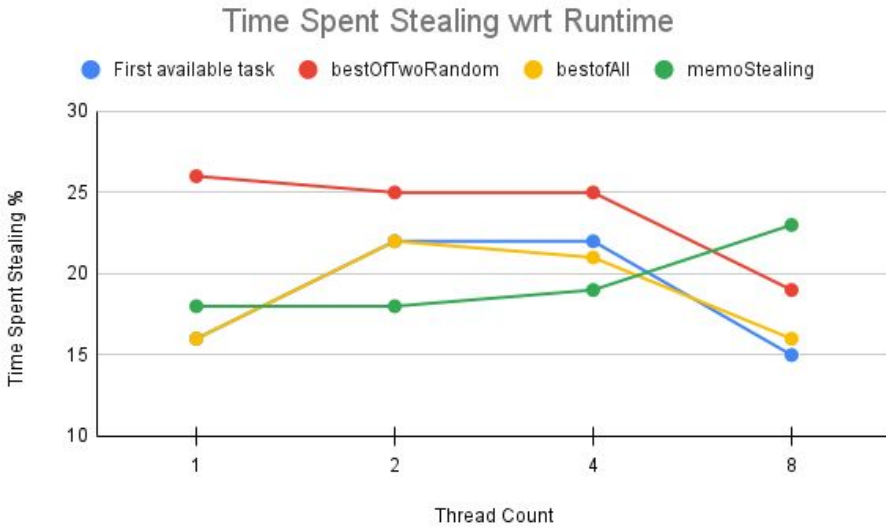
Performance of Thread Selection approach.

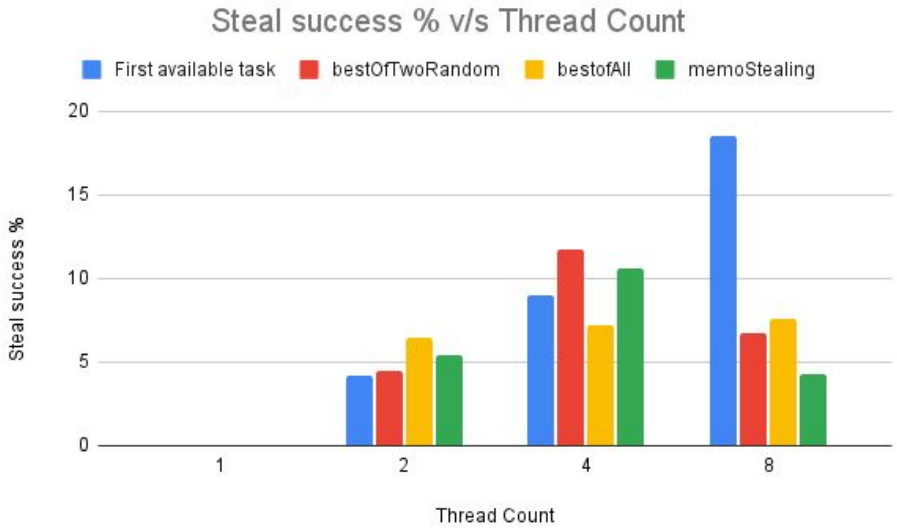




The proposed *memoStealing* spent less time on stealing compared to others. The steal success was better as well. But only until we hit 8 thread counts.

Performance of Thread Selection approach.





The proposed *memoStealing* spent less time on stealing compared to others. The steal success was better as well. But only until we hit 8 thread counts.

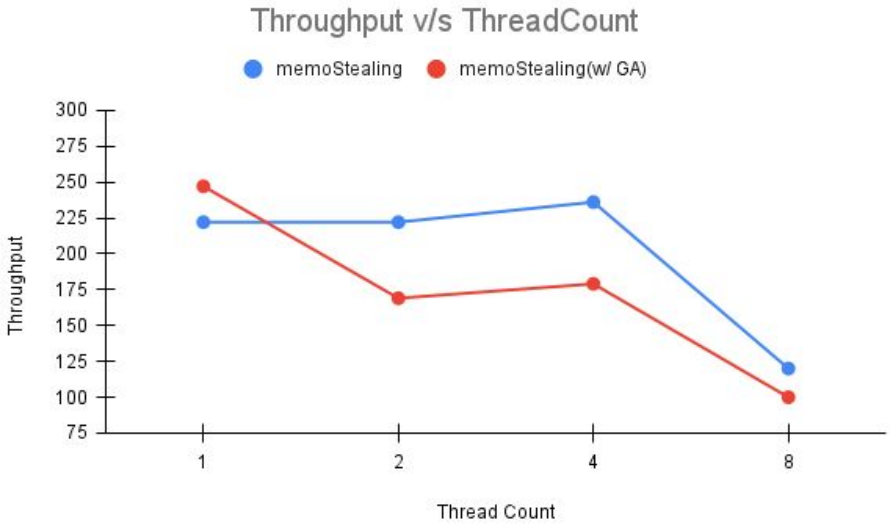
Since all threads begin looking at a single victim, the contention to steal a task increases alot.

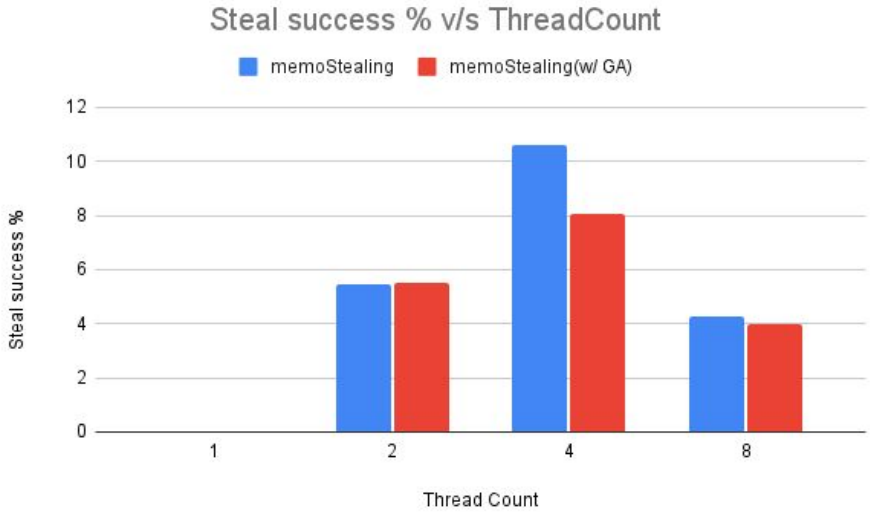
Two-step Analysis

- Performance of Thread Selection approach.

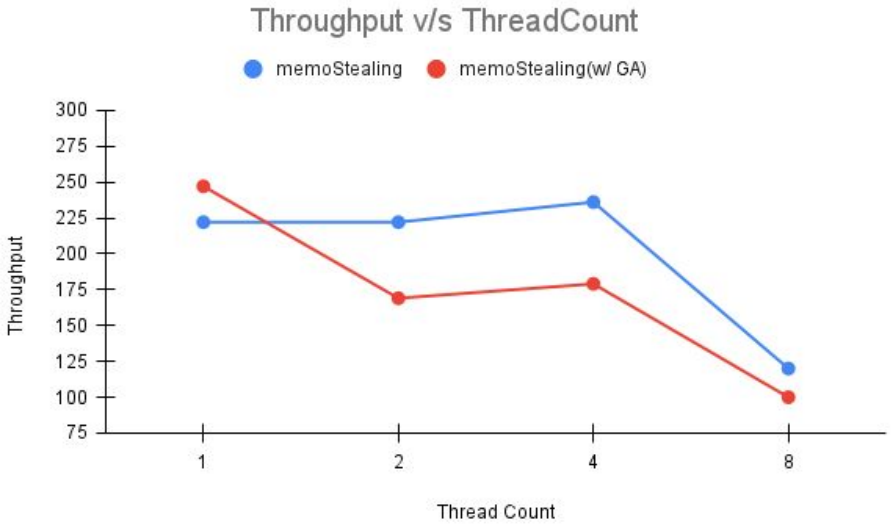
- Performance of hybrid
  (Thread Selection + Granularity adjustment).

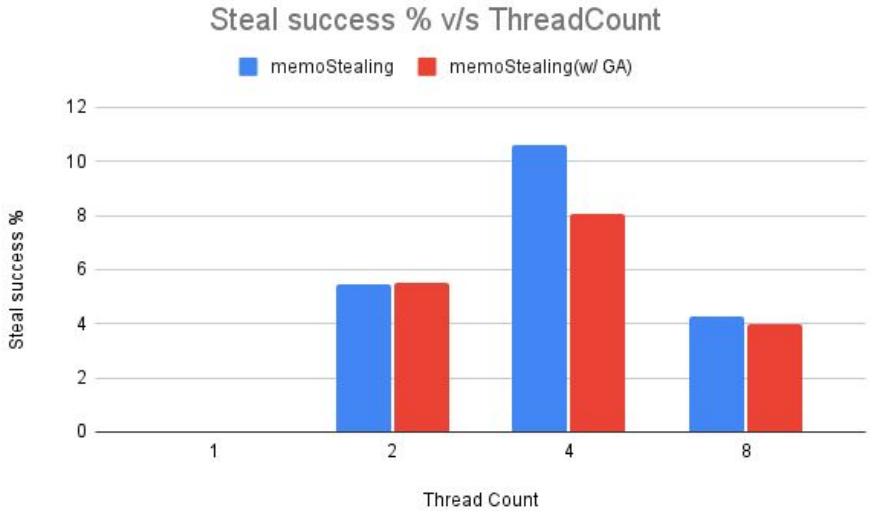
Performance of hybrid





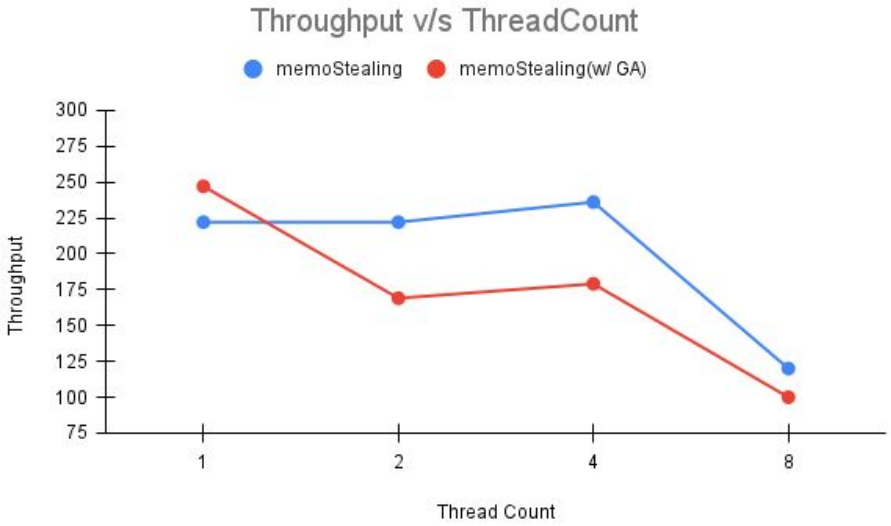
Performance of hybrid

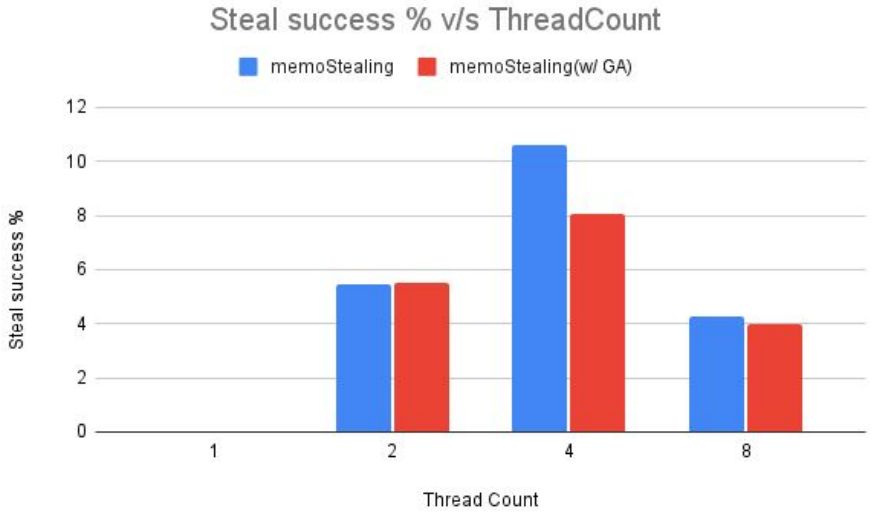




The Granularity Adjustment modification is effective for low contention and its performance benefit becomes an overhead quickly as we go towards higher number of threads.

Performance of hybrid





The Granularity Adjustment modification is effective for low contention and its performance benefit becomes an overhead quickly as we go towards higher number of threads.

The taskQueues are constantly changing, so by the time we decide to reduce the granularity, the taskQueues have already changed making our decision incorrect.

5.
Conclusion

Summary & Future Work

Summary

- Proposed an optimization of work-stealing using modified Thread Selection & Dynamic Granularity Adjustment.

Summary

- Proposed an optimization of work-stealing using modified Thread Selection & Dynamic Granularity Adjustment.

- The Thread Selection performed well until 4 threads in parallel. Higher threads made it slow due to contention.

Summary

- Proposed an optimization of work-stealing using modified Thread Selection & Dynamic Granularity Adjustment.
- The Thread Selection performed well until 4 threads in parallel. Higher threads made it slow due to contention.
- The dynamic granularity adjustment had a high overhead which overpowered its benefits

Future Work

- Bridge the performance gap between our implementation and java's inbuilt implementation.

- Include Local Sensitivity based optimizations to see how the overall implementation performs.

Danke!